A Tour of the Squeak Object Engine

Tim Rowledge, tim@sumeru.stanford.edu

A. Introduction

This chapter will explain the design and operation of the Squeak Object Engine. The term *Object Engine* is a useful phrase that encompasses both the Smalltalk low-level system code (such as the `Context` and `Process` classes) and the Virtual Machine.

We will discuss what a Virtual Machine (VM) is, how it works, what it does for Squeak programmers and users, and how the Squeak VM might develop in the future. Some explanation of system objects such as `Contexts` and `CompiledMethods` will be included.

A. What is a Virtual Machine?

A Virtual Machine is a software layer that provides us with a pretense of having a machine other than the actual hardware in use. Using one allows systems to run as if on hardware designed explicitly for them.

Object Engine is less commonly used but is a useful concept that includes the lowest system areas of the language environment running on the VM. Since there is often some flux in the definition of which components are within the actual VM and which are part of the supported environment, Object Engine is useful as a more inclusive term. In Smalltalk we would usually include classes such as `Context`, `Process`, `Number`, `InstructionStream` and `Class` in the definition of Object Engine.

The term Virtual Machine is used in several ways. When IBM refer to VM/CMS they are referring to a way of making a mainframe behave as if it is many machines, so that programs can assume they have total control even though they do not. Intel provide a somewhat similar facility in the x86 architecture, referred to as Virtual Mode. This sort of VM is a complete hardware simulation, often supported at the lowest level by the hardware.

Another sort of VM is the emulator - SoftWindows for the Mac, Acorn's !PC, Linux's WINE are good examples - where another machine and/or OS is simulated to allow a Mac user to run Windows programs, an Acorn RiscPC or a Linux machine to run Windows98 programs and so on. Emulators of games consoles, such as Bleem, are also popular, if a little legally contentious.

Many languages and even applications, such as some popular word processors, are built on a VM. Various implemetations of the BASIC language are probably the most numerous deployed VMs. BASIC interpreters do not just interpret the BASIC language but have to provide a varying amount of runtime support code depending on the precise system. Perl is another popular language that uses a similar form of VM.

We shall focus in this chapter on the form of VM used for Smalltalk, in particular the Squeak version. Many of the general principles apply equally well to other Smalltalk systems, and often to other dynamic languages such as Lisp, Dylan and even Java.

A. The basic functionality of a Smalltalk Virtual Machine

In a Smalltalk system all we do is

- create objects,
- send them messages which return objects

In order to send messages we must execute the bytecode instructions found in the `CompiledMethod`s belonging to the classes making up our system. Some message sends result in the VM calling primitives to perform low-level operations or to interface to the real host operating system. Eventually most objects are no longer needed and the system will recover the memory via the garbage collector.

In this section we will consider the basics of object allocation, message sending, bytecode and primitive execution, and garbage collection.

1. Creating Objects

Unlike structures in C or records in Pascal, Smalltalk objects are not simply chunks of memory to which we have pointers, and so we need something more sophisticated than the C library `malloc()` function in order to create new ones.

Smalltalk creates objects by allocating a chunk of the object memory and then building a header that provides the VM with important information such as the class of the object, the size and some content format description. It also initialises the contents of the newly allocated object to a safe, predetermined, value. This is important to the VM since it guarantees that any pointers found inside objects are proper object pointers (or *oops*) and not random memory addresses. Programmers also benefit from being able to rely on the fresh object having nothing unexpected inside it.

Smalltalk allows for four kinds of object, which can contain

I. oops, referred to by name. See, for example, class `Class.`

II. oops, referred to by an index. Optionally there can be named variables as in a). See `OrderedCollection`.

III. machine words, usually 32 bit in current implementations, referred to by an index. See `Bitmap`

IV. bytes, referred to by an index. See `String`

If a pointer object (a or b above) were not properly initialised the garbage collector would be prone to attempting to collect random chunks of memory -- any experienced C programmer can tell you tales of the problems that random pointers cause. Objects containing oops are initialised so that all the oops are nil and objects containing words or bytes are filled with zeros.

We cannot mix oops and non-oops in the same object - `CompiledMethod` appears to do this by an ugly sleight of hand and two special primitives (see `Interpreter > primitiveObjectAt` and `objectAtPut`) . Plans exist to correct this situation and to break methods into two normal objects instead of one hybrid.

1. Message sending

In order to do anything in Smalltalk, we have to send a message.

Sending a message is quite different to calling a function; it is a request to perform some action rather than an order to jump to a particular piece of code. This indirection is what provides much of the power of Object Oriented Programming; it allows for encapsulation and polymorphism by allowing the recipient of the request to decide on the appropriate response. Adele Goldberg has neatly characterised this as "Ask, don't touch".

Message sending involves three major components of the Object Engine

- `CompiledMethods`
- `Contexts`
- the VM

A brief explanation of the first two is required before we can explain the VM details of sending. Smalltalk reifies the executable code that it runs and the execution contexts representing the running state. This guarantees that the system can access that state and manipulate it without recourse to outside programs. Thus we can implement the Smalltalk debugger in Smalltalk, portably and extensibly.

a) `CompiledMethod`

These are repositories for the fixed part of a Smalltalk program, holding the compiled bytecode instructions and a literal frame, a list of literal objects used to hold message selectors and objects needed by the code that are not receiver instance variables or method temporary variables. They have a header object which encodes important information such as the number of arguments, literals and temporary variables needed to execute the method as well as an optional primitive number and whether the method requires the normal stack size (16 slots) or the larger stack size (56 slots). All of these quantities are determined by the Compiler when the method code is accepted by the user or filed in from outside.

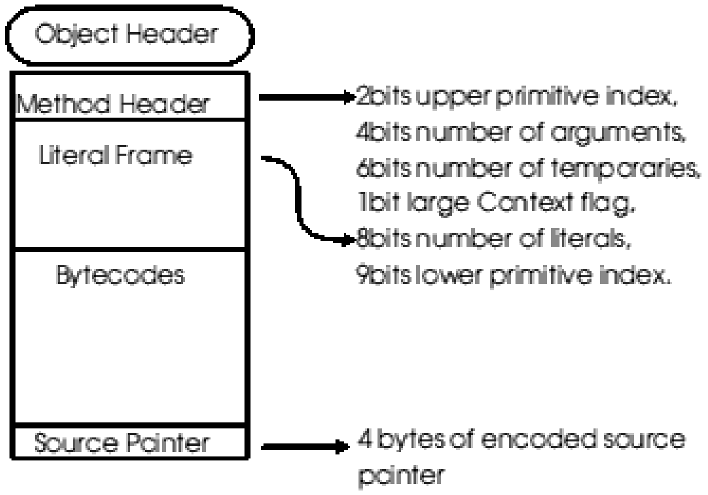
Object Header
Method Header
Literal Frame
Bytecodes
Source Pointer
2bits upper primitive index,
4bits number of arguments,
6bits number of temporaries,
1bit large Context flag,
8bits number of literals,
9bits lower primitive index.
4 bytes of encoded source pointer

figure 1. Format of `CompiledMethod` instances

a) `Context`

Contexts are the activation records for Smalltalk, maintaining the program counter and stack pointer, holding pointers to the sending context, the method for which this is appropriate, etc. There are two types of context:-

`MethodContext` - representing an executing method, it points back to the context from which it was activated, holds onto its receiver and compiled method. Note how similar it is to a stackframe from a C program.

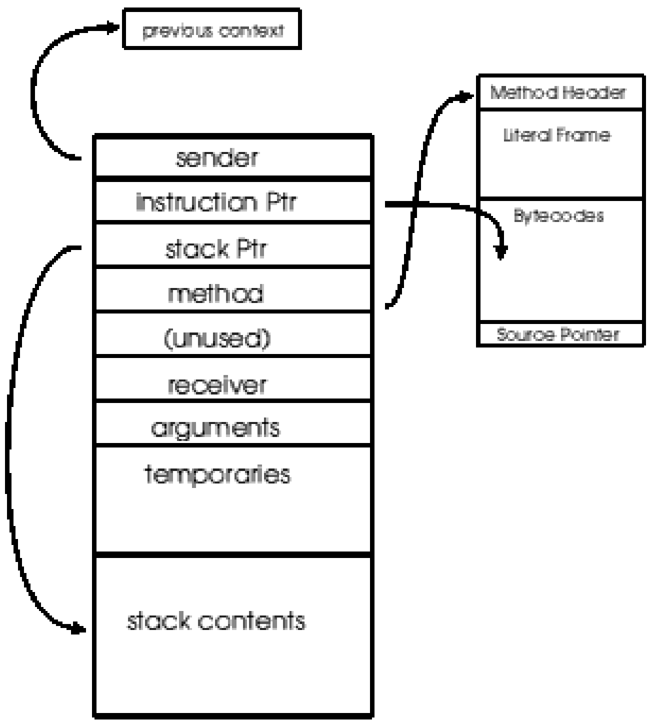

figure 2. Format of `MethodContext` instances

`BlockContext` - an active block of code within some method, it points back to its home context, the MethodContext where it was defined as well as the caller, the context from where it was activated.

