RVM: Recoverable Virtual Memory, Release 1.3

Henry M. Mashburn, Mahadev Satyanarayanan, David Steere, Yui W. Lee,
School of Computer Science, Carnegie Mellon University[1] 17 September 1997

"RVM provides an unstructured recoverable virtual memory. The recoverable storage is represented by Unix files or disk partitions that applications can map at page granularity into the address space of a process. Simple, non-nested atomic transactions guarantee permanence of changes to recoverable storage across system crashes. Applications can schedule transaction logging actions to enhance performance. The design stresses simplicity, ease of use, and high performance. Unix compatibility is standard, while optional Mach-specific extensions are supported for additional flexibility and performance. RVM has been extensively used in the clients and servers of the Coda File System, and in the Venari system.",

Contents

Chapter 1

Introduction

RVM, a *Recoverable Virtual Memory* system, is a mechanism to support persistent virtual memory in the face of system crashes. The current state of affairs is that software rather than hardware is limiting factor is systems reliability [?], and so RVM is intended to assist in building more robust software. RVMs principal reliability criterion is to provide permanence of committed changes and integrity of data over operating system crashes. The fatal nature of these all-too-frequent events often leaves data structures in a shambles. But the all-or-none property of atomic transactions can insure consistency of data structures over crashes and makes the application significantly more robust with small cost. Using simple, non-nested local transactions, it provides a way to automatically checkpoint the state of memory before an operation and restore that state in the event of a crash.

RVM can also provide some resilience in the face of localized media failures, but massive media failures such as head crashes are not recoverable since storage is not mirrored. Applications must rely on periodic backups, or other mechanisms, to recover from these rare events. While mirroring isnt supported, RVM does not preclude it: applications can provide device drivers for storage media that mirror data without RVMs knowledge.

Because RVM is not a server, but a tool for building servers, there are important differences between RVM and database transaction systems. RVM does not provide application level synchronization or serialization functions, although RVM synchronizes its internal actions so that applications can use multiple threads. Also, RVM manages storage as an unstructured address space; no object abstractions are supported.

None of these responsibilities are necessary for simple transactions, although some, or all, will be needed for a given application. RVMs design is restricted to only the minimum functionality necessary to support the restoration mechanisms needed for exception handling in a persistent virtual memory. Other services can be provided by libraries and servers that use RVM, or by application-specific functions.

Much of the motivation for building RVM has come from experience with Camelot [?], a large, full-featured transaction system. While Camelot is much more capable than RVM, it was discovered in building CODA [?], a high-availability file system, that most of the benefit of transactions was gained by using only a fraction of Camelots facilities. RVM is an experiment to discover the minimum transaction processing functionality useful in building complex applications such as file servers and other non-database systems.

1.1 Design Assumptions

The decision to use RVM rather than a more complex transaction system can be made by considering the underlying design assumptions. If the following constraints are not too restrictive for the application, RVM will be a good choice:

- Virtual memory can hold all of the applications temporary data.
- The backing store for virtual memory is orthogonal to data storage.
- Robustness in the face of system crashes is the principal reliability criterion.
- Transactions do not extend across shutdown or system crashes. Transactions must be completed before shutdown and are aborted by a crash.
- Nested transactions are not required.
- A single concurrent server at a single site is sufficient.
- Success of disk operations are correctly reported, and disk sector writes are assumed to be atomic: all or none of a sector is written.

These assumptions greatly simplify a transaction system and reflect RVMs design and implementation philosophy that stresses simplicity and ease of use. To make RVM highly portable, the implementation uses only Unix features that are standard across a wide variety of Unix and Mach/Unix implementations. Mach-specific extensions are used only for optional features and performance enhancements. Optional features, when provided, exact no penalty when not chosen; at worst, one will not receive the benefits of performance enhancing options.

While RVM directly supports only a single server at a single site, an application can provide the communication mechanisms to integrate multiple instantiations of RVM into a widely distributed system. RVMs first application, CODA, will use this technique to manage multiple file servers.

RVM is inoffensive and inconspicuous in action: it does not complicate or preclude the use of standard tools such as profilers and debuggers. Recognizing that the disk operations inherent in transaction processing are expensive, RVM allows the application considerable freedom to manage such operations.

Chapter 2

The RVM Design

The basic structure of RVM is similar to most transaction systems. It has a virtual memory cache manager, disk-based data storage, and a log manager that records changes to the data to provide recovery after a crash. Changes to the data are effected by transactions only. Because RVM supports a byte-addressable recoverable virtual memory rather than a collection of typed records, its disk storage and cache management differ most from record-oriented systems. Each of these systems is detailed in this section.

In addition to the basic design features, RVM offers two additional, optional mechanisms for the convenience of application builders. A loader is provided that allows the creation and maintenance of a load map for recoverable storage. The loader formalizes the layout and mapping of recoverable storage using the mechanisms described in this chapter. Using the loader also minimizes the number of library calls that must be programmed. An allocator permits recoverable storage to also be used as a heap. The functionality is similar to malloc and free, although with persistence. The loader is used to automate reloading of the heap after shutdown or a crash. Details of these options are described in Chapter 5 (RVM Segment Loader) and 6 (RDS, A Dynamic Heap Allocator).

2.1 Recoverable Storage

RVM manages recoverable storage in *segments*, which can be thought of as being similar to Multics segments, although they can be much larger. Any number of segments can be created and used on a machine, limited only by its storage resources. Multiple instantiations of RVM can run simultaneously on the same machine, although sharing of segments among processes is severely constrained.

2.1.1 Segment Structure

Segments are represented by files or disk partitions (or any other random access device) which RVM presumes to be structureless. Since Unix can name partitions as special files, storage for segments will be referred to from now on as files. The file name is also the segments name in RVM. The maximum segment size is determined by the static allocation of the file, with a limit of 2^{64} bytes so that large disks can be supported. The 64 bit addresses are represented as two unsigned long integers since not all machines support an eight byte integer format.

The segment file represents the image of the recoverable memory with an address space from zero to the maximum size of the file in bytes. The contents of the data file is the applications responsibility, although it can only be modified in RVM via transactions.

The address space is linearly mapped onto the data file so locality of data in the address space implies locality in the file. When partitions are used, applications can minimize disk accesses and head movement by allocating related data locally. Alternatively, dispersing data copies in the address space can give some protection from local media errors without resorting to mirrored storage. Since only local damage is the usual failure mode, dispersed copies are not likely to be damaged simultaneously (see [?]).

2.1.2 The Log and Its Operations

To insure the integrity of the data segments across crashes, a dedicated log file is used by RVM to record changes to segments as transactions are processed The log file records changes to all segments used by the application and is declared to RVM at initialization time. The actual file can be a disk partition, but using a RAM disk, which has no seek or rotational latencies, will give better performance. If a RAM disk is used, it should be provided with an uninterruptable power supply for protection from power failures.

If two or more instantiations of RVM are operating on the same machine and disk partitions are used for the log files, the partitions should be allocated on different disk drives if possible. If a single drive must be used, the actual partitions should be placed on adjacent cylinders to minimize seek latencies. Head movement between partitions can be a serious performance limitation.

Following the example of most database management systems, RVM uses *write-ahead* logging to improve performance, recording only changes as a mapped region is modified. The changes are typically smaller than the modified region, and if modifications are scattered through a large region, the change records can be much smaller. Therefore writing only changes will often be cheaper than writing the entire region.

There are two log operations: *flush* and *truncation*. As transactions are performed, records of segment changes are created and are written to the log file in the flush operation. Periodically, the modifications represented by the log records are applied to the data file. After this is done, the records are removed from the log, and the log is said to be *truncated*. These operations are usually automatically performed by RVM, but because log management is the strongest determinant of performance, the operations are made visible and the application can use knowledge of its operations to optimize their timing.

The log file must be initialized before it can be used by RVM. Initialization is usually performed by the init_log command provided by the utility rvmutl, but, for files only, can also be done by a library call, rvm_create_log. The operation is quite simple since only the amount of storage allocated to log records need be specified. The remaining initialization and maintenance is provided by RVM. Details of log initialization and other functions of rvmutl are provided in Chapter 8 (of rvmutl).

2.1.3 Segment Mapping

The mapping of data from the data file to memory, a function supplied by the cache manager in data base systems, is supported differently by RVM. Mappings can not be automatic upon fetch of an object since a segment is an unstructured space. The application must manage the virtual memory cache by explicitly mapping and unmapping segment regions corresponding to application objects, or collections of objects.

The relation of the data cache and the data file is independent of virtual memory and should not be confused with mapping a file into virtual memory. One of the underlying assumptions of RVMs design is that page backing and data storage are orthogonal. For the application designer, this avoids having to pin pages in memory or otherwise having to manage virtual memory, except to allocate its own structures.

When a region is mapped in Unix, it is normally copied from the data file to virtual memory. This copy-on-mapping method is illustrated in Figure 2.1 (of copy on mapping). For large regions, copying can take some time, and designers should consider mapping such regions at initialization and leaving them mapped as long as they are required.

For Mach applications, segment mapping can be done in cooperation with the virtual memory system by

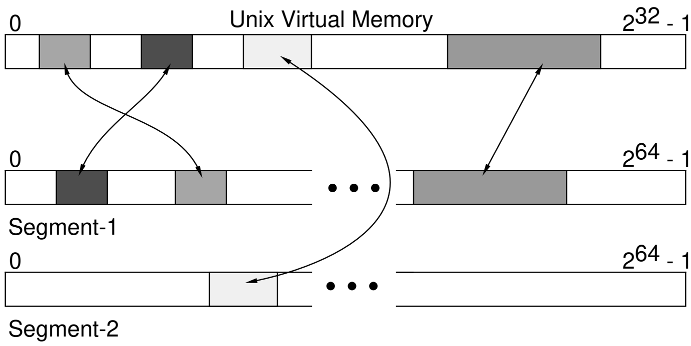

Figure 2.1: Data mapping with the copy-on-map method. All shaded areas are physically copied to the addresses specified during mapping.

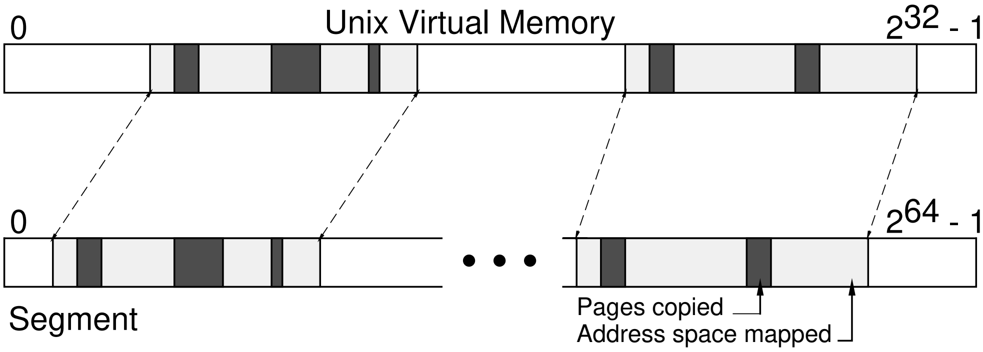

Figure 2.2: Data mapping with the copy-on-demand method. Address space is reserved by mapping (lightly shaded areas), but the pages are not physically copied until referenced (dark shading).

using Machs external pager interface. Mapped regions are copied into virtual memory on demand by the external pager, so there is no I/O delay when a region is mapped. This mapping method is illustrated in Figure 2.2 (of copy on demand).

Because virtual memory contains uncommitted changes, the pager is not permitted to write pages back to the data file. If a page must be forced out of virtual memory by the kernel, the external pager holds the page in its virtual memory, which is, in turn, backed by the Mach default pager. Modifications to the data file can be made only by transactions.

Although management of recoverable virtual memory is greatly simplified by not using the data file as the paging backing store, application designers should be aware that paging activity can seriously degrade performance and should match virtual memory usage to available physical memory for best results.

RVM provides a standard external pager, but does not preempt the external pager mechanism. When used, an external pager backs only the segments for which it was specified; other virtual memory is backed by the Mach default pager. Applications can supply external pagers for special cases, but great care must be taken to coordinate with RVMs log truncation mechanism which updates the data file. A thorough knowledge of the Mach external pager interface (see [?]), and RVM internals will be required.

Mach applications can also use the Unix copy-on-map method. The choice of method is a mapping option that is made when the first region of a segment is mapped, and can be different for each segment. When large regions are mapped and the usage pattern is sparse, using the external pager will provide the best service.

If a region is going to be completely replaced by a process that is not dependent on the original data, the region may optionally be mapped without data copying. In this case mapping simply associates the segment region with the specified virtual memory addresses. This option is available on each mapping and is not precluded by the mapping copy method chosen for the segment.

RVM, via the log, guarantees that data newly mapped by either mapping method represents the committed image of the region. The cache and segment file will remain consistent provided that applications use RVMs transaction mechanisms exclusively to modify mapped regions. However, RVM cannot protect the cached data against memory smashes, and there is no guarantee that corrupted data will not be used in transactions.

Restrictions on segment mapping are minimal, but lead to simplified implementation, and equivalent semantics between Mach and Unix mapping methods. The most important restriction is that no region of a segment may be mapped more than once by the same process. This eliminates the need for RVM to support mechanisms to maintain the consistency of multiply mapped (cached) regions. Also, no mappings can overlap in virtual memory. Attempts to multiply map, or overlap, a region will raise an exception.

In both Mach and Unix, mapping must be done in integral page sized regions, and the regions must be page aligned in virtual memory. Otherwise, regions can be mapped to any area of the address space permitted by Unix or Mach. Since a segment can be much larger than the address space, only the desired region of a segment, rounded to integral page size, need be mapped. As many segments as necessary can be simultaneously mapped by a process.

Regions can be unmapped on demand, but only when they have no uncommitted transactions. Also, RVM retains no information about segment mappings after segments are unmapped, so if absolute pointers are used in a segment, applications must specify the same base address on each mapping.

2.1.4 Segment Sharing

Sharing of a segment by multiple Unix processes or Mach tasks is only weakly supported. Inter-process sharing would greatly increase the complexity of RVM by requiring inter-process communication that can be avoided otherwise. The reason for permitting any inter-process sharing is to support maintenance utilities that will occasionally have to run concurrently with a server built with RVM. Other uses are discouraged.

All Mach threads within a task share segment mappings. If concurrent access to mappings is required in an application, using Mach is strongly suggested. In the rest of this document, the term "thread" will be used to mean either a Mach thread, Unix process, or a coroutine thread. Regardless of method, synchronization of data access is entirely the responsibility of the application.

When it is necessary that two or more processes share a segment, there must be no uncommitted transactions and the log must be empty before control of the segment can be safely transferred. To be sure the log is empty, the relinquishing process must commit or abort all transactions and force a log truncation. A function is provided to query the status of a segment, locate any uncommitted transactions, and return a flag if the log is empty. All regions that could be affected in the shared access should be unmapped before control is transferred.

The process taking control of the segment must insure that other processes that could map the shared region are blocked. It releases the segment by unmapping its regions and truncating the log. When the original process regains control, it restores the committed image in virtual memory by remapping the desired region (s).

Other mechanisms could be created by the application to insure cache consistency and possibly avoid unmapping and remapping. The conditions of no outstanding transactions and log empty are also necessary and sufficient for direct access to the segment file by the application if necessary for special operations. If such mechanisms or direct access are necessary, remember that RVM leaves interprocess communication and synchronization strictly to the application.

2.2 Transactions

RVM provides the basic transaction operations *begin*, *end*, and *abort*. As a transaction proceeds, the state of the byte ranges affected by the transaction are logically checkpointed so that they can be restored if the transaction is aborted. If the transaction commits, the new state is recorded in the log. Since RVM uses write-ahead logging, the region is not updated in the data file immediately; the log entry provides permanence.

The only restrictions on transactions are that all changes must occur within mapped regions, and that nested transactions are not supported.

2.2.1 Transaction Types

RVMs applications will perform transactions on regions of widely varying sizes. To permit the possibility of optimizing log operations, RVM supports several transaction start and commit modes which control the creation of old and new value log records.

Although most transactions are used to provide atomic changes, there are times when an application does not need virtual memory restoration on abort. In these cases, it is known that the current state of memory is worthless, and no effort need be spent preparing for restoration. RVM supports no_restore transactions when old value records are not necessary. Any transaction that cannot be aborted by the application, or does not need virtual memory restoration if aborted, is a candidate for no_restore.

Another opportunity for optimization occurs when many changes are made to a concise region. In these cases, just one old or new value log record can be created provided that the range of changes is known to the transaction manager before the changes are made. This enhances performance because the allocation and initialization costs of creating log records are often greater than the cost of copying the information, particularly for small ranges. Block copy operations are usually cheap.

To capitalize on block copy, both old and new value records are created by byte ranges. Old value records are created as the ranges are specified, while new value records created when the transaction commits will record all modifications. Also, because the ranges of modification are known, no modify function is needed and normal assignment statements can be used. The rvm_set_range function is used to specify ranges and create old value records.

The rvm_modify_bytes function is also provided for convenience when the entire range is to be modified. Both rvm_modify_bytes and rvm_set_range can be used in the same transaction as needed.

The begin transaction modes are summarized below:

restore

restore virtual memory on abort.

no_restore

do not restore virtual memory on abort.

2.2.2 Transaction Abort

Either the application or RVM can abort a transaction. RVM will automatically abort only if there has been an internal problem, such as an I/O error or resource exhaustion, that prevented logging the commit. This rare event will cause an appropriate exception code to be returned.

If a restore mode transaction is aborted, the old value records are used to restore the region affected to its exact state before the transaction began. Applications can abort a transaction at any time, for any reason, with the rvm_abort_transaction function.

2.2.3 Transaction Commit

Once the application has completed all modifications, it then *commits* to the changes by calling rvm_end_transaction. At this time, RVM makes permanent the changes by copying them to the log file.

RVM provides two commit modes with different performance and permanence guarantees. Guaranteeing permanence requires waiting for the log records to be flushed (written) to the log file. After this is done, only a disaster destroying the information in the log file can cause loss of the changes. The recovery mechanisms can restore the changes in a less destructive crash.

The faster mode, no_flush, does not copy changes to the log file before reporting success. There is no guarantee for change permanence since a system crash before the file is written will cause loss of the changes. In this case, a quick commit is preferred to the guarantee of permanence provided by flushing. This is sometimes called a "lazy" commit [?]. Permanence can be insured at a later time by any transaction doing a commit with flush, or by the application directly invoking the rvm_flush function.

The two types of commit are supported by the following options, either of which can be used to commit a transaction begun in any mode:

flush

guaranteed permanence of changes.

no_flush

quick commit, no guarantee of permanence.

2.2.4 Transaction Optimizations

RVM supports two levels of optimization in logging transactions. Intra-transaction optimization coalesces ranges declared within a single transaction. This eliminates redundant log records if the transaction declares overlapping ranges. Also, adjacent ranges are coalesced into a single range for logging.

When no_flush commit is used, a second level of optimization can be used. Ranges in the currently committed transaction can be coalesced with those of previous no_flush committed transactions that have not been flushed. This can eliminate redundant log records if the same offsets in a segment are modified by successive transactions.

These optimizations have together provided about a 50% improvement in log efficiency and are recommended in most cases. Sometimes the log is a useful debugging record, and since the optimizations do alter the exact transaction history by eliminating redundant information, their use may not be desired. The log management utility rvmutl can print the log to show the applications transaction history, and has been a useful debugging tool.

2.3 Concurrency in RVM

Applications have the option of using threads provided by the C Threads [?] programming package. The application is responsible for protecting access to its data structures by concurrent threads, but can otherwise use RVM functions freely, with RVM synchronizing access to its internal structures.

This discussion of concurrency in RVM assumes that Mach preemptive threads are in use (one Mach thread per C thread, -lthreads) library. However, RVM will also work correctly if the coroutine C Thread (-lco_threads) library is linked for debugging since the C Thread synchronization primitives are used exclusively. The choice of thread type is made simply by linking the desired thread library. Linking different versions of RVM for the thread types will not be necessary. The Mach task implementation of C Threads, -ltask_threads library, must not be used.

RVM does not set or modify any of the C Thread control variables such as mutex_spin_limit [1]. This level of tuning is left to the application, which is also responsible for setting thread stack limits via the shell limit command.

Unix applications can be built without thread support. This is the default build procedure in the RVM distribution kit. By defining the Cthreads primitives in terms of other thread package primitives, it is possible to use other thread support. However, no formal interface for this exists at this time.

Inherent file delays and RVMs internal locking structure divide RVM functions into two groups, distinguished by execution speed and probability of contention. Functions that do not access files are inherently fast since they access only virtual memory structures whose locking structure is specifically intended to allow the maximum useful concurrency while minimizing synchronization overhead.

The fast functions are:

```
rvm_begin_transaction  rvm_abort_transaction
rvm_set_range          rvm_modify_bytes
rvm_set_options        rvm_query
rvm_unmap              rvm_terminate
```

Slow functions are those requiring synchronous file access:

```
rvm_map                rvm_end_transaction
rvm_flush              rvm_truncate
```

These functions will be sequentialized on their access to files. First-come, first-served order prevails. Functions operating on separate files will not contend at RVM level, but may at kernel level if those files are supported by disk partitions on the same physical disk or controller. The order of access will be determined by the kernel.

Several of these functions can be in the "fast" category under certain conditions:

- rvm_end_transaction will be quick in no_flush mode.
- rvm_truncate and rvm_flush will be fast if the log is empty.
- rvm_map, when operating on a segment backed by an external pager, will be much faster than when performing copy-on-map.

2.3.1 Concurrency During Flush and Truncation

Because no_flush transactions do not require immediate access to the log file, RVM permits them to commit during flushing or truncation. Transactions that require exclusive access to the log (flush) mode must be serialized, with no guarantee the commit order of simultaneously committing transactions. This is controlled by the C Threads package and should be considered random.

[1] Setting any of the spin limits to other than zero on a uniprocessor is pointless!

Transaction commits in flush mode are allowed during log truncation. RVM internally divides the log into two sections: the area being truncated, and the remainder which becomes the active area of the log. This allows transactions to commit while truncation proceeds, although a small delay might occur due to contention for the log file. Truncation is considered lower priority than commit and will yield control of the log after each access. A committing transaction will flush all waiting no_flush transactions plus itself before permitting truncation to proceed.

Applications can control the timing of truncation by either setting a threshold for initiation, or by doing it themselves. If invoked by threshold, RVM uses an internal thread to execute the truncation, and transactions continue. If the application directly invokes rvm_truncate, truncation is synchronous with the calling thread. Regardless of method, truncation should be initiated soon enough that the remainder of the log is not filled before truncation completes. Otherwise, transactions will be blocked until log space is available. Details of truncation control can be found in the Options section (section 3.2.1 ()).

Log truncation and rvm_map have special concurrency considerations because rvm_map must insure that the data mapped represents the most recent committed image of the segment, a state that is controlled by log truncation. On the first mapping of a segment, a truncation will be needed if the log is not empty. In subsequent mappings, truncation will be needed only if a modified region is unmapped and then remapped with no intervening truncation. In these cases, rvm_map will initiate truncation and must wait for its completion.

RVM attempts to minimize mapping delays due to truncation by keeping careful track of what has been unmapped so that no unnecessary truncations are initiated. The truncation mechanism also posts its progress so that mapping is delayed only until the needed region of the data file has had all relevant log records applied to it.

2.4 Good Programming Practice

Good programming practice in building applications with RVM is quite simple. There are only two areas where some caution must be observed: declaring modification ranges to RVM, and using transactions in critical sections.

2.4.1 Declaring Modification Ranges

Since RVM has no way to detect modifications to mapped regions unless they have been declared by a call to rvm_set_range, programmers must be careful to insure that all modifications are covered by such a declaration. Failure to do so results in errors that are not detectable until the modified region is unmapped, either explicitly, or by termination of execution. When the modified range is again mapped, the undeclared modifications will be missing since they were not included in the new values captured by transaction commit. This has been the most common programming error noted in use of RVM, so if some modifications thought to have been committed are missing after a restart, check the rvm_set_range calls carefully to be sure that all modifications are covered.

The declaration of a modification range should always be made *before* the modifications are actually assigned. This is mandatory when transactions are begun in restore mode since the old values are preserved by rvm_set_range for restoration if rvm_abort_transaction is called. If the assignments are made before calling rvm_set_range, the opportunity to capture the old values is lost and aborting the transaction will result in restoring the modifications rather than the original data. In future versions, making modifications before calling rvm_set_range may result in incorrect operation.

Declarations of modification ranges are most efficiently made by minimizing the number of calls to rvm_set_range or rvm_modify_bytes. Since the overhead of of a modification range record in the log is significant, declaring the range to include the entire modified object is often more efficient than declaring several small changes within the object. This is true for objects up to 100 bytes or so. One must be careful, though, not to include any part of an object that may be modified by a concurrent transaction since this could cause uncommitted state to be made permanent.

A second efficiency consideration is to declare a single modification range to cover an entire array or other structure that is modified sequentially, such as in an initialization. Declaring each element of a contiguous structure individually can cause extra log overhead if transaction optimizations are not enabled.

2.4.2 Transactions in Critical Sections

When using transaction in critical sections, any locks protecting modified objects must be held until after the transaction is either committed or aborted. This is true for locks that are held when a transaction is started, and for those acquired during the transaction.

RVM does not capture the new values for modification ranges until commit time, so if a lock is released before a transaction is committed, there is the possibility that a concurrent access could make additional changes that are not part of the transaction. Similarly, if a lock is released before an abort, the abort could restore values that are not meaningful to a concurrent transaction.

Chapter 3

RVM Library Specifications

RVM is a library of functions to be linked with the application and is executed within the address space and control thread of the calling process. For single process applications, this design eliminates RPC calls, although extensions to RVM for distributed transactions will require them.

The functions implementing the core of RVM are included in one user level library. Two versions of the library are necessary: one for Unix, and one for Mach. The Unix version will run under Mach, but no Mach-specific features, such as external pager support or C Threads, will be available. The interface to C is specified in one header file that is used with both versions.

Auxiliary libraries of useful functions are also available. These can extend the functionality of RVM, or provide convenience features useful to a broad range of applications. Two of these, a utility for loading regions of segments, and an allocator for segments, are documented in Chapter 5 (of RVM Segment Loader) and Chapter 6 (of RDS, A Dynamic Heap Allocator).

