Higher-Order and Symbolic Computation, 18, 327–354, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

Integrating User-Level Threads with Processes in Scsh*

MARTIN GASBICHLER gasbichl@informatik.uni-tuebingen.de
Universität Tübingen, Sand 13, 72076 Tübingen, Germany

MICHAEL SPERBER[†] sperber@deinprogramm.de
Pappelweg 2, 72076 Tübingen, Germany

Abstract. Scsh, the Scheme shell, enables concurrent system programming with portable user-level threads. In scsh, threads behave like processes in many ways: each thread receives its own set of process resources; like Unix processes, new threads can inherit resources from the creating thread. The combination of scsh's interface to the POSIX API with user-level threads creates a number of design and implementation challenges: Scsh's abstractions for managing process resources raise interesting modularity issues, particularly in connection with first-class continuations. Scsh also provides an interface to the `fork` system call; its implementation must avoid common pitfalls that arise with a user-level thread system. This paper describes the design and implementation of the relevant abstractions and discusses the implications for programming-language and system design.

Keywords: threads, Unix, Scheme, continuations

1. Introduction

Scheme 48 [17, 18] is a byte-code implementation of Scheme. It supports concurrent programming via a user-level thread system. Scsh [23] is a variant of Scheme 48 with extensive support for Unix shell and systems programming. Specifically, it provides full access to all basic primitive functions specified by POSIX. Historically, the original version of scsh was single-threaded. This paper describes the highlights of our effort to integrate the scsh API with the multithreading facility of Scheme 48.

The combination of full POSIX access and multithreading in scsh enables a wide spectrum of applications, among them the implementation of efficient Internet servers, system administration daemons, and concurrent user-interface libraries. Moreover, several independent scsh applications can run within the same scsh process in different threads.

The integration of the POSIX process model with a user-level thread system creates some unique challenges for the scsh implementation: Writing multithreaded scsh programs is easiest when threads behave mostly like processes. However, the analogy between processes and threads can never be perfect in Scheme because of the additional power of

*This article is a revised and extended version of the paper *Processes vs. User-Level Threads in Scsh* that appeared in *Proceedings of the Third Workshop on Scheme and Functional Programming*, Pittsburgh, USA, 2002.
[†]Partially supported by Universität Tübingen.

threads in Scheme. Specifically, the use of first-class continuations interferes with the analogy.

Making the POSIX API available to multithreaded Scheme programs requires special design and implementation effort in three general areas:

– POSIX provides a number of *process resources* to each process, such as the environment or current working directory. These resources are process-*local* but thread-*global*. A system that manipulates the process resources directly would cause programs that work correctly in a single-threaded system to interfere with each other when run concurrently in multiple threads, even though there is no explicit shared state or communication with other threads.
– The POSIX `fork` system call copies the entire process. Thus, implemented naively all threads of the parent would also run in the child. This interferes with the intuition of the programmer who expects "only the current thread to fork." Moreover, it causes a number of race conditions associated with the `fork`/`exec` pattern common in POSIX programming. Worse, the programmer cannot work around this problem easily because the primitives of the thread system are not sufficiently powerful.
– The C library causes various problems: The `syslog` interface to the system's message logging facility offers only a single global, implicit connection that needs to be multiplexed among threads. Also, some POSIX library calls block indefinitely, making timely preemption of threads impossible when such calls are running.

This paper describes the steps taken in scsh (from version 0.6 on) towards handling or working around these problems:

– We have virtualized process resources to appear thread-local to Scheme programs. Scsh represents the values of thread-local process resources by *process fluids*, which are thread-local cells that support binding, mutation, and preservation across a `fork`-like operation on threads. Scsh uses lazy synchronization to align the underlying process resources with the resources of the running thread before system calls.
– Scsh's thread system supports a novel primitive called `narrow` that allows implementing a fork operation that forks "only the current thread."
– We have taken specific measures to make various C libraries available to Scheme programs without compromising multithreading.

Overview Section 2 gives a short account of the Scheme 48 thread system. Section 3 describes the POSIX process resource concept, and the scsh API for manipulating process resources. Section 4 describes process fluids which scsh uses to represent process resources. Next, Section 5 describes how scsh uses process fluids to keep the thread-local process state lazily aligned with the actual process state. Scsh's implementation of `fork` is described in Section 6. Section 7 describes some of the miscellaneous problems with integrating common C libraries with user-level threads. Section 9 reviews related work, and Section 10 concludes.

2. The Scheme 48 thread system

Concurrency within scsh is expressed in terms of the user-level thread system of Scheme 48 [2]. Its structure is inspired by nestable engines [4, 11, 12] and is implemented almost entirely in Scheme. Scheme 48's virtual byte-code machine (VM) supports the thread system in two ways:

– It schedules timer interrupts and thus allows preemption of running threads.[1]
– The VM I/O primitives are non-blocking. The VM manages a queue of outstanding I/O requests and schedules interrupts as they become enabled.

The system represents each thread by a Scheme object that, while it is running, keeps track of its remaining time before preemption. As in other engines-based thread systems, Scheme 48 uses continuations for saving and restoring the control context of a thread. For a blocked thread, the thread object contains a saved continuation and an interrupt mask.

As in any continuation-based system with threads, Scheme 48 needs to take `dynamic-wind` [13] into account: Scheme 48 implements both `call-with-current-continuation` and `dynamic-wind` on top of the thread system, preserving the sequential semantics of Scheme in each thread.[2] For context switching, the thread system employs the `primitive-cwcc` VM primitive which merely reifies the VM-level continuation. Each thread object also keeps track of the dynamic environment and the dynamic wind point, which in turn are used to implement `dynamic-wind` and `call-with-current-continuation`.

The dynamic environment contains thread-local bindings for *fluid variables* (or just *fluids*) that implement a form of dynamic binding local to a single thread. For example, Scheme 48 holds the current input and output ports in fluids. Fluids play a crucial role in coordinating thread-local state and process state. Section 4 explains this issue in detail.

Each thread is under the control of a *scheduler*, itself a regular thread. Schedulers nest, so all threads in the running system are organized as a tree. A scheduler runs a thread for a slice of its own time by calling `(run` *thread time*`)`. The call to `run` returns either when the time slice expires or an *event* happens. This event might signify termination, an interrupt, a blocked operation, another thread becoming runnable, or a request from the thread to the scheduler.

An example for such a request from the thread to its scheduler is the implementation of the `spawn` procedure. By calling `spawn`, a thread causes its scheduler to spawn a new thread:

`(spawn` *thunk*`)` $\rightarrow$ *undefined*

`Spawn` creates a new thread and calls its thunk argument in that new thread. The thread terminates once the thunk returns. `Spawn` works by signalling a `spawned` event to the scheduler, which contains the thunk.

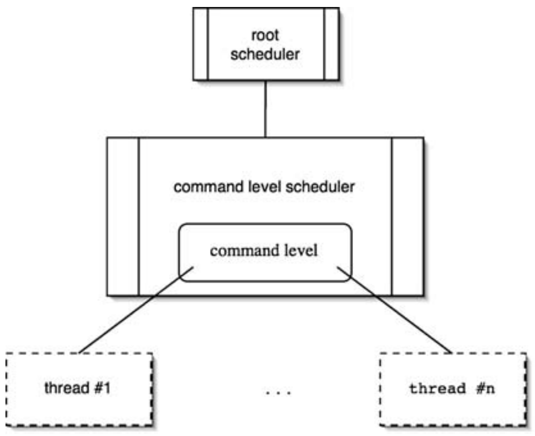

Figure 1. Command levels in Scheme 48.

