COLA Kernel Abstraction

Ian Piumarta

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2009-007

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

COLA Kernel Abstraction

Ian Piumarta

`ian@vpri.org`

2009–08–22

Contents:

1 Introduction

A useful program must interact with its environment. A useful environment isolates programs from resource management, the complexity and diversity of devices, and malignant interference from other programs. Virtualisation of the execution environment has proved effective for all these roles.

Our interest in virtualisation is also driven by the desire to perform cheap and *safe* experiments with novel (and potentially dangerous) language semantics and execution mechanisms, within and without risk of disruption to an interactive environment. Inspired by the nested virtualisation model of Flke [Ford96] we have designed a particularly simple and lightweight model of recursive kernels that promises to be general, extensible, composable and more scalable than most. The symbiotic marriage of language and operating system techniques made possible by the STEPS system is critical to the success of our virtualisation model.

2 Kernels

A *kernel* is a collection of primitive behaviours. An opertaing system kernel is a virtual machine whose abstract operations are the system calls provided to applications and middleware. Our kernels are similar, with each providing a small set of related services often synthesized from lower-level services provided by "more primitive" parent kernels; in addition, they encapsulate any state required by the kernel to provide those services.

Every execution context in the STEPS system is associated with a kernel that provides services to the running program. When a procedure is called a new execution context is created in which the callee will run. The callee context inherits its kernel from the caller. When execution returns from a context, the kernel of the caller is implicitly restored as control is passed back to the caller's context.

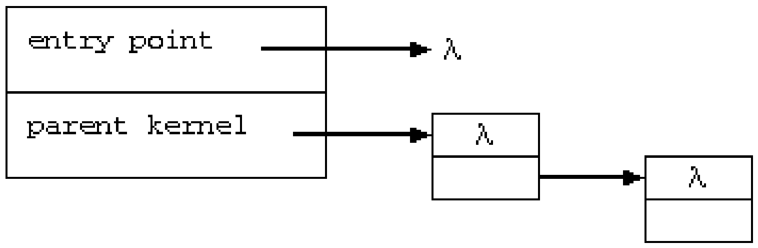
Calling a kernel service is a normal procedure call in which the callee context does not inherit the kernel of the caller. Instead the callee's context will be set to the parent kernel of the caller's kernel (in other words, the kernel responsible for providing services to the caller's kernel). The call to a kernel service can be explicit (similar to a "syscall" in a traditional operating system) or implicit according to some condition

A kernel service can be provided that associates a new kernel with the caller's context. The new kernel will become active when control returns to the caller's context (which is running the application program) and will remain active until that context returns, and will be inherited by any procedures called from that context. The scope of a kernel is therefore determined dynamically by the call graph of the application program.

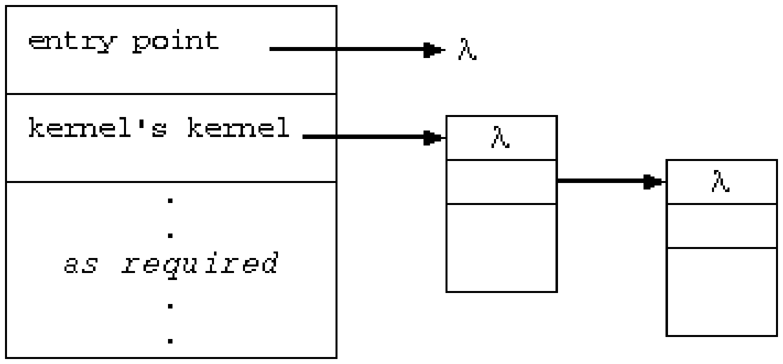
Unlike objects, which typically belong to one type or class for their entire lifetime, an application can rapidly pass through many kernels during its execution. The encapsulation provided by a kernel can be thought of as the dynamic, behavioural dual of the static encapsulation of state provided by objects.

2.1 Kernel representation

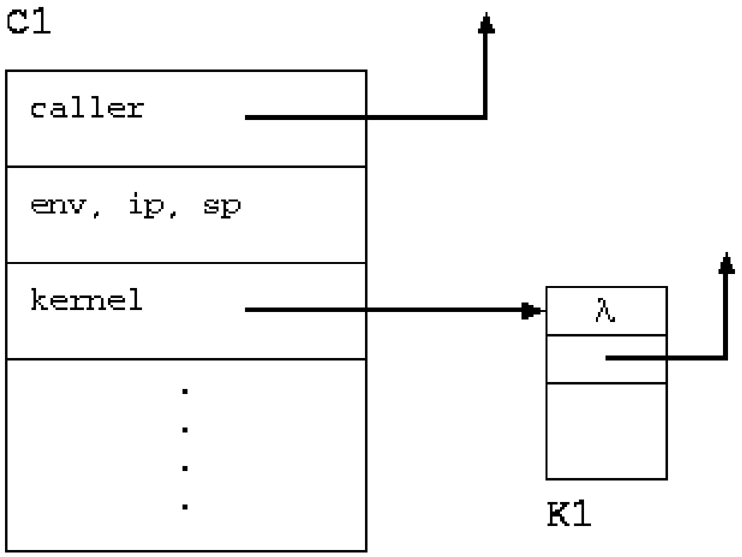
The smallest possible kernel is a tuple containing that kernel's entry point and a reference to its parent kernel.