All RVM functions return success/failure codes of type rvm_return_t[1] . The code, RVM_SUCCESS [2], is returned by all functions to report successful execution. If an exception is encountered in executing a library function, an appropriate code is returned. These codes are specified with the functions. If the RVM_EIO code is returned, the specific error causing it will be left in errno. RVM functions check their parameters at entry, so unless otherwise specified in the function descriptions, a function returning an error code will have had no effect.

Library functions can also return information via output parameters. When this method is used, descriptions of the information returned will be specified with the function.

3.1 RVM Files and Structures

A single header file, rvm.h (included in Appendix B (of C Declaration of RVM)), provides the basic declarations for using RVM. rvm.h recognizes if it has been previously included in a compilation and will not create multiple definitions. In the Carnegie-Mellon programming environment, the enumeration typedef bool, with constants true and false, and the typeless constants TRUE and FALSE are often defined with the include file stdlib.h (or libc.h) at CMU. For convenience, rvm.h also supplies these, but will not attempt to redefine them if stdlib.h has been included.

Other header files are used for optional features of RVM. These include headers for statistics collection and printing, an allocator, and a segment loading utility.

[1] All RVM-specific types are declared as C typedefs and have a "_t" suffix.

[2] Constants and macros defined with #define are in upper case; types, structures, variables, and functions are declared in lower case

In building the application, the choice of Unix or Mach libraries must be made. If the application is using threads, Mach must be used, and the application must also link the C Threads library and the safe version of the C library (see [?]), p. 13). The Unix library will run on Mach/Unix, but will be strictly synchronous so threads must not be used in the application.

The necessary files are:

header files:

`~/include/rvm.h, ~/include/rvm_statistics.h`

library:

`~/lib/librvm.a`

log utility:

`~/bin/rvmutl`

Each structure used in RVM has a self-identifier as its first field (struct_id). These are used by the library functions to validate the record type and are also valuable in debugging since the type, regardless of casts (intentional or otherwise), will be available. The type code will automatically be set by type-specific allocation and initialization routines provided by RVM. The struct_id field must not be modified.

The print names for all RVM codes and typedefs can be obtained with functions similar to strerror. These functions return a pointer to a character array containing the name. These arrays are statically allocated and the pointers must not be deallocated with free; no deallocation is necessary. The functions are:

```
char *rvm_return (code)/* function return codes */
rvm_return_tcode;

char *rvm_mode (mode)/* transaction modes */
rvm_mode_tmode;

char *rvm_type (struct_id)/* typedef/structure  names */
rvm_struct_id_tid;
```

The structures used in RVM have allocation, initialization, copy, and deallocation functions defined for them. The allocation and deallocation functions maintain a cache of heap allocated records within RVM. All type-specific initialization and finalization is automatically done, as needed, by these routines. Initialization for non-heap allocated records is also provided. At present there are three types allocated: rvm_options_t, rvm_region_t, and rvm_tid_t. These represent the options descriptor, discussed below, the mapped regions descriptor, and the transaction identifier.

These functions take their names from the type they support. For example, the allocation function for rvm_options_t is rvm_malloc_options, which requires no parameters and returns a pointer to a new options record. Similarly, the deallocation function for a transaction identifier (rvm_tid_t) is `rvm_free_tid (tid)`, which takes a single parameter, tid, which is a pointer to a transaction identifier record. Strings used in RVM records are always copies of pointers to the previously existing character arrays, so no deallocation is necessary.

The initialization and copy functions are named rvm_init_ and rvm_copy_ respectively, and take a single parameter: the pointer to the record affected. The copy functions return a pointer to the created record. The rvm_init_ functions must be applied to statically, globally, or automatically allocated records before RVM will recognize them. The rvm_free_ functions will ignore such non-heap allocated records. A field, from_heap, is present in all RVM structures so that the space freeing functions can detect non-heap allocated records. It is set by the allocation functions, and must not be altered. The function specifications are:

```
/********/
/* utility routines for rvm_options_t (option descriptor, or opt. desptr.) */
/********/
rvm_options_t *opt;                   /* pointer to option descriptor      */

rvm_options_t *rvm_malloc_options();  /* allocate, initialize opt. desptr. */
                                      /* return pointer                    */

rvm_free_options(opt);                /* free the opt. desptr. pointed to  */
                                      /* by opt                            */

rvm_init_options(opt);                /* initialize non-heap allocated     */
                                      /* opt. desptr.                      */

rvm_options_t *rvm_copy_options(opt); /* copy opt. desptr. pointed to by   */
                                      /* opt to a newly allocated opt.     */
                                      /* desptr., return pointer           */
/********/
/* utility routines for rvm_region_t (region descriptor, or reg. desptr.)  */
/********/
rvm_region_t *reg;                    /* pointer to region descriptor      */

rvm_region_t *rvm_malloc_region();    /* allocate, initialize reg. desptr. */
                                      /* return pointer                    */

rvm_free_region(reg);                 /* free the reg. desptr. pointed to  */
                                      /* by reg                            */

rvm_init_region(reg);                 /* initialize non-heap allocated     */
                                      /* reg. desptr.                      */

rvm_region_t *rvm_copy_region(reg);   /* copy reg. desptr. pointed to by   */
                                      /* reg to a newly allocated reg.     */
                                      /* desptr., return pointer           */
/********/
/* utility routines for rvm_tid_t (transaction identifer, or tid)          */
/********/
rvm_tid_t *tid;                       /* pointer to transaction identifier */

rvm_tid_t *rvm_malloc_tid();          /* allocate, initialize tid          */
                                      /* return pointer                    */

rvm_free_tid(tid);                    /* free the tid pointed to by tid    */

rvm_init_tid(tid);                    /* initialize non-heap allocated tid */

rvm_tid_t *rvm_copy_tid(tid);         /* copy tid pointed to by opt        */
                                      /* to a newly allocated tid          */
                                      /* return pointer                    */
```

Implementing RVM as a library limits freedom to change the program interface because change would require code changes in all applications. To alleviate this, RVM uses *option records* to specify parameters for certain features. New features will be implemented by adding new fields to the records so program changes will be limited to those applications needing the new features. Others will need only to recompile and link, an automated process for most systems.

The option records can specify global settings, particularly for the log, and also options for individual recoverable storage segments. Some features may require setting at initialization time, and can not be changed later. Details of each option are presented as the features are discussed.

3.2 Initialization, Options and Mapping

RVM requires initialization by the client application. In addition to initializing its internal structures, RVM will check for version skews between the version of rvm.h used to build the application and the version of librvm.a linked.

At initialization, the name of the log file is specified and other options can also be specified. Most options (except the log file) can be changed after RVM has been initialized, and there is provision to query the state of all options.

To insure an orderly shutdown of RVM, a termination function is also provided. It will detect and return an error code if any transactions are found to be uncommitted at the time of shutdown. As part of orderly termination, the application designer should consider forcing a log truncation. This will avoid the inconvenience of truncation delay during the first mapping the next time the application is started.

3.2.1 RVM Options Descriptor

When a region is mapped, certain options can be specified in an rvm_options_t record created with rvm_malloc_options, which takes no parameters. Option records can also be used with rvm_query to discover the state of RVM options and resource usage. At present the following mapping option fields are defined:

log_dev

Name of the log file. Required on initialization and cannot be changed thereafter. Returned on query.

truncate

Log truncation threshold: percentage of the log file capacity. When this much of the log is full, truncation will be initiated. Zero inhibits the truncation thread. Actual percentage of log used at time of returned; default: 50%, specified by the constant TRUNCATE.

recovery_buf_len

Length of truncation buffer in bytes.

flush_buf_len

Length of flush buffer in bytes.

max_read_len

Maximum disk transfer length (Mach only).

log_empty

State returned on query (bool).

pager

Name of external pager file (char array).

n_uncommit

Number of uncommitted transactions returned on query.

tid_array

Pointer to an array of transaction identifiers of length n_uncommit returned on query.

flags

Bit vector of option flags.

Unless otherwise specified in rvm.h, all fields are unsigned long integers.

The log_dev field is a pointer to to a character array that contains the Unix file name of the log. This must be specified on initialization and is returned on query. The log file cannot be changed after initialization.

The truncate option, specified as a percentage of log file capacity, indicates the threshold for automatic log truncation. The default is 50%, and the actual threshold is returned on query.

If the value of truncate is non-zero, RVM creates an internal thread to do truncations asynchronously; a zero value inhibits automatic truncations if a thread was already created. When the automatic truncations are inhibited, truncations are initiated by the application with rvm_truncate, or by remapping a previously mapped and modified region.

If automatic truncation is inhibited and the log file overflows, a truncation is performed by the thread encountering the overflow. This is an emergency measure and applications should not allow it to happen since all transactions will be blocked until the truncation completes. However, when running without thread support, this is the only method of initiating truncation.

The recovery_buf_len field specifies the length, in bytes, of the buffer used by truncation. It is set to a default value of 256 Kbytes, which is generally sufficient.

The flush_buf_len field specifies the length, in bytes, of the buffer used to flush log records on raw partitions. It is not used for file-based logs. The default length is 256 Kbytes.

The max_read_len field specifies the length, in bytes, of the maximum read transfer that RVM will perform. This is a Mach-only feature, and is used to limit the amount of kernel buffering required when large segments are mapped. This field is ignored on Unix systems. The default of 512 Kbytes seems to work well.

log_empty will be set by rvm_query if the log is empty. This is to help applications determine when it is safe to transfer access control to a segment between processes.

The pager option is used to specify a Mach external pager to be used with mappings of the segment. This is not implemented at this time.

The fields n_uncommit and tid_array are returned by rvm_query to allow an application to discover uncommitted transactions. This is necessary if either rvm_unmap or rvm_terminate return an error code because such transactions are outstanding. n_uncommit is set to the number of uncommitted transactions outstanding in a region, or in the entire segment, if no region is specified. If n_uncommit is not zero, tid_array will point to an array of length n_uncommit of transaction identifiers.

The flags field is a bit vector used to specify other options to RVM. At present, the transaction optimization options RVM_COALESCE_RANGES and RVM_COALESCE_TRANS can be specified. Both are specified with RVM_ALL_OPTIMIZATIONS. The options in use are returned by rvm_query.

3.2.2 RVM Region Descriptor and Segment Mapping

Regions of recoverable storage are mapped into virtual memory for manipulation by a process. Mapped regions are described by the typedef rvm_region_t. This structure (actually pointers to its instances) is used as a handle to specify regions in mapping and transaction function calls. The fields of the structure are:

data_dev

Name of the segment file.

dev_length

64 bit segment maximum length.

offset

64 bit region offset in segment.

vmaddr

Virtual memory base address of mapped region.

length

Length, in bytes, of region.

no_copy

Do not copy segment data for mapped region if true.

The file name and virtual memory address are pointers to character arrays.

The dev_length field is of type rvm_offset_t, a structure specifying the 64 bit disk offset. It specifies the maximum length of a segment that may be mapped, and permits enforcement of a resource limit on a segment. Optional for segments represented by files, it is required for disk partitions since there is no easy way to discover the partition size at run-time.

The offset field is of type rvm_offset_t, and is the offset in the segment of the first byte of the region.

The length field is of type rvm_length_t. This numeric type is guaranteed to always be of sufficient size to represent the maximum region length meaningful in the host machine address space. Functions and macros to manipulate length and offset types are provided and are described in the next section.

The no_copy field offers a performance enhancement when the data would be entirely replaced by the application. When the option is chosen by setting the field to true, the segment region is associated with the specified virtual memory region, but no data from the segment is copied into memory. This option is available with or without the external pager.

Programs prepare for RVM function calls by creating an rvm_region_t record with the allocator rvm_malloc_region. Statically or globally allocated records must be initialized with rvm_init_region, which requires a pointer to the record. All field values are entered directly into the records.

When the region records are used, the file name is always required, but other fields may sometimes not be used. Certain fields are also used to return information. The descriptions of the functions will specify what is required and what is returned. After a region is mapped, the descriptor is used for all functions referencing the region.

3.3 Statistics Collection

RVM collects various statistics about its performance and the characteristics of the applications transaction behavior. All statistics are collected for both current and cumulative periods. The current period begins at RVM initialization time and continues until the next truncation. At truncation time, the current values are summed with the cumulative values. The current fields are then reset. Reseting the cumulative values can be done only by reinitializing the log, so the cumulative history is maintained from the time the log is initialized.

To read the statistics, a `rvm_statistics_t` record is allocated with `rvm_malloc_statistics`, and passed to the `rvm_statistics` function. The current and cumulative values are copied into the record and it is returned.

As RVM is developed, the statistics are expected to change. Consequently, the definition of the `rvm_statistics_t` record that is used to present the values to the application is not included in the standard RVM library header. A secondary header file, `rvm_statistics.h`, is used so that the statistics version can change independently of the primary program interface. If the application program makes minimal direct use of the fields in the `rvm_statistics_t` record, the anticipated changes should cause little reprogramming inconvenience.

To further insulate the application program from changes in the statistics, a printing function `rvm_print_statistics`, is also included. If statistics are simply captured and printed, the details of the statistics record are completely hidden. The statistics can also be printed with the `rvmutl` log utility.

For detailed specification of the statistics currently collected, consult the header file `rvm_statistics.h`, which printed in Appendix B (of C Declaration of RVM).

3.4 Address, Length, and Offset Arithmetic

The `rvm_length_t` and `rvm_offset_t` types are intended to shield applications from machine-dependent representations of large values. Using basic C types such as unsigned long for region lengths may not work on machines with virtual address spaces greater than 2^{32}. Type long may continue to be defined as a 32-bit quantity in some C compilers so rvm_length_t will always be defined as an integer type wide enough to hold the largest virtual memory region possible on the host machine.

Similarly, disk memories greater than 2^{32} already exist. The rvm_offset_t will be guaranteed to provide 64-bit integer representation for segment offsets. On most present machines, this is implemented as two 32-bit fields, but on machines with 64-bit integer formats, rvm_offset_t may be defined as a single field. Applications are strongly discouraged from directly using the internal representations of either rvm_length_t or rvm_offset_t. Such use may create portability problems.

The following functions and macros are provided for manipulating rvm_length_ts and virtual memory addresses:

```
RVM_ADD_LENGTH_TO_ADDR (vmaddr,length) /* add length to address, */
  char          *vmaddr;  /* returning result as value of macro */
  rvm_length_t  length;   /* with type char *  */
RVM_SUB_LENGTH_FROM_ADDR (vmaddr,length) /* subtract length from address, */
  char          *vmaddr;  /* returning result as value of macro */
  rvm_length_t  length;   /* with type char *   */
```

The following macros are provided for manipulating, comparing, and converting offsets.

```
RVM_ADD_OFFSETS (x,y)  /* add y to x, */
  rvm_offset_t  x;     /* return offset result */
  rvm_offset_t  y;

RVM_SUB_OFFSETS (x,y)  /* subtract y from x, */
  rvm_offset_t  x;     /* return offset result */
  rvm_offset_t  y;

RVM_ADD_LENGTH_TO_OFFSET (x,y)  /* add length y to offset x, */
  rvm_offset_t  x;     /* return offset result */
  rvm_length_t  y;

RVM_SUB_LENGTH_FROM_OFFSET (x,y)  /* subtract length y from offset x, */
  rvm_offset_t  x;     /* return offset result */
  rvm_length_t  y;

RVM_MK_OFFSET (x,y)    /* make an offset from two lengths  x, y, */
                       /* return offset result */
  rvm_length_t  x;     /* high order bits */
  rvm_length_t  y;     /* low order bits */

RVM_OFFSET_INITIALIZER (x,y)  /* compile-time initializer for offsets */
  rvm_length_t  x;     /* high order bits */
  rvm_length_t  y;     /* low order bits */

RVM_ZERO_OFFSET (x)    /* zero all 64 bits of x */
  rvm_offset_t  x;

RVM_OFFSET_TO_LENGTH (x)  /* return low-order bits of offset x as length */
  rvm_offset_t  x;

RVM_OFFSET_HIGH_BITS_TO_LENGTH (x)  /* return the high-order bits offset x as length */
  rvm_offset_t  x;

RVM_LENGTH_TO_OFFSET (x)  /* construct offset from length x */
  rvm_length_t  x;
```

The comparison macros return 1 if the comparison is true; 0 otherwise:

```
RVM_OFFSET_~(x,y)   /* comparators, return 1 if comparison is true */
  rvm_offset_t  x;  /* operators (~): LSS, LEQ, GTR, GEQ, EQL */
  rvm_offset_t  y;

RVM_OFFSET_EQL_ZERO (x)  /* return 1 if offset x is zero */
  rvm_offset_t  x;
```

For convenience in mapping, the following macros provide integral, page-sized conversions:

```
RVM_PAGE_SIZE       /* return page size as rvm_length_t */

RVM_ROUND_LENGTH_UP_TO_PAGE_SIZE (x)  /* return length x rounded up */
```

```
  rvm_length_t  x;  /* to next integral page-sized length */

RVM_ROUND_LENGTH_DOWN_TO_PAGE_SIZE (x)  /* return length x truncated */
  rvm_length_t  x;  /* to integral page-sized length */

RVM_ROUND_ADDR_UP_TO_PAGE_SIZE (x)  /* return address x rounded up */
  char  *x;         /* to next integral page size */

RVM_ROUND_ADDR_DOWN_TO_PAGE_SIZE (x)  /* return address x truncated */
  char  *x;         /* to integral page size */

RVM_ROUND_OFFSET_UP_TO_PAGE_SIZE (x)  /* return offset x rounded up */
  rvm_offset_t  x;  /* to next integral page-sized offset */

RVM_ROUND_OFFSET_DOWN_TO_PAGE_SIZE (x)  /* return offset x truncated */
  rvm_offset_t  x;  /* to integral page-sized offset */
```

Chapter 4

Library Functions

In this and the next two chapters, funtions of the RVM runtime library will be briefly presented in groups, and then the actual man pages of the functions will be presented.

4.1 Initialization, Options, and Mapping Functions

There are five functions for this groups:

rvm_initialize
: Initialization of RVM runtime library

rvm_set_options
: Changes RVM options after initialization

rvm_terminate
: Clean termination of RVM runtime library

rvm_map
: Mapping all or part of data segment to virtual memory

rvm_unmap
: Ummapping previouslly mapped segment

NAME

rvm_initialize - RVM initialization

SYNOPSIS

```
#include "rvm.h"

typedef struct
    {
    char           *log_dev;
    long           truncate;
    rvm_length_t   recovery_buf_len;
    rvm_length_t   flush_buf_len;
    rvm_length_t   max_read_len;
    rvm_length_t   flags;
    }
rvm_options_t;

rvm_return_t rvm_initialize (version, options)
rvm_return_t RVM_INIT (options)

char          *version;   /* RVM library version */
rvm_options_t *options;   /* pointer to a RVM options record */
```

DESCRIPTION

rvm_initialize must be called once to initialize the RVM library, and must be called before any other RVM function. Without successful initialization, any call to RVM functions except rvm_initialize will result in the error return RVM_EINIT. Multiple calls to rvm_initialize are ignored. However, rvm_initialize cannot be used to restart RVM after rvm_terminate has been called.

The version parameter is used to detect version skews. The string constant RVM_VERSION from rvm.h is passed to rvm_initialize, and compared with the value from the header file the library was compiled with. If they are different, RVM_EVERSION_SKEW is returned and RVM will not initialize.

RVM operation option choices are specified with the options parameter. A options record created with typedef rvm_options_t must be used. RVM_EOPTIONS will be returned if options does not point to a valid record.

The name of the log file must be specified in the options record. If either the name is not provided, or the file cannot be opened and initialized, RVM_ELOG will be returned. If options other than the log file are not specified, defaults will be assumed. These can be changed later with rvm_set_options.

rvm.h defines a macro, RVM_INIT, with a single parameter for the options, that calls rvm_initialize and automatically passes RVM_VERSION. This is the best way to initialize RVM.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_EVERSION_SKEW

RVM library version skew

RVM_ELOG

invalid log file

RVM_EIO

I/O or kernel error, errno has code number

RVM_EOPTIONS

invalid RVM options record or pointer

RVM_ENO_MEMORY

heap exhausted

RVM_EINIT

RVM previously terminated

RVM_EPORTABILITY

internal portability problem, see system maintainer

SEE ALSO

`rvm_terminate (3)`, `rvm_set_options (3)`, `rvm_query (3)`

AUTHOR

Hank Mashburn

NAME

rvm_set_options - set RVM options

SYNOPSIS

```
#include "rvm.h"

typedef struct
    {
    char            *log_dev;
    long            truncate;
    rvm_length_t    recovery_buf_len;
    rvm_length_t    flush_buf_len;
    rvm_length_t    max_read_len;
    rvm_length_t    flags;
    }
rvm_options_t;

rvm_return_t rvm_set_options (options)

rvm_options_t    *options;    /* pointer to a options records */
```

DESCRIPTION

rvm_set_options allows RVM options to be changed after initialization. Only the options permitted to be selected after initialization will be considered. Others can be present, but will be ignored. Please refer to the RVM manual for detailed descriptions of the options.

If the options parameter does not point to a valid descriptor, RVM_EOPTIONS will be returned.

At present, the only options that can be changed are the following: truncation threshold and transaction optimization levels. If the truncation threshold is lowered and the log filled beyond the new threshold, a truncation is initiated if one is not in progress. If a truncation must be initiated, it will run synchronously with the calling thread.

Transaction optimization levels can be changed at any time, but the level in effect at the beginning of a transaction remains in effect for that transaction until it is ended.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_EOPTIONS

invalid options record or pointer

RVM_ENO_MEMORY

heap exhausted

RVM_ELOG

invalid log file

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_map (3)`, `rvm_unmap (3)`, and `rvm_query (3)`

AUTHOR

Hank Mashburn

NAME

rvm_terminate - terminate RVM

SYNOPSIS

```
#include "rvm.h"

rvm_return_t rvm_terminate ()
```

DESCRIPTION

rvm_terminate checks for uncommitted transactions and otherwise insures a clean exit from RVM. If an uncommitted transaction is found, RVM_EUNCOMMIT is returned, and the application should call rvm_query to discover such transactions. If any are discovered, this is an indicator of likely program logic errors. If there are no uncommitted transactions, the log is flushed, and all regions unmapped.

RVM cannot be reinitialized after termination.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_EUNCOMMIT

uncommitted transaction(s) pending

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_initialize (3)`, and `rvm_query (3)`

AUTHOR

Hank Mashburn

NAME

rvm_map - map segment region

SYNOPSIS

```
#include "rvm.h"

typedef struct
    {
    char            *data_dev;
    rvm_offset_t    dev_length;
    rvm_offset_t    offset;
    char            *vmaddr;
    rvm_length_t    length;
    bool            no_copy;
    }
rvm_region_t;

typedef struct
    {
    char            *log_dev;
    long            truncate;
    rvm_length_t    recovery_buf_len;
    rvm_length_t    flush_buf_len;
    rvm_length_t    max_read_len;
    rvm_length_t    flags;
    }
rvm_options_t;

rvm_return_t rvm_map (region,options)

rvm_region_t    *region;  /* pointer to a region descriptor [in/out] */
rvm_options_t   *options; /* optional pointer to an options record */
```

DESCRIPTION

rvm_map moves a copy of all or part of the data in a data file to virtual memory. Mapping is necessary before any transactions can be made on a segment. Data can be mapped in any integer multiple of page size that Unix allows. The only restriction is available space in virtual memory.

To insure that the committed image of the data file is brought into memory, rvm_map generally performs log truncation before mapping. One advantage of this is that crash recovery is automatic with the first mapping after the crash. The application need do nothing to restore the state of recoverable storage.

rvm_map requires the name of the segment file to be specified in the descriptor provided by the region parameter. If the region parameter does not point to a legitimate descriptor, or the segment file cannot be opened, RVM_EREGION is returned.

The range to map is specified by the offset and length fields of the region descriptor. The offset is rounded down to a page boundary, and the unrounded offset plus length is rounded up to a page boundary to determine the exact region mapped. The rounded offset plus length must lie at least partially within the segment or RVM_EOFFSET is returned.

For file segments, if length is zero, the number of bytes remaining in the segment from the offset is assumed. Setting the offset and length to zero maps the entire segment. For raw partition segments, the length must be specified. If the range cannot be mapped due to size, RVM_ERANGE will be returned. The rounded offset and the length of the region actually mapped are returned in the region descriptor, and should not be changed since the region descriptor will be the applications record of the mapping and will be required to unmap the region.

The mapping buffer can be optionally specified with a virtual memory address in the vmaddr field of the region descriptor, or set to zero. If zero, RVM will allocate page aligned space for a buffer of the requested length. This space is not heap allocated and can not be returned via free; the program can reuse it after it is unmapped. When allocation is requested, no other threads should also doing allocation, including using malloc. In Unix, both RVM and malloc use the kernel call sbrk and cannot be synchronized without RVM-specific modifications to the C library. Interference between them can cause unreferenced virtual memory, but the results are always correct. To avoid such interference, try to do mapping allocations during application initialization, or other times when concurrency is minimal. Mach versions of RVM use vm_allocate and do not suffer from conflicts with malloc.

When a buffer is allocated, its address is returned in vmaddr. If the buffer cannot be allocated due to space restrictions, RVM_ENO_MEMORY will be returned.

When vmaddr is non-zero, it must refer to a page aligned buffer; the address specified is not rounded. If vmaddr specifies an invalid address, RVM_ERANGE will be returned.

Mapping buffers must not overlap in virtual memory. Neither can the same region of a segment be mapped twice by the same process. In these cases, RVM_EVM_OVERLAP or RVM_EOVERLAP will be returned. If an application splits objects across page boundaries, it must make sure the pages are mapped so that such objects can be accessed.

If any error is encountered, no buffer is allocated, no data mapped, and no modifications are made to the region descriptor.

An optional rvm_options record can be specified with the options parameter. If options is not zero, and points to a valid rvm_options record, the options specified will be used in the mapping. Please refer to the RVM manual (section RVM option descriptor) for details for the available options. Options that must be specified on the first mapping only will be ignored on mappings of other regions of the same segment. If options is not zero and does not point to a valid record, RVM_EOPTIONS will be returned.

If this is the first mapping of a segment and a Mach version of RVM is running then the pager field of the options record is checked. If it is not the null string, rvm_map will attempt to load the file specified into a forked task as an external pager and initialize it. If either the load fails or the task cannot be properly initialized as an external pager, RVM_EPAGER is returned. Any task created will be destroyed if an error is reported. If successful, rvm_map returns and pages will be copied on demand by the external pager. The external pager backs all mapped regions of the segment, and will be in effect until the last region is unmapped. Using an external pager can substantially improve performance, particularly if references to the data in the region are sparse.