Not every scheduler has to handle all events: If a scheduler is unwilling or unable to handle an event it can simply pass the event upwards in the thread tree by re-signalling it to its own scheduler. This makes adding new kinds of events very easy.

Thus, each scheduler performs at least two tasks: it implements a scheduling policy by deciding which threads to run for how long, and it handles events returned by `run`.

A non-interactive Scheme 48 process has only a single *root scheduler*. The root scheduler, in addition to managing its subordinate threads, also periodically wakes sleeping threads and takes care of port flushing. An interactive Scheme 48 also has a scheduler managing a *command level*, sitting beneath the root scheduler (See Figure 1.) Each command level represents an interaction with the user; the system typically creates a new command level when an error occurs, allowing the user to debug and possibly rectify the problem before returning to the previous level. Associating a scheduler with the current command level allows Scheme 48 to cleanly interrupt all running threads at any time by entering a new command level: The interrupt causes the command-level scheduler to swap its queue of runnable threads with that of a newly created level. The scheduler can continue the old level simply by reversing the procedure.

3. Unix process resources and the scsh API

Unix associates each process with an internal data structure. The process can read and write the contents of certain fields of this data structure—the *process resources*. Here are the most important process resources:

– the current working directory,
– the file mode creation mask (*umask*),
– the user and group ID,
– the environment.

The operating system initializes these resources during creation of a process by copying the values from the parent process. The process can then use the following functions to read and write the resources above:

- `chdir` and `getcwd` for the current working directory,
- `umask` for changing the umask (`umask` returns the previous mask),
- `getuid`/`getgid` and `setuid`/`setgid` for the user and group ID,
- `getenv` and `putenv` for reading and writing a single key/value pair of the environment (the environment as a whole is accessible via the global variable `environ` of type `char**`).

A number of system calls implicitly consult the resources of the calling process:

- All system calls that take a filename as an argument interpret the path relative to the current working directory if the path does not start with a slash.
- The mask of the `umask` resource specifies the permission bits not to be set when the `open` system call creates a new file.
- The operating system checks the values of the user and group IDs before accessing files and directories.
- The `execl`/`execv`/`execle`/`execlp`/`execvp` for starting child processes propagate the environment to the child.

As far as process resources are concerned, the scsh API provides the full functionality of POSIX:

- The `chdir` procedure changes the current working directory.
- `Set-umask` sets a new file creation mask.
- `Set-uid`/`set-gid` sets the user and group ID.
- `Alist->env` turns an association list into the current environment.

These procedures are convenient for writing simple sequential scripts. Still, as the process resources constitute global state, they promote non-modular programming: Inserting a piece of a code mutating a resource in the middle of an existing script is likely to cause breakage.

To promote modular programming, scsh augments the imperative interface of the Unix API by constructs that temporarily change the value of the resources within the extent of a piece of code. For each resource, a macro `(with-`*resource val body* `...)` sets *resource* to value *val* during the evaluation of the *body* expressions. For example, `with-cwd` temporarily changes the value of the working directory. The following piece of code deletes all files in `/tmp` without affecting the current working directory permanently:

```
(with-cwd “/tmp”
   (for-each delete-file (directory-files)))
```

Similar to `with-cwd`, `with-umask` temporarily changes the umask, and `with-total-env` temporarily changes the environment. (There is no variant for the user and group IDs: Only the super-user can change the IDs, and reverting to the old value afterwards is not possible for security reasons. Section 5 elaborates on this issue.)

The `with-`*resource* forms work correctly in the face of non-local control flow: If the program calls an escape procedure (created via `call-with-current-continuation`) to escape from the body of such a form, they will reset the process resource to the value it had before evaluating the form. Also, if the program calls an escape procedure captured during the extent of the body of an instance of the form, they will restore the specified value of the process resource accordingly.

4. Thread-local resources in Scheme

The various process resources constitute implicit state, just like the settings for `current-input-port` and `current-output-port` (Scheme's versions of `stdin` and `stdout`). This section discusses common abstractions for managing such settings: *fluids* support dynamic binding, and *thread-local cells* support mutation. Each is capable of covering a portion of the scsh API for process resources, but is insufficient for meeting the needs of the scsh API by itself. Therefore, we introduce a third abstraction called *process fluids* which allows implementing the scsh API in a way compatible with threads.

4.1. Fluids

R^{5}RS [16] implicitly suggests the presence of a *dynamic environment* to hold the values of `current-input-port` and `current-output-port`, and specifies primitives to change them temporarily:

(`with-input-from-file` *string thunk*) → *value of thunk*

(`with-output-to-file` *string thunk*) → *value of thunk*

Both of these procedures change the respective port setting, call *thunk*, and then revert to the old setting.

Scheme 48 lets users extend the dynamic environment with new *fluid bindings*; a *fluid* is a value-holding cell that allows dynamic binding. Here are the operations for fluids:

(`make-fluid` *val*) → *fluid*

(`fluid` *fluid*) → *val*

(`let-fluid` *fluid val thunk*) → *value of thunk*

For an arbitrary value *val*, `(make-fluid` *val*`)` creates a fluid with default value *val*. For a fluid *fluid*, `(fluid` *fluid*`)` returns the value bound to *fluid*. For a fluid *fluid*, a value *val*, and a thunk *thunk* `(let-fluid` *fluid val thunk*`)` calls *thunk* with *fluid* bound to value *val* during the dynamic extent of this call.

Here are some examples from a Scheme 48 session, where > marks the top-level command prompt:

```
> (define f (make-fluid 1))
> (fluid f)
⤳ 1
> (define (p)
    (fluid f))
> (+ (let-fluid f 3 p)
     (fluid f))
⤳ 4
```

Fluids interact with continuations: Conceptually, each of `with-input-from-file`, `with-output-to-file`, and `let-fluid` changes the value of a fluid binding for the extent of its continuation. Thus, it is convenient to view the dynamic environment as part of the current continuation. This intuition also clarifies the relationship between fluids and `call-with-current-continuation`: As it captures the current continuation, it also captures the current dynamic environment, and calling the escape procedure restores both.[3]

A few examples illustrate the relationship between the current continuation and the current dynamic environment. The following program fragment (uninteresting by itself) saves an escape procedure in `*k*`:

```
> (define *k* 'uninitialized)
> (let-fluid f 25
    (lambda ()
      (* (call-with-current-continuation
           (lambda (k)
             (set! *k* k)
             10))
         (fluid f))))
⤳ 250
```

The call to `call-with-current-continuation` captures a continuation corresponding to this control context:

```
(* [] (fluid f))
```

Moreover, the call also captures the dynamic environment with `f` bound to 25. The top-level binding of `f` is still the initialization value:

```
> (fluid f)
⤳ 1
```

Invoking the escape procedure in `*k*` also restores the dynamic environment of the original continuation. Thus, calling `*k*` always multiplies its argument with 25, regardless of any surrounding `let-fluid` bindings to `f`:

```
> (*k* 100)
⤳ 2500
> (let-fluid f 42 (lambda () (*k* 100)))
⤳ 2500
```

The environment that associates fluids with their values is local to each thread. Each newly spawned thread receives a fresh dynamic environment from its scheduler, with all fluids bound to their default values. The following fragment illustrates the (non-)interaction between `let-fluid` and `spawn`:

```
> (let-fluid f 23
    (lambda ()
      (spawn (lambda ()
               (display (fluid f))))))
⤳ prints 1
```

To sum up, fluids are a good match for those scsh abstractions that provide dynamic binding for process resources such as `with-cwd` or `with-env`, especially as these binding forms are the preferred method for altering the process resources. Unfortunately, unlike process resources, fluids do not support mutation, which the scsh API requires.

