Engines Build Process Abstractions

by

C. T. Haynes and D. P. Friedman

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 159

Engines Build Process Abstractions

June, 1984

by

C. T. Haynes and D. P. Friedman

This material is based on work supported by the National Science Foundation under grant numbers MCS 83-04567 and MCS 83-03325.

Engines Build Process Abstractions*

Christopher T. Haynes

Daniel P. Friedman

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, Indiana 47405 USA

Abstract

Engines are a new programming language abstraction for timed preemption. In conjunction with first class continuations, engines allow the language to be extended with a time-sharing implementation of process abstraction facilities. To illustrate engine programming techniques, we implement a round-robin process scheduler. The importance of simple but powerful primitives such as engines is discussed.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**] Applicative (Functional) Programming; D.1.3 [**Programming Techniques**] Concurrent Programming; D.3.2 [**Programming Languages**] Language Classifications—*extensible languages;* D.4.3 [**Programming Languages**] Language Constructs—*concurrent programming structures; control structures;* D.4.1 [**Operating Systems**] Process Management—*concurrency; scheduling;* D.4.6 [**Operating Systems**] Security and Protection—*access controls; security kernels;*

General Terms: Languages, Security

Additional Key Words and Phrases: Engines, first class objects, preemption, continuations

* A preliminary version of this paper was presented at the *1984 ACM Symposium on LISP and Functional Programming*. This material is based on work supported by the National Science Foundation under grant numbers MCS 83-04567 and MCS 83-03325.

1 Introduction

In this paper we introduce a novel facility, called *engines,* that abstract the notion of *timed preemption*. In conjunction with the ability to maintain multiple control environments, engines allow the base language to be extended with a time-sharing implementation of process abstraction facilities.

By designing a base language with a few general abstraction mechanisms such as engines and continuations, a powerful basis for building programming environments is provided. In this paper we are *not* introducing yet another process abstraction; rather, we are proposing a low-level mechanism that allows time-sharing implementation of arbitrary process abstractions. This assures language extensibility and suitability to a wide range of applications, as well as less rapid obsolescence.

In the next section we provide an overview of Scheme 84, a Scheme dialect which provides first class functional and continuation objects. We then define the engine mechanism as it is currently implemented in Scheme 84 [Friedman *et al.* 84], and include some simple examples of its use. Next we illustrate a more elaborate use of engines by implementing a simple time-sharing process scheduler. This implementation is then extended to provide different forms of process abstraction. We conclude with a discussion of the relation between traditional concurrent programming implementation techniques and those employed here.

2 An Overview of Scheme 84

Scheme was designed and first implemented at MIT in 1975 by Gerald Jay Sussman and Guy Lewis Steele, Jr. [Steele 78, Sussman 75] as part of an effort to understand the actor model of computation [Hewitt 77]. Scheme may be described as a dialect of Lisp that is applicative order, lexically scoped, and properly tail-recursive. Most importantly, Scheme—unlike all other Lisp dialects—treats functions and continuations as first class objects.

A subset of Scheme 84 [Friedman 84], an extended version of Scheme with engines, is composed of the following syntactic constructs:

```
⟨expression⟩ ::=
      ⟨constant⟩
    | ⟨identifier⟩
    | ⟨syntactic extension⟩
    | (lambda ( {⟨identifier⟩} ) {⟨expression⟩} )
    | (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩ )
    | (set! ⟨identifier⟩ ⟨expression⟩ )
    | (call/cc ⟨expression⟩ )
    | (engine ⟨expression⟩ )
    | ⟨application⟩

⟨syntactic extension⟩ ::= (⟨keyword⟩ {⟨expression⟩} )

⟨application⟩ ::= ( ⟨expression⟩ {⟨expression⟩} )
```

`lambda` is the sole binding operator of this subset and evaluates its expressions sequentially, returning the value of the last. `set!` side-effects an existing identifier binding or initializes a global identifier. `call/cc` and `engine` are described below. An application evaluates its expressions (in an unspecified order) and applies the value of the first expression to a list of arguments formed from the values of the remaining expressions.

Scheme 84 provides a syntactic preprocessor that examines the first object in each expression. If the object is a syntactic extension (macro) keyword, the procedure associated with the indicated syntactic extension is invoked on the expression, and the expression is replaced by the resulting transformed expression. If the object is not a keyword or core expression identifier (`lambda`, `if`, etc.), then it is assumed that the expression is an application.