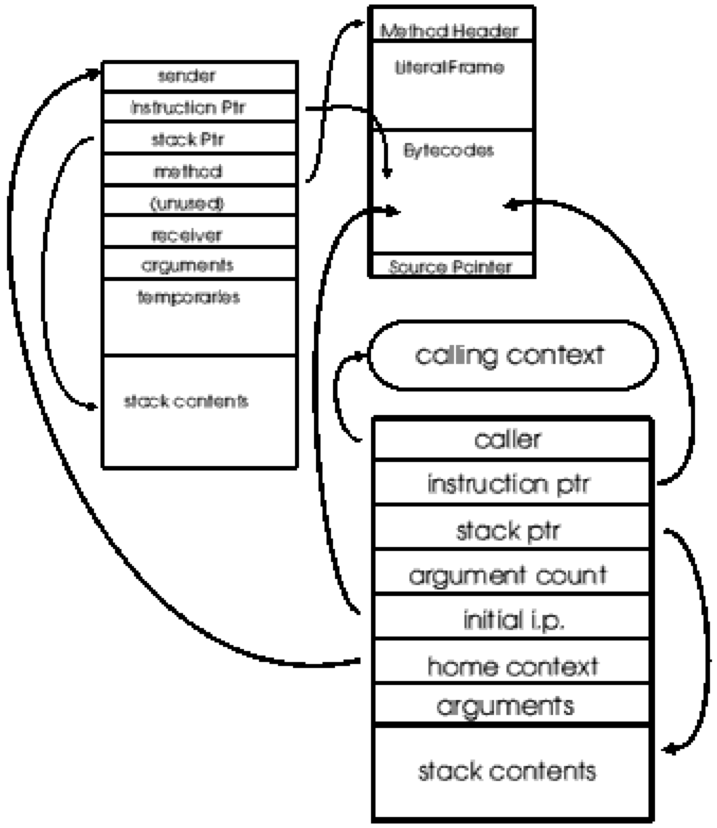

figure 3. Format of `BlockContext` instances

Note that both forms have a stack frame private to their own use. This stack frame is used for the arguments, the local temporary variables and all the working variables the code requires. In practise, the compiler can work out the size of stackframe needed by any code it is compiling, but only two sizes are used in order to aid the implementation of context recycling which helps reduce the workload on the memory system.

To send a message to a receiver, the VM has to:-

I. find the class of the receiver by examining the object's header.

II. lookup the message in the list of messages understood by that class (the class's `MethodDictionary`)

III. if the message is not found, repeat this lookup in successive superclasses of the receiver

- if no class in the superclass chain can understand the message, send the message `doesNotUnderstand:` to the receiver so that the error can be handled in a manner appropriate to that object.

IV. extract the appropriate `CompiledMethod` from the `MethodDictionary` where the message was found and then -

  (i) check for a primitive (see the later section) associated with the method by reading the method header

  (ii) if there is a primitive, execute it.

  (iii) if it completes successfully, return the result object directly to the message sender.

  (iv) otherwise, continue as if there was no primitive called.

V. establish a new activation record, by creating a new `MethodContext` and setting up the program counter, stack pointer, message sending and home contexts, then copy the arguments and receiver from the message sending context's stack to the new stack..

VI. activate that new context and start executing the instructions in the new method.

In a typical system it often turns out that the same message is sent to instances of the same class again and again; consider how often we use arrays of `SmallInteger` or `Character` or `String`. To improve average performance, the VM can cache the found method. If the same combination of the method and the receiver's class are found in the cache, we avoid a repeat of the full search of the `MethodDictionary` chain. See the method `Interpreter > lookupInMethodCacheSel:class:` for the implementation.

VisualWorks and some other commercial Smalltalks use inline cacheing, whereby the cached target and some checking information is included inline with the dynamically translated methods. Although more efficient, it is more complex and has strong interactions with the details of the cpu instruction and data caches.

1. ByteCodes

When a message has been sent and a new `MethodContext` has been activated, execution continues by executing the bytecodes from the `CompiledMethod`.

The bytecodes are byte sized tokens that specify the basic operations of the VM such as pushing variables onto the context's stack, branching, popping objects from the stack back into variables, message sending and returning. Unsurprisingly they bear a strong resemblance to an instruction set for a stack oriented CPU.

As the VM is conceptually a simple bytecode interpreter[1], execution follows this loop:-

- fetch bytecode
- increment VM instruction pointer
- branch to appropriate bytecode routine, usually implemented as a 256 way case statement.
- execute the bytecode routine
- return to top of loop to fetch next bytecode

a) Bytecode categories

Most bytecodes belong to a category where part of the byte is used to specify the basic operation and the rest is used to specify and index of some sort. For example the Squeak bytecode 34 belongs to the *push literal variable* group that starts at bytecode 32 and pushes the (34 - 32 = 2) second literal variable onto the stack. For up to date details on the precise numbering of bytecodes in Squeak, refer to the `Interpreter class > initializeBytecodeTable` method.

(1) Stack pushes

Before sending messages the arguments and the receiver need to be pushed onto the stack. Since the arguments may be receiver variables, temporary variables of the current method, literal variables or constants, or even the current context itself, there are quite a few push bytecodes.

- Push receiver variable 0-15. Fetch the referenced variable of the current receiver and push it.
- Push temporary variable 0-15 from the home context - if the current context is a `BlockContext`, the current home context is the block's home context. See figure 3. above.
- Push literal constant 0-32 from the literal frame of the home context's `CompiledMethod`.
- Push literal variable 0-32 - as above, but assumes the literal is an `Association` and pushes the value variable of that instead of the association itself.
- Push special object - receiver, `true`, `false`, `nil`, or the `SmallIntegers` negative one, zero, one or two. Since these are very frequently used, it is worth using a few bytecodes on them.
- Extended push - uses the byte following the bytecode to allow a larger index into the receiver, temporary or literal variables lists.
- Push active context pushes the actual current context, allowing us to manipulate the execution and build tools such as the debugger, exception handling and so on.

(1) Stack pops and stores

The results of message sends need to be popped from the stack and stored into some suitable place. Just as with the push bytecodes, this may mean receiver variables or temporaries but we do not store into the literal frame nor the current context using bytecodes.

- Store and pop receiver variable 0-8.

[1]Virtually all commercial Smalltalks actually take the bytecode list as input to some form of translator that produces a machine specific subroutine to improve performance. Many techniques both simple and subtle are used. There are plans to incorporate such a dynamic translation or JIT system into Squeak.

- Store and pop temporary variable 0-8.
- Extended pop and store - as above but uses the next byte to extend. the index range usable.
- Extended store - as the above pop & store but does not actually pop the stack.
- Pop stack - just pops the stack.
- Duplicate stack top - pushes the object at the stack top, thus duplicating the stack top.

(1) Jumps

We need to be able to jump within the bytecodes of a `CompiledMethod` so that the optimisations applied to control structures such as `ifTrue:ifFalse:` can be supported. Such message sends are short circuited for performance reasons by using test and jump bytecodes along with unconditional jump bytecodes.

- short unconditional jump - jumps forward by 1 to 8 bytes.
- short conditional jump - as above, but only if the object on the top of the stack is false.
- long unconditional jump - uses the next byte to extend the jump range to -1024 to +1023. A small but important detail is that backwards branches are taken as a hint that we may be in a loop and so a rapid check is made for any pending interrupts.
- long jump if true/false - if the object on the top of the stack is true or false as appropriate, use the next byte to give a jump range of 0 to 1023.

(1) Message Sending

As mentioned above, the main activity in a Smalltalk system is message sending. Once the arguments and receiver have been pushed onto the stack we have to specify the message selector and how many arguments it expects before being able to perform the message lookup. The send bytecodes specify the selector via an index into the `CompiledMethod` literal frame.

- Send literal selector - can refer to any of the first 16 literals and 0, 1 or 2 arguments.
- Single extended send - uses the next byte to extend the range to the first 32 literals and 7 arguments.
- Single extended super send - as above but implements a `super` send instead of a normal send, as in the code
  ```
  arf := super fribble.
  ```
- Second extended send - uses the next byte in a different way to encompass 63 literals and 3 arguments.
- Double extended do-anything - uses the two next bytes to specify an operation, an argument count and a literal. For sends, it can cause normal or super sends with up to 31 arguments and a selector anywhere in the first 256 literals. Other operations include pushes, pops and stores. Rumour has it that it can also make tea and butter your toast.