4.2. Thread-local cells

Scsh implements a primitive mechanism orthogonal to fluids called *thread-local cells* or *thread cells*: a thread cell supports mutation, and mutation is always thread-local:

(`make-thread-cell` *default*) → *thread cell*

(`thread-cell-ref` *thread-cell*) → *value*

(`thread-cell-set!` *thread-cell value*) → *undefined*

`(make-thread-cell` v`)` returns a thread cell with default value v. `(thread-cell-ref` c`)` returns the current contents of the cell, and `(thread-cell-set!` c v`)` mutates the cell's value as seen by the current thread to v. The default value is the initial value of the thread cell—each new thread sees the default value until it mutates the thread cell via `thread-cell-set!`:

```
> (define a-cell (make-thread-cell 23))
> (thread-cell-ref a-cell)
⤳ 23
```

The following fragment starts a new thread that mutates the cell:

```
> (spawn (lambda ()
            (thread-cell-set! a-cell 42)
            (let lp ()
               (display (thread-cell-ref a-cell))
               (lp))))
⇝ keeps printing 42 until the end of days
```

The top-level thread still sees the initial value:

```
> (thread-cell-ref a-cell)
⇝ 23
```

Semantically, a thread cell is a table mapping the current thread to a value. Thus, unlike the dynamic environment, `call-with-current-continuation` does not capture the values of the thread-local cells. The following example—analogous to the first example for fluids—illustrates this:

```
> (define a-cell (make-thread-cell 23))
> (define *k* 'uninitialized)
> (thread-cell-set! a-cell 42)
> (* (call-with-current-continuation
       (lambda (k)
         (set! *k* k)
         10))
      (thread-cell-ref a-cell))
⇝ 420
> (thread-cell-set! a-cell 0)
> (*k* 10)
⇝ 0
```

Unfortunately, unlike process resources in scsh, thread cells do not support dynamic binding, which the scsh API requires.

4.3. Process fluids

As the previous two sections have shown, both fluids and thread cells are convenient abstractions with simple semantics. Unfortunately, the scsh API, for its process resources, must provide both binding and mutation. Consequently, we needed to introduce a third abstraction called *process fluids* specifically for meeting the requirements of the process resource API. Intuitively, a process fluid is simply a combination of a fluid and a thread-local cell—it supports the union of the operations of both:

(`make-process-fluid` *obj*) → *process fluid*

(`process-fluid` *fluid*) → *value*

(`let-process-fluid` *fluid obj thunk*) → *value of thunk*

(`set-process-fluid!` *thread-cell value*) → *undefined*

As before, `make-process-fluid` creates a process fluid with a default value. `Process-fluid` returns the current value of the process fluid. `Let-process-fluid` works just like `let-fluid`: It calls *thunk*, temporily setting *fluid* to *obj* for the extent of the call. Finally, `set-process-fluid!`, like `thread-cell-set!`, mutates the value of the process value.

Whereas the implementations of fluids and thread cells is deeply interwoven with the internals of the thread system, the implementation of process fluids builds on thread cells: A `:process-fluid` object—`:process-fluid` being a record type defined using SRFI 9 [15]—has only a single field with a constant thread cell in it:

```
(define-record-type :process-fluid
  (really-make-process-fluid cell)
  process-fluid?
  (cell process-fluid-cell!))
```

`Make-process-fluid` sets up a thread cell with its argument for a default value:

```
(define (make-process-fluid top)
  (really-make-process-fluid (make-thread-cell top)))
```

Conversely, `process-fluid` dereferences the thread cell in the process fluid, and `set-process-fluid!` sets its value:

```
(define (process-fluid process-fluid)
  (thread-cell-ref (process-fluid-cell process-fluid)))


(define (set-process-fluid! process-fluid val)
  (thread-cell-set! (process-fluid-cell process-fluid) val))
```

Finally, `let-process-fluid` uses `dynamic-wind` to mutate the process fluid for the extent of the call to its thunk argument:

```
(define (let-process-fluid p-fluid val thunk)
  (let ((old-val (process-fluid p-fluid)))
    (dynamic-wind
     (lambda () (set-process-fluid! p-fluid val))
     thunk
     (lambda () (set-process-fluid! p-fluid old-val)))))
```

As long as only a single thread ever accesses a process fluid, it behaves—depending on the operation—like a fluid or thread cell. All the examples for fluids and thread cells in the previous two sections yield the same results. Specifically, a thread newly spawned via `spawn` receives the default values for the process-fluid bindings from its scheduler, rather than from the thread which evaluated the call to `spawn`. This is contrary to how process resources work, where the child inherits from the parent. Lexical bindings allows communicating the value of a process fluid to a spawned thread explicitly:

```
(define p-fluid (make-process-fluid #f))

  ...
  (spawn
    (let ((val (process-fluid p-fluid)))
     (lambda ()
       (let-process-fluid p-fluid val
        ...))))
```

To automate the above process, the implementation of process fluids includes a procedure `preserve-process-fluids`:

(`preserve-process-fluids` *thunk*) → *procedure*

`Preserve-process-fluids` accepts a thunk as an argument, collects a list of all live process fluids and their values, and returns another thunk that applies `let-process-fluid` to the list before forcing the thunk argument of `preserve-process-fluids`. Thus, the above code could be rewritten as:

```
(define p-fluid (make-process-fluid #f))

  ...
  (spawn
    (preserve-process-fluids
     (lambda ()
      ...)))
```

The process-fluids library also defines a procedure `fork-thread` with the following definition:

```
(define (fork-thread thunk)
  (spawn (preserve-process-fluids thunk)))
```

A forked thread thus inherits the values of the process fluid bindings from its parent:

```
> (define f (make-process-fluid 0))
> (let-process-fluid f 1
      (lambda ()
         (fork-thread
            (lambda () (display (process-fluid f))))))
```

$\rightsquigarrow$ prints `1`

Mutation of process fluids is still thread-local:

```
> (begin
     (fork-thread
        (lambda () (set-process-fluid! f -1)))
     (process-fluid f))
```

$\rightsquigarrow$ `0`

4.4. Limitations and pitfalls

The design of the process fluids mechanism was mandated by the requirements of the pre-existing scsh API design which prescribes that scsh programs must be able to mutate process resources, or, in this case, the process fluids representing them. However, this capability comes with its own problems:

– Mutable process fluids are a form of global state, even if they are thread-local. Thus, abstractions mutating process fluids, instead of simply binding them, can lead to serious modularity problems.
– Whereas ordinary fluids are associated with continuations, this is not true for thread-local cells which are associated with threads. This also has a negative impact on the programmer's ability to create modular abstractions.