We use a few syntactic extensions. `let` makes local identifier bindings. `define` changes an existing identifier binding or initializes a global identifier. The syntactic extension

$$\texttt{(rec } I\ E\texttt{)} \equiv \texttt{(let ([}I\ \texttt{'*]) (set! } I\ E\texttt{))}$$

evaluates E in an environment which binds I to the value of E itself.† Evaluating E should not result in dereferencing I. This presents no problem in the usual case when E evaluates to a closure.

$$\texttt{(case } Tag\ \texttt{[}A_1\ \ E_1\texttt{]}\ \ldots\ \texttt{[}A_n\ \ E_n\texttt{])}$$

evaluates Tag and then returns the value of the first E_i such that A_i matches the value of Tag.*

`call/cc` evaluates its argument and applies it to the current continuation represented as a functional object of one argument.‡ Informally, this continuation represents the remainder of the computation from the `call/cc` application point. At any future time this

† $\equiv$ indicates that an expression of the form on the left is transformed into one of the form on the right, and brackets are interchangeable with parentheses.

* Some readers may be put off by the number of parentheses in Scheme programs. However, we feel these parentheses are justified for several reasons. In the first place, there are more advantages of such minimal syntax than is generally realized. A simple, unambiguous syntax aids comprehension and provides the basis for a truly extensible language. (Users may easily define their own syntactic extensions, or macros, thereby extending the syntax of the language to meet their needs.) Secondly, the difficulties associated with heavily parenthesized expressions are greatly reduced with some practice and intelligent tools, such as editors and pretty-printers [Sandewall 78]. Finally, in this paper we are concerned with semantics and do not wish to be burdened by elaborate syntax. The obdurate reader may imagine these semantics expressed in his or her favorite syntax.

‡ Using this primitive we can define `catch`, a version of Landin's J operator [Landin 65, Reynolds 72, Sussman 75].

continuation may be invoked with any value, with the effect that this value is taken as the value of the `call/cc` application. (The continuation of a continuation application is discarded; unless, of course, it has been saved with another `call/cc`.)

To formally specify the current continuation at any point in a computation, a *continuation semantics* is necessary. We provide such a semantics with the meta-circular interpreter of a subset of Scheme 84 in the Appendix. (See [Friedman *et al.* 84] for a meta-circular interpreter of the entire core of Scheme 84.)

In a continuation semantics, every recursive procedure receives a continuation parameter. Rather than simply return, the procedure may pass its result directly to this continuation, or pass the continuation (possibly embellished) to some other procedure, which is expected to finish the computation. For example, consider `meaning` (see the Appendix). The constant, identifier, and lambda expressions pass their values directly to the continuation `k`. In the `if` case, `meaning` is recursively invoked on the `test-pt` (predicate) of the expression in the current environment, and with a new continuation, denoted by a lambda expression, which will be passed the value of the `test-pt`. Depending on this value, `meaning` is invoked on the `then-pt` or `else-pt` with the original continuation `k` of the `if` expression.

The function `tick` and other aspects of engine semantics defined by the meta-circular interpreter will be described in the next section, after an informal introduction to engines.

3 Definition of Engines

Metaphorically, a Scheme 84 engine is run by giving it a quantity of fuel. If the engine completes its computation before running out of fuel, it returns the result of its computation and the quantity of remaining fuel. If it runs out of fuel, a *new* engine is returned that, when run, will continue the computation.

More formally, engine fuel is measured in *ticks*, an unspecified unit of computation. Though in some implementations a tick may represent (as the name suggests) a fixed amount of time, this is not required by the engine abstraction. Ticks measure *computation*, not time.† An *engine* is a functional object of three arguments. An expression of the form

`(engine` *Exp*`)`

returns a new engine that, when applied to a positive integer n, a two-argument *success* function and a one-argument *fail* function, proceeds with the evaluation of *Exp* for n ticks. If the evaluation of *Exp* is completed in m ticks, for some $m \leq n$, then the success function is applied to the value of *Exp* and the number $t = n - m$ of ticks remaining. If the evaluation of *Exp* is not completed in n ticks, then the fail function is applied to a new engine that, when invoked, will continue with the evaluation of *Exp*. Both the success and fail functions are invoked with the continuation of the engine application. Thus if these functions return a value, it is returned as the result of the engine invocation. The evaluation of *Exp* always proceeds in the environment of the original engine expression (`engine` is thus a closing form, like `lambda`).

