Processes vs. User-Level Threads in Scsh

Martin Gasbichler Michael Sperber
Universität Tübingen
{gasbichl,sperber}@informatik.uni-tuebingen.de

Abstract

The new version of scsh 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, forked threads can inherit resources from the parent thread. To store these resources scsh uses *preserved thread fluids*, a special kind of fluid variables. The paper gives a detailed description of an efficient implementation for thread-local process resources. Scsh also provides an interface to the `fork` system calls which avoids common pitfalls which arise with a user-level thread system. Scsh contains a binding for `fork` that forks "only the current thread."

1 Introduction

Scsh [14] is a variant of Scheme 48 [11, 10] with extensive support for Unix systems and shell programming. Specifically, it contains full access to all basic primitive functions specified by POSIX. Scsh 0.1, the first version, came out in 1994.

In late 1999, the scsh maintainers set out to produce a version of scsh capable of multithreading. The main motivation was to improve scsh's abstraction of the operation system [15] as well as to implement multi-threaded Internet servers with scsh. At the time, scsh was based on Scheme 48 version 0.36 which did not support multithreading. Meanwhile, Scheme 48 had reached version 0.53 which did support fast, preemptive user-level multithreading. Hence, the task was originally to disentangle scsh from the underlying 0.36 substrate and port it to 0.53.

However, once the basic porting work was finished, it turned out that some of the POSIX functionality interfered with the user-level threads. Writing multi-threaded scsh programs is easiest when threads behave mostly like processes. However, this analogy breaks in a straightforward implementation of user-level threads and POSIX system calls in two important respects:

- A number of system resources, such as the environment or current working directory are process-*local* but thread-*global*. This would cause programs which would 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 would copy the entire process, and 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 powerful enough.

Moreover, the C library causes some problems: The `syslog` interface to the system's message logging facility offers only a single global, implicit connection which needs to be multiplexed among threads. Also, some POSIX library calls block indefinitely, making timely preemption of threads impossible while they are running. The most notable examples are `gethostbyname` and `gethostbyaddress` whose functionality is indispensable for implementing multi-threaded Internet servers.

This paper describes the steps taken in scsh 0.6 towards handling or working around those problems:

- Scsh represents thread-local process resources by *thread fluids*, thread-local cells which support binding, assignment, and preservation across a `fork`-like operation on threads.
- Scsh uses *resource alignment* to lazily keep the internal representation synchronized with the process state.
- Scsh's thread system supports a novel primitive called `narrow` which allows implementing a fork operation that forks "only the current thread."

Overview Section 2 gives a brief account of the Scheme 48 thread system. Section 3 briefly describes the process resource functionality POSIX offers. Section 4 describes thread fluids which scsh uses to represent process resources. Next, Section 5 describes how to use thread 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 the standard C library with user-level threads. Section 8 reviews some related work, and Section 9 concludes.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

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 [7, 8, 4] and is almost entirely implemented in Scheme, and therefore extensible without changing the VM in any way. The VM supports the thread system in two ways:

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

Each thread is represented by a Scheme object which, 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 contexts of threads. For a blocked thread, the thread object contains a saved continuation and an interrupt mask.

As in any continuations-based system, Scheme 48 needs to take `dynamic-wind` into account: 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 the full-scale `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. Specifically, 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 can run a thread for a slice of its own time by calling `(run` *thread time*`)`. The call to `run` returns either when the time slice has expired or an *event* happened. This event might signify termination, an interrupt, a blocked operation, another thread becoming runnable, or a request from the thread to the scheduler. For example, a thread can cause the scheduler to spawn a new thread by returning a `spawned` event along with a thunk to be run in the new thread. Note that it is easily possible to pass an event upwards in the thread tree if the current scheduler is unwilling to handle it.

Thus, a scheduler performs at least two tasks: it implements a scheduling policy by deciding which threads to run for how long, and it must handle 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 for each *command level* that encapsulates a state of interaction with the user. This allows Scheme 48 to cleanly interrupt all running threads at any time by entering a new command level, and later continue them by throwing back into an old one. The built-in schedulers all use a simple round-robin scheduling policy.

3 Unix Process Resources

The representation of a process within the kernel of a Unix operation system contains several *process resources*. The kernel initializes these resources during creation of a process, typically by copying the values from the parent process. Here are the most important process resources:

- the current working directory,
- the file mode creation mask, called *umask*,
- the user and group ID,
- the environment.

For each resource the kernel provides system calls to read and set the resource. For the current working directory, `getcwd` returns the path as a string and `chdir` sets the directory to a new path.