If RVM is running on Unix, or the pager field is a null pointer, the mapped region is copied into virtual memory in total. If on Mach, the inheritance property of pages will be set to VM_INHERIT_NONE and it recommended that this not be changed.

If the no_copy field is set in the region record, the data for the mapped region will not be copied to virtual memory. This is true with or without the external pager. This option can be used when it is known that the data will be completely replaced.

The virtual memory protection of mapped pages is initially read/write in both Mach and Unix. However, the application can change this if desired by making the appropriate kernel calls.

A disk transfer error can result in corrupting a buffer, so applications must not assume anything about data in an existing buffer unless rvm_map returns successfully. If a transfer error occurs,RVM_EIO will be returned and length will be set to zero, and errno will contain the error number returned by the failing I/O call.

Mapping requires allocation of internal structures. If these cannot be allocated, RVM_ENO_MEMORY is returned and no data mapped.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_EREGION

invalid region descriptor or pointer

RVM_EOFFSET

invalid offset

RVM_ERANGE

invalid virtual memory range

RVM_EOVERLAP

region overlaps existing segment mapping

RVM_EVM_OVERLAP

region overlaps existing virtual memory mapping

RVM_EOPTIONS

invalid options record or pointer

RVM_EIO

transfer error or segment file cannot be opened, errno has code number

RVM_EPAGER

invalid external pager

RVM_ENO_MEMORY

heap exhausted, or buffer cannot be allocated

RVM_ENOT_MAPPED

file to be mapped has zero length

RVM_ELOG

no log file has been specified or invalid file name

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_unmap (3)`, `rvm_set_options (3)`, `rvm_query (3)`, `rvm_create_segment`, and `rvm_load_segment`

AUTHOR

Hank Mashburn

BUGS

There is no provision for extending a mapped region.

NAME

rvm_unmap - unmap segment region

SYNOPSIS

```
#include "rvm.h"

typedef struct
    {
    char            *data_dev;
    rvm_offset_t    dev_length;
    rvm_offset_t    offset;
    char            *vmaddr;
    rvm_length_t    length;
    bool            no_copy;
    }
rvm_region_t;

rvm_return_t rvm_unmap (region)

rvm_region_t *region;   /* pointer to region descriptor */
```

DESCRIPTION

rvm_unmap releases mapped buffers. The virtual memory space is freed for other uses, but not returned to Unix. This is true even if the space was allocated by rvm_map, in which case attempts to deallocate it with free will result in error. If the space was originally allocated from the heap by the application, using free is permissible.

If vmaddr is invalid, length is zero, or the range is not within valid virtual memory limits, RVM_ERANGE is returned. The address range must represent a mapped region of a segment, otherwise RVM_ENOT_MAPPED is returned. The region unmapped must be the same as that mapped, or RVM_ERANGE or RVM_ENOT_MAPPED, as appropriate, will be returned. A region cannot be split by unmapping part of it. Use of the region descriptor as returned by rvm_map for the region is recommended.

Before a region of a segment can be unmapped, there must be no uncommitted transactions affecting that region. RVM checks this and will return the code RVM_EUNCOMMIT if any exist. The function rvm_query can be used to discover the uncommitted transaction (s). Discovery of an uncommitted transaction in a region to be unmapped is a likely indicator of program logic errors.

Invoking log truncation after unmapping modified regions will avoid delay in mapping if those regions are expected to be remapped. If several regions are unmapped, truncation after the last unmapped will insure no mapping delay if any are remapped.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_EREGION

invalid region descriptor or pointer

RVM_ERANGE

virtual memory address range does not match mapped region

RVM_EUNCOMMIT

uncommitted transaction(s) pending

RVM_ENOT_MAPPED

designated region not mapped

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_map (3)`, `rvm_set_options (3)`, and `rvm_query (3)`

AUTHOR

Hank Mashburn

4.2 Transaction Functions

Transactions are used to change recoverable storage. A single transaction can modify any number of mapped regions. Addresses used in modifications to recoverable storage are virtual addresses and only virtual memory is modified directly. RVM will perform the translation into segment offsets based on the mapped region descriptors. The modifications become permanent only when a transaction commits.

To initiate a transaction, a transaction identifier record (rvm_tid_t) is created and presented to rvm_start_transaction which will give the transaction a unique identifier. A pointer to the record becomes a handle that must be used for all subsequent operations on the transaction. The record can be created with the rvm_malloc_tid function (see section 3.1 (of RVM structures)).

The transaction identifier record has two time value fields that mark the approximate time when the transaction was begun. Because the Unix kernel call gettimeofday underlying the timestamps is defined in microseconds, but many machine clocks do not have that fine resolution, the value can be no more precise than the machines clock. Also, RVM may increment the lowest order bits to generate a unique name without having to perform a kernel call. The value can still be presented to the time formatting routines available to extract a printable time. The fields, both integers, are contained in the `struct timeval` field `uname`:

tv_sec

Seconds since Jan. 1, 1970.

tv_usec

Microseconds since last second "tick."

RVM offers several options to control logging in transactions. The intent in all cases is to take advantage of special situations that may be known to the application and can be used to optimize the timing of, or eliminate, log operations. If a transaction cannot abort, using no_restore transactions will enhance performance.

The options supported are:

restore

Create old and new value log records for modifications; restore virtual memory after an abort.

no_restore

Do not create old value log records; virtual memory not restored after an abort.

flush

Flush log on commit.

no_flush

Do not flush log on commit.

Note that options can be specified at transaction start and commit. The restore, and no_restore options are valid only at the beginning of a transaction. The flush and no_flush options are valid only at transaction end (commit). At either transaction beginning or end, only one option can be specified.

NAME

rvm_begin_transaction - begin a transaction

SYNOPSIS

```
#include "rvm.h"

rvm_return_t rvm_begin_transaction (tid,mode)

rvm_tid_t    *tid;    /* pointer to transaction identifier */
rvm_mode_t   mode;    /* transaction begin mode */
```

DESCRIPTION

rvm_begin_transaction initiates a transaction. A unique transaction identifier is assigned and returned in the record specified by tid parameter. If the pointer does not reference a valid rvm_tid_t record, the RVM_ETID exception code is returned. If the transaction cannot be started because of internal memory limitations, RVM_ENO_MEMORY is returned.

rvm_begin_transaction recognizes two transaction modes: restore and no_restore. Transactions begun with restore create old value records so that the state of virtual memory can be restored if the transaction aborts. If mode is not a legal option, RVM_EMODE is returned.

The no_restore mode transactions do not create old value records. Therefore, an abort does not restore the state of virtual memory. They are offered as an additional performance enhancement for situations where there is no possibility that the application will issue an abort for the transaction.

Once a transaction is begun, modifications to mapped regions can be made with rvm_modify_bytes, or by the program in a range specified by rvm_set_range. The only restriction on modifications is that the modified regions must be mapped.

Transactions are ended with rvm_end_transaction, or rvm_abort_transaction, as appropriate.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_ETID

invalid transaction identifier pointer

RVM_EMODE

illegal transaction mode

RVM_ENO_MEMORY

heap exhausted

RVM_ELOG

no log file has been specified

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_set_range` (3), `rvm_modify_bytes` (3), `rvm_end_transaction` (3), and `rvm_abort_transaction` (3)

AUTHOR

Hank Mashburn

NAME

rvm_set_range - define a modification range

SYNOPSIS

```
#include "rvm.h"

rvm_return_t rvm_set_range (tid,dest,length)

rvm_tid_t      *tid;   /* pointer to transaction identifier */
char           *dest;  /* base address of modification range */
unsigned long  length; /* length of range in bytes */
```

DESCRIPTION

rvm_set_range defines a modification range within a transaction. The transaction must have been begun with rvm_begin_transaction, or the RVM_ETID exception code is returned.

The range is specified with the dest and length parameters, and must be in valid virtual memory or RVM_ERANGE is returned. The range must also be within a single mapped region or RVM_ENOT_MAPPED is returned. A range of zero length is ignored.

Range modifications provide efficient logging for transactions modifying memory in a specific range, and is ideal when modifications are not sparse. The original values within the range are preserved for possible restoration upon transaction abort. eliminating the need for a modify operator to log changes as they are made so normal assignment statements can be used. Changes can be made anywhere in the range and need not be sequential.

A common programming error is to make modifications without doing a rvm_set_range call to declare the modifications to RVM. This results in changes that are not permanent: since RVM has not been informed of the changes, it cannot log the new values and the changes are lost when the region is unmapped or if a crash occurs.

The declaration of a modification range should always be made *before* the modifications are actually assigned. This is absolutely required if the transaction was begun in restore mode, since the old values cannot be recorded otherwise and will not be available for restoration upon transaction abort. In future versions, making modifications before calling rvm_set_range may result in incorrect operation.

rvm_set_range can be used as many times as necessary within a transaction to define the ranges of modifications required. rvm_set_range and rvm_modify_bytes are not mutually exclusive, and can be used in the same transaction as is convenient to the programmer.

If the transaction was begun with the no_restore mode, no old value record will be created for the modification range, and an abort will not restore the state of virtual memory. If memory to create an old value record for the range cannot be allocated, RVM_ENO_MEMORY is returned.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_ETID

invalid transaction identifier or pointer

RVM_ERANGE

invalide virtual memory range

RVM_ENOT_MAPPED

virtual memory range not mapped

RVM_ENO_MEMORY

heap exhausted

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_begin_transaction (3)`, `rvm_modify_bytes (3)`, `rvm_end_transaction (3)`, and `rvm_abort_transaction (3)`

AUTHOR

Hank Mashburn

BUGS

A range cannot extend across a region boundary even if virtual memory is contiguously mapped.

NAME

rvm_modify_bytes - recoverable modifications to a segment

SYNOPSIS

```
#include "rvm.h"

rvm_return_t rvm_modify_bytes (tid,dest,src,length)

rvm_tid_t     *tid;    /* pointer to transaction identifier */
char          *dest;   /* base address of modification range */
char          *src;    /* base address of source range */
unsigned long length;  /* number of bytes to modify */
```

DESCRIPTION

rvm_modify_bytes is used to modify recoverable memory in a transaction. The transaction must have been previously initiated with rvm_begin_transaction; otherwise RVM_ETID is returned.

The modification range is specified with the dest and length parameters, and must be in valid virtual memory or RVM_ERANGE is returned. The source of new values is specified with src, and must be in valid virtual memory or RVM_ESRC is returned. The range must also be within a single mapped region specified by the region parameter, or RVM_ENOT_MAPPED is returned. A range of zero length is ignored.

Modification of different mapped regions in successive rvm_modify_bytes calls is permitted. Modifications can also be made with rvm_set_range, and both can be used in the same transaction as is convenient to the programmer. The modifications in virtual memory are visible to the program when the call returns.

If the transaction was begun with the no_restore mode, no old value records will be created for the modification range, and an abort will not restore the state of virtual memory. If memory to create an old value record for the range cannot be allocated, RVM_ENO_MEMORY is returned.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_ETID

invalid transaction identifier or pointer

RVM_ENOT_MAPPED

virtual memory range not mapped

RVM_ESRC

invalid addrss range for new values

RVM_ERANGE

invalide virtual memory range

RVM_ENO_MEMORY

heap exhausted

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_begin_transaction (3)`, `rvm_set_range (3)`, `rvm_abort_transaction (3)`, and `rvm_end_transaction (3)`

AUTHOR

Hank Mashburn

NAME

rvm_abort_transaction - abort specified transaction

SYNOPSIS

```
#include "rvm.h"

rvm_return_t rvm_abort_transaction (tid)

rvm_tid_t    *tid;   /* pointer to transaction identifier */
```

DESCRIPTION

rvm_abort_transaction aborts the specified transaction. Transactions begun with any mode can be aborted with this function. All modifications to segments made in the course of the transaction are restored to their original values (unless the transaction was begun with the no_restore) mode.

The transaction identifier must refer to a previously initiated transaction or RVM_ETID is returned.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_ETID

invalid transaction identifier or pointer

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_begin_transaction (3)`, `rvm_end_transaction (3)`

AUTHOR

Hank Mashburn

NAME

rvm_end_transaction - commit transaction

SYNOPSIS

```
#include "rvm.h"

rvm_return_t rvm_end_transaction (tid,mode)

rvm_tid_t         *tid;  /* pointer to transaction identifier */
rvm_trans_mode_t  mode;  /* transaction commit mode */
```

DESCRIPTION

rvm_end_transaction commits the changes made since the transaction began. All transactions can be committed with this function, regardless of the mode they were begun with. The transaction identifier must refer to a previously initiated transaction or RVM_ETID is returned.

Two commit modes are recognized: flush and no_flush. In a flush commit, the new value records for the transaction are written (flushed) to the log file. Permanence is then guaranteed upon successful return.

A no_flush commit is faster than a commit with log flushing, but provides a weaker guarantee of permanence. The new value records are created, but not written to the log file. Permanence is guaranteed only when they are flushed to the log file. This can be done either by a flush-mode commit of another transaction, or by explicitly flushing the log with rvm_flush. An application can enhance performance by delaying the log flush (and permanence of changes) until a sequence of related transactions has been completed. Note that aborting the last transaction of such a sequence *will not* restore the original state of transactions previously committed with the no_flush option. RVM does not support nested transactions.

DIAGNOSTICS

RVM_SUCCESS
success

RVM_ETID
invalid transaction identifier or pointer

RVM_EMODE
invalide transaction end mode

RVM_EINIT
RVM not initialized

SEE ALSO

`rvm_begin_transaction (3)`, `rvm_set_range (3)`, `rvm_modify_bytes (3)`, and `rvm_flush (3)`

AUTHOR

Hank Mashburn

4.3 Log Control Operations

4.3.1 Log Flush and Truncation

If an application is scheduling log operations, it can explicitly invoke the flush and truncate operations with the rvm_flush and rvm_truncate functions. Both of these functions are considered *slow* operations since they require disk operations, and will lock out other RVM functions requiring disk access, particularly rvm_end_transaction in flush mode. If invoked by more than one thread, these operations will be serialized in the order received by RVM.

RVM will recognize when an operation is not needed because the log is empty, and will return immediately.

4.3.2 Log Initialization

The function rvm_create_log can be used to create log files, *provided* that the file does not already exist. This requirement is made to prevent applications that automatically create a log file on startup from destroying the log records that are necessary to recover from a crash.

Either files or raw disk partitions can be initialized as logs with the utility rvmutl. (Raw disk partitions can only be initialized by rvmutl). Details are found in Chapter 8 (of rvmutl).

NAME

rvm_flush - flush log records to log file

SYNOPSIS

```
#include "rvm.h"

rvm_return_t rvm_flush ()
```

DESCRIPTION

rvm_flush makes visible the log flush operation, which moves transaction records to the log file and insures the permanence of the changes represented by those records. It is normally done with a transaction commit, but is available to the application if it wishes to schedule the flush after a sequence of no_flush transactions.

rvm_flush is synchronous with the calling thread.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_ELOG

no log file has been specified

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_end_transaction (3)`

AUTHOR

Hank Mashburn

NAME

rvm_truncate - truncate the log

SYNOPSIS

```
#include "rvm.h"

rvm_return_t rvm_truncate ()
```

DESCRIPTION

rvm_truncate makes visible the log truncation operation. When the log is truncated, records for committed changes are used to update the committed images of modified segments. After modified data files are successfully updated, the records can be deleted and the log "truncated."

Like log flushing, this operation is usually performed implicitly by RVM, but is available to the application if it wishes to schedule the operation itself. rvm_truncate performs a flush before truncating, and is synchronous with the calling thread.

rvm_truncate requires virtual memory working space. If this space cannot be allocated, RVM_ENO_MEMORY is returned, and the truncation is aborted. If possible, the application can deallocate as much memory as possible and retry the truncation. This can done as many times as necessary. If no memory can be freed, this error should be considered fatal.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_ENO_MEMORY

heap exhausted

RVM_EIO

I/O or kernel error, errno has code number

RVM_ELOG

no log file has been specified

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_flush (3)`

AUTHOR

Hank Mashburn

NAME

rvm_create_log - RVM log file creation

SYNOPSIS

```
#include "rvm.h"

rvm_return_t rvm_create_log (options, length, mode)

rvm_options   *options;  /* pointer to a RVM options record */
rvm_offset_t  *length;   /* length of log record area in file to be created */
long          mode;      /* protection mode of file to be created */
```

DESCRIPTION

rvm_create_log provides a mechanism for creating a log file from within an application. It is intended to allow an application to be self-installing by silently creating the log and other RVM structures it needs. To provide some protection against an applications automatically over-writing a log containing valid records, rvm_create_log can be used to create logs in Unix files only, and only if the file does not already exist. Creating logs in raw partitions must be done with the utility `rvmutl`.

The name of the log file to be created must be specified in the options record. An options record created with typedef rvm_options_t must be used. RVM_EOPTIONS will be returned if options does not point to a valid record. If either the name is not provided, or the file cannot be created and initialized, RVM_ELOG will be returned. Also, the total length of the file to be created must be permitted by the file system. This includes the length specified for log records plus the overhead necessary for the log status area, presently one sector. Options other than the log file name are ignored.

The mode parameter is the Unix file system protection mode and must allow writing by the owner. An i/o error will be returned if the file cannot be created or written.

rvm_create_log does not declare the created log file as the operational log. This must be done by specifying an options descriptor to `rvm_set_options`, `rvm_initialize`, or `rvm_map`.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_ELOG

no log file has been specified

RVM_EIO

I/O or kernel error, errno has code number

RVM_EOPTIONS

invalid RVM options record or pointer

RVM_ENO_MEMORY

heap exhausted

RVM_ENAME_TOO_LONG

file name longer than 1023 characters

RVM_ETOO_BIG

file length requested greater than file system permits

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_set_options (3)`, `rvm_initialize (3)`, `rvm_map (3)`

AUTHOR

Hank Mashburn

4.4 Query and Statistics Functions

RVM allows the application to query the settings of options and some internal state with `rvm_query`. Access to the statistics automatically collected by RVM is provided by `rvm_statistics`.

The query and statistics functions require that the application allocate records for the data to be returned. In general, the application is responsible for deallocation of these records, although they can be used in multiple calls to their respective functions. As with other records types in RVM, the `rvm_options_t` and `rvm_statistics_t` records must be allocated or initialized with the provided type-specific functions.

Because `rvm_query` must return two variable length items, some special allocation conventions must be observed. If the name of the log file is to be returned, the caller must allocate a buffer of `MAXPATHLEN+1` characters and place a pointer to it in the `log_dev` field. The file name will be copied into the buffer. The buffer will not be deallocated by RVM. The `log_dev` field should be null if the name of the log is not to be returned.

The tid_array vector is always returned if uncommitted transactions are present. Since the length isnt known before the call, `rvm_query` must do the allocation. This results in the following deallocation conventions: for option records allocated with `rvm_malloc_options`, the array should not be deallocated by the application since this will be done by `rvm_free_options`. However, if the options record is allocated on the stack, the array must be explicitly deallocated by the application, or space will be lost.

Deallocation of the array in statically allocated options records is optional: repeated use of the record will cause the array to be reallocated as necessary by `rvm_query` with no explicit action by the application. If you do deallocate the array, you *must* null the pointer to avoid a possible internal error due the dangling pointer.

No special deallocation conventions are required for `rvm_statistics_t` records.

NAME

rvm_query - query RVM options

SYNOPSIS

```
#include "rvm.h"

rvm_return_t rvm_query (options,region)

rvm_options_t  *options;  /* pointer to a RVM options record */
rvm_region_t   *region;   /* optional pointer to a region descriptor */
```

DESCRIPTION

rvm_query allows inspection of the RVM options currently set. An options record, specified by options, is filled in with the values currently in effect. If this record is not valid, RVM_EOPTIONS will be returned. Please refer to the RVM manual for the details of options.

If the name of the log file is to be returned, the caller must allocate a buffer of `MAXPATHLEN+1` characters and place a pointer to it in the `log_dev` field. The file name will be copied into the buffer. The buffer will not be deallocated by RVM.

If region is null, the options returned are those global to RVM, otherwise the options will be for the specified region. If the region does not point to a valid region descriptor, RVM_EREGION will be returned.

If only the segment file is specified in the region descriptor, the options are those applicable to all mappings of the segment. If the offset, length, and vmaddr fields of the region descriptor are not zero, the options returned will reflect those of that region. If the region isnt mapped, RVM_ENOT_MAPPED is returned. Certain options reflect the the global state of RVM regardless of the region specified.

The number of uncommitted transactions, if any, is returned in the n_uncommit field of the options descriptor. If uncommitted transactions exist, tid_array will contain a pointer to a vector of length n_uncommit of identifiers for the transactions. If a valid region descriptor has been specified, the uncommitted transaction descriptors returned will be only those modifying the specified region. Otherwise, all uncommitted transactions are returned.

If a tid_array vector exists in the options record when rvm_query is called, it will be replaced to reflect the current status.

For option records allocated with rvm_malloc_options, the array should not be deallocated by the application since this will be done by rvm_free_options. However, if the options record is allocated on the stack, the array must be explicitly deallocated by the application, or space will be lost. Deallocation of the array in statically allocated options records is optional: repeated use of the record will cause the array to be reallocated as necessary by rvm_query with no explicit action by the application. If you do deallocate the array, you *must* null the pointer to avoid a possible internal error due the dangling pointer.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_EREGION

invalide region descriptor or pointer

RVM_ENOT_MAPPED

region not mapped

RVM_EOPTIONS

invalid options record or pointer

RVM_ENO_MEMORY

heap exhausted

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_map (3)`, `rvm_unmap (3)`, `rvm_set_options (3)`, `rvm_abort_transaction (3)`, `rvm_end_transaction (3)`, and `rvm_terminate (3)`

AUTHOR

Hank Mashburn

NAME

rvm_statistics - query collected statistics

SYNOPSIS

```
#include "rvm_statistics.h"

typedef struct
    {
    /* see rvm_statistics.h for current fields */
    }
rvm_statistics_t;

rvm_return_t     rvm_statistics (version, statistics);
char             *version;    /* Statistics version string */
rvm_statistics_t *statistics; /* pointer to a statistics record */

rvm_return_t     RVM_STATISTICS (statistics);
rvm_statistics_t *statistics; /* pointer to a statistics record */
```

DESCRIPTION

`rvm_statistics` copies the statistics collected by RVM to a record provided by the application. The contents of the record can then be analyzed by the application, or formated by `rvm_print_statistics`.

Note that the header `rvm_statistics.h` must be included to use the statistics facility. These declarations are not included in the standard RVM library header since they are expected to change as RVM is further developed. Because of the expectation of change, a version parameter is used to detect version skews between the statistics record format of the application and the RVM library. The string constant RVM_STATISTICS_VERSION from rvm_statistics.h is passed to rvm_statistics, and compared with the value the library was compiled with. If they are different, RVM_ESTAT_VERSION_SKEW is returned and no data is copied to the statistics record.

The macro `RVM_STATISTICS` will automatically pass the version string to the function.

The `statistics` parameter specifies the record to which the statistics will be copied. This record must be of type `rvm_statistics_t` and must have been created by `rvm_malloc_statistics` or initialized with `rvm_init_statistics`. If the record is not properly formated by either of the above functions, the error `RVM_ESTATISTICS` is returned.

For detailed specification of the statistics currently collected, consult the header file `rvm_statistics.h` which is also included in the RVM manual.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_ESTAT_VERSION_SKEW

Statistics version skew

RVM_ESTATISTICS

Invalid statistics record or pointer

RVM_ELOG

no log file has been specified

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_print_statistics (3)`

AUTHOR

Hank Mashburn

NAME

rvm_print_statistics - format and print statistics

SYNOPSIS

```
#include "rvm_statistics.h"

typedef struct
    {
    /* see rvm_statistics.h for current fields */
    }
rvm_statistics_t;]

rvm_return_t      rvm_print_statistics (statistics,out_stream);
rvm_statistics_t *statistics; /* pointer to a statistics record */
FILE              *out_stream; /* pointer to stdio stream */
```

DESCRIPTION

rvm_print_statistics formats an rvm_statistics_t record previously filled in by rvm_statistics. The formated output is set to the specified stream, which must first have been opened with fopen.

For detailed specification of the statistics currently collected, consult the header file `rvm_statistics.h`, which is also included in the RVM manual.

This function can also be invoked from `rvmutl` with the `statistics` command.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_ESTATISTICS

Invalid statistics record or pointer

RVM_EIO

I/O or kernel error, errno has code number

RVM_ELOG

no log file has been specified

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_statistics (3)`, `rvmutl-statistics (1)`

AUTHOR

Hank Mashburn

Chapter 5

RVM Segment Loader

Designed and programmed by David C. Steere

The RVM segment loader provides a mechanism that allows a load map to be created for a segment. Regions can then be mapped to specific virtual memory addresses or to virtual memory allocated by RVM in a single library call. There are only two library functions: `rvm_create_segment` and `rvm_load_segment`.

To use the segment loader, a segment header must be created by `rvm_create_segment`. The header, which occupies the first page of the segment, the load map for regions defined within the segment, and a version identifier so that `rvm_load_segment` can recognize suitable segments. All region lengths must be integral multiple of the page size of the machine, which is available as the macro `RVM_PAGE_SIZE`. Virtual memory addresses must be page-aligned.

The load map is prepared in a vector of descriptors for the desired regions (type `rvm_region_def_t`). The descriptors specify just two parameters: the region length and the virtual memory address where the region is to be loaded. The offsets of the regions in the segment are assigned by `rvm_create_segment`. If pointers are included in the data, a fixed virtual memory address must be specified so that the loader will place the region there at each loading. If there are no pointers, the segment loader can be instructed to allocate space for the region by specifying an address of zero. The segment loader will make virtual memory addressable as necessary when loading.

The virtual memory address for the first region should be specified far enough beyond the applications break point so that the program can grow during development. Otherwise, there will have to be a way to modify the pointers or rebuild the structures.

The segment header establishes the load map for only that segment. If multiple segments are used, each must have its own header, and `rvm_load_segment` must be called for each.

The segment loader is used by the RDS allocator (see Chapter 6 (of RDS)) to automate loading of a persistent heap region and a statically allocated region. A utility program is available to create the segment header and initialize the heap so that it is easily loaded.

The header file giving all necessary definitions for the RVM segment loader is included in Appendix B (of C Declarations).