Engines, like all objects in Scheme 84, are first class: they may be passed to functional objects, be returned by functions, be stored in data structures, and have indefinite extent. Hence engines, and the environment and control information that they contain references

---

† Thus an engine tick is similar to a *computon*, which has been defined as "a mythical subatomic particle that bears the unit quantity of computation" [Steele *et al.* 83].

to, must be reclaimed by a garbage collector when they are no longer accessible.

In the current Scheme 84 implementation of engines, a tick corresponds to the execution of one Virtual Scheme Machine instruction. However, in general, the amount of computation associated with a tick is not defined—it is only assured that a larger tick count is associated with a larger expected amount of computation (in the statistical sense). Thus, engine ticks might be metered by a real-time clock, with an unpredictable amount of time spent handling interrupts.

The invocation of an engine by an already running engine, which we refer to as *nesting* of engines, is not allowed. This considerably simplifies the implementation of engines, and results in no loss of generality. Our experience to date has been that when the need for nested engines is felt, closer examination of the problem usually reveals that nesting is not only unnecessary, but leads to unnecessarily complicated programs. Real need for nesting occurs when the ticks being allotted to an engine should be subdivided among its offspring, and such cases do arise in some operating system and artificial intelligence applications. We have implemented nested engines using the existing unnested engines. This is analogous to implementing a recursively virtuallizable operating system. An algorithm for implementing deeply nested engines efficiently is under development and will be reported elsewhere.

Apart from free variable bindings, engines do not have state! Invoking the same engine twice continues the engine's computation from the same point. This is in contradistinction to the usual interpretation of processes. To record the progress of an engine process, a new engine is created. It would be possible to define an otherwise engine-like mechanism in which the original engine was modified upon exhaustion of the tick count, rather than returning a new engine. However, we feel that the side-effect-free approach we have taken is much to be preferred; it makes engines more suitable than traditional processes for applicative styles of programming in which side effects are disallowed or restricted to a single operation, such as `set!`.†

A more precise semantics of engines is provided by the meta-circular interpreter in the Appendix.‡ In the absence of engines, the `meaning` function returns ordinary values. Given the domain assignments

E		syntactic expressions
R		environments
K	$= D \rightarrow Ans$	expression continuations
Ans	$= D$	answers
D	$= F + K + \ldots$	denotable values
F	$= D^* \times K \rightarrow D$	functional values,

we have `meaning` $: E \times R \times K \rightarrow D$. Note that functions accept a list of arguments from D, and a continuation to which the function passes its value.

In order to include engines, we introduce additional domains and redefine F and K

† The remainder of this section was not included in the preliminary version of this paper presented at the *1984 ACM Symposium on LISP and Functional Programming*.

‡ Meta-circular interpreters, like denotational semantics, often overspecify the semantics: this interpreter defines order of argument evaluation and the unit of computation associated with a tick, both of which are unspecified in the semantics of Scheme 84. Students of denotational semantics will find the style, if not the syntax, of this interpreter to be familiar. Others will likely have difficulty with this interpreter, and may safely proceed to section 4.

as follows:

NIL	$= \{nil\}$	the trivial domain
ET	$=$ *positive integers*	engine ticks
EF	$= NIL + K$	engine failure continuations
ES	$= NIL + (D \times ET \rightarrow G)$	engine success continuation
G	$= ET \rightarrow (ES \times EF) \rightarrow Ans$	meaning values
F	$= D^* \times K \rightarrow G$	functional values
K	$= D \rightarrow G$	expression continuations

`meaning` must now effectively be a function of EF, ES and ET, as well as E, R and K, but by currying it with respect to EF, ES and ET, we avoid the necessity of passing these additional arguments on each application of `meaning`. `meaning` now returns a "general" value from the domain G:

$$\texttt{meaning} : E \times R \times K \rightarrow G.$$

The new function `tick` decrements the tick count each time `meaning` returns a value, unless the count has reached one, in which case the engine failure continuation is invoked with a new engine that will continue the engine computation when invoked. We have:

$$\texttt{tick} : G \rightarrow G.$$

`make-engine` takes a general value which represents a "base" engine state and returns an engine in functional form:

$$\texttt{make-engine} : G \rightarrow F.$$

When this engine is invoked, it absorbs the ticks count and engine success and failure continuations, which should be nil. If they are not, then an engine is already running, which is an error. Otherwise, the engine success and fail continuations are constructed using the corresponding functions applied to the engine and the continuation of the engine invocation, and then the value which `make-engine` was passed is invoked with the ticks count and these continuations.