(a) Common selector message sending

There are a number of messages sent so frequently that it saves space and time to encode them directly as bytecodes. In the current Squeak release they are: `+`, `-` `<`, `>`, `<=`, `>=`, `=`, `~=`, `*`, `/`, `\\`, `@`, `bitShift:`, `//`, `bitAnd:`, `bitOr:`, `at:`, `at:put:`, `size`, `next`, `nextPut:`, `atEnd`, `==`, `class`, `blockCopy:`, `value`, `value:`, `do:`, `new`, `new:`, `x`, `y`. Some of these bytecodes simply send the message, some directly implement the most common case(s) (for example see `Interpreter > bytecodePrimAdd`) and have fall-through code to send the general message for any more complex situation.

(1) Returns

There are two basic forms of return

- The method return, from a `MethodContext` back to its sender, commonly seen in code as
  ```
  ^foo.
  ```
  The same form of return will return the value of a `BlockContext` to the sender of its home context, as in the code
  ```
  boop ifTrue:[ ^self latestAnswer]
  ```
  This variety of return is known as a non local return, since it can pass control back to a `MethodContext` many steps up the context stack. The VM return code has to take some pains to handle this eventuality, particularly if the system uses unwind blocks or exception handling. See the `Interpreter>returnValue:to:` method for details.
- The block return, from a `BlockContext` to its caller, as in the code
  ```
  bar := [thing doThisTo: that] value.
  ```

Both return the object at the top of the stack. For performance optimisation there are also direct bytecodes that implement the method return with the receiver, `true`, `false` or `nil` as the return value. Methods that have no explicit return will use the method return of the receiver as the default.

A good, detailed explanation of the operation of the bytecodes can be found in part 4 of "Smalltalk-80, The Language and its Implementation", otherwise known as the Blue Book [GoR83], in the section on the operation of the VM and need not be repeated here.

The use of bytecodes as a virtual instruction set is one of the main factors allowing such broad portability of Smalltalk systems. Along with the reified object format applied to the compiled code and activation records, it ensures that any properly implemented VM will provide the proper behaviour.

For those people that are fans of the idea of implementing Smalltalk specific hardware, it should be noted that the bytecode implementation part of the VM is typically fairly small and simple. A CPU that used Smalltalk bytecodes as its instruction set would still require all the primitives and the object memory code, most of which is quite complex and would probably require a quite different instruction set.

1. Primitives

Primitives are a good way to improve performance of simple but highly repetitive code or to reach down into the VM for work that can only be done there. This includes the accessing and manipulating of the VM internal state in ways not supported by the bytecode instruction set, interfacing to the world outside the object memory, or functions that must be atomic in order to avoid deadlocks.

They are where a great deal of work gets done and typically a Smalltalk program might spend around 50% of execution time within primitives. For up to date details on the precise numbering of primitives in Squeak, refer to the `Interpreter class > initializePrimitiveTable` method.

Primitives are used to handle activities such as:-

a) Input

The input of keyboard and mouse events normally requires very platform specific code and is implemented in primitives such as `Interpreter >primitiveKbdNext` and `primitiveMouseButtons`. In Squeak, these in turn call functions in the platform specific version of sq{platform}Windows.c and associated files - See the chapter on porting Squeak for details.

a) Output

Traditionally Smalltalk has relied upon the BitBlt graphics engine to provide all the visual output it needs. Assorted extra primitives have been made available on some platforms in some implementations to provide sound output, or serial ports or networking etc.

Squeak has a plethora of new output and interface capabilities provided via a named primitive mechanism that can dynamically load code at need.

a) Arithmetic

The basic arithmetic operations for `SmallIntegers` and `Floats` are implemented in primitives. Since the machine level bit representation is hidden from the Smalltalk code, we use primitives to convert the object representation to a CPU compatible form, perform the arithmetic operation and finally to convert the result back into a Smalltalk form. It would be possible to implement most arithmetic operations directly in Smalltalk code by providing only a few primitives to access the bits of a number and then performing the appropriate boolean and bit functions to derive the result; the performance would be unacceptable for the most common case of small integers that can be handled efficiently by a typical CPU. This is why Smalltalk has the special class of integer known as `SmallInteger`; values that can fit within a machine word (along with a tag that allows the VM to discriminate whether the word is an oop or a `SmallInteger`) can be processed more efficiently than the general case handled by `LargePositiveInteger` and `LargeNegativeInteger`. Many arithmetic primitives are also implemented within special bytecodes that can perform the operation if all the arguments are `SmallIntegers`, passing off the work to more sophisticated code otherwise.

Clearly, it takes many more cycles to perform an apparently simple addition of two plain integers than it would in a compiled C program. Instead of

```
      a + b;
```

being compiled to -

```
ADD Result, Rarg1, Rarg2
```

which takes an ARM cpu a single cycle, we have

```
      a + b.
```

compiled to the special bytecode for the message #+, which is implemented in the VM as bytecode 176 and presented here in pidgin C code -

```
case 176: /* bytecodePrimAdd */
      t1 = *(int*)(stackpointer - (1 * 4));
      t3 = *(int*)(stackpointer - (0 * 4));
      /* fetch the two arguments off the stack - note that this
means they must have been pushed before this bytecode! */
```

```
     /* test the two objects to make sure both are tagged as
SmallIntegers, i.e. both have the bottom bit set */
     if (((t1 & t3) & 1) != 0) {
            /* add the two SmallIntegers by shifting each right
one place to remove the tag bits and then add normally */
            t2 = ((t1 >> 1)) + ((t3 >> 1));
            /* then check the result is a valid SmallInteger
value by making sure the top two bits are the same - this handles
both positive and negative results */
            if ((t2 ^ (t2 << 1)) >= 0) {/* If the value is ok,
convert it back to a SmallInteger by shifting one place left and
setting the bottom tag bit, then push it back onto the Smalltalk
stack */
                   *(int*)(stackpointer -= (2 - 1) * 4) = ((t2
<< 1) | 1));
            }
     } else {
            /* defer to code handling more complex cases */
```

This will take at least fourteen operations on most CPUs, even if the memory system can deliver the stack reads and writes in a single cycle. The most plausible time on an ARM is twelve cycles due to the ability to mix shifts and logic operations. The cost of initially pushing the objects onto the stack and of interpreting the bytecodes can be ignored for this example since it makes little overall difference. The important point to make here is that although the costs for adding 1 to 1 and getting 2 seems high, the system will not have any problems adding 24 billion to 5.6×10^{200} and getting a suitable answer, nor in using the `#+` message to combine two matrices. Mostly, it demonstrates yet again the limited utility of most benchmark programs.

a) Storage handling

In order to create an object we need a primitive (`primitiveNew` for objects with only named instance variables and `primitiveNew:` for objects with indexed variables). These primitives work with the object memory to build the object header and to initialise the contents properly as described above. Object creation needs to be atomic so that there is never a time when the garbage collector might have to try to scan malformed memory.

Reading from and writing to arrays with `at:` and `at:put:` is done by primitives that access the indexed variables of the object. If used with an object that has no indexed variables, the primitive fails, leaving the backup Smalltalk code to decide what to do. Reading or writing to `Stream`s on `String`s or `Array`s is very common and two primitives are provided to improve performance in those cases.

Just as with the simple addition example shown above, the number of CPU cycles required for these primitives seems at first glance to be much higher than you might expect from a compiled C program. In the same way, we gain in flexibility, bounds checking and reliability.

a) Process manipulation

Suspending, resuming and terminating `Process`es, as well as the signalling and waiting upon `Semaphore`s are all done in primitives. Changing structures such as the process lists, the VMs idea of the active process and semaphore signal counts needs to be performed atomically, so primitives are used to ensure no deadlocks occur.

a) Execution control

Smalltalk has two unusual ways of controlling execution, evaluating blocks and the `perform:` manner of sending a message.

When a block is sent the message `value` (or indeed `value:` or one of its relatives) as in the phrase

```
    [burp + self wonk] value.
```

the `primitiveValue` (or `primitiveValueWithArgs`) primitive is used to activate the `BlockContext` and thus to set the program counter, stack pointer, caller and then to copy any arguments into the `BlockContext`. Execution then proceeds with the bytecodes of the block until either we return to the caller context (the sender of the `value` message) or the sender of the block's home method (where the block was defined). See also the above section on return bytecodes.