Whereas the modularity problems caused by the use of global state are widely known, the latter kind of problem is more subtle. It arises from the combination of threads and `call-with-current-continuation`: It is possible to invoke a continuation in a thread different from the one where it was originally captured.[4] Ideally, the continuation would behave in the new thread as in the original one. With ordinary fluid bindings, this is the case, as invoking a continuation restores the bindings active at the capture, no matter in what thread. With process fluids, the situation is different: a process fluid is stored in a thread-local cell. Therefore, its value is associated with the thread where the last binding or mutation happened. Hence, a continuation which migrates between threads sees different values for the process fluids, depending on which thread it runs in.

The following program fragment illustrates the issue:

```
(define *k* 'uninitialized)
(define *result* 'uninitialized)
(define *fluid* (make-process-fluid 'init))

(fork-thread
  (lambda ()
    (call-with-current-continuation
     (lambda (abort)
       (set-process-fluid! *fluid* 'first)
       (call-with-current-continuation
        (lambda (k)
          (set! *k* k)
          (abort 'ignore)))

       (set! *result* (process-fluid *fluid*))))))
```

After the thread has completed, `*k*` holds a continuation that will set `*result*` to the value of the process fluid `*fluid*`. A second thread can now invoke the continuation:

```
(fork-thread
  (lambda ()
    (set-process-fluid! *fluid* 'second)
    (*k* 'ignore)))
```

Surprisingly, `*result*` now holds the symbol `init`. This is because `preserve-process-fluids` in `fork-thread` uses `let-process-fluid` which in turn uses `dynamic-wind`. On the application of `*k*`, the `dynamic-wind` thunks restore the default bindings. If the program uses `let-process-fluid`, it restores the binding upon invocation of the escape procedure as a slight modification of the above example shows:

```
(fork-thread
  (lambda ()
    (call-with-current-continuation
     (lambda (abort)
       (let-process-fluid *fluid* 'bind
          (lambda ()
            (set-process-fluid! *fluid* 'first)
            (call-with-current-continuation
             (lambda (k)
               (set! *k* k)
```

```
              (abort 'ignore)))
            (set! *result* (process-fluid *fluid*))))))))
(fork-thread
  (lambda ()
    (set-process-fluid! *fluid* 'second)
    (*k* 'ignore)))
```

After evaluation of this code fragment, `*result*` holds the symbol `bind`, the value of the last binding with `let-process-fluid`. Again, the mutation via `set-process-fluid!` is gone in the new thread. Thus, programs that mutate process fluids become sensitive to the use of threads in subtle, and possibly non-modular ways.

Unfortunately, the problem is deeply rooted: The given combination of thread locality and mutation makes it impossible for `call-with-current-continuation` to capture the values of process fluids in a meaningful way. In particular, it is unclear *what* values the invocation of a continuation should re-install: those active at the time of capture, at the time of forking the thread, or at invocation time. No such semantic issues occur in programs that only use dynamic binding and never mutate process fluids. Consequently, programmers must take care when writing programs that mutate process resources and migrate continuations between threads.

5. Thread-local process resources

Process fluids are an appropriate abstraction for representing process resources in Scheme. However, scsh must still communicate their values to the underlying Unix process. This section describes scsh's general framework for this procedure, called *resource alignment*, and shows how it applies to the various actual POSIX process resources.

5.1. Resource alignment

A simple approach for implementing thread-local process resources is to adjust the process resources during each thread context switch: Before the scheduler runs the next thread, it updates the process resources to the values of the process fluids of the new thread, *aligning* the process resources with their Scheme counterparts. Unfortunately, this method requires system calls for each context switch and is therefore expensive.

As only certain system calls access the actual process resources, it is not necessary that the process resources and the values of the respective process fluids match all the time. It is sufficient for scsh to align a process resource when the thread actually performs a system call that accesses the resource. For example, the `open` system call would be implemented as follows:

```
(define (open filename)
  (chdir-syscall (process-fluid $cwd))
  (set-umask-syscall (process-fluid $umask))
  (open-syscall filename))
```

Here, `chdir-syscall` and `set-umask-syscall` set the respective process resources directly; `open-syscall` is the interface to the `open` system call. Of course, this code has a race condition: Another thread could align the umask and the current working directory with its own values before the `open`. Locks remedy this problem by performing alignment and the actual system call atomically:

```
(define cwd-lock (make-lock))
(define umask-lock (make-lock))
(define (open filename)
  (obtain-lock cwd-lock)
  (obtain-lock umask-lock)
  (chdir-syscall (process-fluid $cwd))
  (set-umask-syscall (process-fluid $umask)
  (open-syscall filename)
  (release-lock umask-lock)
  (release-lock cwd-lock))
```

`Make-lock` creates a mutex lock. After a thread has called `obtain-lock` on this lock all other threads doing the same will block. Subsequently, each call to `release-lock` will unblock one of the waiting threads.[5]

The performance of this approach is still not optimal: for each `open`, scsh executes one `chdir` and one `set-umask`, regardless of the actual values of the respective resources. To improve this, scsh caches the value of the process resource and compares the cache with the process fluid to determine if the process needs to align the resource.

5.2. *POSIX process resources in scsh*

Not all process resources are equal—the interface for accessing a resource as well as the manner the various system calls deal with a resource vary significantly. Therefore, scsh must use subtle variations for correctly aligning the various resources. This section describes how alignment works for the most important ones.

The file creation mask. The umask case is the simplest. There is a cache, and the implementation of `set-umask` sets the cache[6]:

```
(define *umask-cache* (get-umask))
(define (umask-cache) *umask-cache*)

(define $umask (make-process-fluid (umask-cache)))

(define umask-lock (make-lock))

(define (change-and-cache-umask new-umask)
  (set-umask-syscall new-umask)
  (set! *umask-cache* (get-umask)))
```

The `get-umask` procedure obtains the value of the umask process resource; `set-umask-syscall` changes it. `Change-and-cache-umask` assumes that its caller has already obtained the `umask-lock` lock which synchronizes access to the cache. It uses another call to `get-umask` to feed the cache: this ensures that the cache contains the same value as the actual resource for arguments where the behavior of the system call is not covered by the POSIX standard.

Next, here is the code to access and modify the process fluid:

```
(define (umask) (process-fluid $umask))
(define (thread-set-umask! new-umask)
  (set-process-fluid! $umask new-umask))
(define (let-umask new-umask thunk)
  (let-process-fluid $umask new-umask thunk))
```

The `umask` procedure is part of the scsh API. The implementations of the other API procedures—`set-umask` and `with-umask`—follow:

```
(define (set-umask new-umask)
  (with-lock umask-lock
    (lambda ()
      (change-and-cache-umask new-umask)
      (thread-set-umask! (umask-cache)))))

(define (with-umask* new-umask thunk)
  (let ((changed-umask
         (with-lock umask-lock
           (lambda ()
             (change-and-cache-umask new-umask)
             (umask-cache)))))
    (let-umask changed-umask thunk)))
```

The lock synchronizes the access to the cache. The `with-lock` procedure obtains the lock given as first argument during the evaluation of its second argument, a thunk.