The "base" engine created by the `engine` case of `meaning` invokes `meaning` to evaluate the engine expression. The continuation of this invocation absorbs the engine ticks count and the engine success and fail continuations, and then invokes the success continuation with the value of the engine expression and the ticks count.

4 Simple Examples

(1) The engine facility could be simplified a little further by defining engines to be functions of two arguments, which always perform one tick of computation when invoked. Since it would then be superfluous to pass the number of ticks remaining when an engine computation completes (for it would always be zero), both the success and fail functions take one argument. As a simple example of Scheme 84 engines, we use them to define these simpler engines, which might be called *steppers*. A syntactic extension could be employed that expands expressions of the form `(stepper` *Exp*`)` to `(engine-to-stepper (engine` *Exp*`))`, where

```
(define engine-to-stepper                          [engine-to-stepper]
  (lambda (e)
    (lambda (succ fail)
      (e 1
        (lambda (val ticks) (succ val))
        (lambda (new-e)
          (fail (engine-to-stepper new-e)))))))
```

It is also possible to implement our multi-step engine mechanism using one-step engines. While one-step engines may be more aesthetically pleasing, we have chosen to provide multi-step engines to avoid the inefficiency inherent in any such interpretive implementation.

(2) It is sometimes useful to run engines with an unlimited number of ticks (rather than the least possible number, as above). Thus if the engine's computation is terminating, we will be assured of obtaining its value, along with the number of ticks required to obtain it. We loop repeatedly, using a stepper to coax the engine computation along, until a value for the engine's expression is obtained.

[complete]

```
(define complete
   (lambda (eng)
      (let ([count 0])
         ((rec loop
             (lambda (step-taker)
                (set! count (add1 count))
                (step-taker
                   (lambda (val)(cons val count))
                   loop)))
          (engine-to-stepper eng)))))
```

`complete` returns a pair $(v\ .\ t)$, where v is the value of the engine's computation and t is the number of ticks required for it to complete. Of course, unlike engine invocation, `complete` is not guaranteed to terminate. A syntactic extension that just returns the number of ticks required to complete a computation may now be defined by

$$\texttt{(ticks } E\texttt{)} \equiv \texttt{(cdr (complete (engine } E\texttt{)))}.$$

This might be used, for example, to compare the relative speeds of several algorithms.

5 An Engine Process Scheduler

As an extended example of engine programming techniques, we will now use engines to implement a simple time-sharing operating system kernel. These techniques have broad applicability and may be used to implement a variety of process abstractions. Furthermore, the power of Scheme's scoping control and functional abstraction mechanisms may be used to abstract from the operating system kernel many of its traditional elements and still maintain the necessary security.

The basic technique is simple. The kernel will dispatch a user process by running a corresponding engine with a tick count corresponding to its time slice. When the process's time is exhausted, the engine will return control to the kernel, which may then dispatch the next process. We assume for the moment that processes are represented directly as engines. A queue of these processes is maintained, with the processes stored in queue entries that correspond to the PCBs (process control blocks) of traditional operating systems. The `&process` and `!process` functions extract processes from and store processes in these queue entries. The kernel's queue of processes is maintained by a queue object that responds to at least the `empty?`, `rot!`, `unhook!`, `enq!`, and `*entry` messages by returning functions that perform customary queue operations. The `rot!` operation rotates the queue by performing the equivalent of a dequeue followed by an enqueue, and returns a reference to the affected queue element. `unhook!` removes an indicated entry (not necessarily the first) from the queue. `*entry` returns a new queue entry. (We will consistently use a prefix '*' to indicate a function that returns a value of the type indicated by the rest of the name.) We can now express a simple kernel as follows:

```
(rec loop
   (lambda ()
      (let ([pcb ((q 'rot!))])
         (!process pcb
            ((&process pcb)
             time-slice-length
             (lambda (val ticks-left) ???)
             (lambda (new-eng) new-eng)))
         (loop))))
```

In practical systems it is also necessary to provide a trap mechanism that allows the user process to return control synchronously (without preemption) to the operating system with a request for some service. Thus we need to provide a function `trap` that may be invoked by a user process with a tag, which indicates the type of operating system service required and perhaps additional information specific to the service request. Invoking `trap` will cause control to be returned to the kernel passing the tag and other information. The kernel must then be able to resume the engine's computation.

