HOME

PAGES
About Cog
About this blog
Building a Cog
Development Image
Cog Projects
Collaborators
Compiling the VM
Downloads
Eliot Miranda
On-line Papers and Presentations

CATEGORIES
Cog
Spur

SEARCH
Find

{ 2009 01 14 }

Under Cover Contexts and the Big Frame-Up

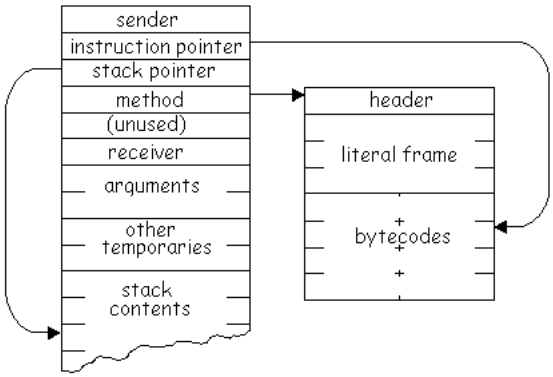
Nobody expects the Smalltalk language system. Our chief weapon is polymorphism… polymorphism and objects…objects and polymorphism. Our two weapons are objects and polymorphism… and inheritance… Our three weapons are objects, and polymorphism and inheritance… and an almost fanatical devotion to contexts…
Our four…no…
Amongst our weapons… Amongst our weaponry… are such elements as objects, po…
I'll come in again.

Contexts are Smalltalk's first-class activation records. thisContext is the "reserved word" (technically, pseudo-variable) for the current activation record. Try "thisContext inspect" and see what you get. Then try "thisContext inspect. self halt" and see what you get.

Contexts are fabulous weapons, even better programming building blocks, and *horribly* expensive. How do I love contexts? Let me count the ways…

Contexts mean Smalltalk has had edit-and-continue debugging since the 1970's.
Contexts allow the implemention of an exception system with no additional support from the virtual machine.
Contexts allow the implemention of dynamic binding, co-routines, tail-recursion-elimination and backtracking with no additional support from the virtual machine.
Contexts enable process persistence and migration.
Contexts allow implementation of continuations (full, delimited/partial or otherwise) without additional VM support (and hence the Seaside web framework).
Contexts are consistent (consistent) with the rest of the "objects all the way down" system, providing activations as first-class objects.
Because they're there.

What does a context look like?



The method is the code to execute. The stack is of finite size, guaranteed by the bytecode compiler. The sender is the caller context, providing a simple and flexible "spaghetti stack" scheme.

But a naive implementation has to allocate a context on each send, move the receiver and arguments from the stack of the caller context to that of the callee, and assign the callee's sender with the caller. For essentially every return the garbage collector eventually has to reclaim, and every return has to nil the sender and instruction pointer fields of, the context being returned from. Here's the blue book code for method activation:

```
activateNewMethod
| contextSize newContext newReceiver |
(self largeContextFlagOf: newMethod) = 1
ifTrue: [contextSize := 32 + TempFrameStart]
ifFalse: [contextSize := 12 + TempFrameStart].
newContext := memory
instantiateClass: ClassMethodContextPointer
withPointers: contextSize.
memory storePointer: SenderIndex
ofObject: newContext
withValue: activeContext.
self storeInstructionPointerValue: (self initialInstructionPointerOfMethod: newMethod)
inContext: newContext.
self storeStackPointerValue: (self temporaryCountof: newMethod)
inContext: newContext.
memory storePointer: MethodIndex
ofObject: newContext
withValue: newMethod.
self transfer: argumentCount + 1
fromIndex: stackPointer – argumentCount
ofObject: activeContext
toIndex: ReceiverIndex
ofObject: newContext.
self pop: argumentCount + 1.
self newActiveContext: newContext
```

If you want to stay with this scheme but make it faster then you could do worse than the current Squeak implementation:

```
activateNewMethod
| newContext methodHeader initialIP tempCount nilOop where |

        methodHeader := self headerOf: newMethod.
newContext := self allocateOrRecycleContext: (methodHeader bitAnd: LargeContextBit).

        initialIP := ((LiteralStart + (self literalCountOfHeader: methodHeader)) * BytesPerWord) + 1.
tempCount := self temporaryCountOfMethodHeader: methodHeader.

        "Assume: newContext will be recorded as a root if necessary by the
call to newActiveContext: below, so we can use unchecked stores."

        where := newContext + BaseHeaderSize.
self longAt: where + (SenderIndex << ShiftForWord) put: activeContext.
self longAt: where + (InstructionPointerIndex << ShiftForWord) put: (self integerObjectOf: initialIP).
self longAt: where + (StackPointerIndex << ShiftForWord) put: (self integerObjectOf: tempCount).
self longAt: where + (MethodIndex << ShiftForWord) put: newMethod.
self longAt: where + (ClosureIndex << ShiftForWord) put: nilObj.

        "Copy the reciever and arguments…"
0 to: argumentCount do:
[:i | self longAt: where + ((ReceiverIndex+i) << ShiftForWord) put: (self stackValue: argumentCount-i)].

        "clear remaining temps to nil in case it has been recycled"
nilOop := nilObj.
argumentCount+1+ReceiverIndex to: tempCount+ReceiverIndex do:
[:i | self longAt: where + (i << ShiftForWord) put: nilOop].

        self pop: argumentCount + 1.
reclaimableContextCount := reclaimableContextCount + 1.
self newActiveContext: newContext
```

The variable reclaimableContextCount counts how many contexts have been allocated that are not referenced by other objects and so may eagerly be reclaimed on return. Whenever thisContext is referenced or a block created reclaimableContextCount gets set to zero. On return if reclaimableContextCount is non-zero the returning context is reclaimed and reclaimableContextCount is decremented.

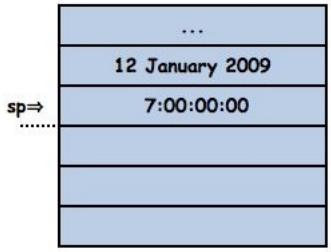
Again in a naive implementation pushing an object on the stack involves a store check. David Ungar's Berkeley Smalltalk nilled out the stack slot when popping off an entry so that a push would only have to increase the reference count of the object pushed instead of also decrementing the count of the object overwritten. The Squeak VM uses a generational collector so there is no reference counting but it does need to make the context a root if it is old (in newActiveContext:) to avoid the store-check on every push. It also carefully protects against access beyond the stack pointer through checks in the at: and at:put: primitives so that allocateOrRecycleContext: does not have to initialize the stack with nils.

But these are all desperate attempts at mitigating the cost of a structure which for the most part is used in last-in, first-out (LIFO) order and only occasionally used to do the cool stuff we love it for. So here's my evolution of Peter Deutsch's and Allan Schiffman's original scheme for having one's cake and eating it too, for providing the illusion of ever-present contexts while only creating them when necessary. This is the fourth variant I've come up with, a hybrid of my earlier schemes and Peter and Allan's, and like any hybrid it is vigourous and healthy 🙂 What's really different about my scheme, unlike Peter and Allan's, is that contexts don't get divorced when you send them a message, a cryptic comment I'll explain presently.

Stack Organization

The basic idea is to organize method activations using a stack discipline and to allow these to be accessed through contexts. The stack discipline is no different to that in conventonal language implementations. Outgoing arguments get pushed on the stack and on activation a frame is built immediately following the arguments which it references via the stack or frame pointer. On return the frame is popped off the stack and the stack pointer and frame pointer revert to referencing the caller frame; no different to stack management in e.g. Pascal or C. The wrinkle is in allowing context objects to be created and manipulated as if they were conventional contexts even though they are stand-ins for underlying stack frames.

Let's look at some activation/return sequences to understand it properly. Here's a stack which has some arguments for a send pushed on it by some already active method, it is about to send + in DateAndTime today + (Duration weeks: 1).



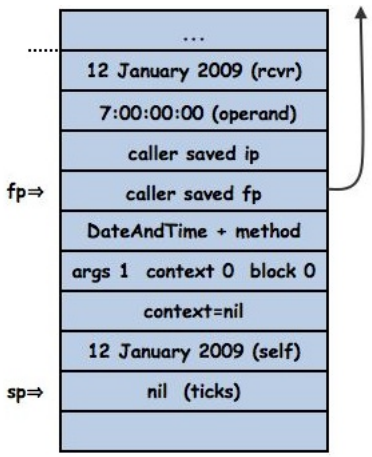
Here's the DateAndTime>>+ method:

DateAndTime methods for ansi protocol
+ operand
"operand conforms to protocol Duration"

```
	| ticks |
ticks := self ticks + (operand asDuration ticks) .
```

```
	^ self class basicNew
ticks: ticks
offset: self offset;
yourself
```

When the message is sent the intetpreter looks up #+ in dateAndTime and finds the method. To build the frame for the method the interpreter pushes its current instruction pointer and frame pointer, saving them for the eventual return, then pushes the method, a flag word and a slot to hold the frame's context if it ever needs one. Finally it pushes the receiver (again) and initializes temporaries (in this case just "ticks") to nil.



all of which is done by the following method

StackInterpreter methods for message sending
internalActivateNewMethod
| methodHeader numTemps rcvr errorCode |
<inline: true>

```
	methodHeader := self headerOf: newMethod.
numTemps := self tempCountOfMethodHeader: methodHeader.
```

```
	rcvr := self internalStackValue: argumentCount. "could new rcvr be set at point of send?"
```

```
	self internalPush: localIP.
self internalPush: localFP.
localFP := localSP.
self internalPush: newMethod.
method := newMethod.
self internalPush: (self
encodeFrameFieldHasContext: false
isBlock: false
numArgs: (self argumentCountOfMethodHeader: methodHeader)).
self internalPush: nilObj. "FxThisContext field"
self internalPush: rcvr.
```

```
	"Initialize temps..."
argumentCount + 1 to: numTemps do:
```

```
[:i | self internalPush: nilObj].

	"-1 to account for pre-increment in fetchNextBytecode"
localIP := self pointerForOop: (self initialPCForHeader: methodHeader method: newMethod) - 1
```

The flag word, tells the interpreter three things, how many arguments the method has, whether the context field is valid or not and whether the frame is a method or block activation.

Each of these fields is a single byte. The interpreter uses the args field to decide where to find a temporary variable. The bytecode set uses push/Store/StorePopTemporary bytecodes to access both arguments and temporaries since in a context the arguments and temporaries are adjacent. But in this stack organization the arguments are at the top of the frame and the temporaries below, so the argument count determines where to find temporaries and hence having fast access to the argument count is essential to access arguments and temporaries with acceptable speed:

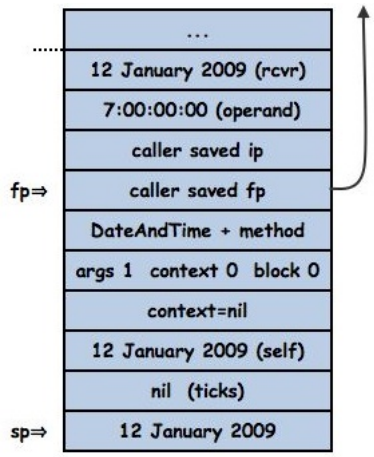
```
*StackInterpreter methods for frame access*
**frameNumArgs:** theFP
"See encodeFrameFieldHasContext:numArgs:"
<inline: true>
<var: #theFP type: #'char *'>
^stackPages byteAt: theFP + **FoxFrameFlags** + 1


*StackInterpreter methods for internal interpreter access*
**temporary:** offset **in:** theFP
"See StackInterpreter class>>initializeFrameIndices"
| frameNumArgs |
<inline: true>
<var: #theFP type: #'char *'>
^offset < (frameNumArgs := self frameNumArgs: theFP)
ifTrue: [stackPages longAt: theFP + **FoxCallerSavedIP** + ((frameNumArgs - offset) * **BytesPerWord**)]
ifFalse: [stackPages longAt: theFP + **FoxReceiver** - **BytesPerWord** + ((frameNumArgs - offset) * **BytesPerWord**)]

**internalPush:** object
"In the StackInterpreter stacks grow down."
stackPages longAtPointer: (localSP := localSP - **BytesPerWord**) put: object


*StackInterpreter methods for stack bytecodes*
**pushTemporaryVariable:** temporaryIndex
self internalPush: (self temporary: temporaryIndex in: localFP)


**pushTemporaryVariableBytecode**
<expandCases>
self fetchNextBytecode.
"this bytecode will be expanded so that refs to currentBytecode below will be constant"
self pushTemporaryVariable: (currentBytecode bitAnd: 16rF)
```

The receiver is pushed also, partly for reasons of efficiency and, as we'll see later, to support blocks. Pushing the receiver after the context field allows the interpreter to find the receiver at a known offset relative to the frame pointer, avoiding having to fetch frameNumArgs for instance variable access:

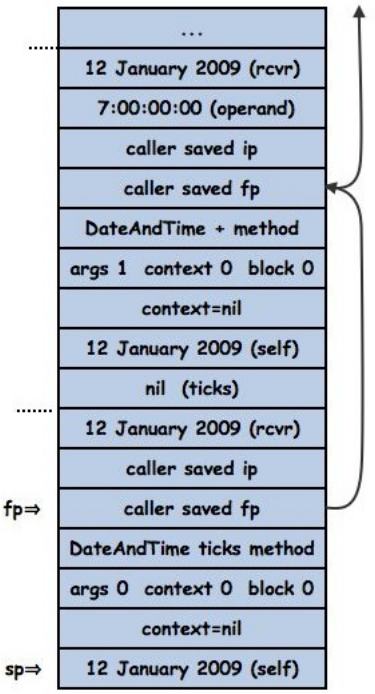
```
*StackInterpreter methods for frame access*
**frameReceiver:** theFP
<inline: true>
<var: #theFP type: #'char *'>
^stackPages longAt: theFP + **FoxReceiver**

*StackInterpreter methods for stack bytecodes*
**pushReceiverVariable:** fieldIndex
self internalPush: (self fetchPointer: fieldIndex ofObject: self receiver)

**pushReceiverVariableBytecode**
<expandCases>
self fetchNextBytecode.
"this bytecode will be expanded so that refs to currentBytecode below will be constant"
self pushReceiverVariable: (currentBytecode bitAnd: 16rF)
```

We arrange that stacks grow down because in the JIT they definitely will. Call, return, push and pop instructions on CISC processors (notably x86/IA32) define that stacks grow downwards, and the JIT will reuse much of the StackInterpreter's machinery. Whenever I talk about a frame being above another one I mean it is later, actually at a lower address, but logically above, towards the head frame and away from the base frame; the "hottest" element on the stack is still called the top of stack.

