The Vulcan Runtime Architecture

by John Ellis, Bill Kalsow, Eric Roberts, Michael Sclafani

Last modified on Thu Sep 27 11:15:55 1990 by kalsow
modified on Fri May 12 16:24:27 1989 by ellis

September 27, 1990

Contents

1. Introduction

This document describes the Vulcan run-time architecture. Vulcan is a Modula-2+ programming environment. This document describes the format of a Vulcan address space, the format of Vulcan object and executable files, the Vulcan register usage conventions, the Vulcan calling sequence, the thread-specific structures, the handling of exceptions and traps, the garbage collection interface, the process of linking, the shared library facility, the "bagged-REF" facility and how Vulcan address spaces are started.

The Vulcan runtime uses three types of pointers. There are absolute pointers and two kinds of relocatable pointers. Absolute pointers are the ones provided by Modula-2+. They are represented as the 32-bit byte address of their referent. Relocatable pointers come in two flavors: long and temporary. Long relocatable pointers are represented as 32-bit signed byte offsets from the location of the pointer to its referent. Temporary relocatable pointers are long relocatable pointers until they are linked into an address space when they become absolute pointers. Zero valued relocatable pointers represent NIL. In this document all pointer types will be declared using the Modula-2+ "`POINTER TO`" declaration. Each declaration of a pointer instance will be annotated with `{L}` (long relocatable), `{T}` (temporary relocatable) or `{A}` (absolute) to indicate how that particular instance of the pointer is represented.

Structures in this document are defined using a few other extensions to the existing Modula-2+ data structures. The extensions are:

- `STRING[n]` is a string of n bytes.
- `STRING[n]` is equivalent to `ARRAY 1..n OF BYTE`.
- `STRING` is a string of unknown length.
- `SEQ n OF T` is a sequence of n T's. Unlike an array, the elements of a sequence may differ in length.
- `SEQ OF T` is a sequence with an unknown number of elements.

The `VuBase` interface defines most of types described below. The useful utililty types are:

```
Card   = BITS 16 FOR [0 .. 65535];        (* a short cardinal *)
Int    = BITS 16 FOR [-32768 .. 32767]; (* a short integer *)
Flag   = BITS  1 FOR BOOLEAN;
```

2. The Vulcan address space

This section describes the contents of a Vulcan address space. The structure of an address space is:

```
AddressSpace = RECORD
  nilHole    : STRING[16K];    (* 0x00000000 *)
  root       : Root;           (* 0x00004000 *)
  user       : ImageRec;       (* 0x00004800 *)
  heap       : STRING;         (* 0xnnnnnnnn *)
  stacks     : SEQ OF Stack;   (* 0x40000000 *)
  libraries  : LibrarySpace;   (* 0x7F780000 *)
  unixStack  : STRING[512K];   (* 0x7FF80000 *)
  system     : STRING[1G];     (* 0x80000000 *)
  io         : STRING[1G];     (* 0xC0000000 *)
END;
```

The *NIL hole* (`nilHole`) is located at address zero. It is a 16K region of memory that is not mapped into the address space. (16K is the maximum page size of the 3 target architectures.) Rather than depending on compiler generated checks, Vulcan uses the NIL hole and the virtual memory hardware to trap dereferences of NIL. `VuBase.NilHoleStart` and `VuBase.NilHoleSize` define the origin and length of the NIL hole.

The *address space root* (`root`), or simply *root*, contains pointers to the primary descriptions of the address space and a few low-level routines. The root is at the fixed fixed address immediately following the NIL hole (16K). `VuBase.RootStart` and `VuBase.RootSize` define the origin and length of the root segment.

The *user region* (`user`) contains the code and data for the address space's main program. This region is initialized from the `a.out` file that created the address space. The user region is variable sized, immediately follows the root, and is mapped as writable memory. `VuBase.UserStart` defines the origin of the user region.

The heap (`heap`) follows user space and fills the remainder of the VAX's P0 space. Heap pages are mapped as "demand-zero" pages.

Following the heap are thread stacks and shared libraries. The initial stack created by the operating system is in `unixStack`. During the boot sequence, this stack is abandoned in favor of one with the other thread stacks. They are in `stacks`, below the shared libraries. Thread stacks are allocated an initial chunk of memory at thread creation time and are extended as needed in the calling sequence. To aid in detecting stack overflows, the pages on either side of a stack segment are not mapped into the address space.

The code and data of the shared libraries is in **libraries**. This region is mapped as "copy-on-write" pages in every address space except the library manager's. The *Library Manager* is a distinguished address space that contains the initial copy of all library space pages. Currently, shared library space is 8MB long and begins 8.5MB from the end of the VAX's P1 space. On the VAX, libraries could be aligned on 64KB boundaries to facilitate sharing of user page tables. `VuBase.LibraryStart` and `VuBase.LibrarySize` define the origin and length of the shared library region.

System and IO spaces (`system` and `io`) are at fixed locations determined by the VAX architecture. Generally, they are not accessible to user processes. But, at the moment the globally shared RPC buffers live in system space. Vulcan does not directly depend on `system` or `io` space. Vulcan does assume that addresses 0 through 0x7fffffff are available.

2.1 Address space root

The address space root which describes the contents of an address space is at a fixed address (16K). This structure is:

```
Root = RECORD
  length      : INTEGER;
  initProc    : PROCEDURE ();
  mainBody    : PROCEDURE ();
  version     : INTEGER;  (* := Version *)
  bootImport  : INTEGER;
  bootModule  : INTEGER;
  bootOffset  : INTEGER;
  linkOffset  : INTEGER;
  self        : Image;        {A}
  images      : ImageMap;     {A}
  types       : TypeMap;      {A}
  machine     : MachineKind;
  argc        : INTEGER;
  argv        : ADDRESS;      {A}
  envp        : ADDRESS;      {A}
  p0size      : INTEGER;
  abortCode   : INTEGER;
  initDone    : INTEGER;
  scratch     : ARRAY [0..3] OF INTEGER;
  bootProc    : PROCEDURE (...);
  linkProc    : PROCEDURE (...): LinkStatus;
  allocator    : PROCEDURE (...): ADDRESS;
  atomicAdd    : PROCEDURE (...): BOOLEAN;
  testAndSet   : PROCEDURE (...): BOOLEAN;
  testAndClear: PROCEDURE (...): BOOLEAN;
  callKernel   : PROCEDURE (...);
  abort        : PROCEDURE (...);
  exit         : PROCEDURE (...);
  getRegister  : PROCEDURE (...): INTEGER;
  setRegister  : PROCEDURE (...);
  throw        : PROCEDURE (...);
  unwindTrap   : PROCEDURE (...);
  unwindRoot   : PROCEDURE (...);
  freeSegment  : PROCEDURE (...);
  block        : PROCEDURE (...);
  wakeup       : PROCEDURE (...);
END;
```

The root acts as the data segment for a builtin module. `length` is the length of its data segment, `initProc` is the module's initialization procedure, and `mainBody` is the module's main body. These fields are initialized by the boot code.

`version` is used by the boot code, operating system, debugger and library manager to ensure that they all agree on the layout of the address space. The version number is set to `VuBase.Version` by the boot code.

`bootImport` and `bootModule` tell the boot code how to start the address space. They indicate the imported library in the `user` image and the module within that library that are to receive control after the boot code is done. `bootOffset` and `linkOffset` are the data segment offsets of the procedures needed to complete booting. `bootProc` and `linkProc` are the procedures. These fields are initialized by the Vulcan bridge.

`self` is a pointer to the builtin root image. It is initialized by the boot code.

`images` is a pointer to a table that describes the images loaded into the address space. Is is initialized by the boot code.

`types` is a pointer to a table that describes the allocated typecodes in the address space. It is initialized by the boot code.

`machine` indicates the processor used to execute this address space. The boot code initializes `machine` to `VuMachine.Self`.

`argc`, `argv` and `envp` are the parameters provided by the operating system. They are initialized by the boot code.