The `perform:` message , as in

```
    arkwright perform: #anatomicallyImpossibleAct.
```

and its siblings, as in

```
    arkwright perform: #ludicrousAct: with: thatThing.
```

allow us to specify a message at runtime rather than compile time and so the primitive has to pull the chosen selector from the stack, slide all other arguments down and then proceed as if the message had been sent in the normal manner.

In general it is best not to use primitives to do complex or heterogenous things; after all, that is something that Smalltalk is good for. A primitive to fill an array with a certain value might well make sense for performance optimisation, but one to fetch, process and render an entire webpage would not, unless you happened to have the code already available in some shared library. In such a case we would use the foreign function interface to make use of outside code.

Primitives that do a great deal of bit-twiddling or arithmetic can also make sense, since Smalltalk is not particularly efficient at that (see above). The sound buffer filling primitive (`PluckedSound>mixSampleCount: into: startingAt: leftVol: rightVol:` is a good example) as are complex vector graphics primitives such as those in the `B3DEnginePlugin 'primitive support'` protocol for examples.

1. Garbage collection

One of the most useful benefits of a good object memory system is that unwanted, or garbage, objects are collected, killed and the memory returned to the system for recycling. There are many schemes for performing this function, and quite a few Ph.D.'s have been awarded for work in this area of computer science. Dr Richard Jones keeps an impressive bibliography of garbage collection literature on his website [Jo99] and is one of the authors of [JoL96], a thoroughly recommended source for further reading.

Garbage collection (GC) is simply a matter of having some way to be able to tell when an object is no longer wanted and then to recycle the memory. Given some known root object(s) it is generally assumed that we can trace all possible accessible objects and then enumerate all those not accessible. If you read [GoR83] you will see that the exemplar system used a reference counting system combined with a mark-sweep system. Most commercial Smalltalks use some variant of a scheme known as Generation Scavenging. Brief explanations of these follow.

a) Reference Counting

Reference counting relies on checking every store into object memory and incrementing a count field (often contained in the header) of the newly stored object and decrementing the count of the over-written object. Any object whose count reaches zero is clearly garbage and must be collected. Any objects that it pointed to must also have their count decremented, with an obvious potential for recursive descent down a long chain of objects.

One problem with a simple reference counting scheme is that cycles of objects can effectively mutually lock a large amount of dead memory since every object in the cycle has at least one reference. Consider a small web of objects with a cycle comprised of C, E and F.

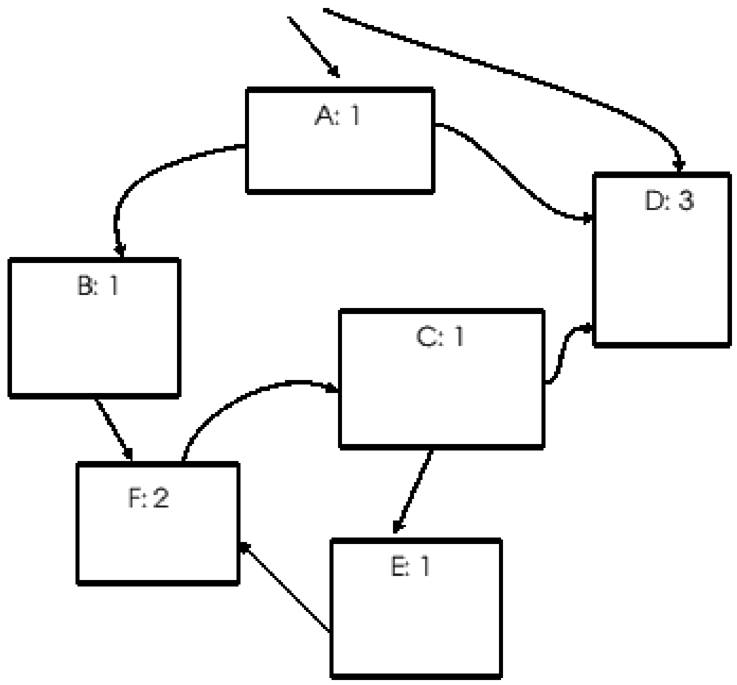

figure 4. Net of object in initial state

If we now replace the reference from B to F, the cycle is orphaned but since each object still has a reference count greater than zero, none of them will be garbage collected.

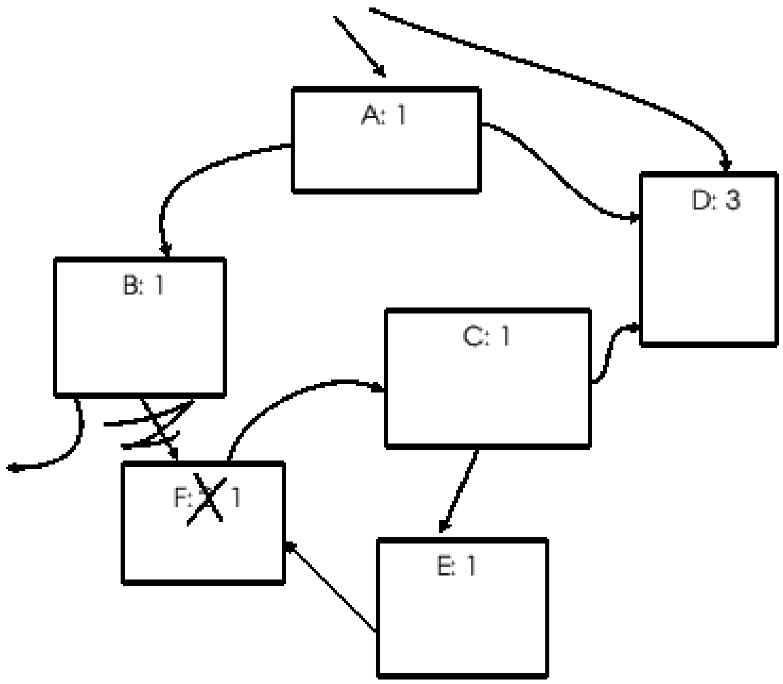

figure 5. Object B no longer points to F - the cycle F/C/E is orphaned

Another problem is that every store into object memory will need the reference incrementing and decrementing to be done, even pushes and pops on the stack. Deutsch and Bobrow [Debo76] developed an improved way of dealing with reference counting the stack, known as *deferred reference counting*. This technique allows us to avoid reference counting pushes and pops by keeping a separate table of objects where the reference count might be zero but that might be on the stack -- if they are on the stack then the actual count cannot be zero. At suitable intervals we scan the stack, correct the count values of the objects in the table and empty the table. Objects that have a zero count at this stage are definitely dead and are removed. Reference counting is still done for all other stores, and those objects that appear to reach a count of zero are added to the table for later processing as above.

a) Mark-Sweep compaction

To avoid completely running out of memory due to cycles, most systems that use reference counting also have a mark-sweep collector that will start at the known important root objects and trace every accessible one, leaving a mark in the object header to check later. In figure 6 the tracing will touch object A, follow its pointer to B and thence out of the diagram. Then the trace will follow the pointer from A to D, where it terminates since D points to no other objects. The second pointer to D will also terminate, but this time because the mark bit in the object header is set. Object F, C and E will not be marked since no other object points to them.

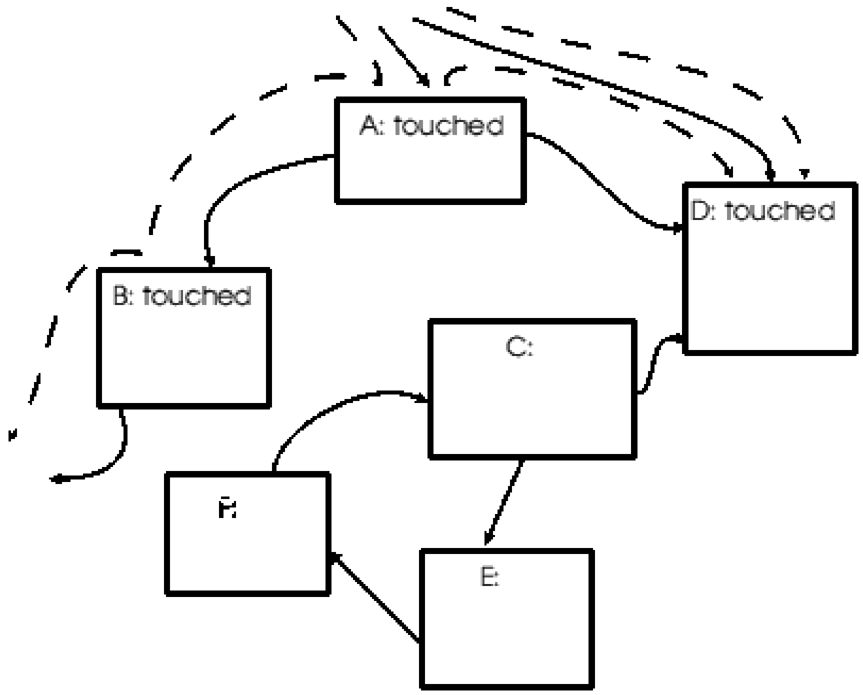

figure 6. Tracing objects in the mark phase

Once the marking phase is completed, every object marked can be swept up (or down) the memory space, effectively removing all the untouched ones, and the mark removed. There are important details to consider with respect to object cycles, which objects are actually roots, how to move the objects memory and update all the reference to it and so on. One useful side effect of the mark-sweep collector is that in compacting all the objects it leaves all the free space in a contiguous lump. In most cases this can be used to make subsequent memory allocation simpler.