Continuing with the execution once DateAndTime>>+ has pushed self (12 January 2009) it is ready to send ticks:



The pushed 12 January 2009 becomes the receiver of the new activation of DateAndTime>>#ticks. The saved instruction pointer is the pointer to the bytecode following the send of ticks. The saved frame pointer is the frame pointer for the DateAndTime>>#+ frame, then comes the DateAndTime>>#ticks method, a flag word indicating no arguments, an invalid context field and the 12 January 2009 receiver:

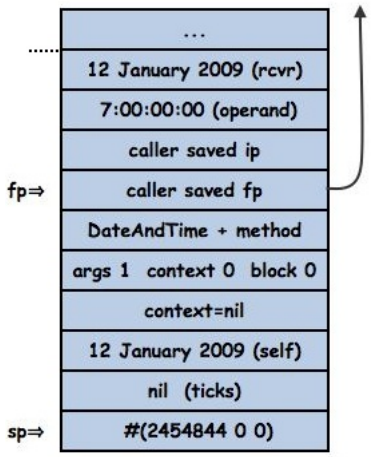


The ticks method will eventually compute #(2454844 0 0)

DateAndTime methods for private
ticks
"Private – answer an array with our instance variables. Assumed to be UTC "

```
^ Array with: jdn with: seconds with: nanos
```

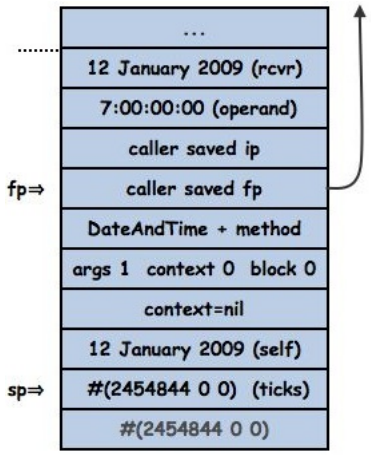
Its return bytecode dismantles the frame, assigning the frame pointer to the stack pointer, popping off the saved frame pointer into the frame pointer, the caller's saved ip into the instruction pointer, removing the arguments, and pushing the result:



which is done by the following method:
commonCallerReturn
"Return to the previous context/frame (sender for method activations, caller for block activations)."
<sharedCodeNamed: 'commonCallerReturn' inCase: 125> "returnTopFromBlock"

```
        localIP := self frameCallerSavedIP: localFP.
localSP := localFP + (self frameStackedReceiverOffset: localFP).
localFP :=  self frameCallerFP: localFP.
stackPages longAt: localSP put: localReturnValue.
self fetchNextBytecode.
method := self frameMethod: localFP
```

and the bytecode after the send of #ticks in DateAndTime>>#+ pops and stores the result into the ticks temporary, leaving us where we started but with ticks holding self ticks. Look ma, no contexts.



Context to Stack Mapping

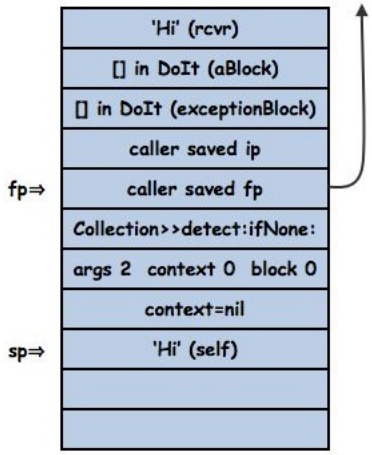
Let's add closures to the mix so we can see contexts related to stack frames. A closure has an explicit reference to its enclosing context, which in our new scheme has to be mapped to a stack frame. Let's evaluate

'Hi!' detect: [:char| '!?' includes: char] ifNone: [^#unemphatic]

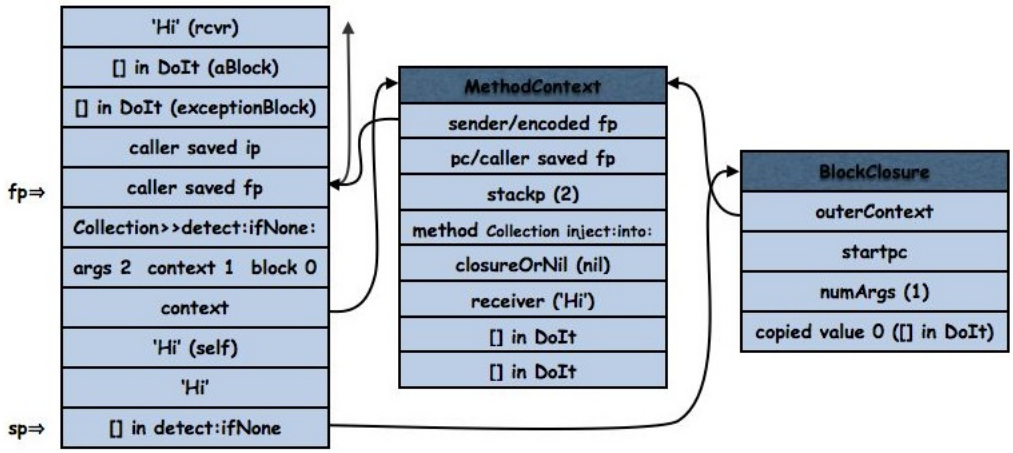
Here's Collection>>#detect:ifNone:

```
Collection methods for enumerating
detect: aBlock ifNone: exceptionBlock
"Evaluate aBlock with each of the receiver's elements as the argument.
Answer the first element for which aBlock evaluates to true. If none
evaluate to true, then evaluate the argument, exceptionBlock."
self do: [:each | (aBlock value: each) ifTrue: [^ each]].
^ exceptionBlock value
```

On activation of Collection>>#detect:ifNone: the top frame on the stack looks like this



The [:each | (aBlock value: each) ifTrue: [^ each]] needs an outerContext but it is executing within the frame activation of Collection>>#detect:ifNone:. So when the block is created we also create a context that is a proxy for the frame (a better reference is here). If the block refers to self or an instance variable it needs to get at the receiver of the outerContext and if it executes the uparrow-return it must return from the Collection>>#detect:ifNone: frame.



To avoid chaos we need a one-to-one relationship between frames and contexts; hence the flag in the frame which ensures we only create one context for a frame. In this implementation we say that a frame is married to a context, and there's no polygamy allowed. Given that a context could be referenced long after its frame exits we also have to detect if a context has been widowed, and convert it into a stable, but returned-from context. As we'll see frames and contexts can get divorced (although that's fatal for the frame) and contexts can re-marry, but frames only marry once if at all; clearly contexts are female, and frames are rather frail males; no widowers here.

To implement non-local return correctly we must be able to determine if a context is single (a normal heap context, not associated with a frame), married or widowed, and, if married, to locate its spouse frame.

At the least we need a pointer from the context to its spouse. For this we use the sender field. The sender field of a context is always either another context or nil (ignoring wilful and inevitably fatal abuse by the programmer; you *can* assign a Point, or any other object, to the sender of thisContext but it'll end your session). So we mark married contexts by storing the frame pointer of their spouse frame with the SmallInteger tag bit set in the sender field. Frames are always word-aligned so frame pointer least significant bits are always zero.

StackInterpreter methods for frame access
isMarriedOrWidowedContext: aContext
^self isIntegerObject: (self fetchPointer: **SenderIndex** ofObject: aContext)

ensureFrameIsMarried: theFP
<inline: true>
<var: #theFP type: #'char *'>
(self frameHasContext: theFP) ifTrue:
[^self frameContext: theFP].

^self marryFrame: theFP

marryFrame: theFP
"Marry an unmarried frame. This means creating a spouse context
initialized with a subset of the frame's state (state through the last argument)
that references the frame."
| theContext methodHeader byteSize tempCount closureOrNil |

```
<inline: false>
<var: #theFP type: #'char *'>
self assert: (self frameHasContext: theFP) not.
```

```
	methodHeader := self headerOf: (self frameMethod: theFP).
```

```
	"This phrase is merely determining how much of the stack to initialize the context with.
It is a lot of work for dubious benefit. Perhaps blocks could encode their num temps in frame flags."
(self frameIsBlockActivation: theFP)
ifTrue: [| numBlockArgs |
numBlockArgs := self frameNumArgs: theFP.
closureOrNil := self pushedReceiverOrClosureOfFrame: theFP.
tempCount := numBlockArgs +(self fetchWordLengthOf: closureOrNil) – ClosureFirstCopiedValueIndex]
ifFalse: [closureOrNil := nilObj.
tempCount := self tempCountOfMethodHeader: methodHeader].
```

```
	byteSize := (methodHeader bitAnd: LargeContextBit) ~= 0
ifTrue: [LargeContextSize]
ifFalse: [SmallContextSize].
theContext := self eeInstantiateContext: (self splObj: ClassMethodContext) sizeInBytes: byteSize.
"Mark context as married by setting its sender to the frame pointer plus SmallInteger
tags and the InstructionPointer to the saved fp (which ensures correct alignment
w.r.t. the frame when we check for validity)"
self storePointerUnchecked: SenderIndex
ofObject: theContext
withValue: (self withSmallIntegerTags: theFP).
self storePointerUnchecked: InstructionPointerIndex
ofObject: theContext
withValue: (self withSmallIntegerTags: (self frameCallerFP: theFP)).
self storePointerUnchecked: StackPointerIndex
ofObject: theContext
withValue: (self integerObjectOf: tempCount).
self storePointerUnchecked: MethodIndex
ofObject: theContext
withValue: (self frameMethod: theFP).
self storePointerUnchecked: ClosureIndex ofObject: theContext withValue: closureOrNil.
self storePointerUnchecked: ReceiverIndex
ofObject: theContext
withValue: (self frameReceiver: theFP).
1 to: tempCount do:
[:i|
self storePointerUnchecked: ReceiverIndex + i
ofObject: theContext
withValue: (self temporary: i – 1 in: theFP)].
```

```
	stackPages longAt: theFP + FoxThisContext put: theContext.
stackPages byteAt: theFP + FoxFrameFlags + 2 put: 1.
```

```
	self assert: (self frameHasContext: theFP).
self assert: (self frameOfMarriedContext: theContext) == theFP.
```

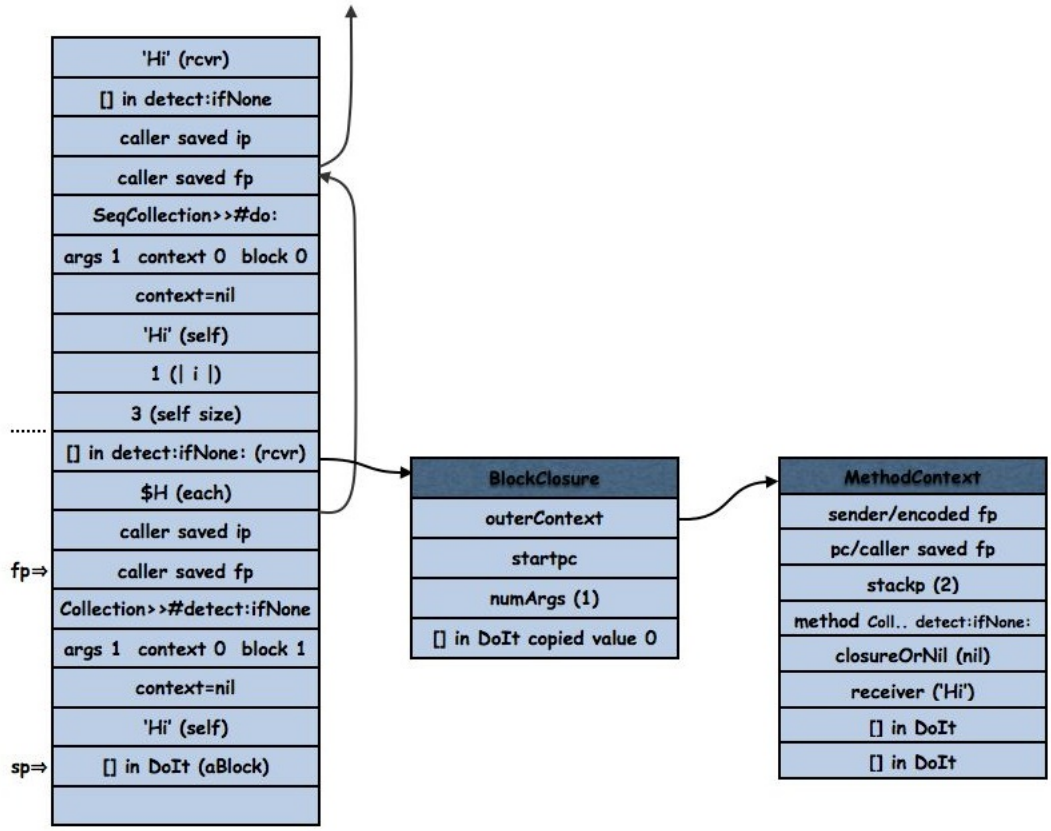
```
	^theContext
```

I think marryFrame: contains a bug in that it copies all temporaries into the spouse context but it needs only to copy the receiver and arguments. But that's how the code stands today. Mea culpa.