`p0size` is a private variable used by the boot code during the first episode of linking.

`abortCode` contains an error status if the address space's boot code aborts, otherwise it is zero.

`initDone` is zero until just before the boot code jumps into compiler-generated code.

The remainder of the root are builtin procedures that are either impossible or difficult to express in Modula-2+. The signatures of these procedures can be found in `VuBase.def`. These procedure values are initialized by the boot code.

`allocator` is the primordial memory allocator. It is only used by the boot code during the initial linking of the address space.

`atomicAdd` atomically adds a 16-bit value into memory. It takes a 16-bit integer and the address of a 16-bit cell in memory. The integer is atomically added to the memory cell. `atomicAdd` returns TRUE iff there was a carry out of the 16-bit result.

`testAndSet` atomically tests and sets a single bit of a 32-bit word. It takes a bit number and the address of a word and returns the previous value of modified bit.

`testAndClear` atomically tests and clears a single bit of a 32-bit word. It takes a bit number and the address of a word and returns the previous value of modified bit.

`callKernel` makes a kernel call. It takes an integer indicating the kernel function to invoke and the address of a VAX format agrument list. Three values are returned: register 0, register 1 and the carry bit.

`abort` kills the address space and causes a core dump. It takes a single integer argument that indicates the reason for the abort.

`exit` terminates the address space. It takes an integer argument that is returned as the address space's status.

`getRegister` reads and returns the value of a machine register. The results are unpredictible for some registers (e.g. `PC` and `SP`).

`setRegister` sets a machine register to a specified value. The results are unpredictible or disastrous for some registers (e.g. `PC` and `SP`).

`throw` implements a non-local goto. This procedure is invoked by the compiler and exception machinery.

`unwindTrap` unwinds the `InterruptT` stack frames that are pushed as a result of machine traps.

`unwindRoot` unwinds the `RootT` stack frames that result from stack extension.

`freeSegment`, `block`, and `wakeup` are procedures implemented in Modula-2+, but imported by the builtin root module.

2.2 Images

Images are collections of object modules together with the information necessary to link them. An image results from combining object files into a program, a shared library, or a dynamic library (section 3, page 17).

Collections of images are described by image maps. An image map is:

```
ImageMap = POINTER TO RECORD
```

```
  nImages : INTEGER;
  map     : ARRAY [0..2046] OF Image; {A}
END;
```

The address space root references its primary image map.

An image is:

```
Image = POINTER TO ImageRec;
ImageRec = RECORD
  length         : INTEGER;
  selfMID        : MID;              {T}
  libraryImports : LibraryImportMap; {T}
  moduleMap      : ModuleMap;        {T}
  linkCmds       : LinkSegment;      {T}
  contents       : STRING;
END;
```

Where, `length` specifies the length of the entire image in bytes, it includes this header. `selfMID` is the modeller unique ID of the image. `libraryImports` defines the imported images. `moduleMap` defines the modules contained in the image. `linkCmds` specifies how to link the modules with the imported images. And finally, `contents` contains the code and data for the modules contained in the image. The relative layout of the components within `contents` is only important to the bridge (since it builds images).

When an image is loaded into an address space, storage is allocated in the heap for the image's global data segments. Images loaded by the library manager reside in library space. When an image is linked into an address space, its temporary relocatable pointers `{T}` are converted to absolute pointers `{A}`.

The `VuImage` interface defines two low-level procedures. `VuImage.Load` reads an image into free memory, links the image with its imported libraries, initializes the image's data segment and types, and executes the main bodies of the modules in the image. Given an image's MID, `VuImage.Lookup` locates that image in the global tables.

2.3 Library imports

A *library import map* defines a mapping from a library index to an imported image. Library import maps are used during linking. The structure of library import maps are:

```
LibraryImportMap = POINTER TO RECORD
  nLibraries : INTEGER;
  libraries  : ARRAY [0..nLibraries-1] OF LibraryImport;
```

```
END;

LibraryImport = RECORD
  MID   : MID;     {T}
  kind  : Kind;
  image : Image;   {A} (* initialized by the linker *)
END;

ImageKind = (PROGRAM, SHARED, DYNAMIC);
```

To resolve an image's external references, the linker must: find all of the libraries whose MID's are in the image's library import map, initialize the `image` fields in that map with pointers to the corresponding libraries, and process the image's link commands using this initialized map.

2.4 Module maps

A *module map* describes the implementation modules contained in an image. A module map is:

```
ModuleMap = POINTER TO RECORD
  nModules : CARDINAL;
  modules  : ARRAY [0..nModules-1] OF ModuleRec;
END;
```

For each implementation module in an image, there is a header:

```
Module = POINTER TO ModuleRec;
ModuleRec = RECORD
  code  : CodeSegment;    {T}
  data  : DataSegment;    {T}
  reloc : RelocationList; {T}
END;
```

Each module in an address space has an entry in some image's module map, a code segment, a data segment, and possible a list of relocations. A module's code segment is not modified after the module has been linked into the address space. A module's code and data segments need not be contiguous.

Before linking, the data fields of the image's module map are `DataDescriptor`'s (section 3.1, page 17). Linking an image into an address space requires that these data descriptors be expanded into the full data segments and that the corresponding entries in the module map be adjusted.

2.5 Link commands

A *link segment* describes the operations required to link an image into an address space. A link segment is a list of link commands:

```
LinkSegment = POINTER TO RECORD
  nLinks : Card;
  links  : ARRAY [0..nLinks-1] OF LinkCmd;
END;
```

Each image contains a link segment.

Each *link command* describes a linkage between the image containing the command and an imported library module.

```
LinkCmd = RECORD
  exportLibrary   : Card;
  exportModule    : Card;
  exportInterface : Card;
  importModule    : Card;
  importLinkage   : Card;
END;
```

The interface exported by the library is defined by the triple (**exportLibrary**, **exportModule**, **exportInterface**) where **exportLibrary** is an index into the importing image's library import map, **exportModule** is an index into the imported library's module map and **exportInterface** is an index into the imported module's export table. The importing module is defined by the pair (**importModule**, **importLinkage**) where **importModule** is an index into the importing image's module map and **importLinkage** is an index into the importing module's import map.

link commands whose **exportLibrary** is zero refer to modules in the importing image.

The set of link commands for an implementation module M, ordered by their **importLinkage** field, is generated by the following algorithm:

```
  AddLinkCmd (M's data segment)
  AddLinkCmd (M2Base)
  FOR x IN M2AST.PreParse(M).exports DO
    FOR y IN M2AST.PreParse(x).imports DO
      AddLinkCmd (y);
    END;
  END;
  FOR x IN M2AST.PreParse(M).imports DO
    AddLinkCmd (x);
  END;
```

2.6 Relocation lists

A *relocation list* describes the relocations needed by a foreign object module. The list is a sequence of:

```
RelocationListRec = RECORD
  header  : RelocationHeader;
  offsets : ARRAY [0..header.count-1] OF INTEGER;
END;
```

where

```
RelocationList    = POINTER TO RelocationListRec;
RelocationSegment = (TextRS, DataRS);
RelocationSize    = (Reloc1, Reloc2, Reloc4, RelocXXXX);
RelocationHeader  = RECORD
  source : BITS 1 FOR RelocationSegment;
  target : BITS 1 FOR RelocationSegment;
  size   : BITS 2 FOR RelocationSize;
  count  : BITS 28 FOR [0..256*1024*1024-1];
END;
```

Relocation lists are terminated by a header with a zero `count`.

Each integer in the `offsets` specifies an offset in the `source` segment that is to be relocated to the `target` segment. The relocation is performed by adding the address of the `target` segment to the bytes specifed by the `offsets`. Relocations may modify 1, 2 or 4 bytes as specified by the `size` field.