Now, the only way to return control from an engine, short of exhausting the tick count, is to return a value as the result of the engine's computation. This is accomplished by invoking the continuation of that computation. Thus the trap function must have access to this continuation. But the continuation of every engine's computation is the same, namely the null continuation that signals return of control to the engine's invoker. This null continuation may be obtained by invoking an engine that immediately grabs its expression's continuation with `call/cc`. Thus the null continuation may be obtained by running to completion an engine that simply returns the null continuation as its value:

```
(car                                          [engine-return]
   (complete
      (engine
         (call/cc
            (lambda (k) k)))))
```

We call this continuation engine-return, for invoking it with a value v at any point in an engine immediately returns v as the value of the engine computation. To enable the kernel to return control to the engine at the point of the trap function call, the trap function first obtains (with call/cc) the continuation k of its invocation.† It then invokes the engine-return continuation, passing it k and the list containing the trap type and argument with which trap was invoked. Thus we have:

```
(define trap                                          [trap]
   (let ([engine-return
            (car (complete
                     (engine
                        (call/cc
                           (lambda (k) k)))))])
      (lambda (type arg)
         (call/cc
            (lambda (k)
               (engine-return
                  (list k type arg)))))))
```

Upon receiving the list (*k type arg*) as a result of a trap, the kernel invokes a trap handler function with this list to perform whatever service is requested by the trap. Using an expression of the form (engine (*k* *v*)) it can then create a new engine (to be run next or at some future time) that will resume the interrupted engine's computation, returning the value v (perhaps returned by the trap handler) as the result of the trap function's application.

† This technique of recording process state with continuations is also useful in the context of multiprocessing and interrupts [Wand 80].

We use functional abstraction to define an operating system kernel that performs only the most essential functions of (1) dispatching processes (running engines) obtained from a process queue, and (2) invoking a trap handler when processes trap for operating system services. In particular, we abstract the queue and trap handling functions from the kernel. In doing so we also maintain a traditional form of operating system security, namely that only the kernel has the ability to dispatch new processes. (This assures that errors resulting from attempts to nest engines cannot occur. We assume that once the kernel has been defined, any further use of the `engine` keyword is prohibited; this may be enforced by defining a dummy syntactic extension for `engine`.)

We define a function `*kernel` that returns a kernel *thunk* (a function of no arguments), invocation of which runs the process scheduler. Since the trap handler and the kernel must share the same ready queue, we pass to `*kernel` the functions `*trap-handler` and `*queue` that return a new trap handler and queue object, respectively. `*kernel` will first create a queue and pass it, among other things, to `*trap-handler` to obtain a local trap handler that uses the same queue.

The trap handler must have the ability to create new processes, as when a fork operation is requested. However, since the security constraint requires that the trap handler be unable to run processes, we cannot represent processes simply as engines. (If the trap handler could create engines, it could also run them!) To avoid this, we represent a process as a function that, upon receiving the appropriate key, returns an engine. (This security technique is similar to that of Morris [73].) Thus within each kernel we define the function `*process` as

```
(lambda (eng)                                        [*process]
   (lambda (key)
      (if (eq? key lock)
          eng
          (error 'privacy-violation))))
```

where lock is a value known only to the kernel.

This function cannot be used directly by the trap handler, since only the kernel can create an engine. For this purpose we also pass to *trap-handler the function forker that forks off a new process. forker is thus similar to a *capability* [Peterson & Silbershatz 83]. Assuming that the queue object returns a new queue entry in response to the *entry message, forker may be defined as follows:

```
(lambda (fun)                                        [forker]
   ((q 'enq!)
    ((q '*entry)
     (*process (engine (fun self))))))
```

The process queue is initially empty, and may become empty in case of deadlock. To handle these cases, we provide the kernel with a *top-level function that may be passed to forker to create a process when the queue is empty.