The following procedure implements the actual alignment of the resource with the process fluid:

```
(define (align-umask!)
  (let ((thread-umask (umask)))
    (if (not (= thread-umask (umask-cache)))
        (change-and-cache-umask thread-umask))))
```

The conditional compares the value of the cache with the process fluid; if they do not match, `align-umask!` adjusts the resource. The caller of `align-umask!` must have obtained the `umask-lock` lock.

The current working directory. For the current working directory, caching is more involved, as the chdir syscall itself reads the current working directory in case the given path is not absolute. Scsh circumvents this case by always making the path absolute:

```
(define (change-and-cache-cwd new-cwd)
  (if (not (file-name-absolute? new-cwd))
      (chdir-syscall (assemble-path (cwd) new-cwd))
      (chdir-syscall new-cwd))
  (set! *cwd-cache* (cwd-syscall)))
```

As in the umask case, the cache is fed by consulting the operating system. This is to find out if the kernel has resolved any symbolic links. Reading, setting and aligning the current working directory is analogous to the umask case:

```
(define (cwd) (process-fluid $cwd))

(define (chdir cwd)
  (with-lock cwd-lock
    (lambda ()
      (change-and-cache-cwd cwd)
      (thread-set-cwd! (cwd-cache)))))

(define (align-cwd!)
  (let ((thread-cwd (cwd)))
    (if (not (string=? thread-cwd (cwd-cache)))
        (change-and-cache-cwd thread-cwd))))
```

The environment. The environment requires special treatment: Instead of being accessible via system calls, it resides in the C variable environ (of type char **). Programs normally access this vector through the getenv, putenv and setenv C library functions. Moreover, the environment affects only a limited number of system calls, namely those called by the execl/execv/execle/execlp/execvp family of functions.

Scsh represents the environment by an association list in Scheme and turns it into an C array on exec... calls only. Moreover, scsh caches the C version of the environment as long as the Scheme code does not modify it. The caching procedure sets environ**:

```
(define (change-and-cache-env env)
  (environ**-set env)
  (set! *env-cache* env))
```

Reading the resource is required only on startup of the system; it turns the C vector into a Scheme list.

The user and group identification. In Unix, user identification comes in three flavors:

1. The *real user ID* encodes the identity of the owner of the process. The operating system copies the value from the parent when creating the process.
2. The *effective user ID* determines which files the process may access.
3. The *saved set-user ID* is set by `exec...` on start of the process and provides an alternative value for the effective user ID.

We discuss only user identification here; group identification works the same way.

For changing the user ID settings, POSIX provides the `setuid` system call. Unfortunately, its semantics depends on the value of the effective user ID: If the current effective user ID is the ID of the super-user, `setuid` changes *all* three values to the *same* but arbitrary ID. However, afterwards, the effective user ID is no longer the ID of the super-user and `setuid` cannot change the IDs any more. Therefore, automatic maintenance as described for the other resources is impossible in general.

For unprivileged users things look slightly different: `setuid` sets *only* the effective user ID to either the real user ID or the saved set-user ID. The other IDs remain untouched. As the real user ID and the saved set-user ID may be different, both can act as a source for the effective user ID. This behavior is desirable for applications that are started with a special saved set-user ID but want to exploit it only for certain tasks such as maintaining lock files. A multithreaded application may want to equip each thread with one of the two IDs. To support this, scsh provides thread-local effective user IDs. The direct binding to the `setuid` function is still part of the scsh API as procedure `set-uid` to allow the super-user a non-reversible change of all three IDs.

The implementation of effective user IDs per thread is analogous to the umask case. A thread can read the effective user ID with `user-effective-uid` and set it with `set-user-effective-uid`. Depending on the platform, scsh uses one of the non-standard system calls `setreuid` or `seteuid` which change only the effective user ID to prevent the super-user from unintentionally changing all three IDs.

Aligning multiple resources simultaneously. Many system calls require the simultaneous alignment of several resources, and thus the acquisition of several locks. To prevent deadlock in general, scsh uses a special data type for resources that holds—along with the aligning procedure and the lock—a unique natural number putting the resources in a fixed order [22]. Here is the type definition for resources, using SRFI 9 records [15]:

```
(define *resource-counter* 0)

(define-record-type :resource
  (really-make-resource count align! lock)
  resource?
  (count resource-count)
  (align! resource-align!)
  (lock resource-lock))
```

```
(define (make-resource align! lock)
  (set! *resource-counter* (+ *resource-counter* 1))
  (really-make-resource *resource-counter* align! lock))
```

The `with-resources-aligned` procedure takes as argument a list of resources and a thunk. It first obtains the locks of the specificed resources in the order specified by the `resource-count` field, aligns the resources, calls the thunk, and returns its value after having released all locks. Defining a resource is trivial:

```
(define umask-resource
  (make-resource align-umask! umask-lock))
```

Now the machinery is in place to define Scheme bindings properly for system calls that access multiple resources. Here, for example, is the implementation of the `open-fdes` procedure which opens a file, given a set of flags and possibly a mode, and returns a file descriptor:

```
(define (open-fdes path flags . maybe-mode)
  (with-resources-aligned
   (list umask-resource
         cwd-resource
         euid-resource
         egid-resource)
   (lambda ()
    (open-syscall path flags
                  (:optional maybe-mode #o666)))))
```

The `open-syscall` procedure opens the file specified by `path` with umask, current working directory, effective user ID and effective group ID aligned. The `:optional` macro returns the default mode `#o666` if the caller supplied no third argument to `open-fdes`.

6. `Fork` vs. threads

The Unix `fork` system call creates and starts a child process that is a copy of the parent process, distinguished from the parent by the return value of `fork`. Moreover, the child has its own process ID, parent process ID and resource utilizations. The child process also gets copies of the file descriptors of the parent that, however, reference the same underlying objects.

In a user-level thread system, all threads are contained in the process. Consequently, the child process—being a duplicate of its parent—also runs duplicates of the threads of the parent process. Depending on the concrete thread system, this is desirable for the system threads, such as those doing I/O cleanup, running finalizers, etc. However, this is usually wrong for the threads explicitly created by a running program.

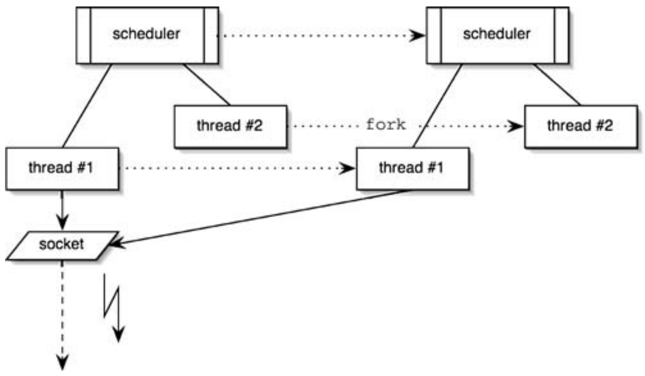

Figure 2. Interference between parent and child in a multithreaded Internet server.