Before we delve into the details of divorce and widowhood, and how we hide all this matrimonial activity from Smalltalk let's finish activating the block within Collection>>#detect:ifNone:. Collection>>#detect:ifNone: sends do: and in this case SequenceableCollection>>#do: will activate the block:

```
*SequenceableCollection methods for enumerating*
do: aBlock
"Refer to the comment in Collection>>do:."
1 to: self size do:
[:index | aBlock value: (self at: index)]
```

The BlockClosure>>value[:value:…] primitives create a frame for the block, setting the "is block" flag, push the method and receiver in the outerContext and push any copied values.



In the block activation the receiver of the value: message, the [] in detect:ifNone: that do: pushed on the stack, is not the receiver ('Hi') of detect:ifNone:. If we need to marry a block activation then we can find its closureOrNil on the stack in the same position as the receiver for normal sends. But as far as accessing self and its instance variables blocks and methods are the same; the field following the context field is always the receiver of the home method activation.

StackInterpreter methods for control primitives
activateNewClosureMethod: blockClosure **numArgs:** numArgs
"Similar to activateNewMethod but for Closure and newMethod."
| numCopied outerContext theMethod closureIP |
<inline: true>
outerContext **:=** self fetchPointer: **ClosureOuterContextIndex** ofObject: blockClosure.
numCopied **:=** (self fetchWordLengthOf: blockClosure) – **ClosureFirstCopiedValueIndex**.

theMethod **:=** self fetchPointer: **MethodIndex** ofObject: outerContext.
self push: instructionPointer.
self push: framePointer.
framePointer **:=** stackPointer.
self push: theMethod.
self push: (self encodeFrameFieldHasContext: false isBlock: true numArgs: numArgs).
self push: nilObj. "FxThisContext field"
self push: (self fetchPointer: **ReceiverIndex** ofObject: outerContext).

"Copy the copied values…"
0 to: numCopied – 1 do:
[:i|
self push: (self
fetchPointer: i + **ClosureFirstCopiedValueIndex**
ofObject: blockClosure)].

self assert: (self frameIsBlockActivation: framePointer).
self assert: (self frameHasContext: framePointer) not.

"The initial instructions in the block nil-out remaining temps."

"the instruction pointer is a pointer variable equal to
method oop + ip + BaseHeaderSize
-1 for 0-based addressing of fetchByte
-1 because it gets incremented BEFORE fetching currentByte"
closureIP **:=** self quickFetchInteger: **ClosureStartPCIndex** ofObject: blockClosure.
instructionPointer **:=** theMethod + closureIP + **BaseHeaderSize** – 2.
method **:=** theMethod

Stack Pages

So far I've left out how the stack is shared amongst Smalltalk's light-weight processes, and how we maintain Smalltalk's immunity from stack overflow. In a context-based Smalltalk the only limit to the depth of a call chain is available heap memory in which to store contexts. In a blue-book Smalltalk implementation infinite recursion manifests itself as a low-space condition. Smalltalk provides light-weight processes, which are simply chains of contexts, and applications like Croquet can create thousands a second and have hundreds active at any one time. Creating a new process involves creating a process object and a context or two. In using a stack we need to both share it effectively between processes and keep process creation cheap.

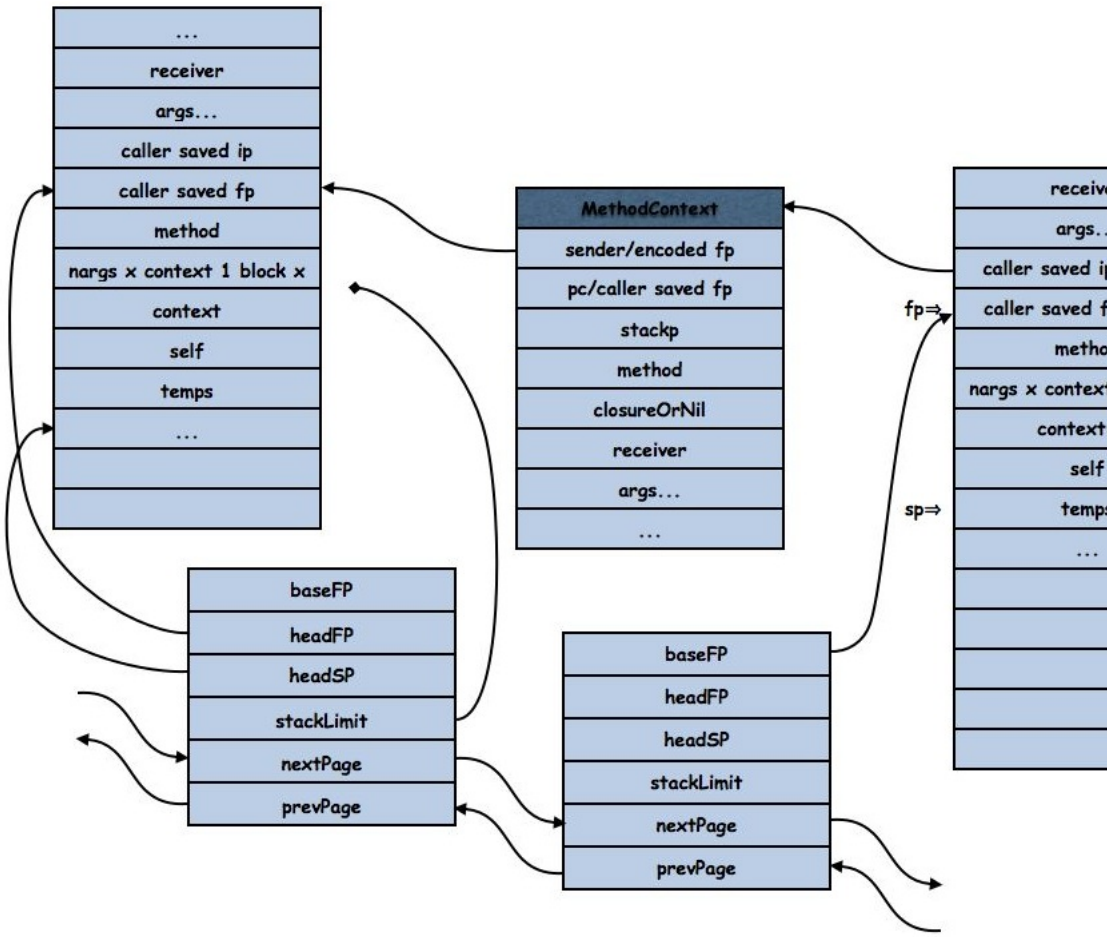
Pater Deutsch's solution is to divide up the stack into pages, each page capable of holding enough contexts so that we're not switching stack pages all the time and few enough such that moving all the frames on the page into the heap in the form of contexts so the page can be reused doesn't take too long. As we'll see with the JIT this does introduce some complications calling run-time routines on the C stack, but on balance it's an excellent solution and I'm sticking with it. So the stack is in fact composed of pages of a fixed size determined statically and quantity determined at start-up. I'm currently using 1024 byte stack pages, which have room for about 20 "average" activations, which is a little tight and so will probably move to 2048 byte pages when I tune the JIT. Current;y inside Qwaq we're using 192 stack pages.

Running on a stack page of a small size introduces three complications, stack page overflow, stack page underflow, and linking stack pages together. We handle overflow by checking a stackLimit on every frame build. So in full detail internalActivateNewMethod is as follows, with the added complications of passing any primitive error code and of checking for stack overflow, which is right at the end of the method:

```
StackInterpreter methods for message sending
internalActivateNewMethod
| methodHeader numTemps rcvr errorCode |
<inline: true>

	methodHeader := self headerOf: newMethod.
numTemps := self tempCountOfMethodHeader: methodHeader.

	rcvr := self internalStackValue: argumentCount. "could new rcvr be set at point of send?"

	self internalPush: localIP.
self internalPush: localFP.
localFP := localSP.
self internalPush: newMethod.
method := newMethod.
self internalPush: (self
encodeFrameFieldHasContext: false
isBlock: false
numArgs: (self argumentCountOfMethodHeader: methodHeader)).
self internalPush: nilObj. "FxThisContext field"
self internalPush: rcvr.

	"Initialize temps..."
argumentCount + 1 to: numTemps do:
[:i | self internalPush: nilObj].

	"-1 to account for pre-increment in fetchNextBytecode"
localIP := self pointerForOop: (self initialPCForHeader: methodHeader method: newMethod) - 1.

	"Pass primitive error code to last temp if method receives it (indicated
by an initial long store temp bytecode). Protect against obsolete values
in primFailCode by checking that newMethod actually has a primitive?"
primFailCode ~= 0 ifTrue:
[((self methodHeaderHasPrimitive: methodHeader)
and: [(self byteAtPointer: localIP + 1) = 129 "long store temp"]) ifTrue:
[errorCode := self getErrorObjectFromPrimFailCode.
self longAt: localSP put: errorCode "nil if primFailCode == 1, or primFailCode"].
primFailCode := 0].

	self assert: (self frameNumArgs: localFP) == argumentCount.
self assert: (self frameIsBlockActivation: localFP) not.
self assert: (self frameHasContext: localFP) not.

	"Now check for stack overflow or an event (interrupt, must scavenge, etc)"
localSP < stackLimit ifTrue:
[self externalizeIPandSP.
self stackOverflowOrEvent: argumentCount mayContextSwitch: true.
self internalizeIPandSP]
```

On overflow we must allocate a new stack page and continue execution there, linking the new bottom-most frame (its base frame) to the current frame. Likewise, return must deal with returning from a base frame and return from one stack frame to the next. We have a mechanism for referring to stack frames, married contexts, so to link a stack page's base frame to the stack

page beneath it we marry the top frame of the stack we're leaving and store this in the base frame of the stack page we're entering. We mark the new stack page's base frame as such by giving it a null caller saved fp and use the caller saved ip field to hold the caller context, the spouse of the stack page beneath's top frame:



We have a linked list of StackPage objects, one for each page, that we use to keep stack pages in order of usage, along with free pages, referenced by a variable called the mostRecentlyUsedPage. Each StackPage keeps track of whether a stack page is in use (baseFP is non-null) and what part of the page is in use (from the first slot through to the headSP) and what the frames are in the page (the list from headFP chained through caller saved fp to the baseFP). The interpreter's current stack page is called stackPage. On stack switch we load stackLimit from stackPage's stackLimit. Peter cleverly realised that one can use the stackLimit check to cause the VM to break out of execution to process input events. The VM is set up to respond to potential input with an interrupt handler that sets the stackLimit to all ones (the highest possible address) so that the next stack overflow check will fail. We also check for stack overflow on backward branch so that we can break out of infinite loops:

```
StackInterpreter methods for jump bytecodes
longUnconditionalJump
| offset |
offset := (((currentBytecode bitAnd: 7) – 4) * 256) + self fetchByte.
localIP := localIP + offset.
(offset < 0 "backward jump means we're in a loop; check for possible interrupts"
and: [localSP < stackLimit]) ifTrue:
[self externalizeIPandSP.
self checkForEventsMayContextSwitch: true.
self browserPluginReturnIfNeeded.
self internalizeIPandSP].
self fetchNextBytecode
```

So stackOverflowOrEvent:mayContextSwitch: checks whether stackLimit differs from stackPage's stack limit to see whether it should check for input and only switches to a new page if the stack pointer is below stackPage's real stack limit.

Of course on stack overflow we may find that there are no free stack pages, in which case the VM simply flushes all the frames on the least-recently used page to the heap in the form of contexts. First it ensures all frames on the page are married and then divorces them all, linking them through their sender fields just as if they were created by a context-only VM, and then reuses the page, mercilessly killing the frames there-on. If any other stack page had the freed stack page beneath it then that stack page's base frame will find its saved caller context is single.

So as execution proceeeds the stack zone holds the top frames of a number of recently executed processes, overflowing the contents of older stack pages to the heap as contexts. The only possible references into that page are from contexts which have all been divorced, so all references remain valid, except now they're to single contexts instead of married ones.

When the system snapshots the image all stack frames are divorced and so the image file contains only single contexts, allowing the image file to be run by a context-only VM or started up on a VM with a different number of stack pages at different addresses. Here's the divorce code:

StackInterpreter methods for image save/restore

```
snapshot: embedded
“update state of active context”
| activeContext activeProc dataSize rcvr setMacType stackIndex |
<var: #setMacType type: ‘void *’>


	“Need to convert all frames into contexts since the snapshot file only holds objects.”
self push: instructionPointer.
activeContext := self divorceAllFrames.
self pushRemappableOop: activeContext.


	“update state of active process”
…etc…

StackInterpreter methods for frame access
divorceAllFrames
| activeContext |
<inline: false>
<var: #aPage type: #’StackPage *’>
self externalWriteBackHeadFramePointers.
activeContext := self ensureFrameIsMarried: framePointer.
0 to: numStackPages – 1 do:
[:i| | aPage |
aPage := stackPages stackPageAt: i.
(stackPages isFree: aPage) ifFalse:
[self divorceFramesIn: aPage]].
stackPage := 0.
^activeContext


divorceFramesIn: aStackPage
| theFP calleeFP theSP theIP calleeContext theContext |
<inline: false>
<var: #aStackPage type: #’StackPage *’>
<var: #theFP type: #’char *’>
<var: #calleeFP type: #’char *’>
<var: #theSP type: #’char *’>


	statStackPageDivorce := statStackPageDivorce + 1.


	theFP := aStackPage headFP.
theSP := aStackPage headSP.
theIP := stackPages longAt: theSP.
theSP := theSP + BytesPerWord. “theSP points at hottest item on frame’s stack”
calleeContext := nil.


	[theContext := self ensureFrameIsMarried: theFP.
self updateStateOfSpouseContextForFrame: theFP WithSP: theSP.
self storePointerUnchecked: InstructionPointerIndex
ofObject: theContext
withValue: (self contextInstructionPointerForFrame: theFP IP: theIP).
self assert: (self frameReceiver: theFP)
== (self fetchPointer: ReceiverIndex ofObject: theContext).
calleeContext ~~ nil ifTrue:
[self storePointer: SenderIndex
ofObject: calleeContext
withValue: theContext].
calleeContext := theContext.
calleeFP := theFP.
theIP := self frameCallerContext: theFP. “a.k.a. frameCallerIP:”
theFP := self frameCallerFP: theFP.
theFP ~= 0] whileTrue:
[“theSP points at stacked hottest item on frame’s stack”
theSP := self frameCallerSP: calleeFP].


	self storePointer: SenderIndex
ofObject: theContext
withValue: theIP. “The ip of the base frame is the caller context”


	“The page is now free; mark it so.”
aStackPage baseFP: 0


updateStateOfSpouseContextForFrame: theFP WithSP: theSP
“Update the frame’s spouse context with the frame’s current state except for the
sender and instruction pointer, which are used to mark the context as married.”
| theContext tempIndex pointer |
<inline: false>
```

```
<var: #theFP type: #'char *'>
<var: #theSP type: #'char *'>
<var: #pointer type: #'char *'>
self assert: (self frameHasContext: theFP).
theContext := self frameContext: theFP.
tempIndex := self frameNumArgs: theFP.
pointer := theFP + FoxReceiver – BytesPerWord.
[pointer >= theSP] whileTrue:
[tempIndex := tempIndex + 1.
self storePointer: ReceiverIndex + tempIndex
ofObject: theContext
withValue: (stackPages longAt: pointer).
pointer := pointer – BytesPerWord].
self storePointerUnchecked: StackPointerIndex
ofObject: theContext
withValue: (self integerObjectOf: tempIndex)
```