2.7 Data segments

A *data segment* contains the global variables and exported interfaces defined by a single implementation module. A data segment is:

```
DataSegment = POINTER TO RECORD
  length   : CARDINAL;
  initProc : PROCEDURE ();
  mainBody : PROCEDURE ();
  exports  : SEQ OF InterfaceRecord;
  private  : SEQ OF Slot;
END;
```

`length` is the length in bytes of the entire data segment. `initProc` is a compiler generated procedure that registers the module's REF types. `mainBody` is the user written main body of the module. `exports` is the interface record(s) exported by the module. `private` is the module's unshared global data.

The private section of a module's data segment contains slots like the interface records (see below). Entries in the private section are not addressed by modules outside the implementation.

After a data segment is initialized, its procedure slots point to the corresponding procedures, its import slots point to imported interface records, and all other values are zero.

After an image is linked into an address space, its `initProcs` and `mainBodys` must be called. Once the `initProcs` have been invoked, the data segments type slots contain registered typecodes. The low-level interface `VuStartup` defines procedures to accomplish this initialization.

2.8 Interface records

An *interface record* is a sequence of slots:

```
InterfaceRecord = RECORD
  exports : SEQ OF Slot;
END;
```

A single interface record exists for each definition module implemented by an implementation module. These interface records are contained in the data segment of the module implementing the interface.

The relative order and size of the slots are compile-time constants stored in the ASTs corresponding to the definition modules. A slot holds a variable, a typecode, a procedure closure, or an import linkage. Slots may be of differing sizes.

A variable slot is a sequence of bytes large enough to hold the variable's value at runtime. Variable slots are initialized to 0(=NIL) prior to program execution.

A typecode slot an INTEGER that contains a typecode that identifies the corresponding REF type. Typecodes are assigned when the image's `initProcs` are invoked.

A procedure slot is an ADDRESS that contains a pointer to the corresponding code body. The pointer to the procedure's code is initialized by the linker.

An import slot is an ADDRESS that contains a pointer to an imported interface record. Import slots are initialized by the linker.

2.9 Code Segments

A *code segment* results from compiling a single implementation module. Code segments are position independent and are contained in object files, shared libraries and loaded units. A code segment is:

```
CodeSegment = POINTER TO RECORD
  length     : CARDINAL;
  imports    : ImportMap;     {T}
  exports    : ExportMap;     {T}
  procedures : Procedure;     {T}
  dataRCMap  : RCMap.T;       {T}
  names      : ADDRESS;       {T}
  contents   : STRING;
END;
```

length is the length in bytes of the entire code segment. **imports** is a vector of chain heads used to link a module with its imports. **exports** defines the offsets of the exported interface records within the module's global data segment. **procedures** is the head of the linked list of all the procedures in the module. **dataRCMap** is the RC map for the module's global data segment. **names** is a pointer to the source names associated with the module. And finally, **contents** contains the module's actual tables and code bodies. The exact layout of **contents** are only important to the compiler.

The source names associated with a module are the name of the implementation module and the names of any interfaces that it implements. The **names** field of a code segment points to the concatenated list of these names. Each name in the list is terminated by a zero byte. The entire list is terminated by zero-length name. The name of the implementation module is the first name in the list.

2.10 Import maps

An **import map** describes the location of linkage record slots that must be relocated when a code segment is linked into an address space. An ImportMap is:

```
ImportMap = POINTER TO RECORD
  nImports : INTEGER;
  imports  : ARRAY [0..nImports-1] OF Import;
END;

Import = RECORD
  offset : INTEGER;
  chain  : ADDRESS; {T}
END;
```

Each imported module is represented by an **offset** and a **chain**. **offset** defines the byte offset in the module's global data segment of a slot that, after linking, will hold the address of the imported interface record. Likewise, `chain` is the head of a linked list of slots within linkage records that must be patched at link time to refer to the imported interface record. The chains are zero-terminated. The image's link command's `linkage` fields index into the `imports` array of the import map.

The mapping from the modeller's MID for an imported interface to its location in the import map is maintained by the bridge. The first import, `imports[0]`, refers to the data segment of this module; `imports[0].offset` is ignored. The remainder of `imports` is in the same order as the imports in the source module, with duplicates removed.

2.11 Export maps

An *export map* contains the offset of the exported interface records within a module's data segment. An export map is:

```
ExportMap = POINTER TO RECORD
  nExports : CARDINAL;
  offsets  : ARRAY [0..nExports-1] OF INTEGER;
END;
```

An image's link commands index entries in the export maps of its imported interfaces.

The mapping from the modeller's MID for an exported interface to its location in the export map is maintained by the bridge.

2.12 Procedures

A `Procedure` encodes the runtime information needed to implement a single Modula-2+ procedure. Procedures are position independent. A procedure is:

```
Procedure = POINTER TO RECORD
  next       : Procedure;      {L}
  frameType  : BITS 3 FOR ProcedureType;
  frameSize  : BITS 29 FOR [0..1024*1024*512-1];
  selfOffset : INTEGER;
  scopes     : ExceptionScope; {L}
  codeLength : INTEGER;
  code       : STRING[codeLength];
END;
```

`next` is the next procedure in the module's chain of procedures. `frameType` and `frameSize` specify the kind of frame that the procedure uses. `selfOffset` specifies the offset of the corresponding procedure slot in the module's data segment. Procedures with no corresponding slots in the data segment must set `selfOffset` to zero. `scopes` is a linked list of the exception scopes within the procedure; the scopes are ordered so that enclosing scopes come after enclosed scopes. `codeLength` is the length in bytes of the code. The exact layout of `code` is only needed by the compiler.

Procedures are classified into several categories based on their stack frames. The categories are:

```
ProcedureType = (
  LightT, FixedT, VaryingT, FinallyT, InterruptT, RootT
);
```

Where,

- `LightT` is a procedure with no stack frame.
- `FixedT` is a procedure with a fixed size frame whose size is no larger than the guard-page size. `frameSize` is the size of the frame.
- `VaryingT` is a procedure whose frame size is larger than the guard-page size or that is computed on procedure entry. `frameSize` gives the length of the procedure's entry sequence. The actual size of the frame is saved at 4(SP).
- `FinallyT` is a procedure that is a finally handler. `frameSize` gives the size of the frame.
- `InterruptT` is a procedure that encapsulates the entire register state. `frameSize` gives the size of the frame.
- `RootT` is the procedure at the root of the stack segment. `frameSize` gives the size of the frame.

2.13 Linkage records

A *linkage record* contains the pointers and data needed by a procedure to make its external references. The linkage record is a sequence of pointers to interface records and non-immediate, link-time constants (like `LONGREAL`s and `Text.Ts`).

Offsets from the code body into the linkage record are PC-relative (VAX) or PB-relative (R-series), compile-time constants.

The external references in the linkage record are established by the linker. The linkage record is a read-only structure once program execution begins.

A linkage record is:

```
LinkageRecord = SEQ OF RECORD
  CASE INTEGER OF
  | 0:   interface : POINTER TO InterfaceRecord;  {T}
  | 1:   const     : STRING;
  END;
END;
```

2.14 Scopes

A *scope descriptor* describes the extent of a TRY scope, its type and how exceptions occuring in the scope are to be handled. Each procedure contains a linked list of TRY scope descriptors. A scope descriptor is:

```
ExceptionScope = POINTER TO RECORD
  next       : ExceptionScope;   {L}
  start      : ADDRESS;          {L}
  length     : Int;
  nClauses   : Int;
  CASE kind  : ExceptionScopeKind OF
  | FinallySK: handler: FinallyHandler;  {L}
  | RaisesSK:  raises:  RaisesList;
  | ExceptSK:  clauses: ClausesList;
  END;
END;
```

Where

```
ExceptionScopeKind = (FinallySK, ExceptSK, RaisesSK);

Exception = ARRAY [0..1] OF INTEGER;

RaisesList = ARRAY [0..nClauses-1] OF Exception;

ClausesList = ARRAY [0..nClauses-1] OF ExceptClause;

ExceptClause = RECORD
  exception : Exception;
  argOffset : INTEGER;
  handler   : ADDRESS; {L}
END;

FinallyHandler = PROCEDURE ((* sp *) ADDRESS,
                            (* pc *) ADDRESS);
```

`next` points to the next scope in the chain. The scopes within a procedure are linked in a single zero-terminated chain. `start` is a pointer to the first byte of the scope. `length` is the length in bytes of the scope. The extent of a TRY scope is [`start` .. `start+length-1`]. `nClauses` is the number of exceptions handled by a `TRY-EXCEPT` statement or passed through a `RAISES` clause. `kind` specifies the type of scope.