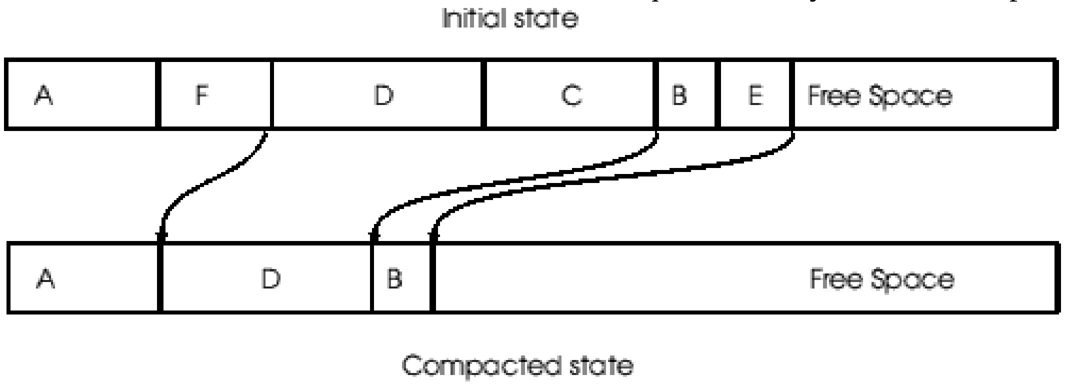

figure 7. Object memory before and after the sweep compaction phase

a) Generation Scavenging

The most commonly used garbage collection scheme in current commercial Smalltalk systems is based on the generation scavenger idea originally developed by David Ungar [Ung87]. He noticed that most new objects live very short lives, and most older objects live effectively forever.

By allocating new objects in a small *eden* area of memory and frequently copying only those still pointed to by a known set of roots into one of a pair of small *survivor* areas, we immediately free the eden area for new allocations. Objects previously in the survivor area that are still live are also copied, leaving one of the survivor spaces empty. Subsequently we keep copying surviving objects from eden and this survivor area into the other and eventually into an *old space* where they are considered tenured. Various refinements in terms of numbers of survivor generations and strategies for moving objects around have been developed.

An important part of a generation scavenger is a table of those old objects (objects existing in old space) into which new objects (objects existing in eden or a survivor space) have been stored. Each time an object is stored into another, a quick check is done to see if the storee is new and the storand is old; this is simpler than the zero count check needed by a reference counting system. If needed, the storand object is added to the table, usually referred to as the *remembered set* or *remembered table*. This table is used as one of the roots when the search for live objects is performed; each of these objects is scanned and new objects they point to are scavenged.

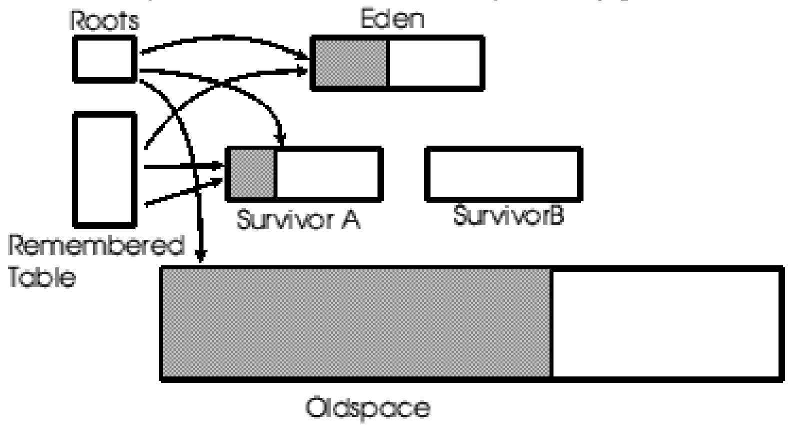

figure 8. Generation scavenging system before a scavenge

In figure 8, notice that the set of roots and the remembered table both refer to objects in eden and a survivor space. The set of roots may also refer to objects in old space. Eden has partially filled with new objects and survivor space A is near empty. Survivor space B is completely empty and not in use at this point.

To perform a scavenge cycle, we first trace the root objects, moving any that are in eden or survivor space A into survivor space B. Then the objects referred to by the remembered set are traced. Once all the new objects pointed to by the table and other roots have been scavenged, the survivors are also scanned and any new objects found are also scavenged. The process continues until no further surviving objects are found. Any old object that no longer points to new objects can be removed from the remembered table.

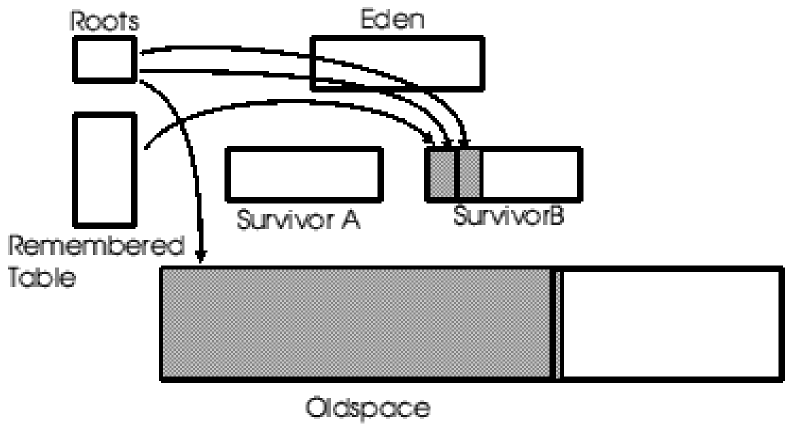

figure 9. Generation scavenging system just after a scavenge

In figure 9, we have just completed a scavenge cycle. Eden is completely emptied, with all surviving object moved to survivor space B. Still-live objects from survivor space A have also been copied to B, or to old space if they met suitable criteria. The remembered table has lost some entries as objects die and get left behind. The next scavenge cycle will copy objects into survivor space A.

Note that scavenging a generation is similar to a mark and sweep of the memory in which the generation lives, though the objects get moved to a different space rather than simply compacted within the same space. The chief benefit of scavenging comes from the reduction in amount of memory touched; only a few live objects are moved in most cases rather than every object being touched to trace through it.

A. Why is an Object Engine better than C and a library?

One might reasonably ask why this rather complex seeming system is better than a 'normal' programming environment and a C library.

The most obvious answer is that the message sending support allows, helps, and strongly encourages an object oriented form of programming. Good object oriented programs form extremely useful libraries that can increase programming productivity greatly. Although it is possible to use almost any language to work in an object oriented style, doing so without proper support makes it much harder than just 'going with the flow' and slipping into procedural programming.

1. Memory handling

Much of the code in a complex C program is taken up with storage management; memory allocating, freeing, initialising, checking and error handling for when it goes wrong. A good VM will handle all this automatically, reliably (making some assumptions about the quality of the VM) and transparently. Although many C programs do not bother to check the bounds of arrays, exceeding those bounds is a major cause of problems. Consider how many computer viruses are spread through or otherwise rely upon buffer overflow and other bounds problems. A Smalltalk VM checks the bounds of any and all accesses to its objects, thus avoiding this problem completely. Furthermore, since an object is absolutely a member of its class (there is no concept of 'casting' in Smalltalk) you cannot fool the system into allowing you to write to improper memory locations. This *referential safety* is a very useful property of Smalltalk systems.

1. Meta-programming capability

One of the great virtues of a virtual machine is that you can precisely define the lowest level behaviour that the higher level code can see. This allows us to provide a reflective system with meta-programming capabilities and thus to write programs that can reason about the structure of the system and its programs. One good example is the Smalltalk debugger. Since the structure of and interface to the execution records (the `BlockContext` and `MethodContext` instances) is defined within Smalltalk, we can manipulate them with a Smalltalk program. We can also store them in memory or in files for later use, which allows for remote debugging and analysis.