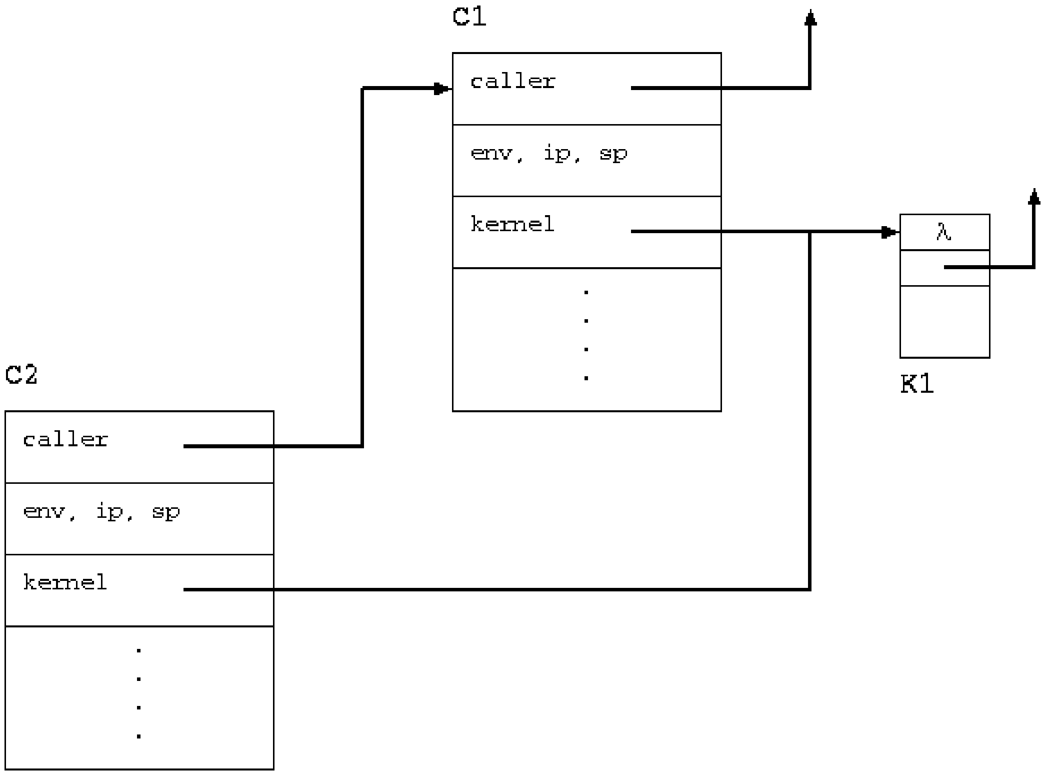
The entry point is a procedure that can have any useful signature, such as the name of the kernel service being invoked followed by a variable number of additional arguments depending on the service. The kernel can contain additional information if required by the kernel for its operation. (We might compare this additional information with the "u area" describing the user state of a Unix process.)



Every context (procedure activation) in the STEPS system is associated with a kernel which provides resources to the program. In the following diagram the context `C1` makes *kernel calls* in `K1` to request system services.