Exceptions are represented as 64-bit UIDs generated by the compiler.

For each exception passed through a `RAISES` clause, there is a `Exception` that names the exception. For each exception handled by a `TRY-EXCEPT` statement, there is an `ExceptClause` that specifies the exception handled and the location of the handler. `ELSE` clauses are represented as 0-valued exceptions. `argOffset` is the frame offset of the local variable that will hold the exception argument. If there is no such variable, `argOffset` is `0`.

The `VuException` interface defines procedures to raise exceptions.

2.15 REF Types

Each unique REF type within an executing program is assigned a small integer called a *typecode.* Logically each REF value carries a typecode, in practice the typecodes are located with the referent. The set of all REF types within a program can be reached from the global typecode map located in the address space root. A typecode is:

```
Typecode = [0..1023];
```

A few typecodes are reserved:

```
CONST
  NilTypecode  = 0;  (* reserved typecode of NIL *)
  TextTypecode = 1;  (* reserved typecode of Text.T *)
  NReservedTypecodes = 2;
```

Typecodes are allocated from the global type map in the address space root. A type map is:

```
TypeMap = POINTER TO RECORD
  nTypes : INTEGER;
  map    : ARRAY Typecode OF TypeDefinition;
END;
```

A type is registered and a typecode assigned by calling `VuType.Register`. `Register` returns a type definition:

```
TypeDefinition = POINTER TO RECORD
  type     : Typecode;
  slot      : POINTER TO INTEGER; {A}
  rcMap     : RCMap.T;            {A}
  fp        : TypeFingerprint;
  partial   : TypeFingerprint;
  opaques   : ADDRESS;
  npr       : INTEGER;
  cleanUpQ  : ADDRESS;
END;
```

`type` contains the typecode assigned to the type. `slot` points to the defining type slot. `rcMap` is a pointer to an map that describes the location of references within the type's referent. `fp` is the unique id for this type. `partial` is a compile-time approximation to the unique id for this type. `opaques` is the address of a NIL-terminated list of pointers to opaque type slots that this type depends on. `npr` used by `ObjectCleanUp` and is the number of "package references" to this type. And `cleanUpQ` is the `ObjectCleanUp.Q` for this type.

A fingerprint is represented as a two-word record:

```
TypeFingerprint = RECORD a, b: INTEGER END;
```

A *type identification* identifies two REF types. A type identification is defined by calling `VuType.Equate` passing the addresses of type slots for the two types to be identified.

2.16 RC maps

An RC map defines the locations of REFs within variables of a particular type. An RC map is:

```
RCMap.T = POINTER TO STRING;
```

RC maps are not elaborated here (see `RCMap.def`).

2.17 Modeller MID's

Unique system modeller id's are stored in images and executables. A modeller id is:

```
MID = POINTER TO RECORD
  fp     : ARRAY [0..1] OF INTEGER;
  length : INTEGER;
  id     : STRING[length];
END;
```

`fp` is the fingerprint of the functional part of the modeller id, i.e. ignoring the recipe. `id` is the textual representation of the complete id, i.e. including the recipe. These id's are used to name shared libraries. When presented with such an id, the system modeller will return the corresponding image (see `VestaUID.def`).

3. Permanent files

Vulcan produces several types of permanent objects. Those types include:

AST the abstract syntax tree resulting from compiling a definition or implementation module. The structure of ASTs is not elaborated in this document (see `M2AST.def`).

object the result of compiling an implementation module (similar to today's ".o" files).

executable a bound collection of object files that includes a main program and some startup code (today's "a.out" file).

load-and-go executable a special form of an executable that loads the required objects into the address space when it is started.

shared library a bound collection of object files that can be managed as a shared library (similar to today's ".a" files).

dynamic library a collection of one or more compiled implementation modules that can be dynamically loaded into an address space.

core file a memory dump of a Vulcan address space.

3.1 Object files

An object file contains a compiled implementation module. The structure of object files is very similar to a module. Rather than an initialized data segment, an object file contains a description of how to build and initialize the data segment. An object file is:

```
Object = POINTER TO RECORD
  length : INTEGER;
  code   : CodeSegment;     {L}
  data   : DataDescriptor;  {L}
  reloc  : RelocationList;  {L}
  rest   : STRING;
END;
```

The `length` field contains the length (in bytes) of the entire object. The actual contents of the code, data and relocation segments are in `rest`.

A data descriptor describes how to build and initialize a data segment. Its layout is:

```
DataDescriptor = POINTER TO RECORD
  length  : INTEGER;
  nChunks : INTEGER;
  chunks  : SEQ nChunks OF DataChunk
END;
```

The `length` field contains the length (in bytes) of the data segment to create. `nChunks` specifies how many initialized data chunks are present. `chunks` are the descriptors for each initialize data chunk.

Each data chunk contains the bits needed to initialize a sub-segment of the data segment. A data chunk is:

```
DataChunk = RECORD
  length : INTEGER;
  offset : INTEGER;
  data   : STRING[length]
END;
```

The `length` and `offset` fields of a chunk bound the region of the data segment to be initialized. Any region of the data segment not described by a chunk is initialized to zero.

3.2 Executable

An executable is a Unix "a.out" file. An executable is built by binding a set of objects into an image and adding an entry point and some boot code. The format of an executable is:

```
Executable = RECORD
  header : ExecutableHeader;
  text   : ARRAY [0..NilHoleSize-1] OF BYTE;
  root   : Root;
  user   : ImageRec;
  rest   : STRING;
END;

ExecutableHeader = RECORD
  magic         : INTEGER := 264; (* = 0410 *)
  textSize      : INTEGER;
  dataSize      : INTEGER;
  bssSize       : INTEGER := 0;
  symSize       : INTEGER := 0;
  entryPoint    : INTEGER := 0;
  textRelocSize : INTEGER := 0;
  dataRelocSize : INTEGER := 0;
END;
```

The actions of the boot code are described in section 12, page 40 .

3.3 Load-and-go executable

A Load-and-go executable is a Unix shell script. The first line of the script names an interpreter for the remainder of the script. The remainder of the script is a modified binary image (`ImageRec`). The modified image's code segment pointers are replaced by `MID`s for the objects that implement the modules. The interpreter is a program that loads the modified image into memory, loads the objects implementing the modules, patchs the in-memory image to refer to the loaded objects, links the image, and finally, runs it.

3.4 Shared library

A shared library is a collection of modules stored as a `ImageRec`. Shared libraries are loaded into a special, per-machine, shared library space and shared by all clients on that single machine.

The modules within a shared library can depend on modules in the library itself as well as other shared libraries. Circular dependencies between shared libraries are not allowed. The bridge is responsible for checking these import restrictions.

When a shared library is built, the list of external references is built, the module map is constructed, the object modules are concatenated, and the link commands necessary to bind the library are generated.

The *library manager* is a vanilla, unprivileged, long-lived Taos process. It manages a single machine's shared library space. The library manager is normally started when the machine is booted. Only a single library manager can be executing at any given time on a given machine.

Client address spaces can, via an RPC interface (`VuLibrary`), request particular shared libraries to be loaded. Libraries are named by their modeller UIDs. Upon receiving such a request, the library manager loads the specified library into its address space, maps that portion of its address space into the client's address space and returns the address of the freshly loaded library. The startup sequence for an executable makes a system call to Taos which acts as a surrogate to do the initial RPCs necessary to load the executable's shared libraries.