The most common use of `fork` in scsh programs is from the `run` and `&` forms that run external programs—`run` runs a program, returning when it is finished, while `&` runs a program in the background and returns immediately. In Unix, the only way to run another program is to replace the running process via `exec...`. Hence, `run` and `&` first fork, and the newly created child then replaces itself by the new program. Unfortunately, the delay between `fork` and `exec...` creates a race condition: other threads of the running program can get scheduled *in the child*.

This race can have disastrous consequences. As an example, consider the CGI interface of the Scheme Untergrund web server [26]. The server starts a separate thread for each connection. Some connection requests require starting an external program such as a CGI script [3]. Now, consider a web server simultaneously serving two connections as shown in Figure 2. Thread #1 is busy serving a connection on the shown socket. Thread #2 forks in order to exec a CGI program. This creates an exact replica of the parent process, including the scheduler and all of its child threads which share access to the file descriptors of the parent process. It is now possible that the child scheduler schedules thread #1, thereby interfering with the parent's thread #1. This at least leads to mangling of the output.

This problem is well known in the realm of OS-level thread systems. Specifically, IEEE 1003.1-2001 [21] specifies that the child runs only the currently executing thread:

> A process shall be created with a single thread. If a multithreaded process calls `fork()`, the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. [...]

"Forking the current thread" is a more useful intuition for what `fork` should do. However, this notion is rife with ambiguity. (For example, what happens if the current thread is holding on to a mutex another thread is blocked on, and then, in the child, releases that mutex?) Moreover, `fork` has been notoriously difficult to implement correctly in Unix systems (see also Section 9).[7]

The general problem of implementing the notion of "forking the current thread" arises in every system combining user-level multithreading with an interface to `fork`.

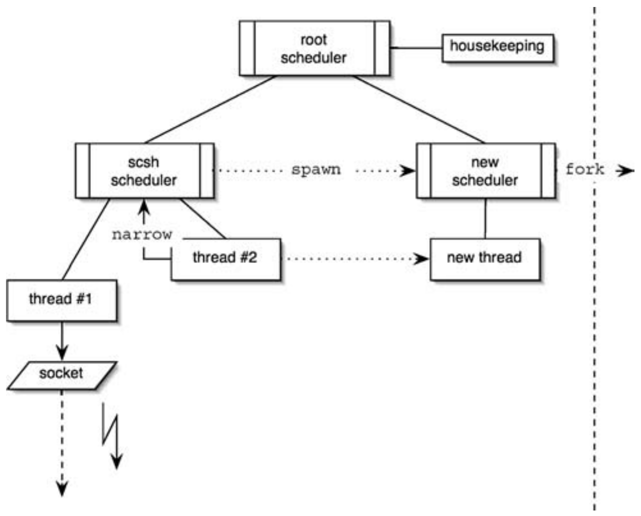

Figure 3. The `narrow` operation.

Fortunately, a nestable-engines-based thread system enables a simple and straightforward solution: Scsh solves the problem by providing a special scheduler that accepts a special kind of event, called `narrow`, from its child threads. `Narrow` accepts a thunk as an argument, and causes the scheduler to spawn a new scheduler and suspend itself until the new scheduler terminates. The new scheduler starts off with a newly created single thread that runs the thunk.

The scsh scheduler sits beneath the root scheduler. Thus, the root scheduler can still perform the necessary housekeeping. Figure 3 shows the setup: The `narrow` call from thread #2 suspends to the scheduler, passing a thunk to run inside the narrowed thread under the new scheduler. The `fork` now happens in the narrowed thread, which also runs the thunk passed to fork. In the parent process, the narrowed thread terminates again which also returns operation to the original scheduler.

Thus, a simplified version of `fork` in scsh (the actual production code needs to perform more complex argument handling and avoid an additional subtle race condition) looks like this:

```
(define (fork thunk)
  (let ((proc #f))
    (narrow
     (preserve-ports
      (preserve-process-fluids
       (lambda ()
         (let ((pid (%%fork)))
           (if (zero? pid)
               (call-terminally thunk) ; Child
               (set! proc (new-child-proc pid))))))))
  proc))
```

`%%fork` is the pure POSIX *fork* system call. It returns 0 in the child, and a non-zero process ID in the parent. `Call-terminally` runs the thunk in an empty continuation to save space and guarantee that the child terminates once `thunk` returns.

Note that, just as with `thread-fork`, `fork` needs to preserve the process fluids via `preserve-process-fluids`. Moreover, `preserve-ports` preserves the usual fluids holding the `current-`{`input,output,error`} ports.

This implementation of `fork` avoids the various semantic pitfalls: All threads are still present inside a `narrow`; they are merely children of a suspended scheduler. Therefore, if, for example, the current thread releases mutex locks other threads are blocked on, these threads are queued with their respective schedulers and can continue after the `narrow` completes. There are no restrictions on what the narrowed thread can do.

The implementation of `fork` actually shipped with scsh also allows duplicating all threads in the child. Consequently, through the use of nested schedulers `narrow` and `spawn`, the programmer has fine-grained control over the set of running threads.

Of course, the user-level program might create its own schedulers beneath the scsh scheduler. This, in general, requires that the new scheduler passes `narrow` events upwards in the scheduler tree to the scsh scheduler, which is trivial in the Scheme 48 thread system. On the other hand, it is possible that an application needs to handle `narrow` in a different way. The key observations of this work are that `narrow` is the appropriate mechanism for the feasible common cases, and that nestable schedulers provide a suitable implementation mechanism for providing a `fork` with well-defined behavior.

7. Miscellaneous Unix services

On Unix systems, the historical assumption that a program conceptually runs a single-threaded process is deeply rooted and pervasive. The process resources and `fork` are the most serious and fundamental issues, but these assumptions cause smaller problems in other, unexpected contexts. This section gives a sampling of some of the other instances in scsh development where we needed to go to extra lengths to make Unix C libraries properly accessible from multithreaded scsh applications:

- The `gethostbyname`/`gethostbyaddr` calls block all threads while performing DNS queries.
- The syslog facility supports only a single connection per process.
- Xlib's functions for event reading, `XPeekEvent` and `XNextEvent`, also block if no event from the X server is available.

7.1. DNS queries

A user-level thread implementation must never call C functions that might block the process and thereby stop all threads. All POSIX system calls can operate in non-blocking mode. Unfortunately, the same is not true for the standard C library. Two problematic cases are the `gethostbyaddr` and `gethostbyname` functions, which turn host names into IP addresses and vice versa. These functions are indispensable for writing almost any kind of

Internet server. They block until they receive an answer or timeout. Thus, the process calling `gethostby...` can block for up to several minutes.[8] To work around this problem, the Scheme Untergrund networking package [26] for scsh provides a library for DNS queries written directly in Scheme.

7.2. Syslog

Another problem is the standard C library's interface to the system message logger: The `openlog` function creates a connection to the `syslogd` daemon. The `syslog` function sends the actual messages to the daemon. The syslog daemon processes the messages according to the parameters of `syslog` and the ones specified by the most recent `openlog` call. Calls to `openlog` may not nest.