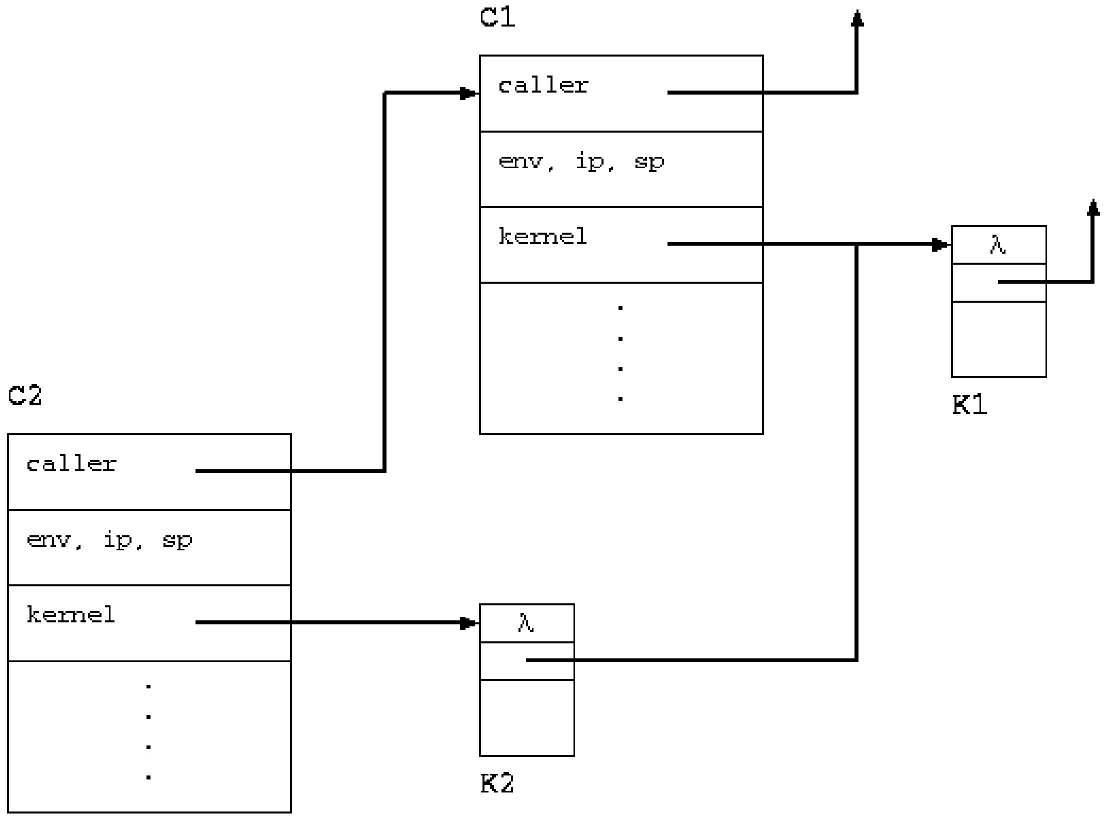
A normal procedure call creates a new context in which the callee will run. The new context inherits its caller's kernel; in other words, normal procedure call does not change the program's provider of system services.



Programs can deviate from this behaviour in two ways: creating a new kernel and calling a kernel entry point.