When loading a library A, the library manager will automatically load any imported libraries that are needed to resolve the A's imports.

The library manager uses VM operations to map a set of its pages into a client address space. The mapped pages appear at the same virtual addresses in both the client and library manager address spaces. The mapped pages are "copy-on-write" in the client's address space. That is, the physical pages are shared until the client attempts to write into them. Any "copy-on-write" page written by a client is copied into a new physical page. The new page is mapped in place of the copy-on-write page in that client's address space. The new page is writable and not shared with any other address spaces. Other clients are not affected.

The library manager can accommodate any alignment requirements imposed by the nub and machine architecture. (e.g. 64K boundaries on a VAX to enable sharing of system page table entries)

3.5 Dynamic Library

A dynamic library is very similar to a shared library. The differences are that a dynamic library is not restricted to importing interfaces from shared libraries and dynamic libraries are not shared among address spaces. A dynamic library may import interfaces from a single executable and arbitrarily many other dynamic libraries. The bridge is responsible for checking these import restrictions.

When a dynamic library is loaded into an address space any shared libraries that it requires are also loaded. It is the responsibility of the client program to ensure that prior to linking a dynamic library any other dynamic libraries that it imports are loaded.

The low-level mechanisms for loading and linking a dynamic library into an address space are provided by the `VuStartUp` interface. A user-level mechanism is not yet defined.

3.6 Core files

When the boot code in a Vulcan program makes the `SYS_Vulcan` system call, Taos sets a flag indicating that the address space is a Vulcan address space. If an address space designated as a Vulcan address space must core dump, a special core file is produced. That core file format is defined in `VuCore.def`. Briefly, it is:

```
CoreFile = RECORD
  version     : INTEGER := 10101;
  cause       : OS.Signal;
  nRuns       : INTEGER;
  nRunBytes   : INTEGER;
  nThreads     : INTEGER;
  runs         : ARRAY [0..nRuns-1] OF Run;
  runBytes     : ARRAY [0..nRunBytes-1] OF BYTE;
  threads      : ARRAY [0..nThreads-1] OF ThreadEntry;
END;
```

Where `Run` and `ThreadEntry` are:

```
Run = RECORD
  start  : ADDRESS;
  length : INTEGER;
END;

ThreadEntry = RECORD
  handle : ThreadsPort.Handle;
  state  : TPSpecial.State;
END;
```

The memory runs include all writable pages of the address space, the root tables that define the set of loaded images and registered ref types, and the code and data segments of the loaded images are included. The contents of the runs are packed contiguously in `runBytes`.

The first thread entry was the "current" thread at the time of the core dump.

4. The calling sequence

The calling sequence is "caller saves" in which the caller is responsible for saving any of its live registers before making a call. The caller must also restore its registers as their values are needed.

As described in section 2.12, page 12 , there are six types of procedure:

- `LightT`: a procedure with no stack frame.
- `FixedT`: a procedure with a fixed size frame whose size is no larger than the guard-page size.
- `VaryingT`: a procedure whose frame size is larger than the guard-page size or that is computed on procedure entry.
- `FinallyT`: a procedure that is a finally handler.
- `InterruptT`: a procedure that saves all registers.
- `RootT`: the procedure at the root of a stack segment.

The VAX and R-series sequences are very similar but enough different so that they are described separately. For each kind of procedure on each architecture, the entry and exit sequences are described, together with stack-walking rules for finding the return address and previous stack frame while walking the stack.

4.1 R-series calling sequence

Register usage:

`$SP` – stack pointer

`$TH` – thread data

`$T1, $T2` – temporaries

`$PB` – procedure base

`$RA` – return address

`$KT0, $KT1` – volatile registers for nub use

`$A1 - $A23` – argument and scratch registers

`$Z0` – constant 0

`$T1`, `$T2`, `$PB`, and `$RA` have special use in the calling sequence, but they are otherwise available for use in procedure bodies. See `VuMIPS.def` for the concrete binding of these names to register values.

The compiler, debugger and runtime system assume that the following instructions are only used in the context specified below:

`jalr $RA, $PB` – procedure call

`jr $RA` – procedure return

`jr $PB` – tail call

The caller of a procedure does:

```
<evaluate parameters into $A1-$An>
<save live registers>
<evaluate procedure into $PB>
jalr    $RA, $PB
```

LightT entry and exit:

```
entry:

exit:
    <evaluate result into $A1, $A2>
    j   $RA

tail call:
    <evaluate procedure into $PB>
    j   $PB
```

LightT stack-walking rules:

- There is no current frame; the return address is in $RA; the previous frame is $SP.

LightT breakpoint restrictions: none.
FixedT entry and exit:

```
entry:
    subu    $SP, frameSize
    sw      $RA, 0($SP)

exit:
    <evaluate result into $A1, $A2>
    <restore $RA if necessary>
    j       $RA
    addu    $SP, frameSize (in delay slot)

tail call:
    <evaluate procedure into $PB>
    <restore $RA if necessary>
    j       $PB
    addu    $SP, frameSize (in delay slot)
```

FixedT stack-walking rules:

- If the PC is at the first instruction of the entry, there is no stack frame, the return address is in $RA, and the previous stack frame is at $SP.
- In the rest of the entry, there is a stack frame of size frameSize at $SP, the return address is in $RA, and the previous stack frame is at $SP + frameSize.

- In the procedure body and exit, there is a stack frame of size `frameSize` at `$SP`, the return address is in `0($SP)`, and the previous stack frame is `$SP + frameSize`.

`FixedT` breakpoint restrictions: none.
`VaryingT` entry and exit:

```
entry:
    move    $T1, $SP
    <compute new $SP>
    lw      $T2, stacklimit($TH)
    bgeu    $SP, $T2, L
        trap <stack overflow>
L:
    sw      $T1, 4($SP)
    sw      $RA, 0($SP)

exit:
    <evaluate result into $A1, $A2>
    <restore $RA if necessary>
    lw      $T1, 4($SP)
    j       $RA
    move    $SP, $T1  (in delay slot)

tail call:
    <evaluate procedure into $PB>
    <restore $RA if necessary>
    lw      $T1, 4($SP)
    j       $PB
    move    $SP, $T1  (in delay slot)
```

`VaryingT` stack-walking rules:

- If the PC is at the first instruction of the entry, there is no stack frame, the return address is in `$RA`, and the previous stack frame is at `$SP`.
- In the rest of the entry, there is a stack frame of size `$T1-$SP` at `$SP`, the return address is in `$RA`, and the previous stack frame is at `$T1`.
- In the body and exit, there is a stack frame of size `4($SP)-$SP` at `$SP`, the return address is in `0($SP)`, and the previous stack frame is in `4($SP)`.

`VaryingT` breakpoint restrictions: none.
`FinallyT` entry and exit:

```
entry:
    <in $A1: $SP of procedure context containing FINALLY
     in $A2: return address into procedure context containing
             FINALLY>
    subu    $SP, frameSize
    sw      $RA, 0($SP)
    sw      $A1, 4($SP)
    sw      $A2, 8($SP)

exit:
    <restore $RA if necessary>
    j       $RA
    addu    $SP, frameSize (in delay slot)

tail call:
    disallowed
```

`FinallyT` stack-walking rules:

- If the PC is in the entry, there is no stack frame, the return address is `$A2`, and the previous stack frame is at `$A1`.
- In the procedure body and exit, there is a stack frame of size `frameSize`, the return address is in `8($SP)`, and the previous stack frame is in `4($SP)`.

`FinallyT` breakpoint restrictions: none.
`InterruptT` entry and exit:

```
entry:
    <none: generated by another thread>

exit:
    <restore all registers except $SP, $RA, $TH>
    lw      $RA, 0(SP)
    j       $RA
    addu    $SP, frameSize (in delay slot)

tail call:
    disallowed
```

`InterruptT` stack-walking rules:

- In the procedure body and exit, there is a stack frame of size `frameSize` at `$SP`, the return address is in `0($SP)`, and the previous stack frame is `$SP + frameSize`.