1. Threads and control structures programmer accessible

When it is possible to cleanly manipulate the contexts, we can add new control structures. See the article "Building Control Structures in the Smalltalk-80 System" by Peter Deutsch in [SCG81] which illustrates this with examples including case statements, generator loops and coroutining. With only a tiny amount of support in the VM, it is possible to add threads to the system; although they are known as `Process`es within the class hierarchy.

1. Portability of the system

In much the same way that the VM allows meta-capability, it can assist in providing high levels of portability. The interface to the machine specific parts of the VM is uniform, the data structures are uniform and thus it is possible to make a system that has total binary portability. No recompiling, conversion filters or other distractions are needed. Squeak, like several commercial Smalltalks, allows you to save an image file on a PC, copy it to a Mac, or a BeOS, or Acorn, or almost any Unix machine and simply run it. The bits are the same and the behaviour is the same, barring of course some machine capability peculiarities.

A. Squeak VM peculiarities

Squeak's VM departs from the design shown in the Blue Book [GoR83] in quite a few ways. It has a quite different object memory format and uses an interesting variant of mark-sweep garbage collection that approaches generation scavenging in many respects. There are many new primitives, often implemented in dynamically loadable VM plugins, a form of shared library or DLL developed by the Squeak community. A completely new form of graphics engine has been introduced, including 3D capabilities. There is extensive sound support and full internet connectivity.

Perhaps most radically for the VM per se, the kernel of the VM and most of the code for the assorted plugins is actually implemented as Smalltalk code that is translated to produce C source code to compile and link with the lowest level platform specific C code.

1. Object format

The Blue Book definition of the ObjectMemory used an Object Table (OT) and a header word for each object to encode the size and some flags. Each object table entry contained some flags (the reference count for example) and a pointer to the memory address for the body of the object. Any access to the instance variables or class of the object required indirection through the object table to find the body.

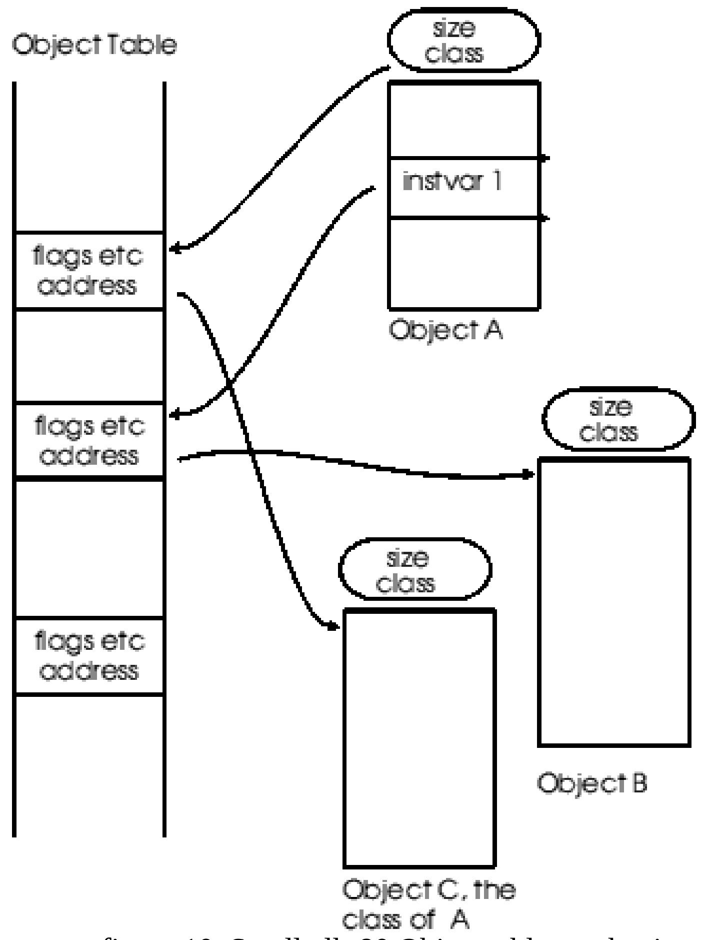

figure 10. Smalltalk-80 Object table mechanism.

In figure 10 we see that to find the class, object 'C', of object 'A', we have to use the oop in the object header to be able to read the entry in the OT and thereby find the address of the body of the class. In a similar manner we have to indirect through the OT to find the body of object 'A's instvar1. When looking up methods as part of a message send, we have to indirect several times in order to find the class of the receiver, the MethodDictionary of that class and so on.

A sometimes useful attribute of an OT is that objects once created keep the same oop throughout their existence, making the oop a very acceptable hash value for most cases.

Squeak uses a more direct method, whereby the oops are actually the memory addresses of a header word at the front of the object body. This saves a memory indirection when accessing the object, but requires a more complicated garbage collection technique because any object that points to a moved object will need the oop updating.

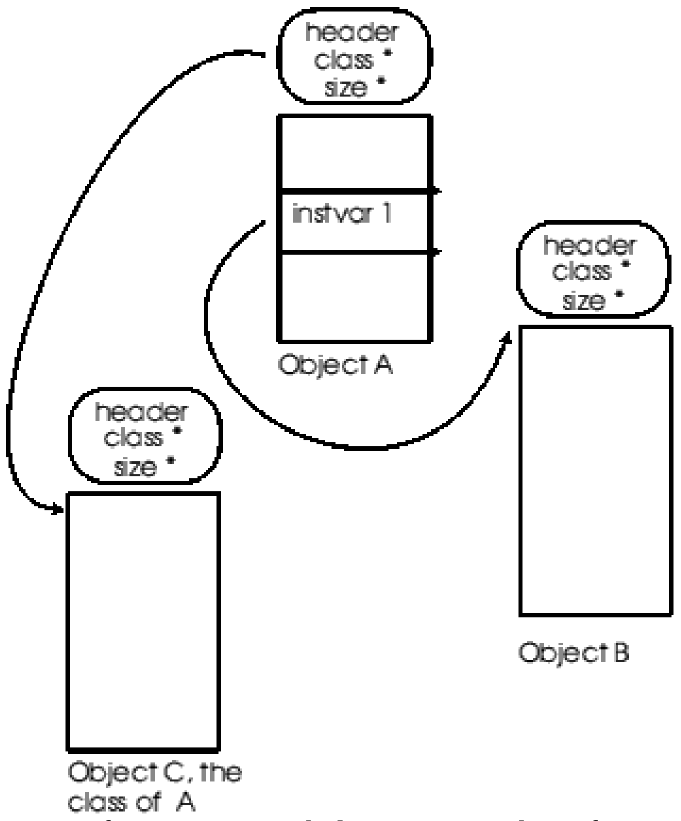

figure 11. Squeak direct pointer object format
* note variable object header format describe below

Since we still need to have the class oop, size and some flags available for each object, Squeak generally uses a larger header. The canonical header is three words:- flags/hash/size encoded in one word, class oop, size. However as a high proportion of objects are instances of a small set of common classes, and since few objects are large, it was decided to use three header formats as follows:-

- One word - all objects have this header.
  - 3 bits reserved for GC state machine (mark, old, dirty)
  - 12 bits object hash (for hashed Set usage)
  - 5 bits compact class index, non-zero if the class is in a group of classes known as 'Compact Classes'
  - 4 bits object format
  - 6 bits object size, in 32-bit words
  - 2 bits header type (0: 3-word, 1: 2-word, 2: free chunk of memory, not an object at all, 3: 1-word)
- Two word - objects that are instances of classes not in the compact classes list. This second word sits in front of the header shown above.
  - 30 bits oop of the class
  - 2 bits header type as above
- Three word - objects that are too big for the size to be encoded in the one-word header, i.e. more than 255 bytes. This third word sits in front of the two shown above.
  - 30 bits of size
  - 2 bits header type as above

Figure 12 illustrates the general case three word header.