A number of system calls implicitly consult the resources of the calling process. In the current working directory example, when the process uses the `open` system call to open a file, the kernel interprets the filename argument of `open` relative to the value of the current working-directory resource. Likewise, `chdir` resolves its path argument relative to the current working directory if it does not start with a slash.

4 Scheme Thread-Local Resources

Threads share state. This enables inter-thread communication by explicitly providing to several threads access to shared state by lexical binding. The various process resources, however, constitute *implicit* state, just like the settings for `current-input-port` and `current-output-port`.

For managing the latter, Scheme 48 keeps their values in *fluid bindings*: a fluid is a cell that allows dynamic binding. `(make-fluid` *v*`)` creates a fluid with default value *v*, `(fluid` *f*`)` references the value bound to a fluid, and `(let-fluid` *f v t*`)` calls thunk *t* with fluid *f* bound to value *v* during the dynamic extent of this call. That is, the fluid mechanism resets the binding to the value before the `let-fluid` if the thunk calls a previously stored continuation, and if the thunk *t* captures a continuation, on a later call to this continuation the fluid mechanism again binds the fluid *f* to the value *v*. Here are some examples from a Scheme 48 session, where > marks the command prompt and `call-with-current-continuation` is abbreviated as `call/cc`:

```
> (define f (make-fluid 1))
> (fluid f)
⤳ 1
```

`Let-fluid` binds the fluid only during the execution of the thunk:

```
> (+ (let-fluid f 3 (lambda () (fluid f)))
     (fluid f))
⤳ 4
```

Save a continuation with a dynamic binding in `*k*`:

```
> (define *k*)
> (let-fluid f 25
      (lambda ()
         (* (call/cc (lambda (k) (set! *k* k) 10))
            (fluid f))))
⤳ 250
```

[1] Scsh restarts system calls interrupted by the timer at the Scheme level.

The top-level binding is still the initialization value:

```
> (fluid f)
⤳ 1
```

Throwing back into the thunk using the stored continuation reactivates the binding introduced by the `let-fluid` above:

```
> (*k* 100)
⤳ 2500
```

Capture a continuation that returns the value of the fluid added to the argument of the continuation:

```
> (define *kk*)
> ((lambda (x) (+ x (fluid f)))
   (call/cc
     (lambda (k)
       (set! *kk* k)
       20)))
⤳ 21
```

Calling the stored continuation amounts to throwing out of the dynamic extent of the thunk:

```
> (let-fluid f -1
      (lambda ()
         (*kk* 3)))
⤳ 4
```

The environment that associates fluids with their values is local to each thread. Each newly spawned thread gets a fresh dynamic environment from its scheduler, typically with all fluids bound to their default values:

```
> (define f (make-fluid 1))
```

Start a new thread:

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

For process resources, sharing their settings among the threads is undesirable, as threads might interfere with each other, even though there is no explicit, intended communication among them. Moreover, it often makes more sense to dynamically bind a process resource rather than mutate it permanently. (To this end, scsh has always offered constructs like `with-cwd`, `with-env` etc.)

Therefore, fluids seem like the right low-level means for implementing thread-local process resources. However, they do not support assignment, primarily because its intended semantics is not immediately obvious: should assignments be visible in other threads?[2] Scsh therefore offers a primitive mechanism orthogonal to fluids called *thread-local cells* or *thread cells*: a thread cell supports assignment, and assignment is always thread-local. `(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*.

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

Start a new thread which 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
```

Moreover, scsh also ships with an abstraction built upon thread cells—*thread fluids*. Thread fluids obey the rules of dynamic binding just as ordinary fluids but also support mutation like thread cells. In fact, a thread fluid corresponds to a fluid containing a thread cell. Here is the transcript of a Scheme 48 session using thread fluids:

```
> (define f (make-thread-fluid 1))
```

Save a continuation with a dynamic binding in `*k*`:

```
> (define *k*)
> (let-thread-fluid f 25
      (lambda ()
         (* (call/cc (lambda (k) (set! *k* k) 10))
            (thread-fluid f))))
⤳ 250
```

Modify the value of the thread fluid:

```
> (set-thread-fluid! f -1)
> (thread-fluid f)
⤳ -1
```

A call to the stored continuation shows that the dynamic binding is still active:

```
> (*k* 100)
⤳ 2500
```

To sum up, a thread fluid supports *both* binding and thread-local assignment, thereby offering the right functionality for representing process resources per thread.

Just as with fluids, a newly spawned thread receives the default values for the thread-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 parents.[3] Simple lexical bindings allows communicating a thread fluid to a spawned thread "by hand:"

[2] The *parameter* mechanism of MzScheme [6] supports both binding and assignment. Assignment is always thread-local. The (as of the time of writing) soon-to-be-released version of Gambit-C also has parameters. These will have "binding-local" assignment: assignment by default is visible in other threads unless there is an intervening binding [5].

[3] In fact, in MzScheme, a spawned thread inherits the parameter bindings from the spawning thread. However, the built-in `error-escape-handler` parameter alone does not propagate to spawned threads—this would cause a space leak [1]. The potential for space leaks alone suggests that the programmer should have control over the propagation of thread fluid values to spawned threads.