Widowhood

Given that contexts are objects they have indefinite extent; they can be stored in instance variables, typically indirectly from blocks that are stored in various places) and so outlive being returned from. For example here's one way to create a widowed context for a block activation:

```
| container |
container := Array new: 1.
#(foo) do: [:ignored| container at: 1 put: thisContext].
container
```

Even simpler would be

```
Object methods for examples
widowedContext
^thisContext
```

At various times, for example whenever accessing the instance variables of contexts, or on return, we need to check whether a context is single, married or widowed and act accordingly. A single context will have a sender that is not a SmallInteger (recall isMarriedOrWidowedContext: above). A married or widowed context however may or or may not have outlived its spouse. Since the system starts up and snapshots with only single contexts we know that all encoded frame pointers in sender fields point into stack pages. We can easily derive the StackPage object for a stack page because each stack page is a power-of-two in size and all stack pages are contiguous. Once we have the StackPage we can find out if it is in use or not, and if the context's spouse frame is in the live portion of the stack (between baseFP and headFP). If so, we can follow the frame pointer and examine the frame's flags and context fields. But there is no guarantee that we're looking at the actual frame. It may have been exited and overwritten by some other frame. So we must assume that the spouse frame pointer is potentially pointing at an arbitrary position in the stack.

Note that we have to keep the current stackPage's headFP and headSP up-to-date because the interpreter of course uses its own localFP and localSP (and, sigh, framePointer and stackPointer, the former being a clever performance hack that tediously causes lots of code duplication, one that the JIT can hopefully do without). writeBackHeadFramePointers and externalWriteBackHeadFramePointers assign stackPage headFP and headSP from localFP & localSP or framePointer and stackPointer respectively (see e.g. divorceAllFrames above).

Because the stack and the heap are disjoint there is no overlap of caller saved fp and other fields on the stack (saved instruction pointers are also pointers but into method bytecodes). So by storing the spouse's caller saved fp in the married context, a match implies the spouse frame pointer is to a frame at that position. If the frame has a context context matches then we know the two are married. If they're not, the frame must have exited, and so we widow the context by converting it into a context that has been returned from, .e. we nil its sender and instruction pointer fields. Now you can see the bug in marryFrame above. The widowed context will have a valid method, receiver and arguments (arguments being read-only in Smalltalk) but shouldn't hold onto the temporaries that existed at the time of its creation because they could be out-of-date. Alas Qwaq is too close to a release right now for me to dare fixing this; I'm going to wait a week or two. That a widowed context doesn't hold onyto non-argument temporaries is, I believe (and fervently hope) the only visible difference between a pure context VM and a stack VM. I think its fair to expect that volatile state does disappear on return, and not holding onto volatile state is good for garbage collection. This change in semantics has never been complained about by the VisualWorks community when I introduced this scheme to VisualWorks in 1999. The only code I've ever seen affected by this was an exception report generator that used to generate its report after the exception handler had squirreled away the context and returned. The fix is either to generate the report in the exception handler or copy the context chain there-in.

With widowhood explained we can have a look at the full glory of return.

Return

As we've seen, most of the time send and return are considerably less complicated than in the context VM. Send is somewhat more complicated when there's a stack overflow. Return is somewhat more complicated when there is a base return:

```
StackInterpreter methods for return bytecodes
commonReturn
"Note: Assumed to be inlined into the dispatch loop."

    | closure home unwindContextOrNilOrZero frameToReturnTo contextToReturnTo theFP callerFP newPage |
<var: #frameToReturnTo type: #'char *'>
<var: #theFP type: #'char *'>
<var: #callerFP type: #'char *'>
<var: #newPage type: #'StackPage *'>
<var: #thePage type: #'StackPage *'>
<sharedCodeNamed: 'commonReturn' inCase: 120>
```

```
        "If this is a method simply return to the sender/caller."
(self frameIsBlockActivation: localFP) ifFalse:
[^self commonCallerReturn].

        … non-local return code; here be beasties …
```

```
commonCallerReturn
"Return to the previous context/frame (sender for method activations, caller for block activations)."
| callersFPOrNull |
<var: #callersFPOrNull type: #'char *'>
<sharedCodeNamed: 'commonCallerReturn' inCase: 125> "returnTopFromBlock"

        callersFPOrNull := self frameCallerFP: localFP.
callersFPOrNull == 0 "baseFrame" ifTrue:
[self assert: localFP = stackPage baseFP.
^self baseReturn].

        localIP := self frameCallerSavedIP: localFP.
localSP := localFP + (self frameStackedReceiverOffset: localFP).
localFP := callersFPOrNull.
stackPages longAt: localSP put: localReturnValue.
self fetchNextBytecode.
method := self frameMethod: localFP
```

```
baseReturn
| contextToReturnTo isContext theFP theSP thePage |
<var: #theFP type: #'char *'>
<var: #theSP type: #'char *'>
<var: #thePage type: #'StackPage *'>
contextToReturnTo := self frameCallerContext: localFP.
isContext := self isContext: contextToReturnTo.
(isContext
and: [self isStillMarriedContext: contextToReturnTo])
ifTrue:
[theFP := self frameOfMarriedContext: contextToReturnTo.
thePage := stackPages stackPageFor: theFP.
self assert: theFP == thePage headFP.
theSP := thePage headSP]
ifFalse:
[(isContext
and: [self isIntegerObject: (self fetchPointer: InstructionPointerIndex ofObject: contextToReturnTo)]) ifFalse:
[^self internalCannotReturn: localReturnValue].
thePage := self makeBaseFrameFor: contextToReturnTo.
theFP := thePage headFP.
theSP := thePage headSP].
stackPages freeStackPageNoAssert: stackPage. "for a short time invariant is violated; assert follows"
self setStackPageAndLimit: thePage.
self assert: (stackPages stackPageFor: theFP) == stackPage.
localSP := theSP.
localFP := theFP.
method := self frameMethod: localFP.
localIP := self pointerForOop: self internalStackTop.
self internalStackTopPut: localReturnValue.
self fetchNextBytecode.
self assert: (self checkIsStillMarriedContext: contextToReturnTo currentFP: localFP)
```

But non-local return is *much* more complicated. What does non-local return do in a pure context VM? It walks the context chain from a block context until it finds the block context's home context, checking for unwnd protect contexts along the way. If it doesn't find the home context it sends cannotReturn: and if it finds an unwnd protect it sends aboutToReturn:through: and allows the image to run unwind-protect blocks itself, otherwise it simply returns from the home context. Using the stack, things are seriously more complicated.

In the StackInterpreter non-local return occurs in some block activation (a frame) and returns from the block activation's home context, which is some distance along the block activation's closure's outerContext chain (its lexical chain). The home context could be single, widowed or married. If married it could be a base frame whose caller could be single, widowed or married. If single we need to marry it to have a frame to resume execution in. If the home's caller is married its spouse frame could be the head frame on its page or some interior frame. So we could be returning within a stack page or across stack pages or across stack pages and single contexts. First we have to walk this chain looking for unwind protects and the home context, then we have to do the return, freeing intervening stack pages. Complicated.

So here it is. Read the comments and hopefully it'll make sense.

```
StackInterpreter methods for return bytecodes
commonReturn
"Note: Assumed to be inlined into the dispatch loop."
```

```
	| closure home unwindContextOrNilOrZero frameToReturnTo contextToReturnTo theFP callerFP newPage |
<var: #frameToReturnTo type: #'char *'>
<var: #theFP type: #'char *'>
<var: #callerFP type: #'char *'>
<var: #newPage type: #'StackPage *'>
<var: #thePage type: #'StackPage *'>
<sharedCodeNamed: 'commonReturn' inCase: 120>


	"If this is a method simply return to the sender/caller."
(self frameIsBlockActivation: localFP) ifFalse:
[^self commonCallerReturn].


	"Since this is a block activation the closure is on the stack above any args and the frame."
closure := self pushedReceiverOrClosureOfFrame: localFP.


	home := nil.
"Walk the closure's lexical chain to find the context or frame to return from (home)."
[closure ~~ nilObj] whileTrue:
[home := self fetchPointer: ClosureOuterContextIndex ofObject: closure.
closure := self fetchPointer: ClosureIndex ofObject: home].
"home is to be returned from provided there is no unwind-protect activation between
this frame and home's sender. Search for an unwind. findUnwindThroughContext:
will answer either the context for an unwind-protect activation or nilObj if the sender
cannot be found or 0 if no unwind is found but the sender is. We must update the
current page's headFrame pointers to enable the search to identify widowed contexts
correctly."
self writeBackHeadFramePointers.
unwindContextOrNilOrZero := self findUnwindThroughContext: home.
unwindContextOrNilOrZero == nilObj ifTrue:
["error: can't find home on chain; cannot return"
^self internalCannotReturn: localReturnValue].
unwindContextOrNilOrZero ~~ 0 ifTrue:
[^self internalAboutToReturn: localReturnValue through: unwindContextOrNilOrZero].


	"Now we know home is on the sender chain.
We could be returning to either a context or a frame. Find out which."
contextToReturnTo := nil.
(self isMarriedOrWidowedContext: home)
ifTrue:
[self assert: (self checkIsStillMarriedContext: home currentFP: localFP).
theFP := self frameOfMarriedContext: home.
(self isBaseFrame: theFP)
ifTrue:
[contextToReturnTo := self frameCallerContext: theFP]
ifFalse:
[frameToReturnTo := self frameCallerFP: theFP]]
ifFalse:
[contextToReturnTo := self fetchPointer: SenderIndex ofObject: home.
(self isMarriedOrWidowedContext: contextToReturnTo) ifTrue:
[self assert: (self checkIsStillMarriedContext: contextToReturnTo currentFP: localFP).
frameToReturnTo := self frameOfMarriedContext: contextToReturnTo.
contextToReturnTo := nil]].
"If returning to a context we must make a frame for it unless it is dead."
contextToReturnTo ~= nil ifTrue:
[frameToReturnTo := self establishFrameForContextToReturnTo: contextToReturnTo.
frameToReturnTo == 0 ifTrue:
["error: home's sender is dead; cannot return"
^self internalCannotReturn: localReturnValue]].


	"Now we have a frame to return to. If it is on a different page we must
free intervening pages and nil out intervening contexts. We must free
intervening stack pages because if we leave the pages to be divorced
then their contexts will be divorced with intact senders and instruction
pointers. This code is similar to primitiveTerminateTo."
newPage := stackPages stackPageFor: frameToReturnTo.
newPage ~~ stackPage ifTrue:
[| currentCtx thePage nextCntx |
currentCtx := self frameCallerContext: stackPage baseFP.
self assert: (self isContext: currentCtx).
stackPages freeStackPage: stackPage.
[(self isMarriedOrWidowedContext: currentCtx)
and: [(stackPages stackPageFor: (theFP := self frameOfMarriedContext: currentCtx)) == newPage]] whileFalse:
[(self isMarriedOrWidowedContext: currentCtx)
ifTrue:
[thePage := stackPages stackPageFor: theFP.
currentCtx := self frameCallerContext: thePage baseFP.
self freeStackPage: thePage]
ifFalse:
[self assert: (self isContext: currentCtx).
nextCntx := self fetchPointer: SenderIndex ofObject: currentCtx.
```

```
self storePointerUnchecked: SenderIndex ofObject: currentCtx withValue: nilObj.
self storePointerUnchecked: InstructionPointerIndex ofObject: currentCtx withValue: nilObj.
currentCtx := nextCntx]].
self setStackPageAndLimit: newPage.
localSP := stackPage headSP.
localFP := stackPage headFP].


	"Two cases. Returning to the top frame or an interior frame.
The top frame has its instruction pointer on top of stack.
An interior frame has its instruction pointer in the caller frame.
We need to peel back any frames on the page until we get to the correct frame."


	localFP == frameToReturnTo
ifTrue: "pop the saved IP, push the return value and continue."
[localIP := self pointerForOop: self internalStackTop]
ifFalse:
[[callerFP := localFP.
localFP := self frameCallerFP: localFP.
localFP ~~ frameToReturnTo] whileTrue.
localIP := self frameCallerSavedIP: callerFP.
localSP := (self frameCallerSP: callerFP) – BytesPerWord].
self internalStackTopPut: localReturnValue.
method := self frameMethod: localFP.
self fetchNextBytecode

findUnwindThroughContext: homeContext
"Search for either an unwind-protect (activation of method with primitive 198)
or homeContext along the sender chain, which ever is found first. If homeContext
is not found answer nilObj, indicating cannotReturn:. If homeContext is found
answer 0. If homeContext is itself an unwind-protect answer the context, not 0."
| ctxtOrNilOrZero theMethod |
self externalizeIPandSP.
"Since nothing changes we don't need to internalize."
ctxtOrNilOrZero := self findMethodWithPrimitive: 198 FromFP: localFP UpToContext: homeContext.
ctxtOrNilOrZero = 0 ifTrue:
[theMethod := self fetchPointer: MethodIndex ofObject: homeContext.
(self primitiveIndexOf: theMethod) == 198 ifTrue:
[^homeContext]].
^ctxtOrNilOrZero


findMethodWithPrimitive: primitive FromFP: startFP UpToContext: homeContext
"See findUnwindThroughContext:. Alas this is mutually recursive with
findMethodWithPrimitive:FromContext:ThroughContext: instead of iterative.
We're doing the simplest thing that could possibly work. Niceties can wait."
| theFP theFPAbove theMethod senderContext |
<var: #startFP type: #'char *'>
<var: #theFP type: #'char *'>
<var: #theFPAbove type: #'char *'>
theFP := startFP.
theFPAbove := startFP.
[((self frameHasContext: theFP)
and: [homeContext == (self frameContext: theFP)]) ifTrue:
[^0].
theMethod := self frameMethod: theFP.
(self primitiveIndexOf: theMethod) == primitive ifTrue:
[^self ensureFrameIsMarried: theFP].
theFPAbove := theFP.
theFP := self frameCallerFP: theFP.
theFP ~= 0] whileTrue.
senderContext := self frameCallerContext: theFPAbove.
(self isContext: senderContext) ifFalse:
[^nilObj].
^self
findMethodWithPrimitive: primitive
FromContext: senderContext
UpToContext: homeContext


findMethodWithPrimitive: primitive FromContext: senderContext UpToContext: homeContext
"See findUnwindThroughContext:. Alas this is mutually recursive with
findMethodWithPrimitive:FromFP:SP:ThroughContext: instead of iterative.
We're doing the simplest thing that could possibly work. Niceties can wait."
| theContext theMethod |
theContext := senderContext.
[self isMarriedOrWidowedContext: theContext] whileFalse:
[theContext = homeContext ifTrue: [^0].
theMethod := self fetchPointer: MethodIndex ofObject: theContext.
(self primitiveIndexOf: theMethod) == primitive ifTrue:
[^theContext].
theContext := self fetchPointer: SenderIndex ofObject: theContext.
theContext = nilObj ifTrue:
[^theContext]].
(self isWidowedContext: theContext) ifTrue:
```

```
[^nilObj].
^self
findMethodWithPrimitive: primitive
FromFP: (self frameOfMarriedContext: theContext)
UpToContext: homeContext
```