`InterruptT` breakpoint restrictions: none.
`RootT` entry and exit:

```
entry:
    <none: generated by fault handler>

exit:
    lw      $RA, 0(SP)
    lw      $T1, 4(SP)
    j       $RA
    addu    $SP, $T1, $Z0 (in delay slot)

tail call:
    disallowed
```

`RootT` stack-walking rules:

- There is a stack frame of size `frameSize` at `$SP`, the return address is in `0($SP)`, and the previous stack frame is in `4($SP)`.

`RootT` breakpoint restrictions: none.

4.2 VAX calling sequence

The VAX sequence differs from that of the R-series because the VAX uses PC-relative addressing and doesn't need `$PB` (procedure base), and because the VAX doesn't have branch-delay slots and thus can't do an atomic "pop stack and return".

Register usage:

`PC` – program counter

`SP` – stack pointer

`TH` – thread data

`T1, T2` – temporaries

`RA` – return address

`A1 - A10` – argument and scratch registers

`T1`, `T2` and `RA` have special use in the calling sequence, but they are otherwise available for use in procedure bodies. See `VuVAX.def` for the concrete binding of these names to register values.

The `jmp` instruction is used for several different purposes: for a long branch, for a procedure call, for a tail call, and for a procedure return. The stack-walking rules need to identify returns and tail calls. A `jmp` of the form `jmp (RA)` is a return. A `jmp` that is not a return and is followed by the one-byte reserved-instruction tag called `tail` is a tail call. See `VuVAX.def` for the concrete binding of `tail`.

The compiler, debugger and runtime system assume that the following instructions are only used in the context specified below:

`jmp (T2)` – procedure call

`jmp (RA)` – procedure return

`jmp ...; tail` – tail call

The caller of a procedure does:

```
    <evaluate parameters into A1-An>
    <save live registers>
    <evaluate procedure into T2>
    moval   1f, RA
    jmp     (T2)
1f:
```

`LightT` entry and exit:

```
entry:


exit:
    <evaluate result into A1, A2>
    jmp     (RA)

tail call:
    jmp     <procedure address>
    tail
```

`LightT` stack-walking rules:

- There is no current frame; the return address is in `RA`; the previous frame is in `SP`.

`LightT` breakpoint restrictions: none.

`FixedT` entry and exit:

```
entry:
    subl2   $frameSize, SP
    movl    RA, 0(SP)

exit:
    <evaluate result into A1, A2>
    <restore RA if necessary>
    addl2   $frameSize, SP
    jmp     (RA)

tail call:
    <restore RA if necessary>
    addl2   $frameSize, SP
    jmp     <procedure address>
    tail
```

`FixedT` stack-walking rules:

- If the PC is at the first instruction of the entry, there is no stack frame, the return address is in `RA`, and the previous stack frame in `SP`.
- In the rest of the entry, there is a stack frame of size `frameSize`, the return address is in `RA`, the previous stack frame is `SP + frameSize`.
- In the procedure body, there is a stack frame of size `frameSize`, the return address is in `0(SP)`, and the previous stack frame is at `SP + $frameSize`.
- If the PC is at a return or tail call `jmp`, there is no stack frame, the return address is in `RA` and the previous stack frame is `SP`.

`FixedT` breakpoint restrictions: no breakpoint on the return or tail call `jmp` instruction.

`VaryingT` entry and exit:

```
entry:
    movl    SP, T1
    <compute new SP>
    cmpl    SP, stacklimit(TH)
    bgequ   L
        trap <stack overflow>
```

```
L:
    movl    T1, 4(SP)
    movl    RA, 0(SP)

exit:
    <evaluate result into A1, A2>
    <restore RA if necessary>
    movl    4(SP), SP
    jmp     (RA)

tail call:
    <restore RA if necessary>
    movl    4(SP), SP
    jmp     (RA)
    tail
```

`VaryingT` stack-walking rules:

- If the PC is at the first instruction of the entry, there is no stack frame, the return address is in `RA`, and the previous stack frame is at `SP`.
- In the rest of the entry, there is a stack frame of size `T1-SP`, the return address is in `RA`, and the previous stack frame is at `T1`.
- In the body and exit, there is a stack frame of size `4(SP)-SP`, the return address is in `0(SP)` and the previous stack frame is in `4(SP)`.
- If the PC is at a return or tail call `jmp`, there is no stack frame, the return address is in `RA` and the previous stack frame is `SP`.

`VaryingT` breakpoint restrictions: no breakpoint on the return or tail call `jmp` instruction.

`FinallyT` entry and exit:

```
entry:
    <in A1: SP of procedure context containing FINALLY
     in A2: return address into procedure context containing
            FINALLY>
    subl2   frameSize, SP
    movl    RA, 0(SP)
    movl    A1, 4(SP)
    movl    A2, 8(SP)
```

```
exit:
    <restore A1 if necessary>
    <restore A2 if necessary>
    <restore RA if necessary>
    addl2   frameSize, SP
    jmp     (RA)

tail call:
    disallowed
```

`FinallyT` stack-walking rules:

- If the PC is in the entry, there is no stack frame, the return address is in `A2`, and the previous stack frame is at `A1`.
- In the procedure body and exit, there is a stack frame of size `frameSize`, the return address is in `8(SP)`, and the previous stack frame is at `4(SP)`.
- If the PC is at the return `jmp`, there is no stack frame, the return address is in `A2` and the previous stack frame is at `A1`.

`FinallyT` breakpoint restrictions: no breakpoint on the return `jmp` instruction.
`InterruptT` entry and exit:

```
entry:
    <none: generated by another thread>

exit:
    <restore all registers except SP, TH, PC, PSL>
    addl2   $frameSize-8, SP
    rei

tail call:
    disallowed
```

`InterruptT` stack-walking rules:

- In the procedure body, there is a stack frame of size `frameSize`, the return address is in `frameSize-8(SP)`, and the previous stack frame is at `SP + $frameSize`.
- If the PC is at an rei, there is an 8 byte stack frame, the return address is in `0(SP)`, the PSL is in `4(SP)` and the previous stack frame is `SP+8`.

`InterruptT` breakpoint restrictions: none.
`RootT` entry and exit:

```
entry:
    <none: generated by the fault handler>

exit:
    movl    0(SP), RA
    movl    4(SP), SP
    jmp     (RA)

tail call:
    disallowed
```

`RootT` stack-walking rules:

- If the PC is at a return, there is no stack frame, the return address is in `RA` and the previous stack frame is `SP`.
- In the body and exit sequence, there is a stack frame of size `frameSize` at `SP`, the return address is in `0(SP)`, and the previous stack frame is in `4(SP)`.

`RootT` breakpoint restrictions: no breakpoint on the return `jmp` instruction.

4.3 Calling non-Modula-2+ code

Calls to non-Modula-2+ procedures are handled by special stub procedures that save the thread data register, repackage the arguments for the target procedures' calling conventions, call the target procedure, repackage the return results, restore the thread data register, and return into the Modula-2+ code. Vulcan will provide some mechanism for generating stubs, otherwise programmers will have to roll their own.

Calls from non-Modula-2+ procedures are handled symmetrically with the above case. Non-Modula-2+ procedures must restore the thread data register before calling a Modula-2+ procedure. To restore the thread data register, the procedure must call `Thread.Self` *(a nub call?)* and use the resulting handle to index a hash table that maps thread handles to thread data registers.

5. Thread stacks

Thread stacks are allocated in fixed-size chunks of memory. A thread is given an initial chunk at creation time. That chunk is extended as needed in the calling sequence. To aid in detecting stack overflows, the pages on either side of a stack segment are not mapped into the address space.

5.1 Stack extension

The stack-extension routine is invoked by an address fault generated when the return address is stored in a fixed stack frame or when the trap is generated while allocating a varying frame. In all cases, the offending thread is stopped, its volatile state is in its registers and nothing has been written in the incomplete stack frame.