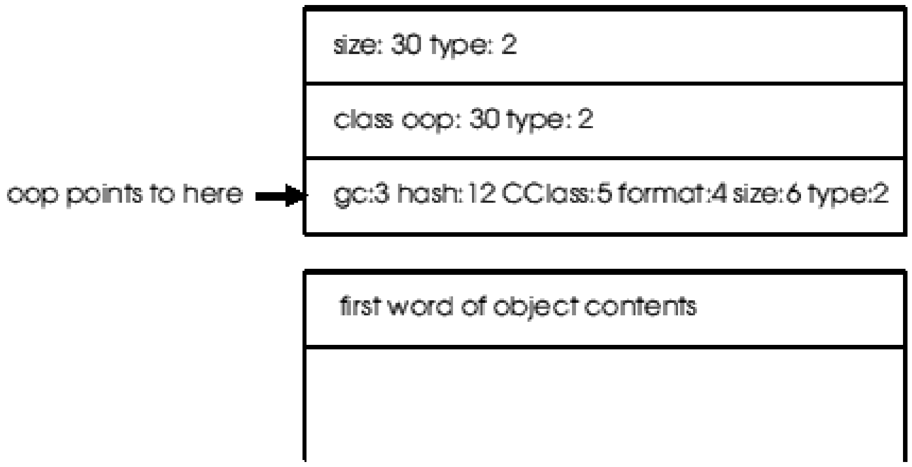
size: 30 type: 2
class oop: 30 type: 2
oop points to here
gc:3 hash:12 CClass:5 format:4 size:6 type:2
first word of object contents

figure 12. Squeak object header layout

The set of compact classes that can be used in the compact headers is a Smalltalk array that can be updated dynamically; you can even decide that no classes should qualify. See the methods `Behavior > becomeCompact` and `becomeUncompact` as well as `ObjectMemory > fetchClassOf:` and `instantiateClass:indexableSize:` for illustration. Whether the space savings are worth the complexity of the triple header design is still an open question in the Squeak systems community. With the flexibility to designate classes as compact or not on the fly, definitive experiments can someday answer the question.

1. Garbage Collection

Squeak uses an interesting hybrid of generation scavenging and mark-sweep collection as its garbage collector. Once the image is loaded into memory the end address of the last object is used as a boundary mark to separate two generations, old objects existing in the lower memory area and new objects are created in the upper memory area.

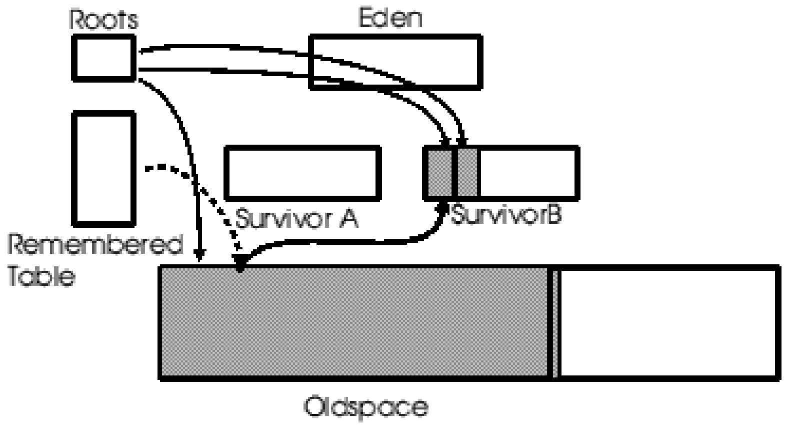

figure 13. Squeak memory organization

When the remaining free memory runs low, or a preset number of objects have been allocated, the VM will pause to garbage collect the new object region (`ObjectMemory > incrementalGC`). This is done with a Mark-Sweep algorithm modified to trace only those objects in the new region, thus touching considerably less memory than an entire-image sweep. As in the generation scavenging scheme described above, the remembered table (the `rootTable` instance variable in `ObjectMemory`) is maintained of old objects that might point to new objects for use as roots when tracing which objects still live. These are added to other important root objects such as the active context and the specialObjects array and used as starting points for the depth first tracing implemented by `ObjectMemory > markPhase`.

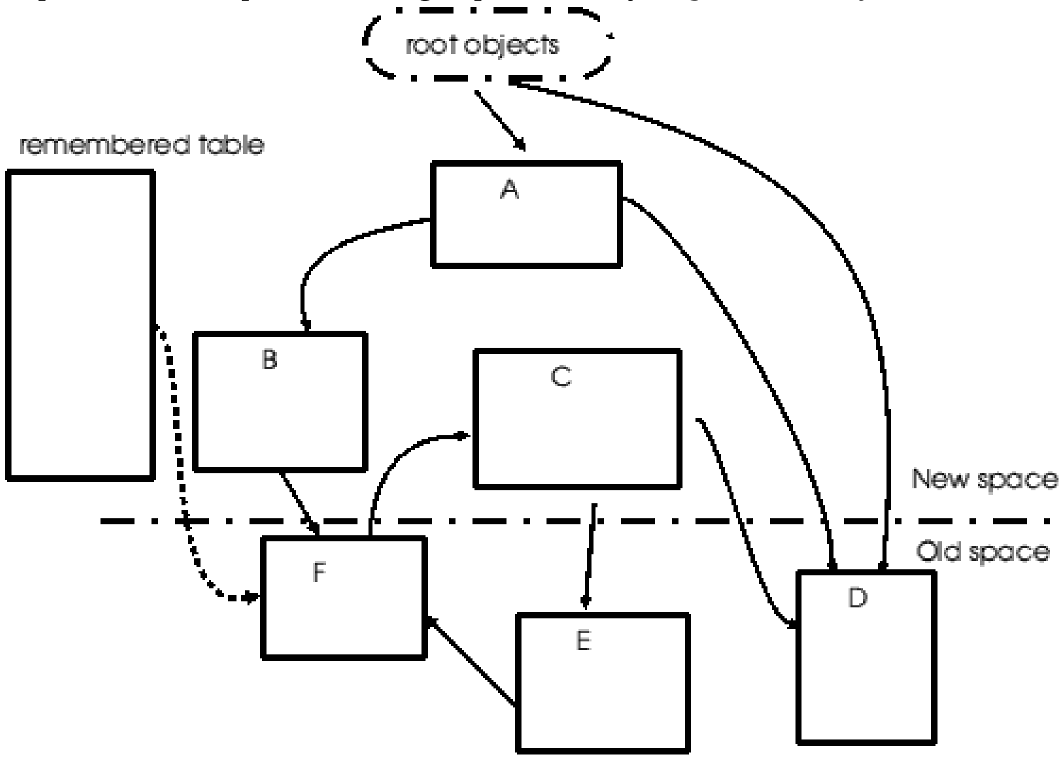

figure 14. Tracing only new objects in Squeak

Tracing starts with the root objects, touching 'A', then 'B' and stopping there since 'F' is an old object. 'D' would not be traced since it is also old. When tracing from the remembered table, we would touch 'C' since it is pointed to by 'F'.

Once the tracing phase has completed the memory region is swept from bottom to top with each object header being examined to see it has been marked. Those that have not been touched are tagged as free chunks of memory. Then a table of address mappings is built , listing each surviving object and the new address it will have. Once all these mappings are known the survivors are updated so that all their oops are correct and finally they are moved down memory to the new addresses. By the end of the compaction the freed memory is all in one chunk at the top of the new region.

This process might seem complicated but the use of the two generations means that it can typically run in a very short time; on an Acorn 200MHz StrongARM machine an average figure is 8-9 ms, on an Apple 400MHz G3 PowerBook it is 3ms. During activities like typing or browsing and coding the system will run an incremental garbage collect one to five times per second. Obviously, the more garbage is created, the more time will be spent on collecting it.

One limitation of an incremental collection is that any objects from the old region that are no longer referenced do not get freed and collected. Of course, the observation that lead to generation scavenging tells us that this usually doesn't matter since old objects generally continue to live, but sometimes we do need to completely clean out the object memory. Many systems that use a generation scavenger for incremental garbage collection also have a secondary collector based on a mark-sweep algorithm. Squeak simply forces the system to believe that the old/new boundary is at the bottom of the old region and thus the entire image is now considered to be new objects and all objects will be fully traced. Look at the `ObjectMemory > fullGC method` for details.

1. Extra Primitives

Squeak has added many primitives to the list given in the Blue Book. They include code for serial ports, sound synthesis and sampling, sockets, MIDI interfaces, file and directory handling, internet access, 3D graphics and a general foreign function interface. Most of them are implemented in VM plugins - see the chapter 'Extending the Squeak VM' for an explanation of the VM plugin mechanism that is used to implement these extra capabilities.

1. Speeding up at: & at:put:

The messages `at:` and `at:put:` are very commonly used - as mentioned above they are both already installed as special bytecode sends. Squeak uses a special cache to further try to speed them up. See `Interpreter > commonAt:` and `commonAtPut:` for the implementation details.

1. Extended BitBlt and Vector Graphics extensions

The original BitBlt worked on monochrome bitmaps to provide all the graphics for Smalltalk-80. Squeak has an extended BitBlt that can operate upon multiple bit-per-pixel bitmaps to provide colour. It can also handle alpha blending in suitable depth bitmaps, which can, for example, give anti-aliasing. Different depth bitmaps can be mixed and BitBlt will convert them as required. New extensions to BitBlt allow for mixed pixel endianness as well as depth, and make external OS bitmaps accessible so that graphics accelerator hardware can be used when provided by the host machine. WarpBlt is a variant of BitBlt that can perform strange transforms on bitmaps by mapping a source quadrilateral to the destination rectangle and interpolating the pixels as needed. There are demonstration statements in the 'Welcome To...' workspace which show some extremes of the distortions possible, but simple scaling and rotation is also possible.