Ouch. That *is* complicated, but at least it is the *most* complicated thing in the StackInterpreter. It is not finished yet. The mutual recursion between findMethodWithPrimitive:FromFP:UpToContext: and findMethodWithPrimitive:FromContext:UpToContext: could conceivably cause the runtime C stack to overflow and should be flattened into a set of loops. But when I wrote this I found it much easier to understand because I was riting it in Smalltalk. I'm not at all convinced that the code inside the VisualWorks VMs correct because, being written in C, it is not nearly as comprehensible.

Those of you who know the Squeak VM well will know there's a primitive terminateTo: that does something similar to the context nilling in non-local return. It is a little simpler than non-local return (although not much) but it is just a variation on the same theme so I'll spare you.

Hiding Stack Frames

Still reading? Good. Now we're ready for some fun. While we've organized the system around stack pages and improved performance markedly (measurements to follow) and prepared the ground for the JIT, we still have to hide these stack pages from the Smalltalk programmer. We need to arrange that whever he or she accesses a context it hides its spose and always pretends to be single, whether we read or write it. We've seen how return copes with this, but what about methods on context such as

```
ContextPart methods for debugger access
swapSender: coroutine
"Replace the receiver's sender with coroutine and answer the receiver's
previous sender. For use in coroutining."

    | oldSender |
oldSender := sender.
sender := coroutine.
^oldSender
```

The problem here is that the current valid state of the context lives in the frame and changing that state needs to either affect the frame or update the context after it has been divorced. One apparently straight-forward way to handle this is to divorce contexts whenever they are sent messages and whenever the system attempts to access a context instance variable. This is the way the VisualWorks VM handled things until I changed the scheme in 5i in 1999. Married contexts (hybrid contexts in the VisualWorks terminology) had special classes that would cause teh send machinery to trap and "stabilize" (divorce) the receiver context. The bytecode compiler marked methods that accessed instance variables which could be run by contexts and the JIT generated special code that would stabilize the receiver context before accessing an instance variable.

The main problem with this approach is that any kind of state access ends up divorcing a context which hurts performance and encourages a proliferation of complex primitives in the VM which try and avoid divorce replicating code that could be written in the image. The other problem is that it is slow. Sends to contexts trap and divorce even for read access. Wen I introduced the scheme I'm using here exception handling code and non-local return roughly doubled in speed. The key in this scheme is to intercept all access to context instace variables and forward the access to the spouse if married.

In the VM the only places the state of a context are accessed outside of the frame management code for send and return and stack page management are the instance variable access bytecodes and the at: at:put: instVarAt: and instVarAt:put: primitives. Instance variable access is very frequent so having to test for accessing a context on every instance variable access bytecode would hurt performance noticeably. Unlike a JIT we can't perform the text for contextness at compile time based on the class of the method being compiled; we have to test on execution of the relevant bytecocde. But as I alluded to in Closures Part III I added a horrible hack to the bytecode compiler to avoid testing on most instace variable accesses.

The compiler allows a class to specify special node classes for handling instance variables. This is used by Tweak to access properties and by InstructionStream, the first superclass of ContextPart to introduce state, to handle context instance variables.

```
Encoder methods for initialize-release
init: aClass context: aContext notifying: req
requestor := req.
class := aClass.
nTemps := 0.
supered := false.
self initScopeAndLiteralTables.
class variablesAndOffsetsDo:
[:variable "<String|CFieldDefinition>" :offset "<Integer|nil>" |
offset isNil
ifTrue: [scopeTable at: variable name put: (FieldNode new fieldDefinition: variable)]
ifFalse: [scopeTable
at: variable
put: (offset >= 0
ifTrue: [InstanceVariableNode new
name: variable index: offset]
ifFalse: [MaybeContextInstanceVariableNode new
name: variable index: offset negated])]].
aContext ~~ nil ifTrue:
[| homeNode |
homeNode := self bindTemp: self doItInContextName.
"0th temp = aContext passed as arg"
aContext tempNames withIndexDo:
[:variable :index|
scopeTable
at: variable
put: (MessageAsTempNode new
receiver: homeNode
```

```
selector: #namedTempAt:
arguments: (Array with: (self encodeLiteral: index))
precedence: 3
from: self)]].
sourceRanges := Dictionary new: 32.
globalSourceRanges := OrderedCollection new: 32
```

CProtoObject class methods for compiling
variablesAndOffsetsDo: aBinaryBlock
"This is the interface between the compiler and a class's instance or field names. The class should enumerate aBinaryBlock with the field definitions (with nil offsets) followed by the instance variable name strings and their integer offsets (1-relative). The order is important; names evaluated later will override the same names occurring earlier."

```
	self allFieldsReverseDo: [:field| aBinaryBlock value: field value: nil].
self instVarNamesAndOffsetsDo: aBinaryBlock
```

Behavior methods for compiling
variablesAndOffsetsDo: aBinaryBlock
"This is the interface between the compiler and a class's instance or field names. The class should enumerate aBinaryBlock with the field definitions (with nil offsets) followed by the instance variable name strings and their integer offsets (1-relative). The order is important; names evaluated later will override the same names occurring earlier."

```
	"Only need to do instance variables here. CProtoObject introduces field definitions."
self instVarNamesAndOffsetsDo: aBinaryBlock
```

instVarNamesAndOffsetsDo: aBinaryBlock
"This is part of the interface between the compiler and a class's instance or field names. The class should enumerate aBinaryBlock with the instance variable name strings and their integer offsets. The order is important. Names evaluated later will override the same names occurring earlier."

```
	"Nothing to do here; ClassDescription introduces named instance variables"
^self
```

ClassDescription methods for compiling
instVarNamesAndOffsetsDo: aBinaryBlock
"This is part of the interface between the compiler and a class's instance or field names. The class should enumerate aBinaryBlock with the instance variable name strings and their integer offsets. The order is important. Names evaluated later will override the same names occurring earlier."

```
	| superInstSize |
(superInstSize := superclass notNil ifTrue: [superclass instSize] ifFalse: [0]) > 0 ifTrue:
[superclass instVarNamesAndOffsetsDo: aBinaryBlock].
1 to: self instSize – superInstSize do:
[:i| aBinaryBlock value: (instanceVariables at: i) value: i + superInstSize]
```

InstructionStream class methods for compiling
instVarNamesAndOffsetsDo: aBinaryBlock
"This is part of the interface between the compiler and a class's instance or field names. We override here to arrange that the compiler will use MaybeContextInstanceVariableNodes for instances variables of ContextPart or any of its superclasses and subclasses. The convention to make the compiler use the special nodes is to use negative indices"

```
	| superInstSize |
(self withAllSubclasses noneSatisfy: [:class|class isContextClass]) ifTrue:
[^super instVarNamesAndOffsetsDo: aBinaryBlock].
(superInstSize := superclass notNil ifTrue: [superclass instSize] ifFalse: [0]) > 0 ifTrue:
[superclass instVarNamesAndOffsetsDo: aBinaryBlock].
1 to: self instSize – superInstSize do:
[:i| aBinaryBlock value: (instanceVariables at: i) value: (i + superInstSize) negated]
```

MaybeContextInstanceVariableNode simply forces the use of the long-form instance variable bytecodes:

MaybeContextInstanceVariableNode methods for code generation (new scheme)
emitCodeForValue: stack **encoder:** encoder
```
stack push: 1.
^encoder genPushInstVarLong: index
```

emitCodeForStorePop: stack **encoder:** encoder
```
encoder genStorePopInstVarLong: index.
stack pop: 1
```

EncoderForV3 methods for bytecode generation
genPushInstVarLong: instVarIndex
"See BlueBook page 596"
"See also MaybeContextInstanceVariableNode"
(instVarIndex >= 0 and: [instVarIndex < 256]) ifTrue:
["132 10000100 iiijjjjj kkkkkkkk (Send, Send Super, Push Receiver Variable, Push Literal Constant, Push Literal Variable, Store Receiver Variable, Store-Pop Receiver Variable, Store Literal Variable)[iii] #kkkkkkkk jjjjj"
stream
nextPut: 132;
nextPut: 64;
nextPut: instVarIndex.
^self].
^self outOfRangeError: 'index' index: instVarIndex range: 0 to: 255

genStorePopInstVarLong: instVarIndex
"See BlueBook page 596"
"See also MaybeContextInstanceVariableNode"
(instVarIndex >= 0 and: [instVarIndex < 256]) ifTrue:
["132 10000100 iiijjjjj kkkkkkkk (Send, Send Super, Push Receiver Variable, Push Literal Constant, Push Literal Variable, Store Receiver Variable, Store-Pop Receiver Variable, Store Literal Variable)[iii] #kkkkkkkk jjjjj"
stream
nextPut: 132;
nextPut: 192;
nextPut: instVarIndex.
^self].
^self outOfRangeError: 'index' index: instVarIndex range: 0 to: 255

The doubleExtendedDoAnythingBytecode (I'm not making this up) only needs to be used for instance variables with an offset greater than 63, which is rare. Since MethodContext only has 6 instance variables we only have to check for the receiver being a contex if we're using the doubleExtendedDoAnythingBytecode to access instance variables 0 through 5. Since the doubleExtendedDoAnythingBytecode has to do a lot of decoding before it computes the instance variable offset and the test is ony a compare of the computed index against the constant 6 the test is essentally free for all but use on actual contexts.

StackInterpreter methods for send bytecodes
doubleExtendedDoAnythingBytecode
"Replaces the Blue Book double-extended send [132], in which the first byte was wasted on 8 bits of argument count.
Here we use 3 bits for the operation sub-type (opType), and the remaining 5 bits for argument count where needed.
The last byte give access to 256 instVars or literals.
See also secondExtendedSendBytecode"
| byte2 byte3 opType top |
byte2 := self fetchByte.
byte3 := self fetchByte.
opType := byte2 >> 5.
opType = 0 ifTrue:
[messageSelector := self literal: byte3.
argumentCount := byte2 bitAnd: 31.
^self normalSend].
opType = 1 ifTrue:
[messageSelector := self literal: byte3.
argumentCount := byte2 bitAnd: 31.
^self superclassSend].
self fetchNextBytecode.
opType = 2 ifTrue: [^self pushMaybeContextReceiverVariable: byte3].
opType = 3 ifTrue: [^self pushLiteralConstant: byte3].
opType = 4 ifTrue: [^self pushLiteralVariable: byte3].
top := self internalStackTop.
opType = 5 ifTrue:
[^self storeMaybeContextReceiverVariable: byte3 withValue: top].
opType = 6 ifTrue:
[self internalPop: 1.
^self storeMaybeContextReceiverVariable: byte3 withValue: top].
opType = 7 ifTrue:
[^self storePointer: **ValueIndex** ofObject: (self literal: byte3) withValue: top]