NAME

rvm_create_segment - create segment header

SYNOPSIS

```
#include "rvm_segment.h

typedef struct
    {
    rvm_offset_t        offset;
    rvm_length_t        length;
    char                *vmaddr;
    }
rvm_region_def_t;

rvm_return_t rvm_create_segment (DevName, DevLength, options,
                                 nregions, regions)

char             *DevName;  /* name of heap segment */
rvm_offset_t     DevLength; /* length of heap raw partition */
rvm_options_t    *options;  /* optional pointer to an options record */
unsigned long    nregions;  /* number of region descriptors */
rvm_region_def_t *region_defs; /* pointer to array of region descriptors */
```

DESCRIPTION

`rvm_create_segment` is used to initialize the segment header with the region load map. The name of the file or partition for the segment is specified by `DevName`, and the length of the partition is specified in `DevLength`. If the segment is represented by a file, `DevLength` should be zero. Also, the file must exist and be at least one page long.

The parameter `region_defs` must point to an array of `rvm_region_def_t` descriptors, with one descriptor for each region to be loaded by `rvm_load_segment`. The descriptor array must be allocated and deallocated by application, and must be as long as the number of descriptors specified by `nregions`. The maximum number of regions that can be described in the header is limited by the page size of the machine, which is available as the macro `RVM_PAGE_SIZE`. If the number of regions requested by the parameter `nregions` causes the header to exceed page size, the error `RVM_ENO_MEMORY` will be returned.

Each `rvm_region_def_t` descriptor requires the regions virtual memory address and its length. The virtual memory address where the region will be loaded must be page-aligned. It can be specified as zero and, if so, will cause the loader to allocate virtual memory for the region. Regions containing pointers must specify an address to guarantee that they are loaded in the same place each time. The region length must be an integral multiple of the page size, or the error code `RVM_ERANGE` will be returned.

Regions are not allowed to overlap in virtual memory. If this condition is not met, the error code `RVM_EVM_OVERLAP` will be returned.

`rvm_create_segment` assigns space in the segment to the header and regions contiguously, beginning with the header at byte zero and with length equal to the page size. The regions are allocated next, in the order of their occurrence in the `region_defs` array. The offset is returned in the `offset` field of each descriptor.

The virtual memory specified in the descriptors does not have to be addressable when calling `rvm_create_segment`. Also, `rvm_create_segment` does not map or alter the applications data in the segment; only the first page of the segment for the header is affected.

Since it is an initialization function, `rvm_create_segment` performs no internal synchronization, so if there is a possibility of concurrent access to the segment, the application must do appropriate serialization.

An optional RVM options descriptor can be passed to `rvm_create_segment` if no log has been previously declared. RVM must be initialized before `rvm_create_segment` is called.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_ERANGE

a region length is not page sized

RVM_EVM_OVERLAP

region mappings oberlap in virtual mapping

RVM_ENO_MEMORY

heap exhausted, or header cannot be allocated

RVM_EOPTIONS

invalid options record or pointer

RVM_ELOG

no log file has been specified

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_load_segment (3)`, `rvm_map (3)`, `rds_zap_heap (3)`, `rds_load_heap (3)`, `rdsinit (1)`

AUTHOR

David C. Steere

BUGS

Specification of region allocations with the `offset` field of the region descriptors is not allowed.

NAME

rvm_load_segment - automatically map regions of segment

SYNOPSIS

```
#include "rvm_segment.h

typedef struct
    {
    rvm_offset_t        offset;
    rvm_length_t        length;
    char                *vmaddr;
    }
rvm_region_def_t;

rvm_return_t rvm_load_segment (DevName, DevLength, options,
                               nregions, regions)

char             *DevName;  /* name of heap segment */
rvm_offset_t     DevLength; /* length of heap raw partition */
rvm_options_t    *options;  /* optional pointer to an options record */
unsigned long    *nregions;  /* number of region descriptors  */
rvm_region_def_t *regions[]; /* pointer to array of region descriptors  */
```

DESCRIPTION

`rvm_load_segment` is used to automatically map the regions of a segment prepared with `rvm_create_segment`. The name of the file or partition for the segment is specified by `DevName`, and the length of the partition is specified in `DevLength`. If the heap is in a file, `DevLength` should be zero.

`rvm_load_segment` will first to load the array of `rvm_region_def_t` descriptors contained in the segment header. If the segment header is not recognizable, the error `RVM_ESEGMENT_HDR` will be returned. A version identifier will also be checked, and if the version of the segment library that created the header does not match the version of the library linked with the application, the error `RVM_EVERSION_SKEW` will be returned. Otherwise, the address and length of the region descriptors array will be returned in regions and `nregions`, respectively. The header array must be deallocated by the application.

After the header is verified, each of the regions will be mapped via `rvm_map`. Allocation of virtual memory is done by `rvm_load_segment`, so the application must not attempt allocation, or the error `RVM_ERANGE` will result. Regions whose virtual memory address is non-zero will be automatically mapped to that address. Regions with zero virtual memory addresses will have space allocated by RVM, and the allocated address will be returned in the `vmaddr` field of the descriptor. After successful completion of `rvm_load_segment`, all data specified by the header is addressable to the application.

Regions are not allowed to overlap in either the segment or virtual memory. If this condition is not met, the error codes `RVM_EOVERLAP` or `RVM_EVM_OVERLAP` will be returned, as appropriate.

Since it is an initialization function, `rvm_load_segment` performs no internal synchronization, so if there is a possibility of concurrent access to the segment, the application must do appropriate serialization.

An optional RVM options descriptor can be passed to `rvm_load_segment` if no log has been previously declared. RVM must be initialized before `rvm_load_segment` is called.

DIAGNOSTICS

RVM_SUCCESS

success

RVM_ESEGMENT_HDR

segment header invalid

RVM_EVERSION_SKEW

segment header built with different version of segment loader library

RVM_EOVERLAP

segment regions overlap

RVM_EVM_OVERLAP

segment regions overlap in virtual memory

RVM_EOPTIONS

invalid options record or pointer

RVM_ENO_MEMORY

heap exhausted, or buffer cannot be allocated

RVM_ELOG

no log file has been specified or invalid file name

RVM_EINIT

RVM not initialized

SEE ALSO

`rvm_create_segment (3)`, `rvm_map (3)`, `rds_load_heap (3)`

AUTHOR

David C. Steere

Chapter 6

RDS, A Dynamic Heap Allocator for RVM

Designed and programmed by David C. Steere

6.1 Introduction to RDS

The Recoverable Dynamic Storage (RDS) library provides a heap similar to that of the C library, although with persistence, for RVM segments. Additional provisions for convenient initialization and reloading of the heap after shutdown or crash are included.

RDS uses the Quick Fit method [?] for allocation. In this method, separate free lists are maintained for a specified number of sizes. Most programs tend to repeatedly allocate a small number of distinct sized blocks. Quick Fit avoids overhead by retaining those block sizes as long as possible in the expectation that they will be re-allocated when freed. Coalescing of adjacent free blocks is deferred until a request cannot be satisfied.

The free list sizes are integer multiples of the *chunk size*, which is specified when the recoverable heap is created and cannot be modified thereafter except by re-initializing the heap. The chunk size must be an integral multiple of `sizeof (char *)`, and be at least `RDS_MIN_CHUNK_SIZE`. The free lists are maintained in sizes of one chunk to n chunks, where n is the maximum number of free lists. The number of lists is also specified at heap creation and cannot be modified thereafter.

When an allocation request is made, the requested size is rounded up to an integral multiple of the chunk size. The free list for that size is checked first, and if not empty, the first block is returned. Otherwise, the next larger free list is checked, and if not empty, the first block is selected, the requested size split off, and the remainder placed on the appropriate free list. If the free list is empty, the next is searched. If no specific-sized free list can allocate the block, the request is taken from the single list of larger blocks by splitting the first block on the list. The remainder is returned to the appropriate free list. Sizes larger than n times the chunksize are also allocated from the large blocks list on a first-fit basis. This list is always created in addition to the fixed size lists.

When a block is freed, it is simply placed on the appropriate free list with no attempt coalesce it with adjacent free blocks. Coalescing is deferred until an allocation request can not be satisfied.

The allocation and deallocation functions, rds_malloc and rds_free, resemble malloc and free, although they are modified for transactions. To help prevent fragmentation, RDS also provides for pre-allocation of pools of block sizes that the application will repeatedly allocate with the function rds_prealloc.

The function rds_load_heap provides a convenient way to use the RVM segment loader to load the persistent heap and a statically allocated region. rds_load_heap will make the necessary RVM mapping calls and initialize RDS. When it completes, the allocations persistent data is ready for use.

Initialization of an RDS heap is done with the function rds_init_heap. A second function, rds_zap_heap, which calls rds_init_heap and rvm_create_segment, can be used to prepare a segment for loading via rds_load_heap. However, heap initialization is usually done with the utility rdsinit, which will prompt for the necessary parameters to call rds_zap_heap.

To assist in detecting buffer overruns and other memory clobbers, RDS puts special *guard* constants at the beginning and end of each block. These are not within the space allocated to satisfy an allocation request and will never be addressed by a correctly operating application. The rounding up of allocation request sizes to integral multiples of the chunk size automatically includes space for the guard constants.

If memory damage is detected by the corruption of one of the guards when a block is deallocated, or when damage to a free list is detected during allocation, the error code ECORRUPT is returned. If this error is detected during deallocation, the block is not put on a free list, but is left unmodified for analysis of how the overwrite occurred.

RDS generally returns error indications in a parameter whose address must be specified in the call. However, a few functions return the error code as their value. RDS error codes are all defined as negative integers in rds.h. Because RDS must perform its actions via transactions, the RVM error codes can also be returned in the error code parameter or as function values. These are defined in `rvm.h`, and are all positive values. Success is always indicated by the code SUCCESS, which is defined to be zero.

The header file giving all necessary definitions for RDS is included in Appendix B (of C declarations).

6.2 Transactions and RDS

All functions in RDS use RVM transactions. Some, particularly rds_malloc and rds_free, must interact with the applications transactions, requiring some care from the caller. RDS does not hold its lock until after commit so that greater concurrency can possibly be achieved in Coda. This is not generally recommended because it permits uncommitted state to be used by other than the calling transaction. See the notes on good programming practice with RVM, Section 2.4.2 (of Transactions in Critical Sections).

Since the lock is not held until commit, some additional complexity must be introduced to insure that allocation and deallocation actions are independent. There are two cases that must be recognized: if the caller guarantees that no context swaps among application threads will occur before transaction commit, and if such swaps must be allowed.

If context swapsare avoided, the application can supply the transaction to be used by RDS in the allocation/deallocation calls and be sure that if the transaction must later be aborted, all will be restored correctly. The transaction must be started in restore mode, which is required for all RDS calls. The transaction will not be committed or aborted by RDS.

If context swaps must be allowed (frequently the case), a null transaction parameter must be specified. This will force an internal transaction to be created by RDS. This transaction will be committed by RDS before the function returns. In this case, if `rds_malloc` is called and the application transaction requesting the allocation must abort, the block must be explicitly deallocated by a call to rds_free. Otherwise, recoverable heap space will be lost.

When the application deallocates blocks in a transaction that could later abort and context swaps are possible, two special functions are provided, rds_fake_free and rds_do_free. These functions manage an *intention list* that records the intended deallocations, but defers the actual deallocation until the application transaction is ready to commit. If the transaction aborts, the intention list is discarded. The application must call rds_fake_free at the point where the block would logically be freed, and then call rds_do_free immediately before committing the application transaction. The application must provide the intention list header, of type `intentionList_t`; RDS will manage the actual list.

Note that these methods are safe *only* in a coroutine thread environment. If pre-emptive threads, or a multiprocessor, are required, RDS cannot safely be used.

NAME

rds_malloc - allocate from recoverable heap

SYNOPSIS

```
#include "rds.h"

int rds_malloc (size, tid, err);

unsigned long size;  /* length of allocation request */
rvm_tid_t     *tid;  /* optional pointer to transaction identifier */
int           *err;  /* pointer to error return location */
```

DESCRIPTION

`rds_malloc` allocates from the recoverable heap a block large enough to hold the request, specified in bytes by the `size` parameter. The address of the allocated block is returned as type `int` and must be cast to the desired type. If the request cannot be satisfied, zero is returned, and the `err` parameter is set to the appropriate error code.

Because allocation actions in the recoverable heap must be done via transactions, `rds_malloc` offers two choices for the transaction. In the first case, `rds_malloc` can be instructed to use an existing transaction begun in `restore` mode, by passing the address of a valid `rvm_tid_t` record in the `tid` parameter. This avoids extra transaction start and commit overhead and provides automatic deallocation if the transaction must later abort. However, no context swap can be permitted between allocation and transaction commit or abort since the modified, but uncommitted, free lists are visible to other RDS actions.

If context swaps cannot be prohibited, or the available transaction was started in `no_restore` mode, the `tid` parameter should be set to null, instructing `rds_malloc` to start an internal transaction. This transaction will be committed in `no_flush` mode if the allocation is made, and aborted otherwise. In this case, an explicit `rds_free` must be done if the allocating transaction later aborts.

DIAGNOSTICS

SUCCESS

success

ENO_ROOM

RDS heap exhausted

ECORRUPT

RDS heap damaged

EHEAP_INIT

RDS not initialized

positive values

RVM return codes

SEE ALSO

`rds_free (3)`

AUTHOR

David C. Steere

BUGS

The internal synchronization is not valid with pre-emptive threads.

NAME

rds_free - return a block to the recoverable heap

SYNOPSIS

```
#include "rds.h"

int rds_free (addr, tid, err)

char          *addr; /* address of block to be freed */
rvm_tid_t     *tid;  /* optional pointer to transaction identifier */
int           *err;  /* pointer to error return location */
```

DESCRIPTION

`rds_free` deallocates recoverable storage allocated by `rds_malloc`. The block to be deallocated is passed in the `addr` parameter. This block must not already be free, or the `EFREED_TWICE` error code will be returned in `err`.

Because deallocation actions in the recoverable heap must be done via transactions, `rds_free` offers two choices for the transaction. In the first case, `rds_free` can be instructed to use an existing transaction by passing the address of a valid `rvm_tid_t` record in the `tid` parameter. This avoids extra transaction start and commit overhead and provides automatic reallocation if the transaction must later abort.

If the available transaction was started in `no_restore` mode, the `tid` parameter should be set to null, instructing `rds_free` to start an internal transaction. This transaction will be committed in `no_flush` mode if the allocation is made, and aborted otherwise.

In neither case can context swaps be permitted between deallocation and transaction commit or abort since the modified, but uncommitted, free lists are visible to other RDS actions. If this condition cannot be met, or if there is a possibility that the transaction will abort, the functions `rds_fake_free` and `rds_do_free` must be used.

DIAGNOSTICS

SUCCESS

success

EFREED_TWICE

block is already free

ECORRUPT

RDS heap damaged

EBAD_ARGS

block is not on 4 byte boundary

EHEAP_INIT

RDS not initialized

positive values

RVM return codes

SEE ALSO

`rds_malloc (3)`, `rds_fake_free (3)`, `rds_do_free (3)`

AUTHOR

David C. Steere

BUGS

The internal synchronization is not valid with pre-emptive threads.

NAME

rds_fake_free - deferred return of blocks to the recoverable heap

SYNOPSIS

```
#include "rds.h"

typedef struct intlist {
    unsigned long          size;
    unsigned long          count;
    char                   **table;
} intentionList_t;

int rds_fake_free(addr, list);
char            *addr; /* address of block to be freed */
intentionList_t *list; /* pointer to intention list */

int rds_do_free(addr, list);
intentionList_t *list; /* pointer to intention list */
rvm_mode_t      mode;  /* transaction commit mode */
```

DESCRIPTION

In cases where the application must free recoverable storage without knowing whether the transaction will later abort, `rds_free` should not be used since the internal free list lock is not held until commit or abort. To handle these cases, RDS allows the creation of an *intention list* for the blocks to be freed. At the point where the block would logically be freed, `rds_fake_free` is called to place the block on the intention list. To actually free the blocks on the intention list, `rds_do_free` is called. This must be done immediately before calling `rvm_end_transaction` for the deallocating transaction, and must not be done if `rvm_abort_transaction` is called. Also, no context swaps can be allowed between the calls on `rds_do_free` and `rvm_end_transaction`.

The intention list header (type `intentionList_t`) must be provided by the application. The `table` field must be null on the first call to `rds_fake_free`. RDS will allocate space as necessary for the blocks to be freed. `rds_do_free` will deallocate the `table` vector. The application is responsible for deallocation of `table` if the transaction is aborted. Deallocation of the `intentionList_t` itself is always the responsibility of the application. RDS does not internally synchronize access to the intention list. If more than one thread is involved in the transaction, synchronization of access to the intention list is the responsibility of the application.

The block to be deallocated is passed in the `addr` parameter to `rds_fake_free` . This block must not already be free, or the `EFREED_TWICE` error code will be returned. The `ECORRUPT` error return is given by `rds_fake_free` if the block guards are damaged. `rds_do_free` returns this code if internal damage to the free lists is found.

These functions return error codes as the value of the function rather than in a parameter.

DIAGNOSTICS

SUCCESS

success

EFREED_TWICE

block is already free

ECORRUPT

block or heap damaged

EBAD_ARGS

block is not on 4 byte boundary

EHEAP_INIT

RDS not initialized

positive values

RVM return codes

SEE ALSO

`rds_free (3)`, `rds_malloc (3)`

AUTHOR

David C. Steere

NAME

rds_init_heap - initialize a recoverable heap

SYNOPSIS

```
#include "rds.h"

int rds_init_heap (base, length, chunkSize, nlists, tid, err)

char          *base;     /* base address of heap */
unsigned long length;    /* length of heap */
unsigned long chunkSize; /* size of allocation unit */
unsigned long nlists;    /* number of allocation lists */
rvm_tid_t     *tid;      /* pointer to transaction identifier */
int           *err;      /* pointer to error return location */
```

DESCRIPTION

`rds_init_heap` initializes a recoverable heap in the previously mapped memory region specified by the address `base` and `length` parameters. Allocation requests will be rounded up to an integral number of allocation units `chunkSize`, which is specified in bytes, and must be an integral multiple of `sizeof (char *)`, and be at least `RDS_MIN_CHUNK_SIZE`.

For rapid allocation, RDS maintains separate allocation lists for blocks of different sizes. The sizes are integrals of `chunkSize`, beginning with 1 and extending to the number of lists specified by `nlists`, which must be at least `RDS_MIN_FREE_LISTS`. These lists are initially empty and the entire heap is placed on the large block list as a single block. `rds_malloc` will split this block as required.

If a number of blocks of specific sizes are required by the application, the function `rds_prealloc` is provided for efficient pre-allocation. It should be called immediately after initializing the heap.

An active transaction must be specified in the `tid` parameter; RDS will not create an internal transaction for this function. This transaction must be committed by the application for the initialization to be permanent.

Since this function is called only to initialize the heap, RDS assumes there is no concurrent access and does no internal locking.

DIAGNOSTICS

SUCCESS

success

ENO_ROOM

heap length not long enough for heap structures

EBAD_CHUNK_SIZE

the chunk size is not large enough or is not a multiple of sizeof(char *)

EBAD_LIST_LENGTH

the number of free lists is not at least RDS_MIN_FREE_LISTS

positive values

RVM return codes

SEE ALSO

`rds_zap_heap (3)`, `rds_prealloc (3)`

AUTHOR

David C. Steere

NAME

rds_load_heap, rds_start_heap - map recoverable storage

SYNOPSIS

```
#include "rds.h"

int rds_load_heap (DevName, DevLength, staticAddr, err);
char         *DevName;  /* name of heap segment */
rvm_length_t DevLength; /* length of heap raw partition */
char         **staticAddr; /* address of static region (out)*/
int          *err;      /* pointer to error return location */


int rds_start_heap (staticAddr, err);
char         **staticAddr; /* address of static region (out)*/
int          *err;      /* pointer to error return location */
```

DESCRIPTION

`rds_load_heap` provides a convenient method of mapping the heap segment and initializing RDS. The name of the file or partition containing the recoverable heap is specified by `DevName`, and the length of the partition is specified in `DevLength`. If the heap is in a file, `DevLength` should be zero.

`rds_load_heap` calls `rvm_load_segment` to map the heap. This requires that the heap file or partition be created with `rds_zap_heap` or `rvm_create_segment`, which is most conveniently done with the `rdsinit` utility. After the heap is mapped, the address of the static region is returned in `staticAddr`. `rds_start_heap` is automatically called to initialize RDS.

If the actual mapping of the heap and static regions is done by the application, `rds_start_heap` should be used to initialize RDS. In this case, the address of the static region must be specified to RDS in the `staticAddr` parameter.

With either function, the version of RDS that the heap was built with is compared with the version of the currently linked RDS library. If there is a mismatch, the error code `EHEAP_VERSION_SKEW` is returned.

Since these functions are called only to initialize the application, RDS assumes there is no concurrent access and does no internal locking.

DIAGNOSTICS

SUCCESS

success

EHEAP_VERSION_SKEW

RDS heap and library have different versions

positive values

RVM return codes

SEE ALSO

rvm_load_segment (3), rvm_map (3), rdsinit (1)

AUTHOR

David C. Steere

NAME

rds_zap_heap - initialize an RVM segment as an RDS heap

SYNOPSIS

```
#include "rds.h"

int rds_zap_heap (DevName, DevLength, startAddr, staticLength,
                  heapLength, nlists, chunkSize, err)


char           *DevName;  /* name of heap segment */
rvm_length_t  DevLength; /* length of heap raw partition */
char           *startAddr;/* base address of heap */
rvm_length_t  staticLength; /* length of static region */
rvm_length_t  heapLength;/* length of heap region */
unsigned long nlist;     /* number of allocation lists */
unsigned long chunkSize; /* size of allocation unit */
int           *err;      /* pointer to error return location */
```

DESCRIPTION

`rds_zap_heap` prepares a recoverable heap for loading by `rds_load_heap`. Two regions are created in the heap segment named by `DevName`, one for the applications statically allocated region and one for the recoverable heap. The lengths of the regions, specified by `staticLength` and `heapLength`, must be integrals of the system page size. The heap region is allocated at `*startAddr` in virtual memory, which must be on a page boundary. This address is the permanent address of the heap. The static region starts at `*startAddr` + `heapLength` and is also permanent. One additional region is created at the beginning of the segment for the segment header, and is one page long. This region is only temporarily mapped by `rds_load_heap`. The heap region follows the header in the segment, and the static region follows the heap.

If the segment is represented by a raw partition, the maximum usable length of the partition must be specified in `DevLength`. For files, this length should be zero.

The region of virtual memory for the heap and static regions should not be allocated by the application. `rds_zap_heap` uses the RVM segment utilities to build the segment header and map the regions. When `rds_zap_heap` is complete, the memory will be mapped.

`rds_zap_heap` uses `rvm_create_segment` to build the segment, and then calls `rds_init_heap` to initialize the free lists as specified by the `nlists` and `chunkSize` parameters. `chunkSize` is specified in bytes, and must be an integral multiple of `sizeof (char *)`, and be at least `RDS_MIN_CHUNK_SIZE`. The number of free lists must be at least `RDS_MIN_FREE_LISTS`. The transaction required by `rds_init_heap` is created and terminated internally.

If a number of blocks of specific sizes are required by the application, the function `rds_prealloc` is provided for efficient pre-allocation. It should be called immediately after initializing the heap.

The utility `rdsinit` is usually used to create recoverable heap segments. It prompts for the parameters and calls `rds_zap_heap`.

Since this function is called only to initialize the heap, RDS assumes there is no concurrent access and does no internal locking.

DIAGNOSTICS

SUCCESS

success

ENO_ROOM

heap length not long enough for header

EBAD_CHUNK_SIZE

the chunk size is not large enough or is not a muliple of `sizeof(char *)`

EBAD_LIST_LENGTH

the number of free lists is not at least `RDS_MIN_FREE_LISTS`

positive values

RVM return codes

SEE ALSO

`rds_init_heap (3)`, `rds_load_heap (3)`, `rvm_create_segment (3)`, `rvm_load_segment (3)`, `rdsinit (1)`, `rds_prealloc (3)`

AUTHOR

David C. Steere

NAME

rds_prealloc - initialize allocation pools

SYNOPSIS

```
#include "rds.h"

int rds_prealloc (size, nblocks, tid, err)

unsigned long size;    /* size of allocation request */
unsigned long nblocks; /* length of allocation request */
rvm_tid_t     *tid;    /* pointer to transaction identifer */
int           *err;    /* pointer to error return location */
```

DESCRIPTION

`rds_prealloc` assists applications in minimizing heap fragmentation by pre-allocating pools of blocks in sizes that the application is known to use in relatively large numbers. `rds_prealloc` is best called immediately after the heap is initialized, although it can be called anytime. When pools of several block sizes are pre-allocated, they should be allocated in increasing block size order so that the larger blocks are not split to make the smaller.

The `size` parameter specifies the size, in bytes, of the blocks to be pre-allocated, and `nblocks` controls the number of blocks pre-allocated.

Because allocation actions in the recoverable heap must be done via transactions, `rds_prealloc` offers two choices for the transaction. In the first case, `rds_prealloc` can be instructed to use an existing transaction begun in `restore` mode, by passing the address of a valid `rvm_tid_t` record in the `tid` parameter. This avoids extra transaction start and commit overhead and provides automatic deallocation if the transaction must later abort. However, no context swap can be permitted between allocation and transaction commit or abort since the uncommitted state is visible to other allocations.

If context swaps cannot be prohibited, or the available transaction was started in `no_restore` mode, the `tid` parameter should be set to null, instructing `rds_prealloc` to start an internal transaction. This transaction will be committed in `no_flush` mode if the allocation is made, and aborted otherwise. In this case, if the allocating transaction later aborts, the pre-allocated blocks remain allocated.

DIAGNOSTICS

SUCCESS

success

ENO_ROOM

RDS heap exhausted

ECORRUPT

RDS heap damaged

EHEAP_INIT

RDS not initialized

positive values

RVM return codes

SEE ALSO

`rds_malloc(3)`, `rds_init_heap(3)`, `rds_zap_heap(3)`

AUTHOR

David C. Steere

BUGS

The internal synchronization is not valid with pre-emptive threads.

NAME

rds_get_stats, rds_clear_stats, rds_print_stats - RDS statistics functions

SYNOPSIS

```
#include "rds.h"

int          rds_get_stats (stats);
rds_stats_t  *stats; /* pointer to RDS statistics record */

int          rds_clear_stats (err);
int          *err;   /* pointer to error return location */

int          rds_print_stats();
```

DESCRIPTION

RDS maintains simple statistics about the recoverable heap and the allocations performed. The statistics area is kept in the recoverable heap region and is initialized when the heap is initialized.