Therefore, scsh treats the connection to the logger analogously to the process resources mentioned in Section 3: The interface to `openlog` virtualizes connections to the loggers as *syslog channels*. The syslog channel records all parameters given to `openlog`. Scsh stores the channel in a process fluid and maintains a cache for the current channel. When another thread calls `syslog` and the cache differs from the thread's connection, scsh closes the connection to the syslog daemon using `closelog` and reconnects with the parameters obtained from the process fluid. Thus, every thread has its own virtual connection to the syslog daemon.

7.3. Xlib event handling

Xlib [20], the low-level library of the X Window System, is another C library with blocking functions: The library uses a queue to collect events sent from the X server, but the functions for dequeuing, `XPeekEvent` and `XNextEvent`, both block if the event queue is empty. They return only after the X server has sent a new event. To check the status of the queue, Xlib provides the `XPending` function that returns the number of pending events. It could be used to avoid blocking on an empty queue. However, this polling strategy is CPU-intensive and not desirable in general.

Fortunately, Xlib exports the socket on which the X server sends the events. By using scsh's interface to the `select` system call, a program can wait for the queue to become non-empty without polling. Scsh's interface to Xlib uses this trick to ensure that it calls `XNextEvent` only if the event queue is non-empty.

8. Lessons learned

Integrating the scsh API with the user-level thread system of Scheme 48 has taught us a number of lessons about system design in the presence of threads. They concern the adaptation of existing APIs to multithreaded environments, the design of thread systems in general, and the integration of foreign code with systems employing user-level thread systems.

8.1. API design

In a sense, the original scsh API repeated the mistakes of Unix standard C library which contains many functions incompatible with multithreading, leading to the complications described here. On the other hand, the complications were hardly avoidable, given scsh's design philosophy of making available the full functionality of POSIX to the Scheme programmer. Given these requirements, we conjecture that scsh's API design has held up reasonably well.

Still, some aspects of the API will probably require reconsideration. Notably, `fork` in scsh also supports an alternative form of invocation (not covered in Section 6) which returns both in the parent and in the child with different return values, rather than calling a thunk in the child, thus mirroring the POSIX invocation of `fork` more closely. Scsh still supports the old form, but, as the call to `fork` returns in the child from the `narrow`, this usage almost certainly leads to the problems described in Section 6. This has, in fact, happened to users, and the difficulty of detecting or understanding the source of the problem suggests that we should abolish this part of the API in a future version.

Consequently, API designers should always consider compatibility with multithreading, even if the system at hand does not support multithreading yet—it probably will at some time in the future.

8.2. User-level thread systems

Introducing user-level threads into an existing programming language implementation running under a Unix-style operating system is a far-reaching procedure: it affects many different parts of the system, both in design and implementation. Thus, it is necessary to conduct a global sweep through the code base to look for places where threads affect the existing functionality. The main issues—preemption and I/O—may be obvious, but many subtle issues lurk in unexpected places. In fact, we shipped the first version of scsh 0.6 without even having noticed the problem with `fork`.

It would be a worthwhile effort to re-examine the issues of this paper making use of operating-system threads which are becoming a reliable part of Unix derivatives. These Unices usually provide thread-compatible variants of some APIs that cause trouble. Still, most problems are likely to remain: It is impractical to map Scheme threads directly to operating-system threads, as the latter kind tends to be prohibitively expensive in terms of time and space. Rather, the system would schedule the Scheme threads on a limited number of operating-system threads, thus having every operating-system thread run multiple language-level threads. Consequently, we expect our work to remain relevant in such a setting.

8.3. Thread system design

Not all programming language implementations offer the luxury of a nestable thread system that allows users to define their own schedulers. Nestable engines clearly subsume traditional, "flat" systems, but our work is an indication that they are work well invested:

they allow fine-grained control over the workings of thread scheduling, specifically in the context of solving the `fork` problem.

More generally, a scheduler in a nestable-engines thread system plays the role of an operating system, allocating resources to its threads and providing a barrier to higher levels in the thread hierarchy. Scsh is a concrete example, as it offers each thread—to the extent possible—its own copy of Unix. Thus, nestable engines are an excellent programming-language-level substrate for implementing operating system facilities, offering control hard to achieve in flat thread systems.

8.4. Foreign code

Generally, threads complicate the integration of code written in a different language, because the languages involved are likely to be using different thread systems. Our work has shown that foreign code must obey some general restrictions:

– Foreign functions must not block indefinitely.
– Implicit state such as the process resources must be multiplexed via thread-local process resources.
– Non-reentrant foreign function APIs such as `syslog` must be virtualized to reentrant interfaces.

If the base language uses resource alignment as presented in this paper, two additional restrictions for the foreign code emerge:

– If the foreign code accesses a process resource, the value of the resource must be aligned in the base language before calling the foreign function.
– The foreign code must not modify a process resource directly as this would undermine the caching mechanism. Instead, the foreign code should call the modification procedure from the base language to change the value of the resource.

9. Related work

The *parameter* mechanism specified in SRFI 39 [6] is conceptually similar to process fluids: parameters also support (dynamic) binding and mutation. SRFI 39 intentionally does not specify the semantics of parameters in the presence of threads. This allows the existing implementations of the parameter API to support the SRFI, even though the semantics differs when it comes to threads.

MzScheme [7] is one of the implementations with both a parameter mechanism and support for threads. In MzScheme, spawned threads always inherit the parameter bindings of the spawning thread. Moreover, unlike scsh, MzScheme uses parameters for the entire dynamic environment, including the settings for `current-input-port` and `current-output-port` and for the current exception handler. However, the `error-escape-handler` parameter for the current exception handler does not propagate to spawned threads—this would cause a space leak [1]. In MzScheme, mutation of parameters is always thread-local. This incurs the same problems as described in Section 4.4. More seriously, MzScheme disallows invoking an escape procedure in a thread other than the one where it was captured.

The (as of the time of writing) soon-to-be-released version 4.0 of Gambit-C also supports SRFI 39's parameters and threads. Just as in MzScheme, a new thread inherits the parameter bindings from its parent thread. Mutation however is "binding-local": by default, it is visible in other threads unless there is an intervening binding [5]. Binding-local mutation avoids the semantic problems mentioned in Section 4.4, but comes with its own problems, as discussed extensively on the SRFI 39 mailing list.

As the discussions on the SRFI 39 mailing list [25] and the semantic differences between parameters in MzScheme in Gambit-C show, the difficulties in integrating threads with first-class continuations, dynamic binding and thread-local storage are numerous. The authors have, together with Knauel and Kelsey, examined the general semantic issues arising from this combination as well as the effect of the various design decisions on the ability to write modular abstractions in a systematic way [9]. Our conclusion is that separating threads from other aspects of language design leads to the most flexible and modular primitive abstractions. This in turn leads to the separation of fluids and thread-local cells, and to the semantics of fluids and `spawn` as described here. Parameters and mutable process fluids are concessions to API demands, but invite modularity violations as described in Section 4.4.