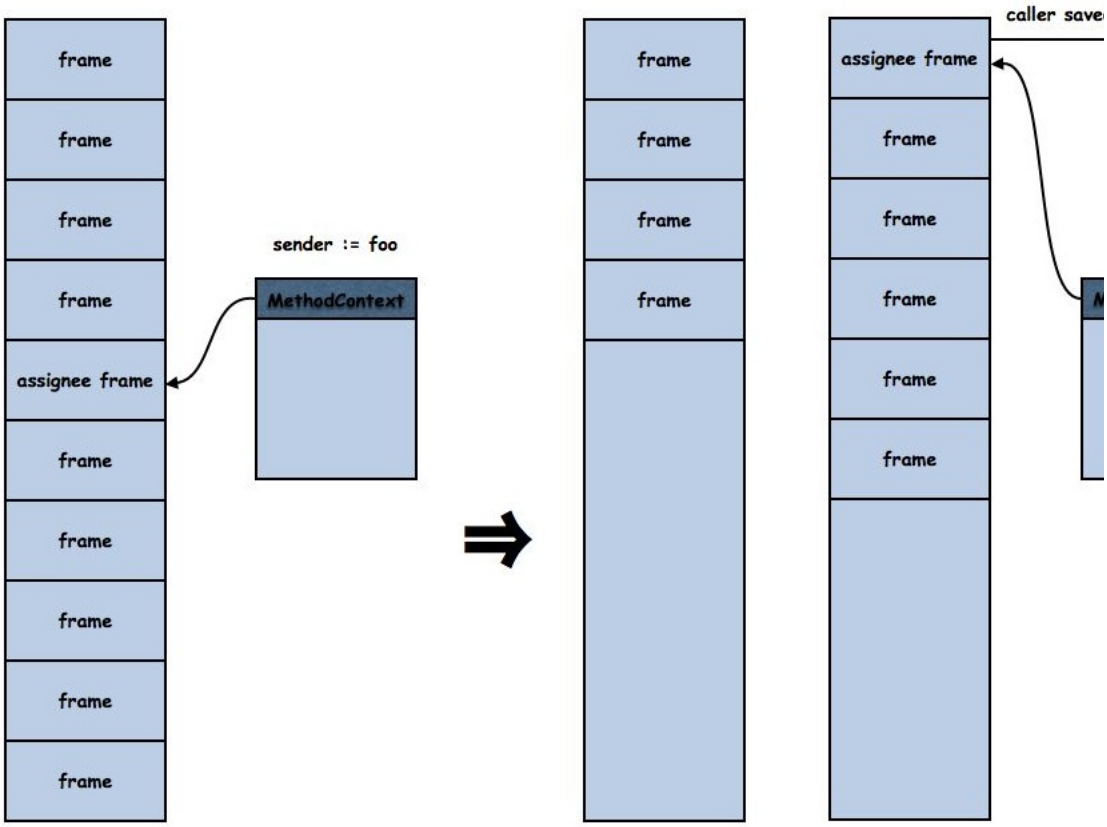
StackInterpreter methods for stack bytecodes
pushMaybeContextReceiverVariable: fieldIndex
"Must trap accesses to married and widowed contexts.
But don't want to check on all inst var accesses. This
method is only used by the long-form bytecodes, evading
the cost. Note that the method, closure and receiver fields
of married contexts are correctly initialized so they don't
need special treatment on read. Only sender, instruction
pointer and stack pointer need to be intercepted on reads."
| rcvr |
<inline: true>
rcvr := self receiver.
(fieldIndex < **MethodIndex**
and: [self isMarriedOrWidowedContext: rcvr])
ifTrue:
[self internalPush: (self instVar: fieldIndex ofContext: rcvr)]
ifFalse:
[self internalPush: (self fetchPointer: fieldIndex ofObject: rcvr)]

```
storeMaybeContextReceiverVariable: fieldIndex withValue: anObject
“Must trap accesses to married and widowed contexts.
But don’t want to check on all inst var accesses. This
method is only used by the long-form bytecodes, evading the cost.”
| rcvr |
rcvr := self receiver.
(fieldIndex <= ReceiverIndex
and: [self isMarriedOrWidowedContext: rcvr])
ifTrue:
[self instVar: fieldIndex ofContext: rcvr put: anObject]
ifFalse:
[self storePointer: fieldIndex ofObject: rcvr withValue: anObject]
```

Reading is easy. Simply compute the relevant value, sender, pc or stackp from the spouse frame.

```
StackInterpreter methods for frame access
instVar: offset ofContext: aOnceMarriedContext
“Fetch an instance avriable from a maybe married context.
If the context is still married compute the value of the
relevant inst var from the spouse frame’s state.”
| spouseFP senderFP |
<inline: true>
<var: #spouseFP type: #’char *’>
<var: #senderFP type: #’char *’>
<var: #thePage type: #’StackPage *’>
<var: #theFPAbove type: #’char *’>
self assert: offset < MethodIndex.
self assert: (self isMarriedOrWidowedContext: aOnceMarriedContext).
self writeBackHeadFramePointers.
(self isWidowedContext: aOnceMarriedContext) ifTrue:
[^self fetchPointer: offset ofObject: aOnceMarriedContext].


	spouseFP := self withoutSmallIntegerTags: (self fetchPointer: SenderIndex ofObject: aOnceMarriedContext).
offset = SenderIndex ifTrue:
[(self isBaseFrame: spouseFP) ifTrue:
[^self frameCallerContext: spouseFP].
senderFP := self frameCallerFP: spouseFP.
^self ensureFrameIsMarried: senderFP].
offset = StackPointerIndex ifTrue:
[^self integerObjectOf: (self stackPointerIndexForFrame: spouseFP)].
offset = InstructionPointerIndex ifTrue:
[| theIP thePage theFPAbove |
spouseFP == localFP
ifTrue: [theIP := self oopForPointer: localIP]
ifFalse:
[thePage := stackPages stackPageFor: spouseFP.
theFPAbove := self findFrameAbove: spouseFP inPage: thePage.
theIP := theFPAbove == 0
ifTrue: [stackPages longAt: thePage headSP]
ifFalse:[self oopForPointer: (self frameCallerSavedIP: theFPAbove)]].
^self contextInstructionPointerForFrame: spouseFP IP: theIP].
self error: ‘bad index’
```

Writing is a different matter. The one case that’s really important for performance is assigning the sender because it is used in e.g. co-routining but also in stack unwinding and termination. It is trivial to assign the sender of a base frame as it already refers to a context, but to assign to the sender of any other frame requires ingenuity something that Peter Deutsch and Allan Schiffman have in abundance and their solution is to split the stack into two.



The assignee and all frames above it are moved to another stack, making the assignee the base frame, rendering assigning the sender trivial. Likewise handlng other assignment cases is done by moving all frames above the assignee to another stack, divorcing the assignee frame and assigning to the resulting single context. Once the assignee frame is a base frame we no longer have to split its stack if its sender is assigned again. So contexts that have their sender assigned often migrate to the bottom of stack pages, allowing the StackInterpreter to keep up with a context interpreter when co-routining.

StackInterpreter methods for frame access
instVar: index **ofContext:** aMarriedContext **put:** anOop
| theFP |
“Assign the field of a married context. The important case to optimize is
assigning the sender. We could also consider optimizing assiging the IP but
typically that is followed by an assignment to the stack pointer and we can’t
efficiently assign the stack pointer because it involves moving frames around.”
<inline: true>
self assert: (self isMarriedOrWidowedContext: aMarriedContext).
theFP := self frameOfMarriedContext: aMarriedContext.
index == **SenderIndex** ifTrue:
[| thePage onCurrentPage |
thePage := stackPages stackPageFor: theFP.
self assert: stackPage == stackPages mostRecentlyUsedPage.
(onCurrentPage := thePage == stackPage) ifTrue:
[self writeBackHeadFramePointers].
self storeSenderOfFrame: theFP withValue: anOop.
onCurrentPage ifTrue:
[localFP := stackPage headFP.
localSP := stackPage headSP].
^self].
self externalizeIPandSP.
self externalDivorceFrame: theFP andContext: aMarriedContext.
self storePointer: index ofObject: aMarriedContext withValue: anOop.
self internalizeIPandSP

storeSenderOfFrame: theFP **withValue:** anOop
“Set the sender of a frame. If the frame is a base frame then this is trivial;
merely store into the FoxCallerSavedIP field. If not, then split the stack at
the frame, moving the frame and those hotter than it to a new stack page.
In the new stack page the frame will be the base frame and storing trivial.
Answer the possibly changed location of theFP.”
| thePage onCurrentPage newPage theMovedFP |
<var: #theFP type: #'char *'>
<var: #thePage type: #'StackPage *'>
<var: #newPage type: #'StackPage *'>
<var: #theMovedFP type: #'char *'>
<returnTypeC: 'char *'>
(self isBaseFrame: theFP) ifTrue:
[stackPages longAt: theFP + **FoxCallerSavedIP** put: anOop.
^theFP].

```
self ensureFrameIsMarried: (self frameCallerFP: theFP).
thePage := stackPages stackPageFor: theFP.
self assert: stackPage == stackPages mostRecentlyUsedPage.
onCurrentPage := thePage == stackPage.
onCurrentPage ifFalse:
["Make sure the frame's page isn't divorced when a new page is allocated."
stackPages markStackPageNextMostRecentlyUsed: thePage].
newPage := self newStackPage.
theMovedFP := self moveFramesIn: thePage through: theFP toPage: newPage.
onCurrentPage ifTrue:
[self setStackPageAndLimit: newPage].
self assert: (self isBaseFrame: theMovedFP).
stackPages longAt: theMovedFP + FoxCallerSavedIP put: anOop.
^theMovedFP
```

```
moveFramesIn: oldPage through: theFP toPage: newPage
"Move frames from the hot end of oldPage through to theFP to newPage.
This has the effect of making theFP a base frame which can be stored into.
Answer theFP's new location."
| newSP newFP stackedReceiverOffset delta callerFP callerIP fpInNewPage offsetCallerFP theContext |
<inline: false>
<var: #oldPage type: #'StackPage *'>
<var: #theFP type: #'char *'>
<var: #newPage type: #'StackPage *'>
<var: #newSP type: #'char *'>
<var: #newFP type: #'char *'>
<var: #callerFP type: #'char *'>
<var: #fpInNewPage type: #'char *'>
<var: #offsetCallerFP type: #'char *'>
<var: #source type: #'char *'>
<returnTypeC: 'char *'>
newSP := newPage baseAddress + BytesPerWord.
stackedReceiverOffset := self frameStackedReceiverOffset: theFP.
"First move the data. We will fix up frame pointers later."
theFP + stackedReceiverOffset
to: oldPage headSP
by: BytesPerWord negated
do: [:source|
newSP := newSP – BytesPerWord.
stackPages longAt: newSP put: (stackPages longAt: source)].
"newSP = oldSP + delta => delta = newSP – oldSP"
delta := newSP – oldPage headSP.
newFP := newPage baseAddress – stackedReceiverOffset.
self setHeadFP: oldPage headFP + delta andSP: newSP inPage: newPage.
newPage baseFP: newFP.
callerFP := self frameCallerFP: theFP.
self assert: (self isBaseFrame: theFP) not.
self assert: (self frameHasContext: callerFP).
callerIP := self oopForPointer: (self frameCallerSavedIP: theFP).
stackPages longAt: theFP + stackedReceiverOffset put: callerIP.
oldPage
headFP: callerFP;
headSP: theFP + stackedReceiverOffset.
"Mark the new base frame in the new page"
stackPages longAt: newFP + FoxCallerSavedIP put: (self frameContext: callerFP).
stackPages longAt: newFP + FoxSavedFP put: 0.
"Now relocate frame pointers, updating married contexts to refer to their moved spouse frames."
fpInNewPage := newPage headFP.
[offsetCallerFP := self frameCallerFP: fpInNewPage.
offsetCallerFP ~= 0 ifTrue:
[offsetCallerFP := offsetCallerFP + delta].
stackPages longAt: fpInNewPage + FoxSavedFP put: (self oopForPointer: offsetCallerFP).
(self frameHasContext: fpInNewPage) ifTrue:
[theContext := self frameContext: fpInNewPage.
self storePointerUnchecked: SenderIndex
ofObject: theContext
withValue: (self withSmallIntegerTags: fpInNewPage).
self storePointerUnchecked: InstructionPointerIndex
ofObject: theContext
withValue: (self withSmallIntegerTags: offsetCallerFP)].
fpInNewPage := offsetCallerFP.
fpInNewPage ~= 0] whileTrue.
^newFP
```

It turns out that moveFramesIn:through:toPage: is useful for one more function that really helps performance, stack overflow. Consider what happens in the execution of benchFib:

```
Integer methods for benchmarks
benchFib
^ self < 2
ifTrue: [1]
ifFalse: [(self–1) benchFib + (self–2) benchFib + 1]
```

If the initial argument is larger than the number of activations of nfib that can fit on the remaining room in the stack page a stack overflow will occur, and probably soon thereafter a base return back to the original stack. At some point the system will thrash doing overflowing sends and underflowing returns. When we first started using the StackInterpreter inside Qwaq benchFib for large N would do well but benchFib for small N would do badly and the problem was overflow/underflow thrashing.