The statistics can be read by `rds_get_stats` into an `rds_stats_t` record for analysis by the application, or printed on `stdout` by `rds_print_stats`. `rds_get_stats (stats)` and `rds_print_stats ()` return error codes as the value of the function rather than in a parameter.

After heap initialization, the statistics area can be cleared by calling `rds_clear_stats`. `rds_clear_stats` simply zeroes the statistics area via an internal transaction. The space allocated and free counts will not be accurate if it is called after initializing the RDS heap. Note that `rds_clear_stats` requires an error return code as a parameter, while the others return the error status as the function value.

RDS performs no internal synchronization for these calls, so if there is a possibility of concurrent access to the statistics, the accesses must be serialized by the application.

DIAGNOSTICS

SUCCESS

success

EHEAP_INIT

RDS not initialized

EBAD_ARGS

null RDS statistics record pointer

AUTHOR

David C. Steere

NAME

rdsinit - RDS heap initialization utility

SYNOPSIS

rdsinit *log data_ seg*

rdsinit *log data_ seg datalen saddr hlen slen nl chunk*

rdsinit *-f log data_ seg datalen saddr hlen slen nl chunk*

DESCRIPTION

`rdsinit` is a utility that constructs an initialized RDS heap in an RVM segment. It is intended to create a structure that can be loaded by `rds_load_heap`.

There are three different ways of using rdsinit. General users are expected to use first two interactive modes, where users may supply parameters for the rds heap interactively or on command line arguments. However, in both cases, users will be asked interactively to confirm their choice of parameters before rdsinit goes ahead to make any permanent change. These are the preferred modes of using rdsinit. Script writers, however, may prefer to supply all the parameters on the command line and *no* confirmation required for those parameters. This is accomodate in the third mode where an additional switch of -f (firm) is supplied on the command line.

In any case, two command-line parameters are always required: `log` and `data_seg`. The former is the name of the RVM log, which must have previously been initialized by `rvmutl`; the latter is the name of the data segment that will be initialized with an RDS heap. If either is missing, a command line error is printed. If the log has not been initialized, an RVM error will result. A short message indicating RVM initialization succeeded is then printed. Both the log and data segment can be regular files or raw partitions.

After the name of log and data segment, there are six other numeric parameters required. They are summarized here and will be explained one by one in the following paragraphs:

datalen

Length of data segment

saddr

Starting address of rvm

hlen

Heap region length

slen

Static region length

nl

Number of lists of free block

chunk

Chunk size

While entering these six numeric parameters, either on command line on via the interactive prompt, users may use numeric number in hexadecimal, decimal or even octal notation. Hexadecimal numbers are preceeded by `0x`, decimal numbers are preceeded by nothing and octal numbers are preceded by `0`.

Special note for long time rdsinit user: the old rdsinit automatically assumed `saddr`, `hlen` and `slen` parameters supplied on command lines are in hexidecimal and *did not* require the prefix `0x`. *This is no longer true with this version of rdsinit.*

Users specify the length of the data segment with the parameter *datalen*. Again, old version of rdsinit did not require this parameter if the data segment was a regular file and it existed already at the time of running rdsinit. This special case is eliminated: *length of data segment must to be supplied in all circumstances.*

Starting address of rvm, or *saddr*, is where heap and static region will be mapped into virtual memory. Heap region is located right at the starting address, while static region is located at starting address plus heap region length. Users may need to have some knowledges of the overall layout of the virtual memory use by the system before they can make the right choice of starting address. For example, the starting address of rvm must be much larger than the largest possible break point of your application, and it should not be in conflict other uses of virtual memory (such as use by shared libraries). It must also be on a page boundary. In CMU, we use 0x20000000 (536870912) with Linux and BSD44, or 0x70000000 (1879048192) with Mach. It is possible to choose other values, but you have to choose them carefully.

Length of regions of heap and static are specified by the parameter *hlen* and *slen* respectively. They both must be integral multiple of pagesize of the system. Also, the combined length of the two regions must be smaller than the length of data segment minus one extra page.

Note that the above three parameters: saddr, hlen, slen, are permanent. They can't be changed without re-initizing (and brain-wiping) the data segment.

The next two parameters: *nl* and *chunk* are related to underlying structure of management of the heap. RDS uses the Quick Fit method for heap allocation. In this method, free blocks are maintained by a number of free lists, each list for one particular size of free blocks. Specifically, there will be `nl` free lists, and each of them will have free blocks of size `1..nl` chunk respectively.

Chunk size must be integral multiple of `sizeof(char *)`, and be at least `RDS_MIN_CHUNK_SIZE`. Number of lists must be at least `RDS_MIN_FREE_LISTS`. For example, a reasonable choice is to have 100 free list with chunk size 32 bytes.

Once all the parameters are chosen, rdsinit will ask user for confirmation before it goes ahead and make permanent change on the log and data segment. Note in the following example that those numerical arguments are presented in both hexidecimal and decimal (in bracket). It is safe to quit at this point and no permanent changes will be made.

```
The following parameters are chosen:
   length of data segment:    0xf5000 (   1003520)
  starting address of rvm: 0x20000000 ( 536870912)
                 heap len:    0xf0000 (    983040)
               static len:     0x4000 (     16384)
                   nlists:       0x64 (       100)
               chunk size:       0x20 (        32)
Do you agree with these parameters ? (y|n|q) y
```

If user supplied the -f (firm) switch on command line, this last confirmation will not show up.

SEE ALSO

`rds_init_heap (3)`, `rds_load_heap (3)`, `rds_zap_heap (3)`, `rvm_create_segment (3)`, `rvm_load_segment (3)`, `rvmutl (1)`

AUTHOR

David C. Steere, created man page

Yui W. Lee, modified (Sept 1997)

Chapter 7

RVM Internals

By Yui Wah LEE

7.1 Overview of RVM internal structure

RVM is designed to be a *run-time library* to be linked with an *external application*. It allows the application to be benefited from *transactional* support. In particular, it allows operations on virtual memory to be *atomic* and *persistent*, provided that the application declares its intention through the RVM application programming interface.

Figure 7.1 () shows the three important components of RVM: *virtual memory*, *data segment* and *log*.

Virtual memory is where the application manipulates its data. RVM allows the application to declare *regions* of virtual memory to be associated with regions in external data segments. The data segments are the persistent backing store of the virtual memory regions: latest committed image of the virtual memory region has a copy in the data segments. Since data segment is on stable storage (disk), it is persistent across program crashes. Should a application crashes, it can always find those memory regions from the data segment. The association of virtual memory regions with segment regions is called *mapping*.

For performance reason, RVM does not write *directly* the committed image of virtual memory regions back to the data segment. Doing so would generate excessive 'seek' operations on the magnetic head of the disk, since the access pattern can be random. Disk head seeks are expensive operation and should be minimized. In the case of RVM, this is achieved by writting *new value* of memory ranges as `records` in an *append-only*, *sequential* log. This is called *flush* operation. Log is also on stable storage (disk or battery-backed ramdisk). Once the new values are written on log (flushed), they become persistent across program crashes. From time to time, records on the log are processed and applied onto the data segment. After that, the log space used by the the records can be re-claimed for further use. This is called *log truncation*. Log truncation is necessary, without it the log will grow unboundedly.

7.2 RVM application programming interface

To facilitate our discussion, let first review briefly the RVM application programming interface. Details of the intefaces can be found in chapter 4 (of RVM Library Functions)

```
rvm_options_t   options;
char            *version;
```

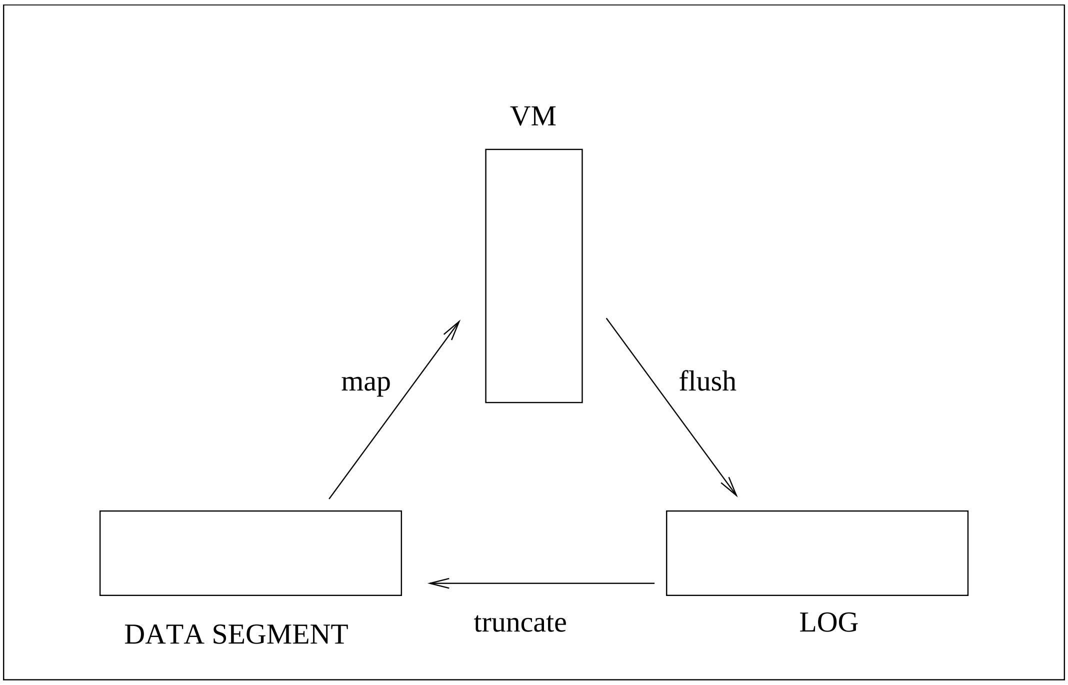

Figure 7.1: Overview

```
rvm_region_t    region;
rvm_tid_t       *tid;
char            *dest, *src;
unsigned long   length;

rvm_initialize(version, options)

rvm_map(region, options);

rvm_begin_transaction(tid, restore);
rvm_begin_transaction(tid, no_restore);

rvm_set_range(tid, dest, length);
rvm_modify_bytes(tid, dest, src, length);

rvm_end_transaction(tid, flush);
rvm_end_transaction(tid, no_flush);
rvm_abort_transaction(tid);

rvm_flush();
rvm_truncate();

rvm_unmap(region);

rvm_terminate();
```

Application declares regions of VM that it want to have persistent image by calling `rvm_map()`. There are two possible modes for `rvm_map()`: `copy` and `no_copy`. The former is the default and will cause the corresponding region in the segment device to be read into virtual memory. The latter is used when the application is sure the image in segment device is useless (e.g., the application is going to replace it immediately).

Application declares the begining of a transaction by calling `rvm_being_transaction()`. The transaction can be committed by calling `rvm_end_transaction()`, or be aborted by calling `rvm_abort_transaction()`. In addition, the application must inform RVM the memory location that will be affected by the transactions. These can be done by calling `rvm_set_range()` or `rvm_modify_bytes()`. RVM guarantees that if the transaction commits, then all the modifications make on declared ranges will be made effective atomically, conversely, if the transaction abort, then none of modifications will be made effective.

Normally, when transactions are aborted, the *old value* in the affected memory ranges will be *restored*. So, a default option for `rvm_begin_transaction` is `restore`. However, in some case, the application can be sure that it will never abort the transaction. For example, if there are no concurrent transaction, the application can be sure it will never abort the transaction explicitly. For these cases, the application can start the transaction with the `no_restore` mode.

If the application want the committed transaction be also recorded in the log device (stable storage) *immediately*, it should commit the transaction with `flush` mode. In this mode, the `end_transaction()` calls return only after the transaction records are written to disk. However, there are cases when the application can accept slight chance of lose of permanency. In those case, it can end the transaction with a `no_flush` mode. There will be a big payoff in performance, as the function `rvm_end_transaction()` will return immediately, without having to the wait for the slow disk I/O to complete. RVM will put the committed transaction in a queue and write them out to the log device later. This is called *lazy commit* and it provides a *bound persistence*. The committed transactions may lost if the program crashed before the queued transaction get a chance to be written out to log. Note that although the transaction is lost, it is lost atomically. Upon the next restart of the application, RVM will restore an old consistent view of the virtual memory.

Commit transaction lazily also give more opportunity to inter- transaction optimization.

To force all previously committed transaction records to be written out to log device, the application can call `rvm_flush()`. Also, whenever the application commit a new transaction with `flush` mode, there will be a side effect that all previous transaction on the queue will also be flushed.

The log device is only a temporary stable storage for commited transactions: its primary purpose is to allow records be written on disk sequentially. Eventually, these records need to be applied to the data segment. The process of applying transaction log records to data segment is called *log truncation*. RVM will carry out log truncation automatically when the log is too full. Users can specify the option `truncate` to set the threshold. Alternatively, the application can initiate a trunction by calling `rvm_truncate()`. Also, note that RVM will do a log truncation upon start up and `rvm_map()` operations. This is to ensure the data segment has the lastest updates before it get mapped into virtual memory.

7.3 Structure of log

7.3.1 Log pointers and limits

The log device can be a

- regular Unix file
- raw partition
- raw partition in a dedicated disk.

It is recommended to use the third option, raw partition in a dedicated disk. As we see in the discussion in section 1, the purpose of the log device is to make the disk I/Os as sequential as possible, having other simultaneous disk I/O will defeat this purpose.

_struct.eps _struct.ps _struct.eps.gz _struct.ps.gz _struct.eps.Z

Figure 7.2: Structure of the log

_trunc.eps _trunc.ps _trunc.eps.gz _trunc.ps.gz _trunc.eps.Z

Figure 7.3: Truncation of the log

Meta data of the log are collectively recorded on the *log status block* which is located at the beginning of the log. It is offset by either `FILE_STATUS_OFFSET` or `RAW_STATUS_OFFSET`, depending on the types of the log devices. The two constants are currently 0 and 8192 bytes (16 sectors) respectively.

Log records are appended sequentially on log area after the log status block. The area starts from `log_start` and is of size `log_size`. (Fig 7.2 ())

As new records are appended, the log pointer `log_tail` will be advanced. `log_head` point to the first record in the log that is not yet subjected to truncation (*current epoch*). If truncation is not in progress, both of the pointers `prev_log_head` and `prev_log_tail` will have a zero value. (Fig 7.3 ()b)

Currently, the log are truncated in a way called *epoch truncation*. In one atomic operation, the function `new_epoch()` will bracket all the current log records by `prev_log_head` and `prev_log_tail`, those records are said to be in *truncation epoch* and are subject to truncation. At the same time `log_head` will be advanced to `log_tail`, effectively resetting the *current epoch* to be empty. New log records may be appended by concurrent threads while truncation is in progress, and `log_tail` will advance to new position. (Fig 7.3 ()a)

It is important to use the log pointer to delimit the log into truncation epoch and current epoch as we don't want the log truncation process to interfere with the log flushing process.

If the truncation goes well and all the records are successfully applied to the data segment. The log pointers `prev_log_head` and `prev_log_tail` will be reset to zero. The area that was once occupied by those truncated records are now reclaimed.

However, it is possible that the program crashes while truncation is in progress. RVM will know this upon next restart, by finding the two log pointers `prev_log_head` and `prev_log_tail` are non-zero. It will then redo the truncation. Truncations can always be redone as it is just writing the same value on the same location again.

7.3.2 Log records

There are five different types of log records. Each of them has a corresponding in-core structure defined. They are listed in the table 7.1 ()

Generic record (`rec_hdr_t`) does not actually exist in the log. It contains the common first four fields as the other five 'real' records. When RVM scans the log and find a record, it may not know the type of the record. It can always cast the record as a generic record, and then interpret the first four fields to determine

Generic record	`rec_hdr_t`	20 bytes
Transaction header	`trans_hdr_t`	48 bytes
New value record	`nv_range_t`	56 + length + pad bytes
Record end marker	`rec_end_t`	28 bytes
Wrap marker	`log_wrap_t`	24 bytes
Special log record	`log_special_t`	24 + special + pad bytes

Table 7.1: Six record types and their in-core structure

_records.eps _records.ps _records.eps.gz _records.ps.gz _records.eps.Z

Figure 7.4: Transaction records

the appropiate action. For example, it can know the types of the records by looking at the first field. (The common four fields are `struct_id`, `rec_length`, `timestamp` and `rec_num`)

In the five 'real' records, type specific fields will follow the common four fields. For example, in the following we show the declaration of `rec_hdr_t` and `trans_hdr_t`.

```
typedef struct
    {
    struct_id_t     struct_id;          /* type of entry */
    rvm_length_t    rec_length;         /* record length */
    struct timeval  timestamp;          /* timestamp of record entry */
    long            rec_num;            /* record number of entry */
    }
rec_hdr_t;

typedef struct
    {
    struct_id_t     struct_id;          /* self-identifier */
    rvm_length_t    rec_length;         /* log record length, displacement to
                                           end mark */
    struct timeval  timestamp;          /* timestamp of log entry */
    rvm_length_t    rec_num;            /* record number of log entry */
    rvm_length_t    num_ranges;         /* number of ranges in record */
    struct timeval  uname;              /* uname of transaction */
    struct timeval  commit_stamp;       /* timestamp of commit */
    rvm_length_t    n_coalesced;        /* count of coalesced transactions */
    rvm_length_t    flags;              /* mode and optimization flags */
    }
trans_hdr_t;
```

`trans_hdr_t`, `nv_range_t` and `rec_end_t` represent the *transaction header*, *new value range header* and *record end marker*. They are the majority of records on the log. Each transaction will be written in a sequence of records: it begins with a transaction header, follows by one or more sub-record (the new value range record) and ends with a record end marker (Fig 7.4 ()). This is for the simple case when the transaction are not split by a wrap marker in between. If the transaction is split by a wrap marker, it will have two sequences of records, each begins and ends with a transaction header and end marker. Details of log wrapping will be discussed in section 7.3.3 ().

New value records have variable lengths, each of them begins with a new value range header and follows by the actual data. The actual data is the image of the virtual memory range that the transaction has committed. The header is of type `nv_range_t` and is 56 bytes long, the actual data length varies. Note that RVM may pad some bytes after the actual data so that the next sub-record will be word-aligned.

To allow RVM to scan the log in both forward and backward direction, different forward and backward links are provided. Generally speaking, `rec_length` is a forward link. For transaction header, it is the forward link to the end marker; for new value record, it is the forward link to the next new value record (or record end marker). However, for end record marker, this value means the `backward` link to the transaction header. (RVM is somewhat inconsistent in naming here). On the other hand, `sub_rec_len` is a `backward` link to the previous sub-record. In new value range header, there is the third lenght field: `length`, it is the length _links.eps _links.ps _links.eps.gz _links.ps.gz _links.eps.Z

Figure 7.5: Forward and backward links of records

of the actual data. `length` may not be equal to `rec_length` because of the padding. See Fig 7.5 () for the details of those links.

The fourth possible record type that could be find on log is the wrap marker. Note that wrapping may not happen exactly at the end of the log, it may happen a little bit earlier. In the latter case, the bytes follow the wrap marker will all be written with value `0xff`.

The fifth possible record type is special log record. Currently, the only special log record is segment mapping descriptor (of type `log_seg_t`). RVM allows more than one data segment for one log. To support this, every new value range will be tagged with a `seg_code` which indicates to which segment that range should be applied to. `seg_code` is only a short hand of the segment. Details of the segment (`num_bytes` and `name`) will be recorded in the segment mapping descriptor. Segment mapping descriptors are inserted into the log whenever a new data segment is mapped the first time. At the time of log truncation, RVM will use those segment mapping descriptors to reconstruct the necessary information to open the segment device.

7.3.3 Log wrapping

When `log_tail` approaches the end of the log, RVM will choose an appropiate point to do a *wrapping*. This is done by writting out a wrap marker. After that, RVM will starting appending new records from `log_start`. There will be no risk of overwriting onto useful records because the old records should have been truncated long before. Even if they are not truncated yet, RVM will call a internal function `wait_for_space()` to force a truncation.

Log wrapping can happens *between* transaction. In this case, it simply leave a wrap marker (of type *log_ wrap_ t* and start a brand new transaction at the beginning of log *log_ start* (Fig 7.6 ()a).

It can also happens *within* a transaction. There are two possibilities: between different new value records or *splitting* a new value records. Since a range can be very long, spliting new value records prevents leaving too much space not used by inserting the wrap marker too early.

In the first case, after the current new value records are written, a record-end marker is inserted into the log, and then followed by a log-wrap marker. Another transaction header will be inserted at `log_start` before the remaining new value records are written. Effectively, the same transaction will have two transaction headers: the first is written before log wrapping, the second is written after log wrapping. The `flag` field of the transaction headers helps to indicates this situation. The first transaction header will only have `FIRST_ENTRY_FLAG` set, while the second transaction header will only have the `LAST_ENTRY_FLAG` set. (If the whole sequence of records for the transaction is not interrupted by a log wrapping, both `FIRST_ENTRY_FLAG` and `LAST_ENTRY_FLAG` of the transaction header will be set.) (Fig 7.6 ()b)

For the latter case where a new value range is split, each of the two portion of the data will has its own new value range header. The first portion of the splitted range is written followed by a record-end marker and a wrap marker. The second transaction header are then written at `log_start`, followed by the second portion of the splitted range. The remaining new value records then follows. There is a flag `is_split` in the new value record header indicates that this happens. (Fig 6c) The flag `FIRST_ENTRY_FLAG` and `LAST_ENTRY_FLAG` will also be set as in the previous case.

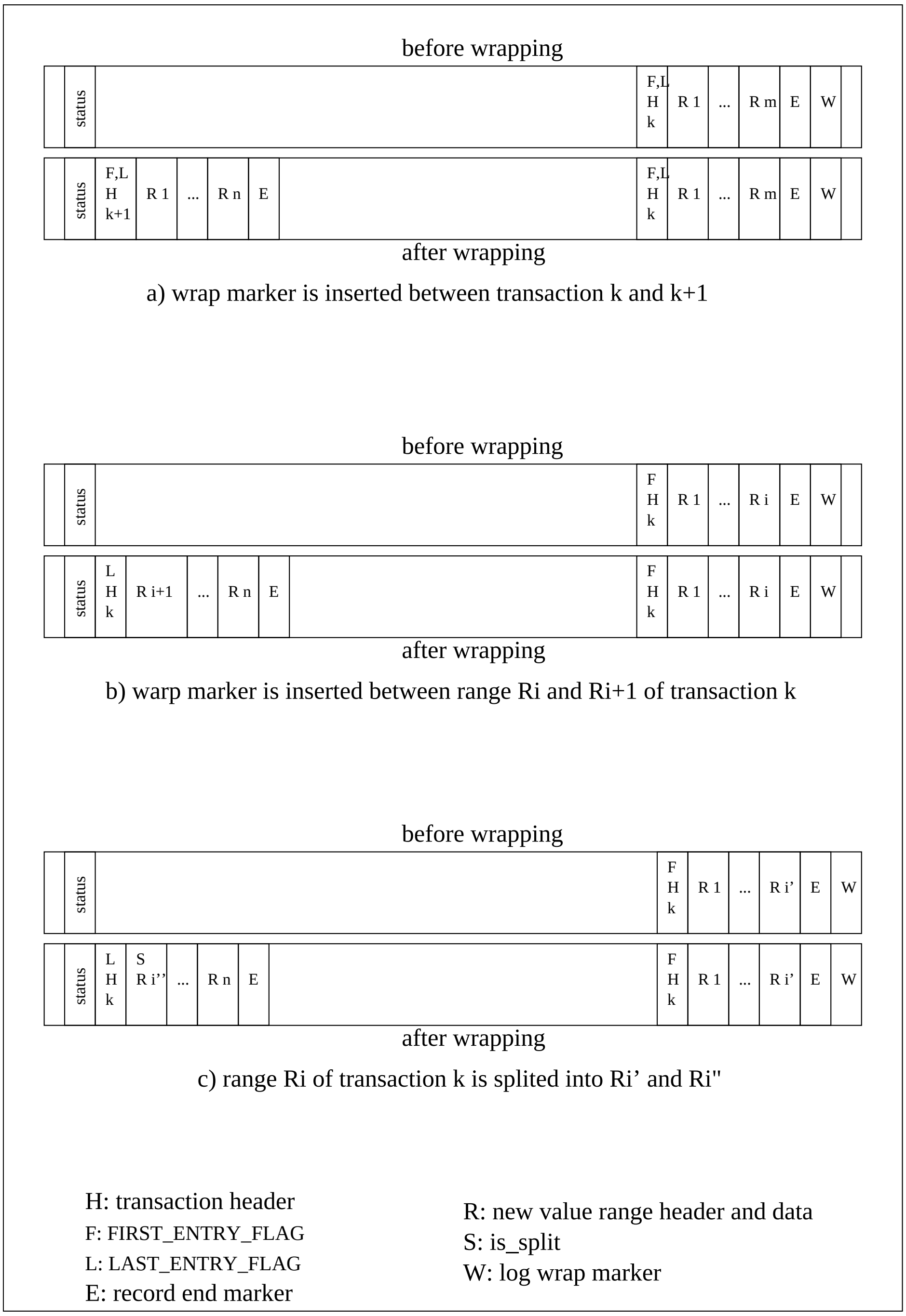

Figure 7.6: Three different possible wrapping

7.4 Internal Data Structure

Before we describe the important data structure used by RVM, we will first discuss some of the building block of these data structure. In next section we will discuss the generic list and generic tree data structure. In the following section we will discuss the synchronization primitive used by RVM that enable multi-threading for RVM.

7.4.1 Generic data structure

A lot of the RVM data structure are organized as lists or trees. To support that, RVM internally defined a generic list type as well as a generic tree type.

Generic list is declared as `list_entry_t`, while generic tree node is declared as `tree_node_t`. Supporting routine for them are `move_list_entry()` and `tree_insert()` etc.

Each of these generic list/tree will contain enough topological information for it to be manuipulated as a list-node or tree node. Generic list and tree are typically not allocated as themselves, rather, they are the first member of other specific data structures that need to be organized. For example, the region descriptor `region_t` is declared with `list_entry_t` as the first member of the structure. That way, the region descriptor can be manuipulated as a list element. We can do the following:

```
seg_t     seg;
region_t  region;

move_list_entry(NULL, &seg->map_list, &region->links);
```

to move a region descriptor `region` to the mapped region list (`map_list`) in the segment descriptor `seg`.