In addition to *trap-handler, lock, and *top-level, we pass to *kernel the values *slice, a thunk that returns a tick count that determines the time slice length, and *queue, a thunk that returns a queue (see Figure 1). After the queue entry of the current process is obtained and bound to pcb, its process field is extracted with &process and invoked. If the engine traps before its time slice is exhausted, trap-val is bound to the returned value and the trap handler is invoked with this value, returning a thunk. A new engine is then created that thaws this thunk. This new engine is returned as the value

```
(define *kernel
   (lambda (*trap-handler *queue *slice lock *top-level)
      (let ([q (*queue)]
            [*process (lambda (eng)
                         (lambda (key)
                            (if (eq? key lock)
                                eng
                                (error 'privacy-violation))))]
            [pcb '*])
         (let ([forker (lambda (thunk)
                          ((q 'enq!)
                           ((q '*entry) (*process (engine (thunk))))))]
               [self (lambda () pcb)])
            (let ([trap-handler (*trap-handler q forker self)])
               (let ([success (lambda (trap-val ticks-remaining)
                                 (let ([th (trap-handler trap-val)])
                                    (engine (th))))]
                     [fail (lambda (eng) eng)])
                  (rec kernel-loop
                     (lambda ()
                        (if ((q 'empty?)) (forker *top-level) nil)
                        (set! pcb ((q 'rot!)))
                        (!process pcb
                           (*process
                              (((&process pcb) lock) (*slice) success fail)))
                        (kernel-loop))))))))))
```

Figure 1. Operating System Kernel

of the engine invocation. (The `ticks-remaining` value is ignored for simplicity.) If the engine is preempted, a new engine that will continue its computation is returned. In either event, the new engine returned is made into a "process" and stored in the original process entry.

A naïve trap handler creation function may be defined as follows:

```
(define *trap-handler
   (lambda (q forker self)
      (lambda (trap-val)
         (let ([arg (arg-pt trap-val)])
            (let ([a (case (type-pt trap-val)
                        [awaken ((q 'enq!) arg)]
                        [block ((q 'unhook!) arg)]
                        [fork (forker arg)]
                        [self (self)]
                        [atomic (arg)])])
               (lambda ()
                  ((k-pt trap-val) a)))))))
```

In addition to `q` and `forker`, the kernel passes `*trap-handler` a thunk `self` that allows the trap handler to return the identity of the trapping process (its PCB) in response to a trap of type `self`. In case the type of the trap value is `awaken` or `block`, the standard queue operations are performed. `fork` creates a new process that evaluates the argument thunk. In every case, the trap handler returns a new process made from a thunk that, when invoked, passes the value resulting from the requested trap operation to the continuation of the original engine's trap application (which is contained in the `k-pt` of the trap value). The `atomic` trap simply thaws its argument as part of the trap operation; this may be used to execute code uninterruptibly.

6 Implementing Semaphores and Parbegin

The process control operations provided by the trap handler of the previous section are very low level, though typical of those provided by operating systems. We now illustrate the implementation of a higher level process abstraction mechanism typical of those provided by concurrent programming languages. For this we define a new trap handler that provides the traditional *parbegin* mechanism for creating processes, and semaphores for process synchronization. An implementation example shows how we may take advantage of the fact that trap-handling has been abstracted from the kernel.

To implement parbegin and semaphores we require the same operations that were provided by the previous low level trap handler. However, we no longer wish the low level operations to be accessible to the user, and we wish to invoke them from within trap handler code (where it is not possible to do additional traps). Thus we move the old trap handler functions into an `inner-handler` that is passed to an `outer-handler` that supports the higher level process abstraction. The `self` function passed to the trap handler by the kernel is also passed to the outer handler. The function `**trap-handler` puts these pieces together, and also abstracts the selection of the trap list components from the trap handlers. The double star prefix indicates that this is a function to which two applications must be made before a trap handler is returned.

```
(define **trap-handler
   (lambda (*outer-handler)
      (lambda (q forker self)
         (let
            ([inner-handler
                (lambda (msg arg)
                   (case msg
                      [awaken ((q 'enq!) arg)]
                      [block ((q 'unhook!) arg)]
                      [fork (forker arg)]
                      [self (self)]))])
            (let ([outer-handler
                     (*outer-handler inner-handler self)])
               (lambda (tv)
                  (let ([ans (outer-handler
                                (type-pt tv)
                                (arg-pt tv))])
                     (lambda ()
                        ((k-pt tv) ans)))))))))
```

The parbegin and semaphore operations require an atomic trap operation to assure that critical sections are executed uninterruptibly. However, we do not wish to make this operation available to the user (for then any process could deadlock the system by looping forever uninterruptibly, and the operating system would no longer be able to account for the use of computation time). This security restriction is implemented by providing a lock, which is local to the outer trap handler. This lock is used (instead of the atom `atomic`) as a trap tag to request uninterruptible execution of a thunk. See Figure 2 for the definition of `*outer-handler`. Note that the function `self` is passed from the kernel to the function returned by `**trap-handler` and finally to `*outer-handler`, but is not accessible to the user. A system may now be generated with

```
(*kernel (**trap-handler *outerhandler) *queue *slice lock *top-level)
```

Note this requires *no* modifications to `*kernel`.