The simple fix is to increase the number of frames moved on each successive overflow from the same page. On the first overflow we move one frame, on the second two and so on. Soon the thrashing is taken care of because the active frames end up located on the same page. So here's stack overflow. It turns out we only need two variables to remember which page last overflowed and a count of frames to move.

```
StackInterpreter methods for frame access
stackOverflowOrEvent: numArgs mayContextSwitch: mayContextSwitch
"The stackPointer is below the stackLimit. This is either because of a
stack overflow or the setting of stackLimit to indicate a possible interrupt.
Check for interrupts and stackOverflow and deal with each appropriately."
| newPage theFP callerFP overflowLimitAddress overflowCount |
<var: #newPage type: #'StackPage *'>
<var: #theFP type: #'char *'>
<var: #callerFP type: #'char *'>
<var: #overflowLimitAddress type: #'char *'>

	"If the stackLimit differs from the realStackLimit then the stackLimit
has been set to indicate an event or interrupt that needs servicing."
stackLimit == stackPage realStackLimit
ifTrue: [self externalWriteBackHeadFramePointers]
ifFalse: [self checkForEventsMayContextSwitch: mayContextSwitch].

	"After checkForInterrupts another event check may have been forced, setting both
stackLimit and stackPage stackLimit to all ones. So here we must check against
the real stackLimit, not the effective stackLimit."
stackPointer < stackPage realStackLimit ifFalse:
[^nil].

	statStackOverflow := statStackOverflow + 1.

	"The stack has overflowed this page. If the system is executing some recursive algorithm,
e.g. fibonacci, then the system could thrash overflowing the stack if the call soon returns
back to the current page. To avoid thrashing, since overflow is quite slow, we can move
more than one frame. The idea is to record which page has overflowed, and the first
time it overflows move one frame, the second time two frames, and so on. We move no
more frames than would leave the page half occupied."
theFP := framePointer.
stackPage == overflowedPage
ifTrue:
[overflowLimitAddress := stackPage baseAddress – stackPages overflowLimit.
overflowCount := extraFramesToMoveOnOverflow := extraFramesToMoveOnOverflow + 1.
[(overflowCount := overflowCount – 1) >= 0
and: [(callerFP := self frameCallerFP: theFP) < overflowLimitAddress
and: [(self isBaseFrame: callerFP) not]]] whileTrue:
[theFP := callerFP]]
ifFalse:
[overflowedPage := stackPage.
extraFramesToMoveOnOverflow := 0].

	self ensureFrameIsMarried: (self frameCallerFP: theFP).
newPage := self newStackPage.
self moveFramesIn: stackPage through: theFP toPage: newPage.
self setStackPageAndLimit: newPage.
framePointer := stackPage headFP.
stackPointer := stackPage headSP.
"To overflow the stack this must be a new frame"
self assert: (self frameHasContext: framePointer) not.
self assert: (self validInstructionPointer: instructionPointer inMethod: method)
```

Garbage Collection

Before we can have a look at measurements I need to discuss garbage collection. There's not much difficulty grabae collecitng references frm stack pages. The garbage collector simply has to traverse the frames on in-use pages. There is a need to void stack pages that are not reachable to stop them holding onto garbage, but this also is easy. The major problem is the Squeak VM's habit of running the garbage collector inside an allocation when memory runs out. This is a feature it shares with a number of other garbage collectors in VMs and it is difficult to be emphatic enough when saying how horrible this is. It is incredibly difficult to live with and a huge source of bugs. Because the garbage collector can run, and hence move objects, on any allocation the VM must hold onto all unreferenced intermediate results it is using while allocating. For example, look at the start of the basic allocation routine in the standard Squeak VM:

```
ObjectMemory methods for allocation
allocate: byteSize headerSize: hdrSize h1: baseHeader h2: classOop h3: extendedSize doFill: doFill with: fillWord
"Allocate a new object of the given size and number of header words. (Note: byteSize already includes space for the base
```

```
header word.) Initialize the header fields of the new object and fill the remainder of the object with the given value.
May cause a GC"

        | newObj remappedClassOop end i |
<inline: true>
<var: #i type: 'usqInt'>
<var: #end type: 'usqInt'>
"remap classOop in case GC happens during allocation"
hdrSize > 1 ifTrue: [self pushRemappableOop: classOop].
newObj := self allocateChunk: byteSize + (hdrSize – 1 * BytesPerWord).
hdrSize > 1 ifTrue: [remappedClassOop := self popRemappableOop].
… etc …
```

Because allocateChunk might cause the GC to run the classOop might move and hence the "solution" (read "slow inconvenient hack") is to provide a stack to hold intermediate results (pushRemappableOop:/popRemappableOop) whose contents the GC updates when it runs. So where is the latent bug in the standard perform primitive?

```
Interpreter methods for control primitives
primitivePerform
| performSelector newReceiver selectorIndex lookupClass performMethod |
performSelector := messageSelector.
performMethod := newMethod.
messageSelector := self stackValue: argumentCount – 1.
newReceiver := self stackValue: argumentCount.

        "NOTE: the following lookup may fail and be converted to #doesNotUnderstand:, so we must adjust argumentCount and
slide args now, so that would work."

        "Slide arguments down over selector"
argumentCount := argumentCount – 1.
selectorIndex := self stackPointerIndex – argumentCount.
self
transfer: argumentCount
fromIndex: selectorIndex + 1
ofObject: activeContext
toIndex: selectorIndex
ofObject: activeContext.
self pop: 1.
lookupClass := self fetchClassOf: newReceiver.
self findNewMethodInClass: lookupClass.

        "Only test CompiledMethods for argument count – other objects will have to take their chances"
(self isCompiledMethod: newMethod)
ifTrue: [self success: (self argumentCountOf: newMethod) = argumentCount].

        successFlag
ifTrue: [self executeNewMethodFromCache.
"Recursive xeq affects successFlag"
successFlag := true]
ifFalse: ["Slide the args back up (sigh) and re-insert the selector. "
1 to: argumentCount do: [:i | self
storePointer: argumentCount – i + 1 + selectorIndex
ofObject: activeContext
withValue: (self fetchPointer: argumentCount – i + selectorIndex ofObject: activeContext)].
self unPop: 1.
self storePointer: selectorIndex
ofObject: activeContext
withValue: messageSelector.
argumentCount := argumentCount + 1.
newMethod := performMethod.
messageSelector := performSelector]
```

Well if the perform is not understood there will be an allocation of a Message within findNewMethodInClass:, which may cause a GC, and if the new method doesn't have the right number of arguments the statements

```
newMethod := performMethod.
messageSelector := performSelector
```

may assign invalid pointers to newMethod and messageSelector. It might bite only once in a blue moon but those are the worst kind.

Things are far worse in the StackInterpreter since any send, or assignment to a context instance variable, may cause an entire stack pages worth of contexts to be allocated as a stack page's frames are divorced so that the page can be reused. It is simply intolerable to live with pushRemappableOop:/popRemappableOop scheme. It has to go, or at least only rear its ugly head occasionally.

The change is to maintain sufficient free space such that we can at least allocate a page worth of contexts, scheduling a GC as soon as we've started eating into this reserve, but deferring the garbage collection until a safe point. I won't bore you wth the details, but there's a subclass of ObjectMemory called NewObjectMemory that coordinates this providing a set of eeInstantiate*

methods (for execution engine) that the VM uses for objects allocated in the course of execution (contexts, closures and messages for doesNotUnderstand:). It then only runs the garbage collector in checkForEventsMayContextSwitch:. The standard allocation primitives primitiveNew and primitiveNewWithArg still use the old scheme which is also provided for plugins so that they don't have to be rewritten, but inside the interpreter the remapBuffer is shunned entirely.

Proportions, Profiles and Prospects

This stack organization does make a difference all by itself. The StackInterpreter is substantially faster for activation/return benchmarks and insubstantially slower for process switch benchmarks. Here are the results of four benchmarks comparing the StackInterpreter against the standard context Interpreter. Both are compiled using the Intel C compiler and run on my 2.16GHz Intel Core Duo MacBook Pro.

{[self nbody: 200000 to: t1]}
took 5.289 seconds
ratio: 0.69 % change: -30.95%

{[self binarytrees: 15 to: t1]}
took 7.583 seconds
ratio: 0.526 % change: -47.4%

{[self chameneosredux: 260000 to: t1]}
took 7.095 seconds
ratio: 0.837 % change: -16.31%

{[self threadring: 10000000]}
took 8.263 seconds
ratio: 0.948 % change: -5.22%
geometric mean 0.733 average speedup -26.74%

These benchmarks are taken directly from the Squeak versions on the computer language shootout site (thanks Isaac!!).

nbody is a floating-point intensive N-body simulation of the Sun, Jupiter, Saturn, Uranus and Neptune. Some of the speedup (about a third) is due to improvements in the primitive floating point code (the use of compact class indices if you must know).

binarytrees is an allocation heavy binary tree creation and traversal benchmark. It benefits both from the stack organization and the GC changes.

Chameneos is a symbolic computation that compares many pairs of objects in one of three states, the colours red, blue and yellow.

threadring is a thread benchmark that passes a token between 503 Smalltalk processes.

As you can see we get worth-while speedups for everything except threadring, and that threading is not any slower.

Running Qwaq Forums, Qwaq's business communications solution written in Croquet, we see about a 15% improvement both in loading times and in frame rate. And we're getting ready to release this VM to the public as I write. I'll be putting a version for general consumption in a Monticello repository near you real soon now™.

If we compare the profiles of benchFib for the context Interpreter

> /Users/eliot/Qwaq/QFCI1.2.28.app/Contents/MacOS/Qwaq VM eem 12/30/2008 18:31
>
> 39 benchFib

gc prior. clear prior.
25.496 seconds; sampling frequency 1465 hz
37317 samples in the Interpreter (37347 samples in the entire program) 99.92% of total

% of interpret	(% of total)		(samples)	(cumulative)
13.46%	(13.45%)	L0internalActivateNewMethod	(5023)	(13.46%)
12.90%	(12.89%)	L0bytecodeDispatch	(4815)	(26.36%)
6.40%	(6.39%)	L0internalStoreContextRegisters	(2387)	(32.76%)
5.99%	(5.99%)	L0commonReturn	(2237)	(38.75%)
5.54%	(5.53%)	L0lookupInMethodCacheSelclass	(2067)	(44.29%)
5.53%	(5.53%)	L0bytecodePrimAdd	(2065)	(49.83%)
5.38%	(5.37%)	L0normalSend	(2007)	(55.21%)
4.95%	(4.95%)	L0bytecodePrimSubtract	(1847)	(60.15%)
4.62%	(4.61%)	L0pushReceiverBytecode	(1723)	(64.77%)
4.21%	(4.20%)	L0internalFetchContextRegisters	(1570)	(68.98%)
3.94%	(3.94%)	L0pushConstantOneBytecode	(1471)	(72.92%)
3.93%	(3.92%)	L0recycleContextIfPossible	(1465)	(76.85%)
3.08%	(3.08%)	L1internalFetchContextRegisters	(1151)	(79.93%)
2.86%	(2.86%)	L0internalQuickCheckForInterrupts	(1068)	(82.79%)
2.81%	(2.81%)	L0pushConstantTwoBytecode	(1049)	(85.60%)
2.63%	(2.63%)	L0sendLiteralSelectorBytecode	(981)	(88.23%)
2.33%	(2.33%)	L0returnTopFromMethod	(869)	(90.56%)
2.22%	(2.21%)	L0internalExecuteNewMethod	(827)	(92.78%)
2.07%	(2.07%)	L0booleanCheat	(774)	(94.85%)
1.68%	(1.68%)	L0pushReceiverVariableBytecode	(626)	(96.53%)
1.46%	(1.46%)	L0longUnconditionalJump	(546)	(97.99%)
1.17%	(1.17%)	L0bytecodePrimLessThan	(436)	(99.16%)
0.84%	(0.84%)	L0storePointerofObjectwithValue	(312)	(100.0%)
0.00%	(0.00%)	...others...	(1)	(100.0%)

to that of the StackInterpreter

/Users/eliot/Qwaq/QFSI1.2.28.app/Contents/MacOS/Qwaq VM eem 1/14/2009 12:23
eden size: 262,144 stack pages: 8

39 benchFib

gc prior. clear prior.
18.282 seconds; sampling frequency 1467 hz
26575 samples in the Interpreter (26824 samples in the entire program) 99.07% of total

% of interpret	(% of total)		(samples)	(cumulative)
20.17%	(19.99%)	L0bytecodeDispatch	(5361)	(20.17%)
9.13%	(9.05%)	L0internalActivateNewMethod	(2427)	(29.31%)
8.87%	(8.79%)	L0internalFindNewMethod	(2357)	(38.17%)
8.58%	(8.50%)	L0bytecodePrimAdd	(2279)	(46.75%)
7.41%	(7.34%)	L0commonCallerReturn	(1968)	(54.16%)
7.35%	(7.28%)	L0bytecodePrimSubtract	(1952)	(61.50%)
6.20%	(6.14%)	L0sendLiteralSelector0ArgsBytecode	(1647)	(67.70%)
5.98%	(5.92%)	L0pushReceiverBytecode	(1588)	(73.67%)
4.77%	(4.72%)	L0pushConstantOneBytecode	(1267)	(78.44%)
4.24%	(4.20%)	L0pushConstantTwoBytecode	(1126)	(82.68%)
4.04%	(4.00%)	L0booleanCheat	(1074)	(86.72%)
3.71%	(3.68%)	L0bytecodePrimLessThan	(987)	(90.43%)
2.40%	(2.38%)	L0pushReceiverVariableBytecode	(638)	(92.84%)
2.29%	(2.27%)	L0internalExecuteNewMethod	(608)	(95.12%)
1.86%	(1.84%)	L0longUnconditionalJump	(493)	(96.98%)
1.77%	(1.76%)	L0commonReturn	(471)	(98.75%)
0.66%	(0.66%)	L0slowPrimitiveResponse	(176)	(99.41%)
0.56%	(0.56%)	L0returnTopFromMethod	(149)	(99.97%)
0.02%	(0.02%)	L0baseReturn	(6)	(100.0%)
0.00%	(0.00%)	...others...	(1)	(100.0%)

we can see where this speedup comes. All of the context management on send and return is substantially cheaper and we see a noticeable increase in the cost of bytecode dispatch, showing that the VM has more time to do real work.

But 15% improvement is nothing for a system that, at least for activation/return benchmarks like benchFib, is some 20 times slower than VisualWorks. The overhead of bytyecode dispatch shows where we have to go next. We need to execute machine code rather than interpret bytecode. We need a JIT.

The StackInterpreter is an intermediate step after closures and before the JIT to ensure steady progress and on-time delivery of a substantially faster VM. The essential point, of course, is that a stack organization suits the use of native call and return instructions whose use, along with in-line cacheing techniques, substantially improve send and return performance. As you can see in my Simulate Out Of The Bochs post I've already started work on the JIT and I hope to be far more current in my blog posts about it than I've been on the StackInterpreter. Apologies.

I'm already enjoying this year immensely. The JIT, the first I've ever written in Smalltalk, is a delight. I hope to blog about it soon.

Thanks for reading.

Posted by admin on Wednesday, January 14th, 2009, at 12:51 pm, and filed under Cog.
Follow any responses to this entry with the RSS 2.0 feed.
You can post a comment, or trackback from your site.

{ 16 }

Comments

1. **Simon Michael** | 14-Jan-09 at 3:07 pm | Permalink

   Thank you Eliot! Great reading.

2. **Gilad Bracha** | 14-Jan-09 at 7:51 pm | Permalink

   Eliot,

   The opening of the post is truly wonderful; As for the rest, I haven't read it yet, because it's hard to read a book online.