7.4.2 Support for multi-threading

RVM is designed to be thread-safe: application is responsible for protecting access to its data structures by concurrent threads, but can otherwise use RVM function freely. RVM will synchronizing its internal operations. This is done by using the following three primitives:

Mutex

Internal data which may be accessed concurrently are protected by mutex of type `RVM_MUTEX`. Routines accessing those data are required to do a `mutex_lock()` before, and do a `mutex_unlock()` after. Alternatively, they can use the macro `CRITICAL()` to bracket their access to critical data.

```
RVM_MUTEX    lck;

mutex_init(&lck);   /* initialization */
...

mutex_lock(&lck);   /* begin    lck crit. sec */
  /* body */;           /* codes in lck crit. sec */
mutex_unlock(&lck); /* end      lck crit. sec */

                    /* alternatively use macro */
CRITICAL(lck,       /* begin    lck crit. sec */
```

```
  /* body */;       /* codes in lck crit. sec */
);                  /* end      lck crit. sec */
```

Read/Write Lock

Sometimes a data can be read by many threads simultaneously, provided none of them attempts to update on the data. This kind of concurrency is supported by read/write lock of type `rw_lock_t`. Readers access those data with `rw_lock(...,r)`, they will be allowed to proceed immediately if there are no other writer, otherwise, they will be blocked until the writer releases the lock. Writers access those data with `rw_lock(...,w)`, they will be allowed to proceed immediately if there are no other readers or writers, otherwise, they will be blocked until all of them release the lock. Both readers and writers have to do a `rw_un_lock()` when they are done accessing the protected data. There is also a macro `RW_CRITICAL()` for bracketing accesses to critical data protected by read/write lock.

```
rw_lock_t     rwl;

init_rw_lock(&rwl); /* initialization */
...

/* reader */
rw_lock(&rwl,r);    /* begin rwl crit. sec. */
  /* body */;           /* read immediately if no other
  /* body */;           /*  writer, otherwise, block until
  /* body */;           /*  it releases lock */
rw_unlock(&rwl,r);  /* end   rwl crit. sec. */

...
/* writer */
rw_lock(&rwl,w);    /* begin rwl crit. sec. */
  /* body */;           /* write immediately if no other */
  /* body */;           /*  writer/readers otherwise block */
  /* body */;           /*  until they release lock */
rw_unlock(&rwl,w);  /* end   rwl crit. sec. */

...
RW_CRITICAL(rwl, mode, /* begin rwl crit. sec., mode=r|w */
  /* body */;
);                     /* end   rwl crit. sec. */
```

Condition variable

Two concurrent threads can synchronize their actions by using the condition variable of type `RVM_CONDITION`. Thread A can sleep by calling `condition_wait()`, waiting to be waked up by thread B when it calls `condition_signal()`.

This techniques is used for the asynchronous log truncation. A seperate thread will execute the code of log_daemon() if there are multi-thread support (either mach's cthread, or unix's lwp or pthread). Normally this thread is block waiting, other thread(s) will run. From time to time, when transactions are committed, the other thread(s) will check and see if the log is too full, if it is, it will wake up the log_daemon() which will then truncate the log asynchronously.

```
RVM_CONDITION     code;

/* thread of log_daemon() */
  CRITICAL(lock,
    {
    while (state == rvm_idle)
      condition_wait(&code, &lock);
    });
  /* async. log truncation, or terminate
   * log daemon here */

/* thread of log_tid() */
  CRITICAL(lock,
    {
      if (need_truncation) {
        if (state == rvm_idle) {
          state = truncating;
          condition_signal(&code);
        }
      }
    });
```

7.4.3 Important Data Structure

The following structure are all declared in the file `rvm_private.h`. The important data structures are the following.

Segment Descriptor (`seg_t`)

There can be more than one segment per process. Each of the segment has a segment descriptor. These segment descriptor are linked together as a list, and the list header is the global variable `seg_root`.

The segment descriptor contains the header `map_list` for an important list: list of mapped region, which will be discussed in more details in section 7.4.3 ()

There can be more than one segment per log. In fact, each individual new value record can be applied to different segment. This is achieved by tagging a short-name of segment `seg_code` with each new value record. The short name of each segment is also stored in its own descriptor.

Log descriptor (`log_t`)

Like the segment descriptors, log descriptors are also connected as a list (header is a global variable `log_root`. However, currently RVM only support one log in a process, which is pointed to by the global pointer `default_log`.

`log_status` is the in-core copy of the log status block, it will be explained in more detail in the next section. Similarily, `trans_hdr`, `rec_end`, `log_wrap` are the area for preparing the corresponing records on log (c.f. section 7.4.3 ()).

`log_buf` is the log recovery buffer descriptor and it will explained in the section 7.4.3 ().

Log descriptor contains headers for three important list: `tid_list`, `flush_list` and `special_list`. The first link up transaction that have beginned but not yet ended. The second link up transaction that have been committed but not yet flushed.

Log status area (`log_status`)

Log status has a in-core copy as well as in-disk copy. The in-core copy is declared as type `log_status_t`. The in-core copy is stored as a sub-structure of the log descriptor, and the in-disk copy is written on the log status block. Ideally, we would want the in-core copy to written to disk everytime the log status change, however, doing so will require a lot of disk head seeks between the log status block and the log tail, which defeat the purpose of the log, so RVM updates the on-disk log status block only every UPDATE_STATUS (currently the value is 100) of status change. This is keep tracked by the count `update_cnt`. Because of that, RVM will not complete trust on the value in the log status block. For example, when doing log truncation, RVM trusted the value `prev_log_head` but not the value `log_tail`, so it will do a `locate_tail()` to make sure it can find the last records written on the log.

Log status area records important log pointers: `log_head`, `log_tail`, `prev_log_head`, `prev_log_tail` and log limit: `log_start` and `log_size`. Besides, it also records a huge numbers of consistency check fields, transaction statistics and log statistics.

Area for preparing log records

Each of the log record types has a corresponding in-core data structure. They are used to prepare the records before they are written out to log. There are only one copy for `trans_hdr_t`, `rec_ent_t` and `log_wrap_t` per-log and they are part of the log descriptor. For new value records, areas are required for the headers and the actual data and they are per-range. Headers are of type `nv_range_t` and are store in the range descriptors (described in the following section). Actual data are stored in a sperately allocated buffer pointed to by the field `data` in range descriptor. Log special records are associated with the log, and they dynamically allocated and are organized as a list (header: `log_special_list`) in the log descriptor.

Log recovery buffer (`log_buf->buf`) and its descriptor (`log_buf`)

Log recovery buffer (also called log buffer) is used in log truncation. It is used to keep an image of log records for processing. The descriptor of the buffer is called log recovery buffer descriptor (type `log_buf_t`). It is a sub-structure of log descriptor. The actually buffer is pointed to by the field `buf`, the length of the buffer is stored by the field `buf_len`.

Two routines can be used to fill and refill the log recovery buffer:

- `init_buffer(log, offset, direction, ...)`
- `refill_buffer(log, direction, ...)`

The log can be scanned in two direction. This is indicated by the argument `direction` of the two routines. They will try to read as much as possible of data from the log device, subject to the length of the log buffer and other constraints:

Upon returns of these two routines, `log_buf->ptr` will be the index into log buffer that corresponds the current log position, and `log_buf->offset` will correspond to the log offset of byte 0 in log buffer. `log_buf->offset` will be sector aligned. (Figure 7.7 ())

The following is a typical sequence code in log truncation:

_buf.eps _buf.ps _buf.eps.gz _buf.ps.gz _buf.eps.Z

Figure 7.7: log buffer and init_buffer()

```
...
r_length      = init_buffer(log, off, REVERSE, ...);
log_buf->ptr -= sizeof(rec_end_t);
rec_end       = (rec_end_t *) &log_buf->buf[log_buf->ptr];
if (rec_end->struct_id == rec_end_id) {
  ...
```

In the code, `off` is the current log position. Routine `init_buffer()` is instructed to initialize the buffer by scanning the log in the reverse direction. When it returns, `r_length` number of bytes are read into the log recovery buffer (pointed to by `log_buf->buf`), `log_buf->ptr` is index of the byte in log buffer that corresponds to `off`. Assume the routine here is expecting a end record in log, it decrements `log_buf->ptr` by the known size of `rec_end_t` so as to point to the end record. After the decrement, the address of `log_buf->buf[log_buf->ptr]` should be the beginning of the in core image of the end record. It type-cast it as `(rec_end_t *)` and stores it in `rec_end`. After that it can manipulate the record as it likes. In the example, it examines and see if `struct_id` of the end record is really a `rec_end_id` ...

Region descriptors (`region_t`)

Each segment will have a list of regions that are mapped. So in each segment descriptor (type `seg_t`) there is a member `map_list` which is the head of this list. Each node in the list is a region descriptr of type `region_t`.

It records the `offset` and `end_offset` of the region in the data segment, as well as the `vmaddr` and `length` of region that is mapped to.

The member `n_uncommit` records the number of uncommitted range in this region, it is incremented everytime application does a `rvm_set_range()`, it will be decrement when a transaction is committed or aborted. When application tries to do `rvm_unmap()`, this value must be zero otherwise it will return RVM_EUNCOMMIT.

`region_t` may be confused with `rvm_region_t` and `mem_region_t`. We will explain the former in the following and the latter in the next section. `rvm_region_t` is part of the API, it is the structure used by the application to indicate the mapping parameters: `data_dev`, `offset`, `length`, `vmaddr` and `no_copy` etc. RVM does not use this structure internally, once it has got all the parameters from the application and has transferred them to `region_t`.

Memory region descriptors (`mem_region_t`)

In addition to the per segment region list described above, there is also a per process memory region tree. It is used to record the regions in the virtual memory space that are mapped, and it is arranged in the order of vm address.

When application does a `rvm_map()`, this tree is consulted to make sure the requested virtual memory region is not already mapped. If it is already mapped, RVM_EVM_OVERLAP will be returned. Also, when application does a `rvm_set_range()`, this tree is consulted to make sure the requested range is totally contained within a single virtual memory region. If it isn't, RVM_ENOT_MAPPED will be returned.

The root of the tree is `region_tree`. Each node of the tree is of type `mem_region_t`.

Transaction descriptor (int_tid_t)

Whenever the application does a rvm_begin_transaction(), rvm will allocate a transaction descriptor of type int_tid_t. These descriptors are linked up into list. Before the transaction commits, the descriptor is in the list headed by log->tid_list. If the transaction is committed, it will be moved to another list headed by log->flush_list. If the transaction record is flushed, it will be removed from the flush list.

Typically, the application may inform RVM one or more memory range which will be affected by the transaction. This is done by issuing one or more rvm_set_range() or rvm_modify_bytes() after rvm_begin_transaction(). Internally, these memory ranges are organized as a tree rooted at range_tree (of type tree_root_t). The tree root is a field of the transaction descriptor.

int_tid_t may be confused with rvm_tid_t. The latter is the transaction identifier used by applications while the former is used internally by RVM. Althought rvm_tid_t contains a pointer to int_tid_t, applications are not expected to use or touch the pointer.

modification range descriptor (range_t)

When application specifies modification range by rvm_set_range() or rvm_modify_bytes(), the vm address (range->nv.vmaddr) and the length (data_len) of the range is stored internally in the range descriptor (range_t). When the transaction commits, the content of virtual memory pointed to by (range->nv.vmaddr) will be copied to the buffer pointed to by range->data. The same buffer has another purpose: if the transaction is in restore mode, the buffer is used also for saving the old image of VM. In case the transaction is aborted, the old image can be restored.

Range descriptors are connected together as a tree, with the tree root in the corresponding transaction descriptor. If the optimization flag RVM_COALESCE_RANGES is specified, the tree is sorted according to the vm address, and ranges are coalesced before they are inserted into the tree. If RVM_COALESCE_RANGES is not specified, the tree is simply sorted by the range number.

There is also an area of type nv_range_t called nv which is used as the in-memory image of the nv range record on log.

storage device region node (dev_region_t)

This is not to be confused with region_t and mem_region_t. dev_region_t is used only in log truncation. During log trunctation, all sub-records of all transactions in the truncation epoch are processed. A tree called mod_tree is built. Each node of the tree is of type dev_region_t. The tree is sorted according to the segment offset. The purpose of the tree is to make the disk write operation on the segment device sequential.

Device descriptor (device_t)

Device descriptor (of type device_t) contains the details of the phyical devices: name, handle, size. Also, it contains the pointers to the gather-write I/O vectors.

7.5 RVM in action

7.5.1 rvm_initialize()

Global lists and trees are initialized (seg_root, log_root, region_tree). A log truncation (log_recover()) is performed.

7.5.2 `rvm_map()`

There is always a log truncation at the begin of mapping. This is necessary to ensure the mapping will get the latest committed image from the data segment. It also checks if application has specified a new segment device, if so, it will create a special record of type `log_seg_t` and queue it on `log->special_list`. This record will be flushed to the log before any next transaction.

Application may or may not specified the mapped-to address for the region. If it is not specified, `rvm_map()` will do a `page_alloc()` to find a VM address for the request region. Either case, the mapped-to address will be stored in `region->vmaddr` as well as `mem_region->vmaddr`.

Before carry out the mapping, `rvm_map()` also need to check that the requested mapping is not overlapped with existing mapping in both the VM address space as well as the segment offset space. The former is detected with the aid of `region_tree` while the latter is detected with the aid of `seg->map_list`. If there are overlapping, RVM_EVM_OVERLAP will be returned in the former case and RVM_EOVERLAP will be returned in the latter case.

Finally, the content of the data segment at the specified offset is copied into the VM space (unless `no_copy` mode is specified).

7.5.3 `rvm_begin_transaction()`

A transaction descriptor (of type `int_tid_t`) will be allocated and inserted in `log->tid_list`. The address of that descriptor is also noted in `rvm_tid->tid`.

7.5.4 `rvm_set_range()` and `rvm_modify_bytes()`

The range declared must be totally included inside a mapped region, other RVM_ENOT_MAPPED will be returned. This is done by consulting `region_tree`.

A new range descriptor (of type `range_t` will be allocated and inserted into `tid->range_tree`. Depending on the optimization flag of the transaction, the range tree may be sorted by vm address or sorted by range numbers.

If transaction is in `restore` mode, the current image of VM in the range will be first saved to a buffer pointed to by `range->data`.

VM address of the range declared is stored in `range->nv.vmaddr`.

7.5.5 `rvm_end_transaction()`

When application calls `rvm_end_transaction()`, for each of the ranges in `tid_range_tree`, the vm area pointed to by `range->nv.vmaddr` is copied to `range->data`. This is to ensure the atomicity of transaction: when `rvm_end_transaction()` returns, all declared ranges have been copied to a seperate location. If the program crashes before `rvm_end_transaction()` returns, then the data segment will contain the old value of the of the declared ranged. Applications are free to write new data on the ranges after `rvm_end_transaction()` returns.

The transaction descriptor `tid` is dequeued from `log->tid_list` and is queued to `log->flush_list`.

If RVM_COALESCE_TRANS flag is set for the transaction, an inter-transaction coalescing will be carried out.

If the transaction is committed in `flush` mode, `flush_log()` is called.

7.5.6 flush_log()

Every transaction descriptors in log->flush_list are written out to the log, this is done by the routine log_tid(). First, the transaction header of type trans_hdr is built and added to the gather write vector dev->iov. Then, for every ranges in tid->range_tree, the new value range header of type nv_range_t is built and added to the gather write vector, followed by the actual new value data. After all the new value records are written, a record end marker (of type rec_end_t is built and added to the gather write vector.

If log wrapping is need, there will be intervening record end marker, log wrap marker and transaction header record inserted, as described in section 7.3.3 ().

7.5.7 rvm_truncate()

Log truncation is carried out by log_recover(). For each segment, a modification tree mod_tree is built. Each node is of type dev_region_t, it is arranged in order of segment offset. For each new value records of each transactions on the log, a node of type dev_region_t is built and inserted into the modification tree. If there are overlaps in segment offset , the overlapped portion is not repeatedly inserted into the modification tree. In this case, it is not an error to have overlapped segment, for example, the same memory range can be updated by different transaction. Also note that since the log is truncated in last-in-first-out order, i.e. new value records that was flushed last is inserted into tree first, we always have the most up-to-date range inserted into the tree.

When all the log records bracket by the log pointers prev_log_head and prev_log_tail are scanned and inserted into the modification tree. The whole tree is written to data device (apply_mods()).

7.6 RVMUTL

The program rvmutl is a utility program that allows:

- initialization of a log
- debug via the log

The following is an incomplete list of commands available in rvmutl:

- init_log: initialize a log
- open_log: open an exising log
- status : display information on log status block
- head : locate and display the first record of current epoch
- tail : locate the display the last record of current epoch
- prev (n): locate and display previous (n) record(s)
- next (n): locate and display next (n) reocrds(s)

In the rvm source codes, you can often see the global variable rvm_utlsw. RVM is designed in mind that there are two possible types of users for rvm's routines: ordinary applications or the program rvmutl. This variable allows a differential treatment on exception condition. For example, in the routine scan_forward(), if the record type is found not to be one that the routine is expecting (possibly means a corrupted log) the program will:

- abort, if it is called by an general application
- continue with error return, if it is called by `rvmutl`

The latter treatment allow manual handling of log corruption via `rvmutl`.

Chapter 8

rvmutl, the RVM Maintenance Utility

rvmutl is the maintenance utility for RVM. Its principle features are log creation and maintenance, but there are also status and statistic display commands as well as printing commands for log records. A search facility for modifications to segments gains additional functionality from the log by allowing its use as a debugging history.

rvmutl is included in the `/bin` directory of the RVM distribution directory. It can be invoked with a single, optional parameter, which is interpreted as the name of a log file to be opened. If the file doesnt exist, rvmutl will ask if you wish to create and initialize a log with the specified name.

rvmutl prompts for commands with a * prompt, after which commands can be typed. Most commands have one or more parameters, which are separated by one or more spaces. In general, command names can be abbreviated, provided that sufficient characters are typed to uniquely specify a command. To allow typing short-cuts, some non-unique abbreviations have defined interpretations. These are specified with the command documentation

As an example, the following sequence will initialize the file log_file to have one megabyte of log record space:

```
% rvmutl
* init_log log_file 1M
* quit
```

After this simple sequence, the log file is ready for use.

The following pages describe the commands available in rvmutl.

NAME

rvmutl – maintenance utility for RVM log

SYNOPSIS

rvmutl *log*

rvmutl

DESCRIPTION

The program `rvmutl` is the maintenance utility for RVM log. Its priniciple features are log creation and maintenance, but there are also status and statistic display commands, as well as printing commands for log records. There is also a search facility for modifications to segments. You may find rvmutl a great debugging tools.

The are two ways to use `rvmutl`: You may supply the name of the RVM log on the command line, or you do not supply the name on command line but specify the log within `rvmutl` (see below).

`rvmutl` prompts for commands with a * (asterisk) prompt. Most commands have one or more parameters, which are separated by one or more spaces. Some commands have abbreviations, these will be specified in the command section.

For example, to examine the status of a RVM log (called logfile in the example), you can do

```
> rvmutl
* open_log logfile
* status

Status of log:           /home/clement/linuxobj8/rvm-src/tests/logfile

  log created on:        Tue Nov  4 1997 17:31:10.244793
  log created with:      RVM Interface Version 1.3  7 Mar 1994
  (more display omitted ...)
* quit
```

As another example, suppose you want to create a RVM log device on the partition /dev/hdc7, of size 1 Megabyte, you can do

```
> rvmutl
* i /dev/hdc7 1M
* quit
```

Note that in the preceding example, we use the abbreviation "i" for the command "init_log", and we can use the unit "M" to stand for Megabyte.

I/O REDIRECTION

You can redirect input and output as in typical Unix shells. For example, the following command will have all display redirected to the file allrec:

```
* show all > allrec
*
```

COMMANDS

In the following, I will detail all the available commands:

all_spec

Show all special log records. Special log records are those non-transaction records, such as wrap-around marker and segment dictionary entry.

build_seg_dict [all [**clear**]]

seg_dict [all [**clear**]]

Build the segment dictionary by scanning through the log. Segment dictionary is a in-memory structure that maps segment code to segment file. Segment code is used in each transaction range as a short hand of the segment file involved. If the **all** option is specified, the whole log will be scanned, otherwise the scanning will stop when one segment dictionary entry is found. If the **clear** option is specified, the segment dictionary will be cleared before starting the scanning process.

close_log

Close the log file.

copy_log

This provide a handy way to copy an existing log to another file or partition. Since rvm logs start on different offset, depending on whether they are files or partitions, using stock Unix commands such as cp or dd sometimes is not as nice as using this command. Some meta data (such as name of log) in the log status area in the destination log will also be updated to reflect new meta data after log copying.

earliest

Show the earliest transaction record in the log.

init_log [*log* [*length*]]

Initialize a log if it does not exist, or re-initialize it if it does exist. The log will have the name *log* and the length of data area will be *length*. The actual size of the log file or partition will be slightly bigger because: 1. there is a log status area (of size 3 sectors); and 2. for log used on raw partition, there will be an offset (of 16 sectors) to jump off possible disklabel of the partition.

The *length* parameter is specified as an integer, optionally followed by a single character size code. The following size codes are defined: **K** or **k** for kilobytes, **M** or **m** for megabytes, **S** or **s** for sectors (512 bytes), **P** or **p** for pages (which have system dependent sizes) and **B** or **b** for bytes (default).

If either parameter is not specified, **init_log** will prompt for the missing information.

RVM maintains the log continuously after creation. If you are going to re-initialize an existing log, make sure the log is empty before the re-initialization, otherwise, the transaction records in the log would be lost (i.e., they would not be applied to the data segment). You can apply all transaction records to the data segments and empty the log (also known as truncate in this manual) by the **recover** command. You can use the command **status** to check whether a log is empty.

find_earliest

Print the offset of the earliest transaction record in the log.

find_tail

Print the offset of the tail of the log.

head

Print the transaction record at the head of log. (Note that there could be even earlier than the head record – smaller record number and smaller time stamp. They are considered truncated from the log and can be overwritten upon them freely. They are still on the log just because it happens that rvm has not yet overwritten on them, and they may be useful for debugging. The command **earliest** prints the earliest record.)

log [*log* [no_tail] [no_update | update]]

open_log [*log* [no_tail] [no_update | update]]

Open the rvm log named *log*. You must open a log before you can use most of the rest commands. The optional **no_tail** switch turns of the search of tail of the log (for example, when the log tail is corrupted). The optional **no_update** and **update** switch controls whether update will be made on the log and segment (they turn on the rvm internal global variable `rvm_no_update`).

n [*num*]

next [*num*]

Print the next *num* records. If the optional argument *num* is not specified, then the next one record will be printed.

ns [*num*]

next_sub_rec [*num*]

Print the next *num* sub-records. Sub-records are the individual ranges of a transaction. If the optional argument *num* is not specified, then the next one sub-record will be printed.

p

peek [*file* *offset* / [*csf*] Peek on (or dump) the content of *file*. You must specify the argument *file* when you first peek or poke on a file, afterward you can omit the argument and rvmutl will continue using the current file. The argument *offset* determines from where the dump should start. The argument *csf* determines how the dump should be done, it comprises of three components: c, s and f; each of them represents the count, size, and format respectively. c is an integer and determines how many units, with size specified by s, should be dumped, and f specifies what is the dump format.]

In the following two examples, the first command dumps, from offset 0, eight bytes in hexadecimal format; the second command dumps two long integers in decimal format. Note that how the same file content can be dumped into different format.

```
* peek 0 / 8xb
00000000000: 0x18 0000 0000 0000 0x23 0x17 0x14 0x63
* peek 0 / 2dl
00000000000:          24   1662261027
```

There are seven different possible sizes: **b** for byte, **l** for long, **s** for short, **O** for offset (two long integers representing the high and low order bits respectively), **f** for float, **D** for double and **t** for time value.

There are nine different possible formats: **c** for character, **d** for decimal, **o** for octal, **x** for hexidecimal, **u** for unsigned, **f** for float, **D** for double, **O** for offset (two long integers representing the high and low order bits respectively) and **t** for time value.

You may omit the arguments c, s and f, and rvmutl will use the defaults, which are 1, d (decimal) and l (long) respectively, for the three arguments respectively.

poke [*file* *offset* ÷ *cs* = *val* [*val* ...]]

Poke on (modify) the content of *file*. You must specify the argument *file* when you first peek or poke on a file, afterward you can omit the argument and rvmutl will continue using the current file. The argument *offset* determines from where the modification should start. The argument *cs* determines how the modifications should be done, it comprises of two components: *c* and *s*; each of them represents the count and size respectively. *c* is an integer and determines how many units, as specified by *s*, should be modified.

Examples:

```
* poke 512 / 4db = 25
* peek 512 / 8db
      512:  25  25  25  25   0   0   0   0

* poke 512 / 2xl = 25
* peek 512 / 8db
      512:  25   0   0   0  25   0   0   0

* poke 512 / 2b = 0x26 0x27
* peek 512 / 8db
      512:  38  39  38  39  25   0   0   0

* poke 512 / b = "abcde"
* peek 512 / 8db
      512:  97  98  99 100 101   0   0   0
```

Note that in the second example, the character **x**, which indicates hexidecimal format in peek, is sliently ignored. To enter a hexidecimal number, you should us prefix '0x' as in the third example. Note also that how a pattern is repeated by using the argument *c* in first and third example.

There are seven different possible sizes: **b** for byte, **l** for long, **s** for short, **O** for offset (two long integers representing the high and low order bits respectively), **f** for float, **D** for double and **t** for time value.