The address fault trap handler first determines whether the PC is in the entry of `FixedT`, `VaryingT`, or `FinallyT` procedure. If so, it invokes the stack-extension routine; if not, the fault is a non-recoverable address fault.

The stack-extension routine allocates a new stack segment, writes a fake stack frame on the new stack, resets the stopped thread's registers to refer to the new stack, and resumes the thread. The procedure that caused the stack overflow will return into a runtime routine that uses the fake stack frame to unwind back to the original stack segment.

To avoid expensive stack fault/allocate/unwind/deallocate cycles, the stack-extension routine and the unwind routine will keep a pointer to the last allocated stack segment in the thread data. If the stack extension routine is invoked and a large-enough free segment is in the thread data, the routine will use it and not allocate another.

Tricky points: The stack extension routine must atomically reset the offending thread's registers. Asynchronous stack walkers (e.g. Collector) must not reference the thread while the extension routine is running. Likewise, the unwind routine probably needs to be atomic with respect to asynchronous stack walkers. A mutex in the thread data should probably serve to provide mutual exclusion between the stack extender, the collector, the stack unwinder and any other asynchronous thread mutators.

(What happens when the Collector looks at a thread that has taken a stack fault but not yet gotten serviced by the stack extension thread? Does the low-level fault handler acquire the thread's transaction queue lock? does the collector wait?)

5.2 Stack walking

A stack walk visits each procedure context of a thread (even procedure contexts that have zero-sized frames). The `VuStack` interface provides a procedure for walking stacks:

```
TYPE Context = RECORD
  proc : Procedure;
  pc   : ADDRESS;  (* where in the procedure *)
  sp   : ADDRESS;  (* the procedure's frame *)
  ra   : ADDRESS;  (* return address register *)
  t1   : ADDRESS;  (* temp 1 register *)
  a1   : ADDRESS;  (* arg 1 register *)
  a2    : ADDRESS;  (* arg 2 register *)
END;

PROCEDURE Top (pc        : ADDRESS;
       VAR IN  registers : ARRAY OF INTEGER;
       VAR OUT context   : Context);

PROCEDURE Self (VAR OUT context: Context);

PROCEDURE Previous (VAR IN a: Context; VAR OUT b: Context);
PROCEDURE PreviousUser (VAR IN a: Context; VAR OUT b: Context);
PROCEDURE PreviousActive (VAR IN a: Context; VAR OUT b: Context);

PROCEDURE FinallyParent (VAR IN a: Context;  VAR OUT b: Context);
```

A `Context` describes a single procedure activation. If the procedures is in its entry or exit sequence it may not have a valid stack frame, but as long as the PC remains in the procedure, it has a context. `pc` and `sp` are always valid, `ra`, `t1`, `a1` and `a2` may not be valid in all contexts. The only valid `Contexts` are those returned by procedures in this interface.

`Top` returns the top context of a thread, given the registers and PC of that thread.

`Self` returns the context of its caller.

`Previous` returns in `b` the context immediately prior to `a`. If there is no prior context, it returns with `b.proc = NIL`. It is an unchecked runtime error if `a` and `b` are aliased to one another.

`PreviousUser` returns in `b` the first "user context" prior to `a`. If there is no such context, it returns with `b.proc = NIL`. Contexts from `InterrputT` and `RootT` procedures are skipped, all other constexts are "user contexts" and are returned. It is an unchecked runtime error if `a` and `b` are aliased to one another.

`PreviousActive` returns in `b` the first "active context" prior to `a`. If there is no such context, it returns with `b.proc = NIL`. Active contexts are all user contexts except those that occur between a finally handler's context and its enclosing procedure's context. If `a` is a finally handler, `b` will be the caller of the procedure that contained the finally statement. It is an unchecked runtime error if `a` and `b` are aliased to one another.

`FinallyParent` returns the context of the active procedure that contains the finally statement being handled by `a`. If `a` is not a finally handler, `b` is returned unchanged. It is an unchecked runtime error if `a` and `b` are aliased to one another.

These procedures use the stack-walking rules in section 4, page 21 to get from one context to another.

If a thread wishes to walk another's stack, the first must ensure that the latter is stopped during the walk.

All of these procedures need to map a PC to the containing Procedure. The runtime and debugger maintain hash tables that map PCs to procedures. These tables are constructed on-the-fly, as needed.

6. Thread data

A dedicated register th carries a pointer to the thread's state (wrt. the Vulcan runtime). A thread's state is:

```
VuThreadStack.Rec = REF RECORD
  base  : ADDRESS;      {A}
  limit : ADDRESS;      {A}
  spare : ADDRESS;      {A}
  mu    : Thread.Mutex;
  gc    : GCData;
END;
```

base and limit define the extent of the thread's current stack segment and spare points to a hot spare segment. gc contains the threads transaction queue. It is defined and maintained by the garbage collector (see TQ.def). mu is used by the runtime to serialize access to the thread state.

The VuThreadStack interface provides procedures for accessing thread stacks.

7. Exceptions

Exceptions are raised by calling RAISE or as a side-effect of a machine trap. This section describes how an exception is handled after RAISE has been called. Section 8, page 39 describes how traps are converted into exceptions.

RAISE is implemented as a normal procedure with a stack frame. It makes two passes through the stack. The first pass checks to ensure that some except handler will field the exception. If no handler is found, the debugger is invoked. If a handler is found, the second pass invokes finally handlers and then resumes control at the except handler.

RAISE uses VuStack.Caller and VuStack.MyCaller to walk the stack. The procedures enumerate the procedure contexts of a thread stack, returning the PC, SP, and Procedure of each context.

For each procedure context, RAISE traverses the list of the procedure's scopes to find the enclosing scopes of the context's pc. (The scopes are ordered so that enclosing scopes come after enclosed scopes.)

See section 2.14, page 14 for details about scope descriptors.

Finally handlers are normal procedures invoked by RAISE.

```
FinallyHandler = PROCEDURE ((* sp *) ADDRESS,
                            (* pc *) ADDRESS);
```

The arguments sp and pc describe the stack frame and pc of the procedure context containing the FINALLY clause. (See section 4, page 21 .) Because the stack frame ("static link") of the containing procedure is the first argument, the compiler treats finally handlers like nested procedures.

Except handlers are simply destination addresses within the containing procedure and are invoked by Throw (see below).