MzScheme also generally addresses the programming-language-as-operating-system issue [8] in the context of a flat thread system: MzScheme's development environment, DrScheme, needs to be able to completely control user programs started from it. This is one of the primary reasons for the inheritance of parameter values—inheritance prevents threads started from user programs from accessing security-relevant objects from higher levels in the thread hierarchy. Given the semantic problems caused by inheritance in conjunction with continuations, engines seem a more elegant approach to the issue and allow significantly more flexibility in designing operating-system-like facilities.

POSIX [21] specifies that `fork` should replicate only the calling thread. The man-page also mentions a proposed `forkall` function that replicates all running threads in the child. However, `forkall` was rejected for inclusion in the standard. The man-page lists a number of semantic issues for both `fork` and `forkall` that arise in the context of the Unix API. Specifically, a kernel-level thread system needs to deal with threads that are stuck in the kernel at the time of the `fork`. Reports of problems with handling or implementing `fork` with the proper semantics abound. Examples can be found in the FreeBSD commit logs and various Linux forums. Details vary greatly depending on implementation details of the operating system kernel and the thread system at hand.

Several other programming language implementations combine access to the underlying `fork` system call with a user-level thread system, notably Python, Ruby, and Perl. None of them, however, address the problem of duplicating all running threads in the child.

The problem of the blocking `gethostbyname`/`gethostbyaddr` functions is well recognized in the C community: The GNU adns C library [14] also provides an implementation of asynchronous DNS lookups.

10. Conclusion

Scsh combines user-level threads and the Unix API to yield a powerful tool for concurrent systems programming. The scsh API tries to maintain an analogy between threads and processes wherever possible. Specifically, threads see process resources as thread-local, and `fork` only "forks the current thread." The API issues involved are not new, but they occur in new forms in the context of Scheme 48's user-level thread system, scsh's support for the full POSIX API, and first-class continuations. Specifically, the solutions have led to the design of the process-fluid mechanism for managing thread-local dynamic bindings as well as of the `narrow` thread primitive which allows, together with nested threads, more fine-grained control over the set of running threads.

Acknowledgments

The implementation of thread-local process resources was developed in collaboration with Olin Shivers during Martin Gasbichler's visit to MIT. An email discussion with Richard Kelsey eventually led to the design of thread cells and process fluids. He specifically proposed separating fluids from thread-local cells. We thank our reviewers who provided excellent feedback—their comments led to significant improvements in our presentation. We would also like to thank the users of scsh for constant feedback and valuable bug reports.

Notes

1. Scsh restarts system calls interrupted by the timer at the Scheme level.
2. If the thread system would implement context switching on top of the regular R^{5}RS `call-with-current-continuation`, each context switch would run `dynamic-wind` thunks, causing surprising effects with typical uses of `dynamic-wind`.
3. In Scheme 48, the current exception handler is also a regular fluid. All non-local control transfer happens via `call-with-current-continuation`, so no new semantic issues arise in connection with the exception system.
4. This situation is not hypothetical as it may look at first: multithreaded Web servers are a natural application for such migration of continuations between threads [10].
5. Scsh always acquires the locks according to a fixed partial order, thus preventing deadlocks. The end of this section explains this mechanism.
6. The actual implementation initializes the cache when the system starts.
7. In systems where threads are implemented as processes, the correct implementation of `fork` is trivial. However, then the implementation of `exec...` becomes a problem because the new program must replace *all* threads of the old one. On the other hand, the implementation of `exec` is trivially correct in scsh.
8. Internet applications such as Netscape [19] and the Squid web cache [24] work around this problem by launching a second process to perform DNS queries. This allows the normal process to continue asynchronously or poll a pipe to the helper process using `select`.

References

1. Biagioni, E., Cline, K., Lee, P., Okasaki, C., and Stone, C. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, **11**(2) (1998) 209–225.

2. Cejtin, H., Jagannathan, S., and Kelsey, R. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, **17**(5) (1995) 704–739.
3. CGI 1.0: 1999, CGI: Common Gateway Interface. `http://www.w3.org/CGI/`.
4. Dybvig, R.K. and Hieb, R. Engines from continuations. *Computer Languages*, **14**(2) (1989) 109–123.
5. Feeley, M. Parameters in Gambit-C. Personal communication, (2001).
6. Feeley, M. SRFI 39: Parameter objects, (2002) `http://srfi.schemers.org/srfi-39/`.
7. Flatt, M. PLT MzScheme: Language Manual. Rice University, University of Utah. Version 203, (2003).
8. Flatt, M., Findler, R.B., Krishnamurthi, S., and Felleisen, M. Programming languages as operating systems. In *Proc. International Conference on Functional Programming 1999* P. Lee (Ed.) Paris, France, ACM Press: New York, 1999, pp. 138–147.
9. Gasbichler, M., Knauel, E., Sperber, M., and Kelsey, R. How to add threads to a sequential language without getting tangled up. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming* M. Flatt (Ed.). Boston, 2003.
10. Graunke, P., Krishnamurthi, S., Hoeven, S.V.D., and Felleisen, M. Programming the web with high-level programming languages. In *Proceedings of the 2001 European Symposium on Programming* D. Sands (Ed.) Springer-Verlag Genova, Italy, pp. 122–136, 2001.
11. Haynes, C.T. and Friedman, D.P. Engines build process abstractions. In *ACM Conference on Lisp and Functional Programming* 1984, pp. 18–24.
12. Haynes, C.T. and Friedman, D.P. Abstracting timed preemption with engines. *Computer Languages*, **12**(2) (1987a) 109–121.
13. Haynes, C.T. and Friedman, D.P. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, **9**(4) (1987b) 582–598.
14. Jackson, I. and Finch, T. GNU adns, (2000) `http://www.chiark.greenend.org.uk/~ian/adns/`.
15. Kelsey, R. SRFI 9: Defining Record Types, (1999) `http://srfi.schemers.org/srfi-9/`.
16. Kelsey, R., Clinger, W., and Rees, J. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, **11**(1) (1998) 7–105.
17. Kelsey, R. and Rees, J. Scheme 48 reference manual. Part of the Scheme 48 distribution at `http://www.s48.org/`, 2002.
18. Kelsey, R.A. and Rees, J.A. A tractable Scheme implementation. *Lisp and Symbolic Computation*, **7**(4) (1995) 315–335.
19. Netscape: Netscape Browser Central, 2002. `http://browsers.netscape.com/browsers/main.tmpl`.
20. Nye, A. *Xlib Reference Manual*, Vol. 2 of *The Definitive Guides to the X Window System.* O'Reilly, 1992.
21. POSIX. The Open Group Base Specifications Issue 6 IEEE Std 1003.1-2001, 2001. `http://www.opengroup.org/onlinepubs/007904975/`.
22. Reppy, J.H. *Concurrent Programming in ML.* Cambridge University Press, 1999.
23. Shivers, O. A Scheme shell. Technical Report TR-635, Massachusetts Institute of Technology, Laboratory for Computer Science, 1994.
24. Squid: Squid Web Proxy Cache, 2002, `http://www.squid-cache.org/`.
25. SRFI-39 mailing list. 2002. `http://srfi.schemers.org/srfi-39/mail-archive/maillist.html/`.
26. SUnet. The Scheme Untergrund Networking Package, 2003. `http://www.scsh.net/sunet/`.