```
(define t-fluid (make-thread-fluid #f))
  ...
  (spawn
    (let ((val (thread-fluid t-fluid)))
     (lambda ()
       (let-thread-fluid t-fluid val
        ...))))
```

The thread-fluids library exports two procedures `make-preserved-thread-fluid` and `preserve-thread-fluids`: `make-preserved-thread-fluid` is just like `make-thread-fluid`, but marks the thread fluid for preservation. `Preserve-thread-fluids` accepts a thunk as an argument and returns another thunk wrapped in pairs of `let` and `let-thread-fluid` forms for all live thread fluids marked for preservation. Thus, the above code could be rewritten as:

```
(define t-fluid (make-preserved-thread-fluid #f))

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

The `thread-fluids` package also exports a procedure `fork-thread` with the following definition:[4]

```
(define (fork-thread thunk . rest)
  (apply spawn (preserve-thread-fluids thunk) rest))
```

Now a forked thread can inherit values from its parent:

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

Mutation of preserved thread fluids is still thread-local:

```
> (begin
     (let-thread-fluid f 1
         (lambda ()
            (fork-thread
               (lambda () (set-thread-fluid! f -1)))))
    (thread-fluid f))
⇝ 0
```

5 Thread-Local Process Resources

To enable modular system programming in the presence of threads the values of process resources must be local to each thread. To mimic processes, freshly created threads should inherit the resources from their parents. Preserved thread fluids provide the right vehicle to store the values within the threads, but communicating the values to the actual process resources requires additional machinery.

A simple approach to implement thread-local process resources is to adjust the process resources on a thread context switch: If the scheduler suspends the current thread the values of all resources are saved in thread fluids. Before the scheduler runs the next thread, it updates the process resources with the values of the thread fluid of the respective thread. This means that the process resources are *aligned* with the thread fluids on a context switch. Unfortunately, this method requires system calls for saving and restoring on each context switch as well as crossing the C foreign function interface boundary, both of which are comparatively expensive.

As the kernel inspects the process resources only during certain system calls, it is not required that process resources and thread fluids match all the time. It is sufficient to align a process resource when the thread actually performs a system call which is affected by the resource. The `open` system call would then be defined as:

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

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 (thread-fluid $cwd))
  (open-syscall filename)
  (release-lock umask-lock)
  (release-lock cwd-lock))
```

`Make-lock` creates a standard mutex lock. After one thread has called `obtain-lock` on this lock all other threads doing the same will block until the lock is released by `release-lock`.

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. Scsh caches the value of the process resource whenever it is changed and compares the cache with the thread fluid to determine if the process needs to align with the resource. The rest of the section describes how scsh implements this strategy for the various process resources.

The umask case is the simplest. There is a cache and a replacement for `set-umask` that sets the cache:

```
(define *umask-cache* (process-umask)[5])
(define umask-lock (make-lock))
(define $umask (make-preserved-thread-fluid (umask-cache)))

(define (umask-cache)
  *umask-cache*)

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

This code uses another call to `umask` to feed the cache: this ensures proper error detection in case the specified value was not valid.

[4] One reviewer rightly noted that “A fluid friendly version of `fork` would have to be called `spoon`.” The next version of scsh will feature this alias.

[5] The actual implementation initializes the cache when the system starts.

Next, there is code to access and modify the thread fluid:

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

To change the umask scsh provides the following procedure:

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

A lock is required to synchronize the access to the cache. The following procedure aligns the resource with the thread fluid:

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

The test of the conditional compares the value of the cache with the thread fluid; the code in the consequence adjusts the resource in case of a mismatch. The following procedure aligns the umask and then calls its argument which is typically the actual system call wrapped in a thunk:

```
(define (with-umask-aligned* thunk)
  (obtain-lock umask-lock)
  (align-umask!)
  (with-handler
   (lambda (cond more)
     (release-lock umask-lock)
     (more))
   (lambda ()
     (let ((ret (thunk)))
       (release-lock umask-lock)
       ret))))
```

The lock prevents another thread from aligning the umask with its own value before the system call completes. As always with locks, some care must be taken to ensure the code releases the lock under unusual circumstances. The thunk argument usually contains only the call to the C function which in turn performs the system call so throwing out and back into its execution state by saved continuations is not an issue. However, in case the system call fails the C code will immediately raise an exception which allows execution to resume at a different point. To release the lock in this case the code above installs an exception handler which releases the lock and passes the exception along to the next handler: The with-handler procedure installs its first argument as a exception handler for the second argument. The handler releases the lock and calls the surrounding handler passed as argument more afterwards.