7.1 RAISE

Here's a fairly complete implementation of RAISE:

```
PROCEDURE VuException.Raise (exception : Exception;
                              arg       : ARRAY OF BYTE);
  VAR
    scope     : Scope;
    failScope : Scope;
    exScope   : Scope;
    exClause  : ExceptClause;
    ex        : Exception;
    myCaller  : VuStack.Context;
    context   : VuStack.Context;
    next      : VuStack.Context;
    handler   : FinallyHandler;
  BEGIN
    VuStack.Self (context);
    VuStack.PreviousActive (context, myCaller);
    failScope := NIL;
    exScope   := NIL;

    (* pass 1: find a handler *)
    context := myCaller;
    scope   := RELOCATE (context.procedure^.scopes);
    ex      := exception;
    LOOP
      WHILE scope = NIL DO
        VuStack.PreviousActive (context, next);
        next := context;
      IF context.sp = NIL THEN Debugger.Call(); END;
      scope := RELOCATE (context.procedure^.scopes);
    END;

    IF (scope^.start <= context.pc)
      AND (context.pc < scope^.start + scope^.length) THEN
      CASE scope^.kind OF
      | FinallySK:
          (* ignore *)
      | ExceptSK:
          IF FindHandler (scope, ex, exClause) THEN
            exScope := scope;
            EXIT;
          END;
      | RaisesSK:
          IF (failScope = NIL)
            AND NOT MatchingRaises (scope, ex) THEN
            failScope := scope;
            ex := Fail;
          END;
      END;
    END;

    scope := RELOCATE (scope^.next);
  END;

    (* pass 2 *)
  context := myCaller;
  scope   := RELOCATE (context.procedure^.scopes);
  ex      := exception;
  LOOP
    WHILE scope = NIL DO
      VuStack.PreviousActive (context, next);
      next := context;
      scope := RELOCATE (context.procedure^.scopes);
    END;

    IF (scope^.start <= context.pc)
      AND (context.pc < scope^.start + scope^.length) THEN
```

```
        CASE scope^.kind OF
        | FinallySK:
            handler := LOOPHOLE (RELOCATE (scope^.handler),
                                 FinallyHandler);
            IF (context.proc^.frameType = VuBase.FinallyT)
            THEN
              VuStack.FinallyParent (context, next);
              handler (next.sp, next.pc);
            ELSE
              handler (context.sp, context.pc);
            END;
        | ExceptSK:
            IF (scope = exScope) THEN
              IF (exClause.argOffset > 0) THEN
                COPY (ADDR (arg),
                   context.sp + exClause.argOffset,
                   NUMBER (arg));
              END;
              Throw (exClause.handler, context.sp);
            END;
        | RaisesSK:
            IF (scope = failScope) THEN
              ex := Fail;
            END;
        END;
      END;

      scope := RELOCATE (scope^.next);
    END;

  END Raise;

PROCEDURE FindHandler (scope     : Scope;
                       exception : Exception;
              VAR OUT exClause   : ExceptClause): BOOLEAN;
  VAR i: INTEGER;  ex: Exception;
  BEGIN
    FOR i := 0 TO scope^.nClauses - 1 DO
      WITH Z = scope^.clauses[i] DO
```

```
        ex := Z.exception;
        IF (ex = exception) OR (ex = <0,0>) THEN
          exClause.handler := RELOCATE (Z.handler);
          exClause.argOffset := Z.argOffset;
          RETURN TRUE;
        END;
      END;
    END;
    RETURN FALSE;
  END FindHandler;

PROCEDURE MatchingRaises (s: Scope;  ex: Exception): BOOLEAN;
  BEGIN
    IF (ex = Fail) THEN RETURN TRUE; END;
    FOR i := 0 TO s^.nClauses - 1 DO
      IF (ex = s^.exceptions[i]) THEN
        RETURN TRUE;
      END;
    END;
    RETURN FALSE;
  END MatchingRaises;
```

The Throw procedure resumes control at newPC, popping the stack back to newSP. To pop the stack with multiple segments, Throw first checks to see if the current SP and the new SP are on the same page, and if so, does nothing else (stacks are allocated in multiple page units, aligned on page boundaries). In the uncommon case where the old and new SPs are on different pages, Throw uses thread data to find the stack segments and free any popped segments.

There are two assembly versions below, one for the R-series and one for the VAX. Note that no special tracing rules are required for these implementations; they behave exactly like FixedT procedures with respect to the tracing rules.

```
PROCEDURE Throw(newPC, newSP: ADDRESS); (* R-series *)
    subu    $SP, frameSize
    sw      $RA, 0($SP)
    <free any intervening stack segments>
    move    $RA, $A1
    j       $RA
    move    $SP, $A2
    END Throw;
```

```
PROCEDURE Throw(newPC, newSP: ADDRESS); (* VAX *)
    subl2   $frameSize, SP
    movl    RA, 0(SP)
    <free any intervening stack segments>
    movl    A1, RA
    movl    A2, SP
    jmp     (RA)
    END Throw;
```

8. Traps

Traps are events that are converted into exceptions. Such events are: divide by zero, dereference NIL or stack overflow. When a trap occurs in a thread, another thread is notified. The notified thread must convert the trap into a call of RAISE on the trapped thread's stack.

A trap handler must be able to take an arbitrary PC value and parse the trapped thread's stack. Starting in the image maps, the handler can find all procedures, their stack sizes and entry sequence lengths.

9. Garbage Collection

The garbage collector must periodically scan all thread stacks and global data segments. From the image maps the collector can find all data segments. Using the image maps, the collector can also convert arbitrary PC values into meaningful stack interpretations.

(What are the details?)

(What is the ref-counting interface?)

10. Linking

The linker will be responsible for binding images into an address space.

There are three primary activities during linking: resolving inter-module references, resolving intra-module references, and assigning type codes.

- *Inter-module references (1/linkage slot).* There is a single form of inter-module linkage. A module's linkage record contains references other module's interface records. The Vulcan bridge will produce the link commands needed to drive the linking.

- *Intra-module references (1/procedure slot).* There is a single form of intra-module linkage. A module's data segment (interface record) contains pointers to its procedures' bodies. The procedure slots that must be fixed are located through the `selfOffset` field in the module's procedure chain.
- *Run-time types (1/REF type).* The linker must also invoke the modules' `initProcs`. The `initProcs` will register new REF types and equate old ones.

11. Bagged refs

A modified version of the `Bag` program will take a pickle (in a file) and produce a Vulcan interface and an implementing object containing the pickle as initialized data in the data segment:

```
SAFE DEFINITION MODULE MyBag;

IMPORT Pkl, Storage;

PROCEDURE Unpickle(
  specials:    Pkl.ReadSpecials := NIL;
  clientState: REFANY := NIL
  ): REFANY
  RAISES {Pkl.Error, Pkl.TypeFault, Storage.MemoryExhausted};

END MyBag.
```

To implement this, the `Pkl` package will be modified to include a new procedure: `Pkl.ReadInPlace`. `ReadInPlace` will unpickle the object without copying it.

12. Starting Vulcan address spaces

A Vulcan address space may be started in one of two ways: by the Vulcan "load-and-go" interpreter or by executing an "a.out" file containing a Vulcan executable.

12.1 Executable

Executables produced by Vulcan contain the standard "text", "data" and "bss" segments. The text segment contains a short segment of assembly language instructions that boot the process. The data segment contains space for the address space root and an image. Usually, the bss segment is empty. Execution begins when Taos calls the first instruction in the text segment.

The boot code is a pre-assembled sequence of machine instructions independent of the program contained in the executable. The boot code is called by Taos with the standard Unix parameters: argc, argv and envp. The boot code has three tasks: map in the required shared libraries, initialize the address space root, and start the program.

To map in the required libraries, the boot code will load the address of the user's image (an assembly time constant) into R0 and make a system call. The system call, `SYS_Vulcan`, is handled by Taos. On behalf of the new process, Taos makes an RPC call to the library manager passing the address space number and R0.

When the library manager is called for a new process it: loads the libraries named in the image's library import map into its own address space, map the relevant portions of the library space into the process's address space, and locate the boot module identified in the process's address space root. The library manager may choose not to load some libraries. If a library is not loaded by the library manager, it is the responsibility of the address space to dynamically load its own libraries.

When the system call returns, the boot code calls the first procedure of the boot module. This procedure links the user's image with the shared libraries and executes the image's `mainBodys`.

A pure Unix executable does not use shared libraries. Its boot code does not make the `SYS_Vulcan` system call. The boot code locates the boot module itself.

The library manager uses the pure Unix boot code. The library manager executable contains the libraries needed by the manager.

12.2 Load-and-go

The "load-and-go" startup proceeds in four phases. First, the load-and-go interpreter is started from an executable as described above. Second, the interpreter reads the modified image from the load-and-go executable. Third, the objects composing the image are loaded and the image is patched. Finally, the load-and-go program links and executes the new image.

To minimize startup time, the bulk of the load-and-go interpreter will be in a shared library. Consequently, the executable containing the interpreter is very small.

12.3 Debugging

When a Vulcan address space is started it checks the last runtime argument, `argv(argc-1)`, for special values to indicate that the address space should stop for debugging. If one of the special values is detected, the argument is removed from the list and the address space stops at the designated point. The recognized values are:

`!!!` stop just prior to executing any user code (module main bodies).

`!!1` stop just prior to executing any compiler generated initialization code (e.g. type registration)

`!!0` stop just prior to linking the address space.

At the moment the Load-N-Go loader doesn't recognize these flags.