You can also enter string by using the double-quote mark ("), as in the last example.

pr [*num*]

prev [*num*]

Print the previous *num* records. If the optional argument*num* is not specified, then the previous one record will be printed.

ps [*num*]

prev_sub_rec [*num*]

Print the previous *num* sub-records. Sub-records are the individual ranges of a transaction. If the optional argument *num* is not specified, then the previous one sub-record will be printed.

quit

Quit rvmutl.

read_status

Re-read the log status from on-disk log device (file or partition) to in-memory structure (`status`). Note that the command **status** show the content of the in-memory structure, but not the on-disk device. If you have made changes to the on-disk device, you may want to re-read them into the memory.

monitor [clear [no_update | update]]

You can specify memory ranges to be monitored by this command. When a memory range is monitored, there will be message printed out whenever there are modification within the range (E.g. in **recover**). The switch **clear** will clear all the monitor previous set. The switch **no_update** and **update** carry the usual meaning (c.f. **open_log**). You will be asked, interactively, for the addresses and formats for the ranges you want to monitor, you should use the syntax similar to **poke**. The following example shows how to use **monitor**, **show monitor**, as well as how **recover** prints out the modification.

```
* open_log log3
* monitor update
  More ranges (y or n [n])? y

  Enter list of addresses/format, terminate with null line
:  0x200d7550/1l
:
* show monitor

Range    Address/format     Length
   1    0x200d7550/1dl         4

* recover
do_nv: data from log record 2101, range 52
  monitored range 1, 0x200d7550, length 4 matched by
  modified range 0x200d7550, length 4
  0x200d7550:    537286660

change_tree_insert: inserting entire range
  monitored range 1, 0x200d7550, length 4 matched by
  modified range 0x200d7550, length 4
  0x200d7550:    537286660


do_nv: data from log record 2101, range 51
  monitored range 1, 0x200d7550, length 4 matched by
  modified range 0x200ced40, length 8367
  0x200d7550:            0

change_tree_insert: inserting entire range
  monitored range 1, 0x200d7550, length 4 matched by
  modified range 0x200ced40, length 8367
  0x200d7550:            0

do_nv: data from log record 2101, range 50
  monitored range 1, 0x200d7550, length 4 matched by
  modified range 0x200cdcc4, length 16
  0x200d7550:            0

change_tree_insert: inserting entire range
  monitored range 1, 0x200d7550, length 4 matched by
  modified range 0x200cdcc4, length 16
```

recover [clear [**file**] [**no_update** | **update**]]

Truncate the log. Truncation means applying the transaction records, that were logged on the log, to the data segment, and then reclaiming the space on the log used by those records.

set [seg_dict *field* | *addr* = *val*]

Set *field* (or memeory locateion at *addr*) to *val*. There are five fields that you can set: **head**, **tail**, **prev_head**, **prev_tail** and **log_start**. (You can use **prev head** and **prev tail** to represent **prev_head** and **prev_tail** respectively.) Only the in-memory structure of the field is changed by this command, you can use **write_status** to write the in-memory structure (status area) to disk.

s

show

all all records

all_records all records

earliest earliest record

head record at head of log

mods *of* ÷ *cs* [= *val*]

modificationss *of* ÷ *cs* [= *val*]

E.g.

```
* show mods 0x2000005c / 8b

Record number: 40756 modifies specified range:

  Modification range:        1   Log offset:    224664
    VM address:     0x2000005c   Length:        52
    Segment code:            1   Offset:      4188
    Record length:         108   Back link:     48

  0x2000005c:  74 -107   0   0   0   0   0   0

* show mods 0x2000005c / 1b = 74

Record number: 40756 assigns specified values

  Modification range:        1   Log offset:    224664
    VM address:     0x2000005c   Length:        52
    Segment code:            1   Offset:      4188
    Record length:         108   Back link:     48

  0x2000005c:  74

* show mods 0x2000005c / 1b = 75
  -- no more records
```

monitor all the range being monitored.

next next record

next_sub_rec next subrecord

ns next subrecord

***num* by record number**

prev previous record

previous previous record

ps previous subrecord

prev_sub_rec previous subrecord

rec_number *num* by record number

remaining all remaining records

seg_dict segment dictionary

seg_dictionary segment dictionary

statistics statistics of rvm activities

status status area of the log

log_status status area of the log

sr current subrecord

sub_rec currect subrecord

tail tail record

timestamp by record timestamp (note: no effect)

sizeof *struct* | all

Show the size of a the give structure *struct*, or size of all structure. The unit is byte. The following structure can be shown: **condition**, **device_t** (dev), **dev_region_t**, **FLUSH_BUF_LEN**, **free_page_t**, **MAXPATHLEN**, **int**, **list_entry_t**, **log_t** (log), **log_buf_t**, **LOG_DEV_STATUS_SIZE**, **log_dev_status_t**, **log_seg_t**, **LOG_SPECIAL_IOV_MAX**, **LOG_SPECIAL_SIZE**, **log_special_t**, **log_status_t** (status), **log_wrap_t**, **long**, **MAX_READ_LEN**, **mem_region_t**, **MIN_NV_RANGE_SIZE**, **MIN_FLUSH_BUF_LEN**, **MIN_RECOVERY_BUF_LEN**, **MIN_TRANS_SIZE**, **mutex**, **NUM_CACHE_TYPES**, **NV_LOCAL_MAX**, **nv_range_t** (nv), **NV_RANGE_OVERHEAD**, **page_size**, **range_t**, **rec_end_t**, **rec_hdr_t**, **RECOVERY_BUF_LEN**, **region_t**, **rvm_length_t**, **rvm_offset_t**, **rvm_options_t**, **rvm_region_t**, **rvm_tid_t**, **rw_lock_t** (rw_lock), **rw_lock_mode_t**, **seg_t**, **struct_id_t**, **int_tid_t**, **timeval**, **trans_hdr_t**, **TRANS_SIZE**, **tree_links_t**, **tree_node_t**, **TRUNCATE**, **ulong**, **unsigned**.

status

log_status

Show the log status area, which is the meta data about the log (include head offset, tail offset, space used by records, total log size, first record number, last record number, log creation time, last truncation time etc).

sr

sub_rec

Show the current subrecord.

statistics

Show the statistics of rvm activities.

tail

Show the transaction record at the tail of log.

update

Turn on update (i.e. set to false the rvm-internal global variable `rvm_no_update`.) Update will not be made on the log and segment.

no_update

Turn off update (i.e. set to true the rvm-internal global variable `rvm_no_update`.) Update will be made on the log and segment.

write_status

Write out the in-memory log status structure to the log status block on disk.

BUGS

The command **find_hole** is not yet documented.

Peeking using the format or size of time value (**t**) does not work.

The way to poke an offset value is not yet documented.

sizeof of some structs (e.g. **condition**, **int**) wrongly displays the page size rather than the size of the structure.

The use of **no_update** does not quite work. Rvmutl easily crashes when the switch is used.

The exact semantic of **replay** command is not documented, and the command itself crashes quite easily.

The use of **seg_dict** in **set** is not yet documented.

The command **show timestamp** has no effect.

AUTHOR

Yui Wah LEE completely rewrote this man page (Nov. 1997)

Appendix A

RVM Import and Installation Instructions

This appendix contains instructions to import and install RVM. The process is set up to be a compatible as possible with both Mach 2.6 and Unix BSD 4.3. Support for other versions of Unix is limited, but in general, RVM attempts to keep its system requirements to a minimum to enhance portability. Where necessary for specific versions of Unix and related systems, special instructions are included below.

Please note that RVM is copyrighted by Carnegie Mellon University, and the terms of distribution are specified by the following copyright notice:

```
changes without encumbrance.  Such improvements and extensions
should be returned to Software.Distribution@cs.cmu.edu.
```

A.1 Differences from Previous Versions

A.1.1 Current Release (RVM 1.3, 29 June 1994)

This release corrects some error reporting bugs in `rvmutl` and the segment loader. No reinitialization of log files is necessary. All that is required to use this version is to recompile and relink your applications after importing and installing RVM.

A.1.2 Release (RVM 1.3, 25 April 1994

This release corrects a few small bugs and synchronizes the export and Coda versions. Re-initialization of your log files is recommended.

To reinitialize log files, for each existing log file (or partition), use your present version of `rvmutl` to force any remaining records to be reflected in the modified segments, and then reinitialize the log with the new version of `rvmutl`:

```
.../RVM1.2/bin/rvmutl        # RVM 1.2
* open_log <log file name>
* recover
* quit

.../RVM1.3/bin/rvmutl        # RVM 1.3
* init_log <log file name> <size>
* quit
```

A.1.3 Release RVM 1.3, 4 April 1994

This release contains a more complete verion of the manual and corrects a bug in `rvm_basher` that caused reports of virtual memory not matching the disk. The problem was in the test program itself, not the RVM library.

Other changes are addition of error checking in the segment loader and RDS initialization functions. Also, the utility `rdsinit` has been made somewhat more forgiving. You can now enter lengths in decimal as well as octal and hexadecimal. The values input are checked for minimal agreement with the requirements of RDS and a size check on the file being created is made to insure that the regions declared will fit in the file.

No recompilation or log re-initialization is necessary if you are upgrading from the 14 March 1994 release. Those who are using the pre-public release, RVM 1.2, should follow the instructions below.

A.1.4 Earlier Release (RVM 1.3, 14 March 1994)

This release was the first public release, but for those presently using the informal release, RVM 1.2, recompilation of modules using RVM and re-initialization of the log is necessary. To reinitialize the log, use the procedure specified above.

A.2 Importing RVM

To import RVM, use FTP to get the file RVM.TAR into the directory from which you wish to build RVM. Use the CMU Computer Science FTP server:

```
cd <your directory>
ftp FTP.CS.CMU.EDU
Name: anonymous
password: <userid>@host
cd project/rvm-releases
get README              # this text
get rvm_manual.ps       # RVM manual (PostScript)
binary                  # required for tar files
get RVM1.3.TAR          # packed source files
quit
```

You should also send mail to rvm@cs.cmu.edu to be placed on a mailing list for future updates.

Then unpack the files:

```
tar xf RVM1.3.TAR
```

At this point, you can delete the RVM1.3.TAR file to recover disc space.

A.3 Machine/OS Dependent Fixes

If you are building for any of the following machine and/or types, you must make the following modifications before continuing the installation. If your machine/OS is not mentioned, no special action is needed.

A.3.1 OSF/1

Delete the following files:

```
.../RVM1.3/Makeconf
.../RVM1.3/rvm/Makeconf
.../RVM1.3/rds/Makeconf
.../RVM1.3/seg/Makeconf
.../RVM1.3/tests/Makeconf
.../RVM1.3/bench-SOSP93/Makeconf
```

A.3.2 HP 9000 "snakes"

make doesnt support the include directive, so you will have to modify the Makefiles. In the following files, replace the include command after the "get basic definitions" with the definitions and macros in the build_defs file which will be found in the RVM directory created by unpacking the tar file:

```
.../RVM1.3/rvm/Makefile
.../RVM1.3/rds/Makefile
.../RVM1.3/seg/Makefile
.../RVM1.3/tests/Makefile
.../RVM1.3/bench-SOSP93/Makefile
```

A.4 Installing RVM

Execute the following commands:

```
cd RVM1.3
make initialize
```

The above sequences will create the necessary directory structure rooted as `RVM1.3` in the directory where the tar file was unpacked, and compile the system. When complete, you will have the following directories under RVM1.3:

lib

repository for the RVM, RDS and SEG libraries.

bin

repository for RVM-related programs, rvmutl, rdsinit, and the test programs.

include

repository for header files for RVM subsystems.

rvm

source code for RVM.

rds

source code for RDS, a dynamic allocator.

seg

source code for SEG, a loader for RVM segments.

tests

source code and test files for RVM.

bench-SOSP93

source code for the benchmark used in the SOSP-93 paper [**?**].

OBJS

(Mach only) directory structure for machine-specific build areas, normally cleared after the initial build process.

These directories and files are fully built (for Mach) in the RVM1.3 directory on the FTP server, so you can see what you should get if there is trouble.

The lib, bin, and include directories are the interface for application build procedures. References to them will get the correct headers, libraries, and utility programs. They are particularly important under Mach since lib and bin are symbolic links to machine dependent files. Under Unix, they are directories with the actual files; no support is provided for compilation for multiple machine types. Also, the build procedure under Unix leaves all derived files in the same directories as the sources since the OBJS directory structure is not built. The structure used by the Mach build process could also be used for Unix by modifying the Makefiles to use the machine specifier used on your system to replace the @sys construct used in Mach.

A.4.1 The Makefiles

The Makefiles included are intended to both install the system and help with the building of customized versions. The process can be controlled from two levels: the top level, at the RVM1.3 directory, for builds of all the subsystems. The second level is builds of a specific subsystem from its source directory. From the top level, the following build options are available:

make initialize

creates directory structures for derived files and then compiles all libraries, utility and test programs. This should be done only once.

make install

does a complete recompilation and installation. If the necessary directories are not present, they will be built. All existing derived files are removed before the build so that a complete rebuild is guaranteed.

make release

builds all subsystems from existing derived files, and copies the interface files to the appropriate directories. Intended for a release after work has been done on one or more subsystems.

make all

builds all subsystems from existing derived files, but does not copy files to the interface directories.

make clean

removes all derived files from the subsystem directories and OBJS, if it is built. The `lib` and `bin` directories are not affected. This is recommended to recover disk space after the initial build of RVM if you do not plan on making modifications to RVM.

make {rvm, rds, seg, tests, bench-SOSP93}

builds the specified subsystem.

From the subsystem level, the install, reinstall, all, and clean options are also available, and do their respected operation for only that subsystem. Further options are also available to create specific libraries, utilities, and test programs. These can be determined by examining the subsystem Makefiles.

The version of Make used in Mach 2.6 will automatically include a file called Makeconf if it exists. This is used for common configuration macros. Since Unix Make doesnt support this, but does have an include facility, most configuration macros are actually put in the file RVM1.3/build_defs, which is included are necessary. The Makeconf files contain Mach specific things only. To allow the same Makefiles to be used for both operating systems, most directories are referred to by macro names which will evaluate appropriately for the system. This is done using only the most basic Make features in the hope that the build will be as portable as possible. One artifact of this method is that the installation will produce a number of error messages followed by messages of the form `"*** Exit 1 (ignored)"` as the inappropriate commands fail. Just note that this is expected behavior.

There are a number of other build options that are only used for the Coda File System. These should be ignored since they require many files not included in the RVM distribution.

A.4.2 Threads

Under Mach, RVM is internally multi-threaded and expects to be linked with the Cthreads package. Either the Mach threads or coroutine libraries can be used. Coroutines are recommended while debugging since program behavior can be repeatable.

Under Unix, RVM is at this time single threaded. To allow RVM to compile, symbolic links are created for the header file cthreads.h to the file dummy_cthreads.h. No thread library will be required at link time. If you have a thread package that you would like to try to use, the file rvm_lwp.h can be used as a model of how to create a header file for another thread package. This not recommended for the faint of heart!

A.5 Testing RVM

This distribution of RVM includes two test programs: testrvm, a basic "sanity checker", and rvm_basher, a stress test. You should run these after installing RVM. Running the simple test involves two steps: creating a log file with rvmutl, and then running the test. Its most easily done from the RVM1.3/tests directory, but if you copy the map_*_files and the t*_chk_files to any other place, you can run it there. The procedure is:

```
.../RVM1.3/bin/rvmutl
* init_log log_file 10k
* quit

.../RVM1.3/bin/testrvm
```

It will then ask three questions, all of which you should simply enter the default "No" by hitting return. It will do some basic transactions and compare virtual memory with the map_chk_file for correct operation. It will report correct operation. Any other termination is an indication of trouble.

rvm_basher is a much more thorough test and should be run for several hours (we have fixed bugs that would only show up after running it for many days on Pentium 90 computers!). The program requires that a data segment be built as well as a log file. The tests will randomly allocate, modify, and deallocate recoverable memory, periodically stopping to check virtual memory against the data segment. Any discrepancy will be reported and the program will stop. Otherwise it will run until you stop it, given the parameters supplied in this distribution.

rvm_basher can also be used a a sample RVM application since it uses most of the RVM library and also the dynamic allocator, RDS, and segment loader. The setup procedure for running it is similar to what most applications will require, although their heap and static sizes wiil most likely differ.

First, cd to a directory in which you can build about 2M of files. Create a log file using rvmutl as in the the previous test:

```
.../RVM1.3/bin/rvmutl
* init_log basher_log 600k
* sizeof page
* quit
```

To prepare a data segment, first load ...RVM1.3/bin/rvm_basher and run the following commands:

```
.../RVM1.3/bin/rvm_basher
* log basher_log
* show_break
* run
```

The program will load and print the current break point (with sufficient space added to compensate for internal requirements). This will become the load point for the data segment. Application programs will also have to determine their break point after initialization. If you plan to use pointers in recoverable storage, be sure to add enough to the breakpoint to allow growth of your application. Otherwise you will have to modify the pointers or re-initialize recoverable storage to allow the break point to be moved.

Next, create a data segment with an RDS heap using the rdsinit utility. The static length must be the size of a page on your machine as indicated by rvmutl when you created the log. (Note: all mappings must be integral page sized in RVM; your application can use as many pages as need in both the heap and static regions – the basher needs only the minimum.)

```
.../RVM1.3/bin/rdsinit basher_log basher_data
Enter the length of the file/device basher_data: 1000000
   <rdsinit will zero the file and initialize RVM>
starting address of rvm: <something much above the break point>
heap len: 0xf0000
static len: <static len>
nlists: 100
chunksize: 128
```

NOTE: On Linux Sparc, you must give page size 8192, not 4096.

You are now ready to run the basher:

```
.../RVM1.3/bin/rvm_basher < .../RVM1.3/tests/basher_parms
```

The file .../RVM1.3/tests/basher_parms contains all the parameters to run a thorough test of RVM and its subsystems. As indicated earlier, allowing it to run for at least several hours is recommended.

A.6 The Benchmark

The benchmark program used for the SOSP 1993 paper, *Lightweight Recoverable Virtual Memory* [**?**], is included in this distribution. It is not known to work on Unix or on versions of Mach outside the Computer Science Dept. at CMU. Unlike RVM, it is dependent on C++, and may have dependencies on the exact compiler version. The version of C++ used at CMU is AT&T C++ Version It is included as documentation for the methods used to produce the data for the paper and most likely will not work as distributed on your machine.

One known problem is that the CMU CS C++ systems use <libc.h> instead of <stdlib.h> which does not work correctly here. Before attempting to compile the benchmark, you will have to change this, assuming your <stdlib.c> works. The inclusion of <mach.h>, which is required for compilation, will also cause trouble on Unix systems. Also, attempts to use other C++ systems have produced multiple definition errors for some of the standard functions such perror.

Because of these problems, no attempt is made to compile the benchmark. However, a Makefile that works at CMU is included as a starting point if you wish to try building it.

A.7 Contact with RVM Maintainers

Email should be sent to rvm@cs.cmu.edu. If you send mail to this account when you import RVM, you will be put on a mailing list of future upgrades and bug fixes.

Please understand that if you modify RVM, or any other parts of this distribution, we cannot provide assistance in debugging your modifications.

Appendix B

C Declaration for RVM

This appendix contains listings of the C header files used to construct programs with RVM. The following files are included:

rvm.h

main RVM library declarations.

rvm_statistics.h

RVM statistics declarations.

rds.h

recoverable heap allocator declarations.

rvm_segment.h

segment loader declarations.