   Let me say a couple of nasty things (no one expect me to say anything nice) about contexts:

   Contexts destroy procedural abstraction. I suppose many people will want that explained – it will, in re-education camp.

   Among other things, this means that contexts are a giant security hole. In a world where cyber-warfare is a reality, and cyber-terrorism not far behind, this is a problem.

   So contexts need to be tamed – by mirrors.

   All this is orthogonal to your main points, which are all about implementation.

   Cheers, Gilad

3. **Jecel Assumpção Jr** | 16-Jan-09 at 1:02 pm | Permalink

   Great description! One detail wasn't clear to me, however. Do processes share stack pages? My first impression was that you had gone to some trouble to allow this, but on a second reading this doesn't seem to be the case. This would mean that in the benchmark with the 503 processes the behaviour would fall back to nearly the same as the context interpreter given that there are only 192 pages.

   Btw, I would have gone in a completely different direction if I had tried to do this. I would allocate two stack per process (64 words each might be a good start) with one being the data stack and the other the return stack (yes, I did design too many Forth processors). The return stack would have frames of four words each: base (pointer to the receiver in data stack), contextOrNil, method and ip. There would be no copying at all on the data stack when adding a frame to the return stack.

4. **S Krishnamachari** | 17-Jan-09 at 1:52 pm | Permalink

   Great Explanation..! Takes me down a memory lane from where I left nearly 3 yrs back. Motivating to head back and look through this again from the Slang implementation perspective. As for the Context, I thought the link to the Context Management would be pertinent here.

   Would you be releasing the Slang Code of the VM or the converted C version..?

   Will follow this more often..

5. **Eliot Miranda** | 18-Jan-09 at 10:47 pm | Permalink

   Hi Gilad,

   to agree contrarily, I'm not sure contexts should be so roughly accused. Common or garden stacks are extremely useful and yet can be exploited in buffer overrun attacks, but only if the computational context allows it. In a safe language a buffer overrun attack is impossible. Exceptions and setjmp/longjmp catch/throw all break procedural abstraction, but they are none the less useful for that. As you point out a system with contexts can still be secure (e.g. by using mirrors). So I wouldn't go as far as to call them evil, merely live 🙂

   But yes, I'm merely discussing how to implement them efficiently. I'm implementing a VM for a language that has unsafe contexts, as efficiently as I can, not doing the harder thing of designing a new language that provides the power of contexts along with safety guarantees. When I'm up to that challenge and given the opportunity I shall be extremely eager to be re-educated.

   Yes, it is my intent to write at least a draft of a book on dynamic language implementation that presents a high-performance design warts and all. Perhaps the medium isn't ideal but it is for me great practice. I've previously been overawed by the prospect but writing a blog allows me to write a chapter at a time at my own pace. Please don't be put off by my prolixity. I very very much appreciate your being a reader.

   Best
   Eliot

6. **Eliot Miranda** | 18-Jan-09 at 10:56 pm | Permalink

   Hi Jecel,

   I'll revise my post to clarify as soon as I have time. Yes, processes share the stack zone, no, processes don't share stack pages. The frames and contexts on a stack page all belong to exactly one process.

   Yes, this implies that when the number of active processes exceeds the number of stack pages every context switch must flush the least recently used stack page to make room for its top context. So yes, performance suffers when this happens, and you can see form the benchmarks that threadRing: has a far smaller speedup that the other three. However, a ring of processes of the same priority is an artificial benchmark which has little to do with typical process usage patterns in real applications and one can have a lot of stack pages on current machines.

   Your different direction sounds very similar to the one I took in my BrouHaHa VM as described in my OOPSLA '87 paper. Yes this works but not as well as Peter and Allan's stack page scheme. The problem is that in this scheme there is a context overflow on every send as soon as one has more frames than the one stack page. Whereas with the stack page scheme typical usage patterns imply that the stack zone is rarely full and on most overflows a free stack page is available.

   Cheers,
   Eliot

7. **Eliot Miranda** | 18-Jan-09 at 11:00 pm | Permalink

   Hi Skrish,

   yes I'll be releasing the full Smalltalk code. The C code wouldn't be much use, as one couldn't practically add another processor to it. I fervently hope that others will port to new processors and improve the code in any way.

   regards
   Eliot

8. **Jecel Assumpção Jr** | 22-Jan-09 at 10:26 am | Permalink

   I'll try to read your BrouHaHa paper the next time I go to the university (I can access the ACM portal from there). My main worry was about the additional complexity of #temporary:in: since it is so basic. But perhaps only very specific benchmarks would reflect the extra cost (even the "sieve", used to calculate bytecodes/sec, is probably dominated by #at:put: rather than temporary variable access). And for a Jitter you would take advantage of the fact that the offset argument is always a constant to compile away the "if" and nearly all of the math (so you would end with just "theFP + compileTimeConstant").

   About overflows, there is always the solution that is so popular in Smalltalk's collection library: allocate a new stack twice as large and copy everything from the old one there. This gets slower and slower but should always be a small percentage of the total computation that creates this situation. I am supposing that these stacks are heap allocated objects which can be arbitrarily large, though I hope it would be very rare for the initial size of 64 words not to be enough, much more so for kilobytes or even megabytes to be needed.

9. **tim@rowledge.org** | 02-Apr-09 at 11:10 am | Permalink

   I suspect that stackpage overflows and the resultant flushing (and re-caching) can be avoided relatively simply. If the stack chunks are allocated within OM rather than outside then more chunks can very easily be created at any point where there is any available memory.

   You could of course dynamically create stack pages in C memory but why implement another memory management system when you already have a fairly good one? Creating bytearray objects is simple enough; you could specialise and avoid the zeroing out unless the guaranteed 0 content turns out to be useful.

The obvious problem is the moving around of these stackframe objects during GC but really all it is is pointer swizzling after a move; no move, no swizzle – and rather a lot of these stackframes are going to end up in oldspace anyway. Hmm, that means remembered table work… but we know they are special and that should help optimise out the issue.

It's a while since I thought about any of these issues but I think that keeping memory usage under GC control is generally a good move.

10. **Eliot Miranda** | 02-Apr-09 at 11:31 am | Permalink

    Hi Tim,

    great to see your face round these parts again!
    Stack pages don't really need managing like GC'ed memory. They are a finite resource allocated at start-up. Instead the management is above them; mapping activations to stack pages, rather than allocating/reclaiming pages.

    Actually you don't want to add new pages because pages are roots to the scavenger and full garbage collector. So adding pages slows down garbage collection because there are more roots to scan.

    Further, allocating stack pages in GC'ed memory can't work simply because the GC will move pages, which would complicate the whole context-to-stack mapping scheme. To map a context to its stack frame one stashes the frame pointer inside the context, and updating these frame pointers when a page moved is not easy to do efficiently. Also, the cost of the check for a valid frame pointer in a context would also be much higher if stack pages could be located any where in memory.

    If one looks at actual logs of activity one does see plenty of overflows (because a call chain is often deep enough to reach the end of a page and need to continue on a new page) but the number of divorces (when an overflow needs to vacate a page already in use because there are no empty pages left) is a rather rare event (tens a second on very fast machines and a vanishingly small percentage of overall execution time). So it turns out not to be something to worry about.

    One of the benefits of this stack organization for Squeak that I hadn't realised is that of avoiding zeroing the reclaimableContextsCount on context switch. One has to zero this count on context switch to avoid incorrectly reclaiming contexts when continuing after a context switch, so in a normal Squeak VM context allocations spike after each context switch. Andreas was profiling our server the other day and saw much better context switch performance and realised it was due to the stack VM not having to allocate contexts on context switch (because it doesn't allocate contexts on send and hence doesn't need to use the reclaimableContextsCount hack).

    To sum up, keeping the number of pages finite is good for locality and GC performance, keeping them contiguous is essential for efficient context-to-stack mapping and in practice divorces aren't an issue.

11. **Chris Cunnington** | 05-Jul-09 at 10:11 am | Permalink

    This is fairly noob questioning, it won't help anybody build a VM, but … if you're here.

    I was playing with the Vice emulator recently, which emulates all the Commodore computers, specifically the PET 2001. I found my book on assembler and started boning up on that, which is good fun. Then I thought about how this transferred to Smalltalk. Sigh.

    I would like to say I'm stunned that the process I use to make HelloWorld in Seaside is basically the same as used by CompiledMethod or MethodContext. That seems a pretty staggering level of internal consistency, to me.

    I understand CompiledMethod and MethodDictionary pretty well. I don't understand exactly how thisContext relates to MethodContext. Not a lot of different contexts on the PET 2001. A lot of pigeon holes for a piece for a datum; maybe two levels of scoping; no recursion. Not a mind bender, Commodore BASIC.

    Context is a … mirrored concept in Smalltalk, I'd say. You define your own. Saving thisContext as a chunk to create a continuation. OK. Tell thisContext to stop and shoehorn in the old list of processed commands to call a continuation. OK. But Smalltalk seems no nebulous. What am I saving a chunk of?

    You gave two examples at the top of the article. thisContext inspect and thisContext inspect. self halt. Everything else after that was pretty much lost … no, totally lost on me.

    What's the difference between the result of these two pieces of code evaluated in Workspace? Both contain a CompiledMethod with what I assume is a header number. I'm going to assume that the program counter in Smalltalk is the same as on a 6502 — a list of numbers iterated one at a time beside which are op codes. 26 is woefully low. A PET 2001 has far more. Your first post in 2006 said each context had it's own stack. Maybe the numbers are low on these stacks? Mini stacks? Stack-lettes?

    And the final difference is the class and selector. Again, why did you choose these two… activation record… can be MethodContext or BlockContext (thank you, Tim) … two kinds … thisContext will give you one or the other…

    No, I've lost it. I'm looking at these two MethodContexts, and I don't understand what I'm supposed to glean from the contrast. If you'd tell me, I'd really appreciate it.

12. **Eliot Miranda** | 05-Jul-09 at 1:48 pm | Permalink

    Hi Chris,

    the difference between them is that the one from

    ```
    thisContext inspect
    ```

    has been returned from, and so you'll notice that it's sender an dpc are both nil, whereas the one from

    ```
    thisContext inspect. self halt.
    ```

    is still active and its sender is the context in which the compilation of "thisContext inspect. self halt." was invoked, and its pc is pointing to the bytecode following the send bytecode for the halt message. So with the latter you can explore the current computation using either the inspector or the explorer or the Debugger which is of course available to you because "self halt" brought one up.

The relationship between a MethodContext and thisContext is that method activations are represented by instances of methodContext and so inside a method thisContext refers to an instance of MethodContext. Conceptually a MethodContext is instantiated whenever a message is sent that activates a method. The rest of the post explains how to avoid the overheads involved in doing so.

BTW, a context's pc is the pc in its method and methods are small and so pc values are small. A context's pc isn't a pc into some large flat code space. There's a lot here I'm leaving out which is why you might want to read the online implementation section of the "Blue Book" which is here before diving into my blog:
Blue Book Implementation Chapters on the Wayback machine

13. **Dave** | 10-Mar-10 at 7:07 pm | Permalink

    You've been fairly quiet on your blog Eliot. I have missed your in-depth posts. Any chance of something new for 2010? 😀

    (Having said that I realise you are a constant contributor on the squeak/pharo mailing lists)

14. **Measuring Simplicity « Squeaking Along** | 29-Apr-10 at 7:47 pm | Permalink

    […] in particular when trying to implement the model efficiently, can be highly complex (check out Eliot's comments on context to stack mapping in […]

15. **Ryan Macnak** | 08-Dec-18 at 1:34 pm | Permalink

    Hi Eliot,

    When marrying a frame, you copy the arguments to the context. Why not skip this and say arguments are unavailable for a returned-from context like the other temporaries? Have you encountered code that relied on the arguments being available in some context after it had returned?

16. **admin** | 09-Dec-18 at 10:34 pm | Permalink

    Hi Ryan,

    I choose to do this to maintain some equivalence between the new design and the classic Smalltalk-80 semantics that arise because of the use of real context objects. In Smalltalk-80 one can always rely on the arguments of a Context being valid (arguments are not writable in Smalltalk-80, and because Contexts are real objects, any arguments in a Context will be preserved until the Context is garbage collected). So it seemed to me that Contexts in the new closure design should at least preserve arguments.

    The situation arises where one wants to determine where some long-lived block was created. For example, it may be the case that one has populated some command-name to action Dictionary with mappings from command names to blocks, and it may be that the original methods in which those blocks were c related got redefined and therefore the methods for the blocks are now unbound, and hence that those blocks are unbound. Having the selector name, the class and arguments for the home context of the block are all useful in trying to determine the history of those blocks and hence help in making sense of a confused situation.

    There's another reason, which is consistency between a closure and its enclosing contexts w.r.t. copied values. It would;d be strange for a closure to hold onto specific copied values and find that its enclosing contexts, retained because the closure references them, which at one time referred to those same copied values because they were the source of the closure's copied values, were no longer referenced from the context. Alas my implementation does indeed maintain this consistency only for values copied from arguments. Updating the stack of a married context when a closure is created would indeed be expensive.

    It seems to me a reasonable performance compromise to not preserve the values of local variables (after all, these widowed contexts have been returned from), and hence not to have to do any work at return time, but to preserve their arguments, since copying arguments from the stack to the Context object when contexts are married, involves only a little more work, and has relatively low performance overhead. And it should be the case that Scorch/Sista reduces this overhead by virtue of inlining. You may come to a different conclusion for your system, valuing performance over debuggability. That's a perfectly reasonable position to take.

    As far as whether I've seen code that relied on it, I have a nagging memory that indeed I have seen such code, but I can't be sure. I should have taken notes 😉 I have forgotten.

    Anyway, you asked for my reasons and they are as stated 🙂

Post a Comment

Your email is *never* published nor shared. Required fields are marked *

Name *

Email *

Website

Message

Post

« THE IDÉE FIXE AND THE PERFECTED PROFILER

ESUG SCREENCAST »