Parbegin cannot be a function, since its arguments (the expressions to be evaluated in parallel) must not be evaluated at the time of call. Instead, the following syntactic extension is used:

```
(parbegin E1 ... En) ≡
   (trap 'parbegin-thunks
      (list (lambda () E1) ... (lambda () En)))
```

Here the `parbegin-thunks` trap receives a list of thunks and forks off a process for each thunk that thaws the thunk and then executes an atomic trap which performs the parbegin "join" operation. The process that performs the parbegin, or `father` process, blocks itself after performing these fork operations and is unblocked by the last forked process to terminate.

The semaphore trap returns an object with the local semaphore queue and count that responds to `wait` and `signal` messages by performing atomic trap operations.

7 Scope yields rings of protection

The simple operating system of section 5 is based upon three rings of security. Innermost is the kernel that has all privileges, including the ability to create and run engines. Next is the trap handler that has the ability, through the `q` and `forker` values that are passed to it by the kernel, to access the process queue and to create processes. These processes are simply protected engines that the trap handler can manipulate, but which only the kernel can run. Outermost are the user's processes that can neither create nor manipulate processes.

The more elaborate trap handler of the last section adds yet another level of security. Between the inner handler, which provides the low level process manipulation operations, and the user's processes we have added an outer handler that supports the semaphore and parbegin operations. The user then has access only to these higher level operations.

These security structures are obtained using only the lexical scope control inherent in Scheme's functional abstraction mechanism (and the trivial use of a dummy syntactic extension to prohibit use of the `engine` keyword outside the kernel). In particular, we do not require the supervisor/user mode hardware mechanisms usually necessary for operating system security.

8 Conclusion

Many process abstraction facilities have been proposed, with a wide variety of design goals [Filman & Friedman 84]. A language designer who chooses one of these facilities, or opts to invent yet another, risks rapid design obsolescence. The resulting language will also be ill-suited to a variety of applications that do not fall within the language's design goals. Attempts to avoid the latter problem by introducing intricate facilities, such as those of Ada, result in a complex design that is optimally suited to few applications.

As an alternative to committing a language design to a specific process abstraction facility, we have proposed the primitive engine mechanism that may be used to provide time-sharing implementations of process abstractions. To demonstrate the use of engines, we have implemented a sample operating system kernel and two trap handlers: one that supports the traditional low level operations of forking, blocking and unblocking processes, and another that provides the higher level semaphore and parbegin operations. We have also shown that with first class functional and control abstractions it is possible to enforce security constraints which have traditionally been performed by low-level unstructured code and specialized hardware. It is our belief that research now underway on efficient implementation techniques for Scheme-like languages will allow systems of this type to compete in efficiency with those developed with more traditional means, while providing a more structured and extensible basis for operating system implementation.

Acknowledgements: We gratefully acknowledge Eugene Kohlbecker's help in the development of an earlier engine operating system, and his suggestion of the term ***engine***. Mitchell Wand contributed to the development of the interpreter in the Appendix.

```
(define *outer-handler
   (lambda (inner-handler self)
      (let ([atomic (*lock)])
         (rec outer-handler
            (lambda (msg arg)
               (if (eq? msg atomic) (arg)
                   (case msg
                      [parbegin-thunks
                         (let ([n (length arg)] [father (self)])
                            (mapc (lambda (thunk)
                                        (inner-handler 'fork
                                           (lambda ()
                                              (thunk)
                                              (trap atomic
                                                 (lambda ()
                                                    (set! n (sub1 n))
                                                    (if (=0 n)
                                                        (inner-handler
                                                           'awaken father))))
                                              (inner-handler 'block (self)))))
                                  arg)
                            (inner-handler 'block father)
                            nil)]
                      [semaphore
                         (let ([n arg] [semq (*queue)])
                            (lambda (msg)
                               (case msg
                                  [wait
                                     (trap atomic
                                        (lambda ()
                                           (if (=0 n)
                                               (let ([myself (self)])
                                                  (inner-handler 'block myself)
                                                  ((semq 'enq!) myself))
                                               (set! n (sub1 n)))))
                                     arg]
                                  [signal
                                     (trap atomic
                                        (lambda ()
                                           (if ((semq 'empty?))
                                               (set! n (add1 n))
                                               (inner-handler 'awaken
                                                  ((semq 'deq!))))))
                                    arg]))))])))))))
```