Perhaps most exciting, there is a new geometry based rendering system known as Balloon, which can be used for 2D or 3D graphics. This has introduced sophisticated vector based graphics to Smalltalk and is the basis of the Alice implementation included in the system (see the chapter 'Alice in a Squeak Wonderland' later in this book).

1. VM kernel written in Smalltalk

The VM is largely generated from Squeak code that can actually run as a simulation of the VM, which has proven useful in the development of the system. See the `Interpreter` and `ObjectMemory` classes for most of the source code.

Note how the VM is written in a fairly stilted style of Smalltalk/C hybrid that has come to be known as *Slang*. Slang requires type information hints that are passed through to the final C source code anytime you need variables that are not C integers. A somewhat more readable dialect of Slang is used in the VM plugins described in the chapter on extending the VM.

You can try out the C code generator by looking at the classes TestCClass1/2 or 3. Printing

```
 TestCClass2 test
```

will run a translation on a fairly large example set of methods and return the source code string that would be passed to a C compiler. To build a new VM for your machine, see the chapter on porting Squeak for instructions on how to generate all the files needed and compile them.

A. Things for the future

Squeak is not finished and hopefully never will be. Already it represents one of the most interesting programming systems in existence and people are constantly expanding their vision of what it might be used for. The following are some notes on what might be done in pursuit of a few of the author's interests.

1. Bigger systems

Probably the largest known Squeak image is the one used by Alan Kay for public demos. Without any special changes to the VM it was quite happy with over a hundred Projects and one hundred and sixty megabytes of object space. To extend Squeak's range to truly large systems we would probably need to consider changing to a 64bit architecture and a more sophisticated garbage collection system. Large systems are often very concerned with reliability and data safety - a corporate payroll system ought not crash too often. An object memory that incorporates extra checking, transaction rollbacks, logging, interfaces into secure databases etc. might be an interesting project.

1. Smaller systems, particularly embedded ones.

It is possible to squeeze a usable Squeak development image down to approximately 600kb, although there is not very much left by then. To get to this size involves removing almost everything beyond the most basic development environment and tools - look at the `SystemDictionary > majorShrink` method.

Building a small system for an embedded application would involve a more sophisticated tool, such as the `SystemTracer`, to create the image, but what Object Engine changes would be useful?

Since such a system would not need any of the development tools, we could remove some variables from, or change the structure of, classes and methods and method dictionaries.

From classes we could almost certainly remove,

- organisation - this simply categorizes the class's methods for development tools
- name - it is unlikely that a canned application would have any use for this
- instanceVariables - this is a string of the names of the instance variables
- subclasses - the list of subclasses is rarely accessed.

None of these would require major VM changes except in some debugging routines such as `Interpreter > printCallStack` that attempt to extract a class' name.

From compiled methods we could remove the source code pointer and possibly go so far as to use integers instead of symbols to identify the message selectors. This would work in Squeak since the method dictionary search code already handles such a possibility. This would reduce the size of the Symbol global table significantly.

`MethodDictionary` relies on having a size that is a power of two for the purposes of the message lookup algorithm and will double the size of the dictionary anytime it gets to 3/4 full - typically the dictionaries are a little under half full. By changing the system to allow correctly sized dictionaries we might save some crucial tens of kilobytes.

Depending on the application involved, it might also be possible to make most of the classes in use be compact classes. This would allow most objects to have one word object headers and reduce the average object overhead in the system.

The VM itself can be shrunk quite simply. In Squeak 2.8, the VM was broken into a number of plugin modules to accompany the kernel. Systems that do not need sound, sockets or serial ports need not have any of the code present. A Linux VM can be built as small as 300Kb with just the basic kernel capabilities.

1. Headless server systems.

If we want to use Squeak as a server program we need to be able to run headless, which is to say without a GUI.

Most of the changes needed are in the image, in low-level code just outside what we would normally consider part of the object engine. For example, there are methods in the FileStream classes that open dialogues to ask a user whether to overwrite a file or not; clearly this is not appropriate in a server. Proper use of `Exceptions` and exception handlers would be required to correct this problem, with probable changes to code in the object engine in order to signal errors and to raise some of the appropriate exceptions.

One major change needed in the VM is to avoid attempting to open the main Squeak window as soon as Squeak starts up. Headless systems would not need nor support this window. The Acorn port already handles this by not creating and opening the window until and unless the `DisplayScreen > beDisplay` method is executed.

Another important capability would be to communicate with the running system via some suitable channel. Although Squeak already supports sockets, it seems that an interface to Unix style stdin/stdout streams would be useful.

1. Faster systems.

Although Squeak is amazingly fast - any reasonably modern machine can execute bytecodes faster than most machines could execute native instructions just a few years ago - we would like still more performance.

The Squeak VM is a fairly simple bytecode interpreter, with some neat tricks in the memory manager and primitives to help the speed. To drastically improve performance we would need to move to a quite different execution model that can remove the bytecode fetch-dispatch loop costs, improve the primitive calling interface speed, reduce the time spent pushing and popping the stack and reduces the runtime cost of having reified contexts. At the same time, this new system must not break any system invariants nor reduce the portability.

The overhead of using bytecodes can be almost eliminated by using some form of runtime translation to native machine code [DeS84]. The VM would need extending to provide a subsystem that can use the bytecodes as input to a machine specific compiler. This has been done in most commercial Smalltalks; it adds a considerable degree of complexity to the VM and particularly to porting the VM to a new machine. The porter has to know the machine CPU architecture and the VM meta-architecture well enough to combine them. Subtleties of the CPU can become major problems - are registers usable for both data and addresses? Does the data cache interact with the instruction cache in a useful manner or not? Do these details change across models of the CPU?

Since the Squeak execution model is a simple stack oriented machine and most modern CPUs are register oriented, we lose a lot of time in manipulating a stack instead of filling registers. When code is translated to native instructions, we can avoid a lot of this by taking advantage of the opportunity to optimise out many stack movements. This will likely interact with the use of Contexts with their distinct individual stackframes and potentially non-linear caller/sender relationships; CPUs normally work with a single contiguous stack and a strict call/return convention. Fortunately most Smalltalk code is quite straightforward and it is possible to implement a Context cacheing system that performs almost as well as an ordinary stack and yet handles the exceptional conditions caused by BlockContext returns or references to the activeContext [Mi87].

The best news of all is that by the time this book is published, Squeak will very likely have a VM making use of all these performance improving techniques!

A. References

GoR83

Adele Goldberg & David Robson, "Smalltalk-80: the Language and its Implementation", Addison-Wesley May 1983

Currently out of print. A second, smaller edition was published as -

GoR89

Adele Goldberg & David Robson, "Smalltalk-80: The Language", Addison-Wesley 1989

The section on Smalltalk implementation that was removed from GoR83 is available online at http://users.ipa.net/~dwighth/

DeS84

L. Peter Deutsch & Allan M. Schiffman, "Efficient Implementation of the Smalltalk-80 system", Proc. POPL 1984.

DeBo76

L. Peter Deutsch and Daniel G. Bobrow, "An efficient, incremental, automatic garbage collector", Comm.ACM Sept 1976

SCG 81

Members of the Xerox Palo Alto Research Centre systems Concepts Group, Byte August 1981 special edition on Smalltalk-80.

Ung87

David Ungar, "The Design and Evaluation of a High Performance Smalltalk System", Addison-Wesley 1987.

Mi87

Eliot E. Miranda, "BrouHaHa - A portable Smalltalk interpreter", Proc.OOPLSA 1987.

JoL96

Richard Jones and Rafeal Lins, "Garbage collection: algorithms for automatic dynamic memory management", Wiley 1996

Jo99

Richard Jones' website at :-
`http://www.cs.ukc.ac.uk/people/staff/rej/gcbib/gcbib.html`

1. Bibliography: Papers of interest not directly referenced in the text

Glenn Krasner, ed. "Smalltalk-80: Bits of History, Words of Advice", Addison-Wesley 1983

Patrick J. Caudill and Allen Wirfs-Brock, "A third generation Smalltalk-80 implementation", Proc.OOPSLA 1986.

Alan C. Kay, "The early history of Smalltalk", ACM SigPLAN Notices March 1993

David M. Ungar, "Generation Scavenging: A non disruptive high performance storage reclamation algorithm", ACM Practical Programming Environments Conference, pp153-173, April 1984.