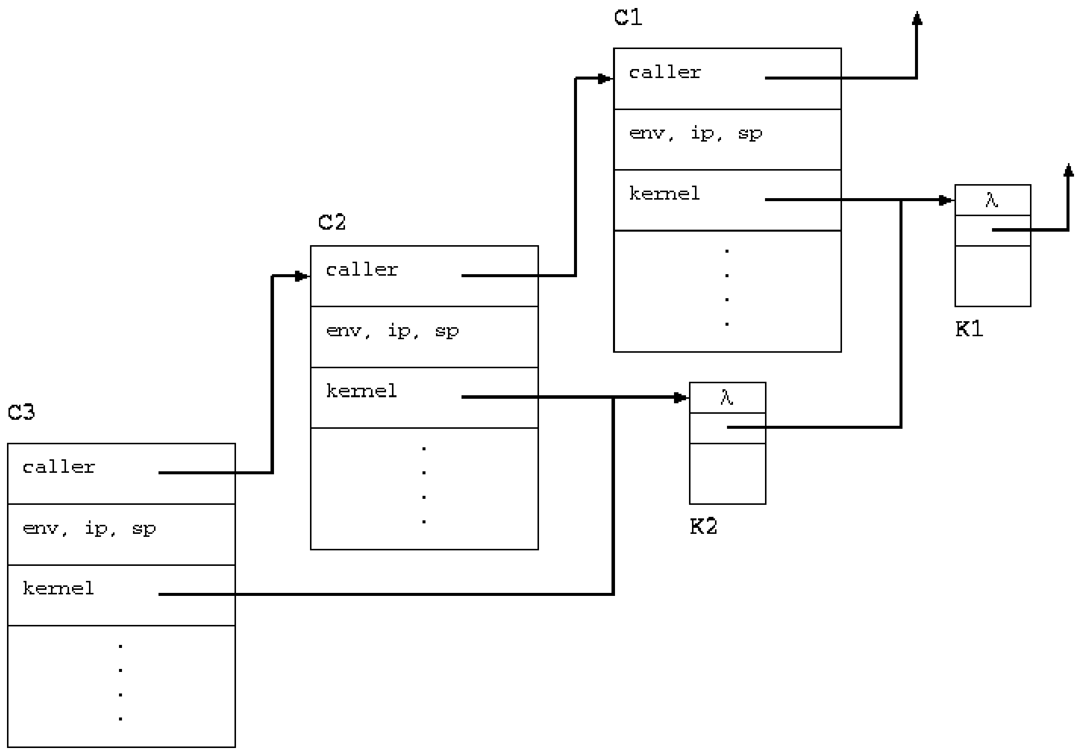
2.2 Creating a new kernel

A program can change the kernel associated with the active context. In the diagram above `C2` has inherited `K1` from its caller. The program can associate a new kernel `K2` with the running context `C2` by "pushing" `K2` onto the "kernel stack".



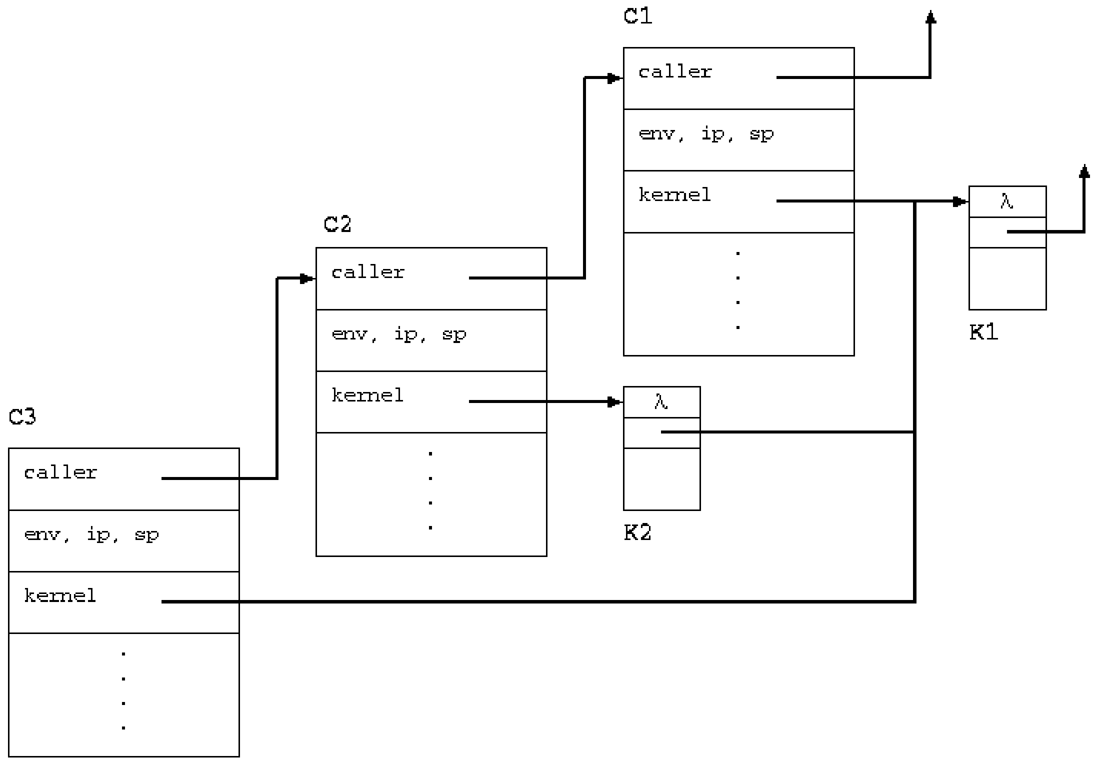
The procedure running in `C2`, as well as any new procedures called from it, will use `K2` to provide system services. `K2` remains the active kernel until `C2` returns to C1 which implicitly makes `K1` active again.

A normal procedure call made in `C2` will behave as described above, creating a context `C3` (with inherited kernel) to run the callee procedure.



2.3 Calling a kernel entry point

A kernel entry point is a procedure that runs on behalf of a kernel rather than the user program. The kernel's parent kernel must become active during the call. This is analogous to entering a higher level of privilege for the duration of a system call.



The kernel procedure associated with `K2` runs in `C3` using resources provided by `K1`. When control returns from the kernel procedure to the application procedure running in `C2`, the active kernel is restored to `K2`.

3 Complete example kernel: pre-emptive multitasking

This example adds pre-emptive multitasking to the STEPS core programming language as a symbiotic combination of language and kernel support.

A "budget" counter is added to the minimal kernel above. The budget counts (approximately) the number of loop iterations and procedure calls allowed in the process before the kernel is given an opportunity to preempt it in favour of some other quiescent process.