Figure 2. Outer Trap Handler

References

[Filman & Friedman 84]
Filman, R., and Friedman, D. P. *Coordinated Computing: Tools and Techniques for Distributed Software*, McGraw-Hill, 1984.

[Friedman *et al.* 84]
Friedman, D.P., Haynes, C.T., Kohlbecker, E., and Wand, M. "The Scheme 84 Reference Manual" Indiana University Computer Science Department Technical Report No. 153, March, 1984.

[Landin 65]
Landin, P. "A correspondence between ALGOL 60 and Church's Lambda Notation", *CACM* **8**, 2–3, February and March 1965, pages 89–101 and 158–165.

[Morris 73]
Morris, J., "Protection in Programming Languages," *CACM* **16**, 1973, pages 15–21.

[Peterson & Silberschatz 83]
Peterson, J., and Silberschatz, A. *Operating System Concepts*, Addison Wesley, 1983.

[Sandewall 78]
Sandewall, E., "Programming in an interactive environment: the "Lisp" experience," *Computing Surveys* **10**, March 1978, pages 35–71.

[Steele *et al.* 83]
Steele, G., Woods, D., Finkel, R., Crispin, M., Stallman, R., and Goodfellow, G., *The Hacker's Dictionary*, Harper & Row, 1983.

[Wand 80]
Wand, M. "Continuation-based multiprocessing," *Conf. Record of the 1980 Lisp Conference*, August 1980, pages 19–28.

Appendix: Meta-circular interpreter for a subset of Scheme 84

```
(define meaning
  (lambda (e r k) ; e = expression, r = environment, k = continuation
    (tick
      (case (type-of-expression e)
        [constant (k e)]
        [quote (k (literal-pt e))]
        [identifier (k (R-lookup e r))]
        [lambda (k (lambda (d* k)
                     (evaluate-all
                       (body-pts e)
                       (extend-r (formals-pt e) r d*)
                       k)))]
        [engine (k (make-engine
                     (lambda (et)
                       (lambda (es ef)
                         (((meaning (engine-body-pt e)
                              r
                              (lambda (d)
                                (lambda (et)
                                  (lambda (es ef) (es d et)))))
                           et)
                          es ef)))))]
        [set! (meaning (val-pt e) r
                (lambda (d)
                  (k (store! (L-lookup (id-pt e) r) d))))]
        [if (meaning (test-pt e) r
              (lambda (d)
                (if d (meaning (then-pt e) r k)
                      (meaning (else-pt e) r k))))]
        [call/cc (meaning (fn-pt e) r
                   (lambda (g)
                     (g (list (lambda (d* dummy) (k (car d*))))
                        k)))]
        [application
          (meaning-of-all e r
            (lambda (d*)
              ((car d*) (cdr d*) k)))]))))
```

```
(define evaluate-all
   (lambda (e-list r k)
      (meaning (car e-list) r
         (if (null? (cdr e-list))
             k
             (lambda (d) (evaluate-all (cdr e-list) r k))))))

(define meaning-of-all
   (lambda (e-list r k)
      (meaning (car e-list) r
         (lambda (d)
            (if (null? (cdr e-list))
                (k (cons d nil))
                (meaning-of-all (cdr e-list) r
                   (lambda (d*) (k (cons d d*)))))))))

(define tick
   (lambda (g)
      (lambda (et)
         (if (= 1 et)
             (lambda (es ef) (((ef (make-engine g)) 0) nil nil))
             (g (sub1 et))))))

(define make-engine
   (lambda (g)
      (lambda (d* k)
         (lambda (et)
            (lambda (es ef)
               (if (null? es)
                   ((g (ticks-arg d*))
                    (lambda (d et)
                       ((((succ-arg d*) (list d et) k) 0) nil nil))
                    (lambda (d)
                       ((((fail-arg d*) (list d) k) 0) nil nil)))
                   (error "can't nest engines")))))))
```