The header `rvm.h` is required for all uses of RVM. The other should be included as needed. The order of inclusion is not important, but Unix system files are often included first. Should the files be included more than once, only the first is processed.

```
/*
*                       Definitions for RVM
*
*
*/

/*LINTLIBRARY*/

/* permit multiple includes */
#ifndef RVM_VERSION

/* Version string for initialization */
#define RVM_VERSION         "RVM Interface Version 1.3  7 Mar 1994"
#define RVM_VERSION_MAX     128         /* 128 char maximum version str length */

/* be sure parallel libraries are used */
#ifndef _PARALLEL_LIBRARIES
#define _PARALLEL_LIBRARIES 1
#endif

/* get timestamp structures and system constants */
#include <sys/time.h>
#include <sys/param.h>

#ifndef C_ARGS
#if defined(c_plusplus) || defined(__STDC__)
#define C_ARGS(arglist) arglist
#else   /* c_plusplus || __STDC__ */
#define C_ARGS(arglist) ()
#endif  /* c_plusplus || __STDC__ */
#endif  /* C_ARGS */

/* define bool, TRUE, and FALSE if libc.h hasn't been included
   these definitions will not be repeated in libc.h if it is
   included later.
*/

#ifndef TRUE
#define TRUE 1
#endif  TRUE

#ifndef FALSE
#define FALSE 0
#endif  FALSE

/* RVM's use of false, true and bool causes trouble with versions of gcc
   above 2.6 or so; because of this, the names of RVM's definitions
   have been changed to rvm_{false,true,bool_t}. (Originally changed
   in rvm.h by Satya (7/31/96); propogated to the rest of the RVM code
   8/23/96 by tilt */
```

```
typedef enum { rvm_false = 0, rvm_true = 1 } rvm_bool_t;
```

```
/*  structure identifiers: rvm_struct_id_t
    codes placed in the first field of each
    structure instance to identify the object.
*/
typedef enum
    {
    rvm_first_struct_id = 39,           /* internal use only */

    rvm_region_id,                      /* identifier for rvm_region's */
    rvm_options_id,                     /* identifier for rvm_options */
    rvm_tid_id,                         /* identifier for rvm_tid's */
    rvm_statistics_id,                  /* identifier for rvm_statistics rec's */
    rvm_last_struct_id                  /* internal use only */
    }
rvm_struct_id_t;

/*  Transaction mode codes: rvm_mode_t */
typedef enum
    {
    rvm_first_mode = 139,               /* internal use only */

    restore,                            /* restore memory on abort */
    no_restore,                         /* do not restore memory on abort */
    flush,                              /* flush records to logdev on commit */
    no_flush,                           /* do not flush records on commit */

    rvm_last_mode                       /* internal use only */
    }
rvm_mode_t;
```

```
/*  Function return codes:  rvm_return_t */
typedef enum {
    RVM_SUCCESS = 0,                    /* success return code */

    rvm_first_code = 199,               /* internal use only */

    RVM_EINIT,                          /* RVM not initialized */
    RVM_EINTERNAL,                      /* internal error, see rvm_errmsg */
    RVM_EIO,                            /* I/O error, see errno */
    RVM_ELOG,                           /* invalid log device */
    RVM_ELOG_VERSION_SKEW,              /* RVM log format version skew */
    RVM_EMODE,                          /* invalid transaction begin/end mode */
    RVM_ENAME_TOO_LONG,                 /* device name longer than 1023 chars */
    RVM_ENO_MEMORY,                     /* heap exhausted */
    RVM_ENOT_MAPPED,                    /* designated region not mapped */
    RVM_EOFFSET,                        /* invalid segment offset */
    RVM_EOPTIONS,                       /* invalid options record or pointer */
    RVM_EOVERLAP,                       /* region overlaps existing seg mapping */
    RVM_EPAGER,                         /* invalid external pager */
    RVM_ERANGE,                         /* invalid virtual memory address */
    RVM_EREGION,                        /* invalid region descriptor or pointer */
    RVM_EREGION_DEF,                    /* invalid region definition descriptor */
    RVM_ESRC,                           /* invalid address range for new
                                           values */
    RVM_ESTATISTICS,                    /* invalid statistics record */
    RVM_ESTAT_VERSION_SKEW,             /* RVM statistics format version skew */
    RVM_ETERMINATED,                    /* terminated by error already reported */
    RVM_ETHREADS,                       /* illegal C Thread library */
    RVM_ETID,                           /* invalid transaction identifier or ptr */
    RVM_ETOO_BIG,                       /* internal resouces exceeded */
    RVM_EUNCOMMIT,                      /* uncommitted transaction(s) pending */
    RVM_EVERSION_SKEW,                  /* RVM library version skew */
    RVM_EVM_OVERLAP,                    /* region overlaps existing vm mapping */

    rvm_last_code                       /* internal use only */
    }
rvm_return_t;

/* Enumeration type print name functions */
extern char *rvm_return C_ARGS
    ((
    rvm_return_t        code            /* return code */
    ));

extern char *rvm_mode C_ARGS
    ((
    rvm_mode_t          mode            /* transaction mode */
    ));

extern char *rvm_type C_ARGS
```

```
((
rvm_struct_id_t     id              /* structure identifier */
));
```

```
/*  RVM basic length and offset types:
    these types are used throughout RVM to hide machine-dependent
    representations of maximum virtual memory region lengths and
    64 bit offsets.  Do not use internal types or fields or
    portability can be compromised
*/
/*  region length: rvm_length_t
    size must be >= sizeof(char *),
    type must be unsigned arithmetic */

typedef unsigned long rvm_length_t;

/*  region offset descriptor: rvm_offset_t supports 64 bit unsigned integers
    struct unecessary if machine has 64-bit ops */

typedef struct
    {                                   /* INTERNAL FIELDS PRIVATE */
    rvm_length_t    high;               /* private */
    rvm_length_t    low;                /* private */
    }
rvm_offset_t;

/* construct offset from two rvm_length_t sized quantities x,y
   -- this will construct an offset from two lengths even if
   8*sizeof(rvm_length_t) is > sizeof(rvm_offset_t); the "extra"
   bits, the highest order bits of parameter y, will be discarded */
#define RVM_MK_OFFSET(x,y)      rvm_mk_offset((rvm_length_t)(x), \
                                              (rvm_length_t)(y))

/* offset initializer -- same as RVM_MK_OFFSET, but compile time */
#define RVM_OFFSET_INITIALIZER(x,y)     {(x),(y)}

/* Zero an offset: create a zero offset and assign it to the parameter. */
#define RVM_ZERO_OFFSET(x)      (x) = RVM_MK_OFFSET(0,0)

/* offset and length conversion macros */

/* return low-order bits of offset x as length
   -- "low-order bits" are the lowest numerically valued bits
   of the same size as rvm_length_t */
#define RVM_OFFSET_TO_LENGTH(x) ((x).low)

/* return high order bits of offset x as length
   -- "high-order bits" are defined as the highest ordered
   bits remaining after the low-order bits are extracted
   they are returned as rvm_length_t */
#define RVM_OFFSET_HIGH_BITS_TO_LENGTH(x) ((x).high)

/* return length x as rvm_offset_t */
#define RVM_LENGTH_TO_OFFSET(x) RVM_MK_OFFSET(0,(x))
```

```
/* rvm_offset_t and rvm_length_t arithmetic support */

/* add rvm_offset to rvm_offset; returns result (x+y)
   implemented as function call -- or simple add if
   machine has 64-bit integer operations */
#define RVM_ADD_OFFSETS(x,y) \
    rvm_add_offsets(&(x),&(y))

/* add rvm_length to rvm_offset; return result (length+offset)
   as rvm_offset_t
   implemented as function call -- or simple add if
   machine has 64-bit integer operations */
#define RVM_ADD_LENGTH_TO_OFFSET(x,y) \
    rvm_add_length_to_offset(&(x),(y))

/* add rvm_length_t to vm address; returns address (char *)
   always implemented as simple add */
#define RVM_ADD_LENGTH_TO_ADDR(length,vmaddr) \
    ((char *)((rvm_length_t)(length)+(rvm_length_t)(vmaddr)))

/* subtract rvm_offset from rvm_offset; return result (x-y) as rvm_offset_t
   implemented as function call -- or simple subtract if
   machine has 64-bit integer operations */
#define RVM_SUB_OFFSETS(x,y) \
    rvm_sub_offsets(&(x),&(y))

/* subtract rvm_length from rvm_offset; return result (offset-length)
   as rvm_offset_t
   implemented as function call or simple subtract if
   machine has 64-bit integer operations  */
#define RVM_SUB_LENGTH_FROM_OFFSET(x,y) \
    rvm_sub_length_from_offset(&(x),(y))

/* subtract rvm_length_t from vm address; returns address (char *)
   always implemented as simple subtract */
#define RVM_SUB_LENGTH_FROM_ADDR(vmaddr,length) \
    ((char *)((rvm_length_t)(vmaddr)-(rvm_length_t)(length)))

/*  rvm_offset_t comparison macros */
#define RVM_OFFSET_LSS(x,y)     (((x).high < (y).high) || \
                                 ((((x).high == (y).high) && \
                                 ((x).low < (y).low))))
#define RVM_OFFSET_GTR(x,y)     (((x).high > (y).high) || \
                                 ((((x).high == (y).high) && \
                                 ((x).low > (y).low))))
#define RVM_OFFSET_LEQ(x,y)     (!RVM_OFFSET_GTR((x),(y)))
#define RVM_OFFSET_GEQ(x,y)     (!RVM_OFFSET_LSS((x),(y)))
#define RVM_OFFSET_EQL(x,y)     (((x).high == (y).high) && \
                                 ((x).low == (y).low))
#define RVM_OFFSET_EQL_ZERO(x)  (((x).high == 0) && ((x).low == 0))
```

```
/* page-size rounding macros */

/* return page size as rvm_length_t */
#define RVM_PAGE_SIZE           rvm_page_size()

/* return rvm_length x rounded up to next integral page-size length */
#define RVM_ROUND_LENGTH_UP_TO_PAGE_SIZE(x)  ((rvm_length_t)( \
    ((rvm_length_t)(x)+rvm_page_size()-1) & rvm_page_mask()))

/* return rvm_length x rounded down to integral page-size length */
#define RVM_ROUND_LENGTH_DOWN_TO_PAGE_SIZE(x)  ((rvm_length_t)( \
     (rvm_length_t)(x) & rvm_page_mask()))

/* return address x rounded up to next page boundary */
#define RVM_ROUND_ADDR_UP_TO_PAGE_SIZE(x)  ((char *)( \
    ((rvm_length_t)(x)+rvm_page_size()-1) & rvm_page_mask()))

/* return address x rounded down to page boundary */
#define RVM_ROUND_ADDR_DOWN_TO_PAGE_SIZE(x)  ((char *)( \
    (rvm_length_t)(x) & rvm_page_mask()))

/* return rvm_offset x rounded up to next integral page-size offset */
#define RVM_ROUND_OFFSET_UP_TO_PAGE_SIZE(x)  \
    rvm_rnd_offset_up_to_page(&(x))

/* return rvm_offset x rounded down to integral page-size offset */
#define RVM_ROUND_OFFSET_DOWN_TO_PAGE_SIZE(x)  \
    rvm_rnd_offset_dn_to_page(&(x))
```

```
/* transaction identifier descriptor */
typedef struct
    {
    rvm_struct_id_t struct_id;          /* self-identifier, do not change */
    rvm_bool_t      from_heap;          /* true if heap allocated;
                                           do not change */
    struct timeval  uname;              /* unique name (timestamp) */

    void            *tid;               /* internal use only */
    rvm_length_t    reserved;           /* internal use only */
    }
rvm_tid_t;

/* rvm_tid_t initializer, copier & finalizer */

extern rvm_tid_t    *rvm_malloc_tid C_ARGS(());

extern void rvm_init_tid C_ARGS
    ((
    rvm_tid_t       *tid                /* pointer to record to initialize */
    ));
extern rvm_tid_t *rvm_copy_tid C_ARGS
    ((
    rvm_tid_t       *tid                /* pointer to record to be copied */
    ));
extern void rvm_free_tid C_ARGS
    ((
    rvm_tid_t       *tid                /* pointer to record to be copied */
    ));
```

```
/*  options descriptor:  rvm_options_t */
typedef struct
    {
    rvm_struct_id_t struct_id;         /* self-identifier, do not change */
    rvm_bool_t      from_heap;         /* true if heap allocated;
                                          do not change */

    char            *log_dev;          /* device name */
    long            truncate;          /* truncation threshold, % of log */
    rvm_length_t    recovery_buf_len;  /* length of recovery buffer */
    rvm_length_t    flush_buf_len;     /* length of flush buffer (partitions only) */
    rvm_length_t    max_read_len;      /* maximum single read length (MACH
                                          only)  */
    rvm_bool_t      log_empty;         /* TRUE  ==> log empty */
    char            *pager;            /* char array for external pager name */
    long            n_uncommit;        /* length of uncommitted tid array */
    rvm_tid_t       *tid_array;        /* ptr to array of uncommitted tid's */

    rvm_length_t    flags;             /* bit vector for optimization and
                                          other flags */
    }
rvm_options_t;

/* rvm_options default values and other constants */

#define TRUNCATE            50         /* 50% default truncation threshold */
#define RECOVERY_BUF_LEN    (256*1024)  /* default recovery buffer length */
#define MIN_RECOVERY_BUF_LEN (64*1024)  /* minimum recovery buffer length */
#define FLUSH_BUF_LEN       (256*1024)  /* default flush buffer length */
#define MIN_FLUSH_BUF_LEN    (64*1024)  /* minimum flush buffer length */
#define MAX_READ_LEN        (512*1024)  /* default maximum single read length */

#define RVM_COALESCE_RANGES 1          /* coalesce adjacent or shadowed
                                          ranges within a transaction */
#define RVM_COALESCE_TRANS  2          /* coalesce adjacent or shadowed ranges
                                          within no_flush transactions */

#define RVM_ALL_OPTIMIZATIONS   (RVM_COALESCE_RANGES | RVM_COALESCE_TRANS)

/* rvm_options_t initializer, copier & finalizer */

extern rvm_options_t *rvm_malloc_options C_ARGS(());

extern void rvm_init_options C_ARGS
    ((
    rvm_options_t   *options           /* pointer to record to initialize */
    ));
extern rvm_options_t *rvm_copy_options C_ARGS
    ((
    rvm_options_t   *options           /* pointer to record to be copied */
```

```
    ));
extern void rvm_free_options C_ARGS
    ((
    rvm_options_t   *options            /* pointer to record to be freed */
    ));
```

```
/*  region descriptor: rvm_region_t */
typedef struct
    {
    rvm_struct_id_t struct_id;          /* self-identifier, do not change */
    rvm_bool_t      from_heap;          /* true if heap allocated;
                                           do not change */

    char            *data_dev;          /* device name */
    rvm_offset_t    dev_length;         /* maximum device length */
    rvm_offset_t    offset;             /* offset of region in segment */
    char            *vmaddr;            /* vm address of region/range */
    rvm_length_t    length;             /* length of region/range */
    rvm_bool_t      no_copy;            /* do not copy mapped data if true */
    }
rvm_region_t;

/* rvm_region_t allocator, initializer, copier & finalizer */
extern rvm_region_t *rvm_malloc_region C_ARGS(());

extern void rvm_init_region C_ARGS
    ((
    rvm_region_t    *region             /* pointer to record to initialize */
    ));

/* note: copier copies pointers to the char arrays */
extern rvm_region_t *rvm_copy_region C_ARGS
    ((
    rvm_region_t    *region             /* pointer to record to be copied */
    ));

extern void rvm_free_region C_ARGS
    ((
    rvm_region_t *region                /* pointer to record to be freed */
    ));
```

```
/*
        Main Function Declarations
*/

/* RVM initialization */
extern rvm_return_t rvm_initialize C_ARGS
    ((
    char                *version,       /* pointer to RVM lib version string */
    rvm_options_t       *options        /* optional pointer to global options
                                           descriptor */
    ));
/* init macro */
#define RVM_INIT(options) rvm_initialize(RVM_VERSION,(options))

/* shutdown RVM */
extern rvm_return_t rvm_terminate C_ARGS(());   /* no parameters */

/* map recoverable storage */
extern rvm_return_t rvm_map C_ARGS
    ((
    rvm_region_t         *region,       /* pointer to region descriptor */
    rvm_options_t        *options       /* optional ptr to option descriptor */
    ));

/* unmap recoverable storage */
extern rvm_return_t rvm_unmap C_ARGS
    ((
    rvm_region_t        *region         /* pointer to region descriptor */
    ));

/* set RVM options */
extern rvm_return_t rvm_set_options C_ARGS
    ((
    rvm_options_t       *options        /* pointer to option descriptor */
    ));

/* query RVM options */
extern rvm_return_t rvm_query C_ARGS
    ((
    rvm_options_t       *options,       /* address of pointer to option
                                           descriptor [out] */
    rvm_region_t        *region         /* optional pointer to region descriptor */
    ));
```

```
/* begin a transaction */
extern rvm_return_t rvm_begin_transaction C_ARGS
    ((
    rvm_tid_t           *tid,           /* pointer to transaction identifier */
    rvm_mode_t          mode            /* transaction begin mode */
    ));

/* declare a modification region for a transaction */
extern rvm_return_t rvm_set_range C_ARGS
    ((
    rvm_tid_t           *tid,           /* pointer to transaction identifier */
    void                *dest,          /* base address of modification range */
    rvm_length_t        length          /* length of modification range */
    ));

/* modification of a region for a transaction */
extern rvm_return_t rvm_modify_bytes C_ARGS
    ((
    rvm_tid_t           *tid,           /* pointer to transaction identifier */
    void                *dest,          /* base address of modification range */
    void                *src,           /* base address of source range */
    rvm_length_t        length          /* length of modification range */
    ));

/* abort a transaction */
extern rvm_return_t rvm_abort_transaction C_ARGS
    ((
    rvm_tid_t           *tid            /* pointer to transaction identifier */
    ));

/* commit a transaction */
extern rvm_return_t rvm_end_transaction C_ARGS
    ((
    rvm_tid_t           *tid,           /* pointer to transaction identifier */
    rvm_mode_t          mode            /* transaction commit mode */
    ));

/* flush log cache buffer to log device */
extern rvm_return_t rvm_flush C_ARGS(( )); /* no parameters */

/* apply logged changes to segments and garbage collect the log device */
extern rvm_return_t rvm_truncate C_ARGS(( )); /* no parameters */

/* initialize log */
extern rvm_return_t rvm_create_log C_ARGS
    ((
    rvm_options_t   *rvm_options,       /* ptr to options record */
    rvm_offset_t    *log_len,           /* length of log data area */
    long            mode                /* file creation protection mode */
    ));
```

```
/* underlying support functions for length, offset, and rounding macros

   *** use outside of the macros can compromise portability ***

   these functions will not be implemented on machines with 64-bit
   integer formats since their operations will be available in the
   native instruction set
*/
extern rvm_offset_t rvm_mk_offset C_ARGS
    ((
    rvm_length_t        x,
    rvm_length_t        y
    ));
extern rvm_offset_t rvm_add_offsets C_ARGS
    ((
    rvm_offset_t        *x,
    rvm_offset_t        *y
    ));
extern rvm_offset_t rvm_add_length_to_offset C_ARGS
    ((
    rvm_offset_t        *offset,
    rvm_length_t        length
    ));
extern rvm_offset_t rvm_sub_offsets C_ARGS
    ((
    rvm_offset_t        *x,
    rvm_offset_t        *y
    ));
extern rvm_offset_t rvm_sub_length_from_offset C_ARGS
    ((
    rvm_offset_t        *offset,
    rvm_length_t        length
    ));

/* private functions to support page rounding */

extern rvm_length_t rvm_page_size C_ARGS(());
extern rvm_length_t rvm_page_mask C_ARGS(());
extern rvm_offset_t rvm_rnd_offset_up_to_page C_ARGS
    ((
    rvm_offset_t        *x
    ));
extern rvm_offset_t rvm_rnd_offset_dn_to_page C_ARGS
    ((
    rvm_offset_t        *x
    ));


#endif RVM_VERSION
```

```
/*
*
*                 Definitions for RVM Statistics Structures
*
*/
/*LINTLIBRARY*/

/* permit multiple includes */
#ifndef RVM_STATISTICS_VERSION

#define RVM_STATISTICS_VERSION  "RVM Statistics Version 1.1 8 Dec 1992"

#include <stdio.h>

/* histgram definitions */

#define flush_times_len         10      /* length of flush timing vectors */
#define flush_times_dist                /* timing distribution in millisecs */ \
    25,50,100,250,500,1000,2500,5000,10000 /* use as array initializer */

#define truncation_times_len    5       /* length of truncation timing vectors */
#define truncation_times_dist           /* timing distribution in seconds */ \
    1,10,100,500

#define range_lengths_len       13      /* length of range length histogram */
#define range_lengths_dist              /* range lengths in bytes */ \
    0,4,8,16,32,64,128,256,512,1024,2048,4096

#define range_overlaps_len      13      /* length of overlaps eliminated by
                                           range coalesce */
#define range_overlaps_dist             /* range lengths in bytes */ \
    0,4,8,16,32,64,128,256,512,1024,2048,4096
```

```
#define trans_overlaps_len      13      /* length of overlaps eliminated by
                                           transaction coalesce */
#define trans_overlaps_dist             /* range lengths in bytes */ \
    0,4,8,16,32,64,128,256,512,1024,2048,4096

#define range_elims_len          6      /* ranges eliminated from log by
                                           range coalesce */
#define range_elims_dist                /* number of ranges */ \
    0,5,10,50,100

#define trans_elims_len          6      /* ranges eliminated from log by
                                           trans coalesce */
#define trans_elims_dist                /* number of ranges */ \
    0,5,10,50,100

#define trans_coalesces_len     6       /* transactions coalesced by
                                           trans coalesce */
#define trans_coalesces_dist            /* number of transactions */ \
    0,5,10,50,100
```

```
/* RVM statistics record */
typedef struct
    {
    rvm_struct_id_t struct_id;          /* self-identifier, do not change */
    rvm_bool_t      from_heap;          /* true if heap allocated;
                                           do not change */

                                        /* transaction statistics --
                                           current epoch */

    rvm_length_t    n_abort;            /* number of transactions aborted */
    rvm_length_t    n_flush_commit;     /* number of flush mode commits */
    rvm_length_t    n_no_flush_commit;  /* number of no_flush mode commits */
    rvm_length_t    n_uncommit;         /* number of uncommited transactions */
    rvm_length_t    n_no_flush;         /* number of queued no_flush transactions */
    rvm_length_t    n_truncation_wait;  /* total transactions delayed by truncation */
    rvm_offset_t    no_flush_length;    /* length of queued no_flush transactions */
                                        /* log statistics -- current epoch */

    rvm_length_t    n_split;            /* number trans split for log wrap */
    rvm_length_t    n_flush;            /* number of internal flushes */
    rvm_length_t    n_rvm_flush;        /* number of explicit flush calls */
    rvm_length_t    n_special;          /* number of special log records */
    rvm_length_t    n_wrap;             /* number of log wrap-arounds (0 or 1) */
    rvm_length_t    log_dev_cur;        /* current % log device in use */
    rvm_offset_t    log_written;        /* current length of writes to log */
    rvm_offset_t    range_overlap;      /* current overlap eliminated by range coalesce */
    rvm_offset_t    trans_overlap;      /* current overlap eliminated by trans coalesce */
    rvm_length_t    n_range_elim;       /* current number of ranges eliminated by
                                           range coalesce/flush */
    rvm_length_t    n_trans_elim;       /* current number of ranges eliminated by
                                           trans coalesce/flush */
    rvm_length_t    n_trans_coalesced;  /* number of transactions coalesced in
                                           this flush cycle */
    struct timeval  flush_time;         /* time spent in flushes */
    rvm_length_t    last_flush_time;    /* duration of last flush (msec) */
    rvm_length_t    last_truncation_time; /* duration of last truncation (sec) */
    rvm_length_t    last_tree_build_time; /* duration of tree build (sec) */
    rvm_length_t    last_tree_apply_time; /* duration of tree apply phase
                                             (sec) */
```

```
                                        /* histogram vectors */

rvm_length_t    flush_times[flush_times_len]; /* flush timings (msec) */
rvm_length_t    range_lengths[range_lengths_len]; /* range lengths flushed */
rvm_length_t    range_elims[range_elims_len]; /* num ranges eliminated by
                                                 range coalesce/flush */
rvm_length_t    trans_elims[trans_elims_len]; /* num ranges eliminated by
                                                 trans coalesce/flush */
rvm_length_t    range_overlaps[range_overlaps_len]; /* space saved by
                                                       range coalesce/flush */
rvm_length_t    trans_overlaps[range_overlaps_len]; /* space saved by
                                                       trans coalesce/flush */
```

```
                                /* transaction stats -- cumulative since log init */

    rvm_length_t    tot_abort;          /* total aborted transactions */
    rvm_length_t    tot_flush_commit;   /* total flush commits */
    rvm_length_t    tot_no_flush_commit; /* total no_flush commits */

                                /* log stats -- cumulative */

    rvm_length_t    tot_split;          /* total transactions split for log wrap-around */
    rvm_length_t    tot_flush;          /* total internal flush calls  */
    rvm_length_t    tot_rvm_flush;      /* total explicit rvm_flush calls  */
    rvm_length_t    tot_special;        /* total special log records */
    rvm_length_t    tot_wrap;           /* total log wrap-arounds */
    rvm_length_t    log_dev_max;        /* maximum % log device used so far */
    rvm_offset_t    tot_log_written;    /* total length of all writes to log */
    rvm_offset_t    tot_range_overlap;  /* total overlap eliminated by range coalesce */
    rvm_offset_t    tot_trans_overlap;  /* total overlap eliminated by trans coalesce */
    rvm_length_t    tot_range_elim;     /* total number of ranges eliminated by
                                           range coalesce */
    rvm_length_t    tot_trans_elim;     /* total number of ranges eliminated by
                                           trans coalesce */
    rvm_length_t    tot_trans_coalesced; /* total number of transactions coalesced */

                                /* truncation stats -- cummulative */

    rvm_length_t    tot_rvm_truncate;   /* total explicit rvm_truncate calls */
    rvm_length_t    tot_async_truncation; /* total asynchronous truncations */
    rvm_length_t    tot_sync_truncation; /* total forced synchronous truncations */
    rvm_length_t    tot_truncation_wait; /* total transactions delayed by truncation */
    rvm_length_t    tot_recovery;       /* total recovery truncations */
    struct timeval  tot_flush_time;     /* total time spent in flush */
    struct timeval  tot_truncation_time; /* cumulative truncation time */

                                /* histogram vectors */
                                /* truncation timings (sec) */
    rvm_length_t    tot_tree_build_times[truncation_times_len];
    rvm_length_t    tot_tree_apply_times[truncation_times_len];
    rvm_length_t    tot_truncation_times[truncation_times_len];
                                /* cummulative flush timings (msec) */
    rvm_length_t    tot_flush_times[flush_times_len];
                                /* cummulative range lengths */
    rvm_length_t    tot_range_lengths[range_lengths_len];
                                /* total num ranges eliminated by
                                   range coalesce/flush */
    rvm_length_t    tot_range_elims[range_elims_len];
                                /* total num ranges eliminated by
                                   trans coalesce/flush */
    rvm_length_t    tot_trans_elims[trans_elims_len];
                                /* space saved by range coalesce/flush */
    rvm_length_t    tot_range_overlaps[range_overlaps_len];
```

```
                                        /* space saved by trans coalesce/flush */
    rvm_length_t    tot_trans_overlaps[range_overlaps_len];
                                        /* transactions coalesced per flush  */
    rvm_length_t    tot_trans_coalesces[trans_coalesces_len];
    }
rvm_statistics_t;
```

```
/* get RVM statistics */
extern rvm_return_t rvm_statistics C_ARGS
    ((
    char                *version,       /* pointer to RVM statistics version string */
    rvm_statistics_t    *statistics     /* address of pointer to statistics
                                           descriptor] */
    ));
#define RVM_STATISTICS(statistics) \
    rvm_statistics(RVM_STATISTICS_VERSION,(statistics))

/* rvm_statistics_t initializer, copier & finalizer */

extern rvm_statistics_t *rvm_malloc_statistics C_ARGS(());

extern void rvm_init_statistics C_ARGS
    ((
    rvm_statistics_t    *statistics     /* pointer to record to initialize */
    ));
extern rvm_statistics_t *rvm_copy_statistics C_ARGS
    ((
    rvm_statistics_t    *statistics     /* pointer to record to be copied */
    ));
extern void rvm_free_statistics C_ARGS
    ((
    rvm_statistics_t    *statistics     /* pointer to record to be freed */
    ));

/* rvm_statistics_t printer */
extern rvm_return_t rvm_print_statistics C_ARGS
    ((
    rvm_statistics_t    *statistics,    /* pointer to record to be printed */
    FILE                *out_stream     /* output stream */
    ));


#endif _RVM_STATISTICS_VERSION
```

```
/*
 * Public definitions for the Recoverable Dynamic Storage package.
 */

#ifndef _RDS_H_
#define _RDS_H_

#include "rvm.h"

#ifndef C_ARGS
#if c_plusplus
#define C_ARGS(arglist) arglist
#else   c_plusplus
#define C_ARGS(arglist) ()
#endif  c_plusplus
#endif  C_ARGS

/* Error codes */

#define SUCCESS             0
#define ERVM_FAILED         -1
#define EBAD_LIST           -2
#define EBAD_SEGMENT_HDR    -3
#define EHEAP_VERSION_SKEW  -4
#define EHEAP_INIT          -5
#define EBAD_ARGS           -6
#define ECORRUPT            -7
#define EFREED_TWICE        -8
#define ENO_ROOM            -9
```

```
/* Function definitions */

extern int rds_zap_heap C_ARGS
    ((
      char                  *DevName,
      rvm_offset_t          DevLength,
      char                  *startAddr,
      rvm_length_t          staticLength,
      rvm_length_t          heapLength,
      unsigned long         nlists,
      unsigned long         chunkSize,
      int                   *err
     ));

extern int rds_init_heap C_ARGS
    ((
      char                  *base,
      rvm_length_t          length,
      unsigned long         chunkSize,
      unsigned long         nlists,
      rvm_tid_t             *tid,
      int                   *err
    ));

extern int rds_load_heap C_ARGS
    ((
      char                  *DevName,
      rvm_offset_t          DevLength,
      char                  **staticAddr,
      int                   *err
    ));

extern int rds_start_heap C_ARGS
    ((
      char                  *startAddr,
      int                   *err
    ));

extern int rds_prealloc C_ARGS
    ((
      unsigned long         size,
      unsigned long         nblocks,
      rvm_tid_t             *tid,
      int                   *err
    ));

extern char *rds_malloc C_ARGS
    ((
      unsigned long size,
      rvm_tid_t             *tid,
```

```
  int                      *err
));
```

```
extern int rds_free C_ARGS
    ((
      char                  *addr,
      rvm_tid_t             *tid,
      int                   *err
    ));

/*
 * Because of bugs (see comment in rds_free.c) we don't actually want to free
 * objects until the end of the transaction. So fake_free records our intention
 * to free an object. do_free actually frees the object. It's called as part
 * of the commit.
 */

typedef struct intlist {
    unsigned long           size;
    unsigned long           count;
    char                    **table;
} intentionList_t;

#define STARTSIZE 128                   /* Initial size of list, may grow over time */

extern int rds_fake_free C_ARGS
    ((
      char                  *addr,
      intentionList_t       *list
    ));

extern int rds_do_free C_ARGS
    ((
      intentionList_t       *list,
      rvm_mode_t            mode
    ));
```

```
/* Heap statistics reporting */
typedef struct {
    unsigned            malloc;         /* Allocation requests */
    unsigned            prealloc;       /* Preallocation requests */
    unsigned            free;           /* Block free requests */
    unsigned            coalesce;       /* Heap coalesce count */
    unsigned            hits;           /* No need to split */
    unsigned            misses;         /* Split required */
    unsigned            large_list;     /* Largest list pointer changed */
    unsigned            large_hits;     /* Large blocks present in list */
    unsigned            large_misses;   /* Large block split required */
    unsigned            merged;         /* Objects merged from coalesce */
    unsigned            unmerged;       /* Objects not merged in coalesce */
    unsigned            freebytes;      /* Number of free bytes in heap */
    unsigned            mallocbytes;    /* Bytes allocated */
} rds_stats_t;

extern rds_print_stats C_ARGS(());
extern rds_clear_stats C_ARGS
    ((
      int                         *err
    ));
extern rds_get_stats   C_ARGS((rds_stats_t *stats));

extern rds_trace_on    C_ARGS((void (*printer)(char *, ...)));
extern rds_trace_off   C_ARGS(());
extern rds_trace_dump_heap  C_ARGS(());

#endif _RDS_H_
```

```
/*
 * Segment Loader public definitions
 */

#ifndef _RVM_SEGMENT_H_
#define _RVM_SEGMENT_H_

#include "rvm.h"

/* region definition descriptor */
typedef struct
    {
    rvm_offset_t        offset;         /* region's offset in segment */
    rvm_length_t        length;         /* region length */
    char                *vmaddr;        /* mapping address for region */
    }
rvm_region_def_t;

/* initializer for region definition descriptor */
#define RVM_INIT_REGION(region,off,len,addr) \
    (region).length = (len); \
    (region).vmaddr = (addr); \
    (region).offset = (off);

/* error code for damaged segment header */
#define RVM_ESEGMENT_HDR 2000
```

```
/* define regions within a segment for segement loader */
extern rvm_return_t rvm_create_segment C_ARGS
    ((
    char                *DevName,       /* pointer to data device name */
    rvm_offset_t        DevLength,      /* Length of dataDev if really a device */
    rvm_options_t       *options,       /* options record for RVM */
    rvm_length_t        nregions,       /* number of regions defined for segment*/
    rvm_region_def_t    *region_defs    /* array of region defs for segment */
    ));

/* load regions of a segment */
extern rvm_return_t rvm_load_segment C_ARGS
    ((
    char                *DevName,       /* pointer to data device name */
    rvm_offset_t        DevLength,      /* Length of dataDev if really a device */
    rvm_options_t       *options,       /* options record for RVM */
    unsigned long       *nregions,      /* returned -- number of regions mapped */
    rvm_region_def_t    *regions[]      /* returned array of region descriptors */
    ));

#endif _RVM_SEGMENT_H_
```