3.1 Overview

We will arrange for the budget to be decremented before every backward branch and in every function prologue. A new instruction `tick` will be added to the instruction set to perform this decrement and test.

```
(define-instruction tick   (let ((budget (kernel-budget
kernel)))     (if budget         (let ()           (set
(kernel-budget kernel) (set budget (- budget 2)))
(if (and (== budget 0) (kernel-entry kernel))
(let ()         ;; make the current context's kernel be the
kernel's kernel               (set (context-kernel
context (kernel-parent kernel)))             ;; call the
kernel entry point with the original kernel as argument
(push kernel)               (push (kernel-entry kernel))
(push 1)              (apply))       ; the 4 items left
on the stack will be popped by a following 'drop 4'
instruction            (set ip (+ ip 1)))) ; budget ok:
skip the following 'drop 4' instruction      (set ip (+ ip
1)))))      ; no budget counter: skip the following 'drop
4' instruction
```

The 'tick' instruction may or may not enter the kernel. To enter the kernel the 'tick' instruction sets the active kernel to the kernel's kernel, pushes several items onto the stack and then 'applies' the kernel entry point to them. The caller is responsible for popping these four values (caller's kernel, entry point, argument count and return value) off the stack, so the 'tick' instruction will always be followed by a 'drop 4' instruction. If the kernel is not entered then the 'tick' instruction skips over the 'drop 4' instruction.

```
        ...
        branch-always test
label:  loop body ...
        tick                    ; pushes 0 or 4 items, skips 1 or 0 instructions
        drop 4
test:   iteration test
        branch-true label
        ...
```

3.2 Kernel support

The kernel entry point detects entry from a tick instruction by the absence of an operation argument. Since the kernel's kernel was made current by installing it in the caller's context just before entering the kernel, the original user kernel must first be restored explicitly. Then the kernel can elect to pre-empt the caller or reset its budget to the initial value.

```
(define kernel-entry   (lambda (caller-kernel operation .
arguments)     (if operation     ;; kernel called explicitly
(perform the requested operation and return to caller)
;; kernel called implicitly from 'tick' instruction
(pre-empt the caller if appropriate)       (set (kernel-
budget caller-kernel) initial-budget)       (set (context-
kernel (context-caller (current-context)) caller-
kernel)))))
```

3.3 Compiler support

The STEPS language compiler generates an intermediate representation that can be interpreted or further compiled to native code. Two compiler functions are of interest to us: `compile-body` generates the body of a function (the binding of the incoming arguments, the body of the function, and the return of the final value); `compile-while` generates a looping construct (loop body, test, backwards jump to body). A program cannot make progress (or consume resources) without frequently either calling a procedure or making a backward branch.

Each of these code generation functions are augmented with a single extra line:

```
(define while (syntax   (lambda (program env . form)
(let ((cond (cadr form))          (prog (cddr form)))
(emit program 'bt   nil)       (let ((loop (pc program)))
(compile-expr program cond env)         (let ((test (pc
program)))           (compile-program program prog env)
(compile-tick    program) ;;; <--- this line added
(emit program 'pop  nil)           (set-cdr (car loop) (pc
program))           (emit program 'br test)           (emit
program 'push nil)           nil))))))  (define compile-
body   (lambda (body scope env)     (let ((program (cons
scope '((return . 1)))))       (compile-program program
body env)       (compile-tick    program) ;;; <--- this
line added       program)))  (define compile-tick       ;;
kernel software interrupt check   (lambda (program)
(emit program 'drop 4)   ;; result nArgs=1 entryFunc
userKernel     (emit program 'tick nil)))
```

These two additional lines call `compile-tick` which emits two instructions: a `tick` instruction which will consume one unit of budget and either invoke the kernel entry point or skip over the following `drop 4` instruction which is executed only when the kernel has been entered, to clean the arguments and return value from the stack.

4 Some other uses for kernels

We plan to use kernels for encapsulation of program semantics and pragmatics (system services), with both global and locally-scoped kernels. In addition to those described above:

- Various kinds of concurrency behind a single interface: processes, threads, user-level "green" threads.
- Synchronisation primitives from which locks and semaphores are constructed.
- Memory management: both varying collection strategies and isolation between zones using the same collection strategy.

- Communication services that configure themselves according to the kinds of concurrency present in the system.
- Kernels as containers for whole-language semantics: the meaning of an abstract representation determined by the particular kernel active during its interpretation.

5 References

[Ford96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back and Clawson, Stephen. *Microkernels meet recursive virtual machines*. Proceedings of the second USENIX symposium on Operating Systems Design and Implementation (OSDI '96), pp. 137–151. ISBN 1-880446-82-0.