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 making the path absolute:

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

Again, the cache is fed by consulting the kernel, this time to find out if the kernel has resolved any symbolic links. Setting and aligning the current working directory is completely analogous to the umask case:

```
(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 requires special treatment: First, there is a direct access to the resource itself. It is stored in the C variable environ of type char **. Programs normally access this vector through the functions getenv, putenv and setenv provided by the C library. Moreover, the only system call the environment influences is exec*. Therefore, scsh represents the environment by an association list in Scheme and turns it into an C array on exec* only. In this case scsh maintains an association of the Scheme list and the C array to allow the latter to be reused and automatically deleted. The caching procedure sets environ**:

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

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

The last remaining process resource is the user identification.[6] In Unix, user identification comes in three flavors:

1. The *real user ID* encodes the identity of the owner of the process. The kernel 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.

For changing these values, POSIX specifies the system call setuid. Unfortunately, its semantics depends on the value of the effective user ID: If the 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. Automatic maintenance as described for the other resources is therefore not possible in general.

For unprivileged users things look slightly different: here 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 in turn. This behavior is desirable for applications which are started with a special saved set-user ID but want to exploit it only for certain tasks such as maintaining lock files.

[6] The following description translates literally to group identification. The presentation therefore does not consider group IDs further.

A multi-threaded application possibly wants to equip each thread with one of the two IDs. To support this, scsh provides thread-local effective user 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`. Scsh guards system calls operation on files with the `with-euid-aligned` macro. 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.

Now the machinery is in place to properly define Scheme bindings for resource-accessing system calls:

```
(define (open-fdes path flags . maybe-mode)
  (with-cwd-aligned
   (with-umask-aligned
     (with-euid-aligned
       (with-egid-aligned
         (%open path
                flags
                (:optional maybe-mode #o666)))))))
```

The `%open` procedure is bound to the `open` system-call. It 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 counterpart to `spawn`/`fork-thread` in the realm of Unix processes is called `fork`: it 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 childhas its own process ID, parent process ID and resource utilizations. The child process also gets copies of the parents file descriptors which, however, reference the same underlying objects.

In a user-level thread system, all threads are contained in the process. Consequently, the child process also runs duplicates of the threads of the parent process. Depending on the concrete thread system, this is desirable for the the system threads, such as those doing I/O cleanup, run finalizers, etc. However, this is usually wrong for the threads explicitly created by a running program. The most common use of `fork` in scsh programs is from the `&` and `run` forms that run external programs: in Unix, the only way to run another program is to replace the running process by it via `exec*`(3). Hence, `run` and `&` first fork, and the newly created child then replaces itself by the new program. Unfortunately, the delay between `fork` and `exec*` create a race condition: other threads of the running program can get scheduled *in the child*.

This race can have disastrous consequences: the Scheme Underground web server [17] 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 1. 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 children 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 thread #1. This at least leads to mangling of the output.

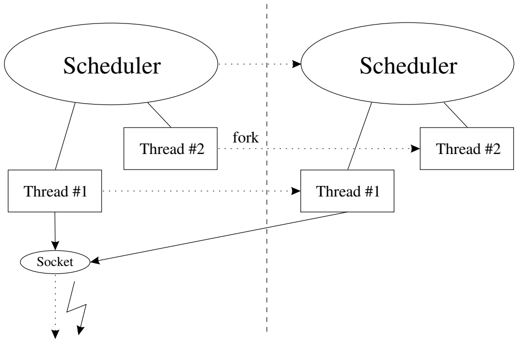

Figure 1. Interference between parent and child in a multi-threaded Internet server

This problem is well known in the realm of OS-level thread systems. Specifically, IEEE 1003.1-2001 [13] specifies that the child only runs the currently executing thread:

> A process shall be created with a single thread. If a multi-threaded 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 as such 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 8).[7]

Fortunately, the implementation issues in the context of a nestable-engines-based thread system are entirely different ones from more traditional settings: Scsh solves the problem by providing a special scheduler which accepts an additional kind of event from its children threads called `narrow`. `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 2 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 a subtle race condition) looks like this:

[7] In systems where threads are implemented as processes, the correct implementation of `fork` is trivial. However, then the implementation of `exec*`(2) 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.

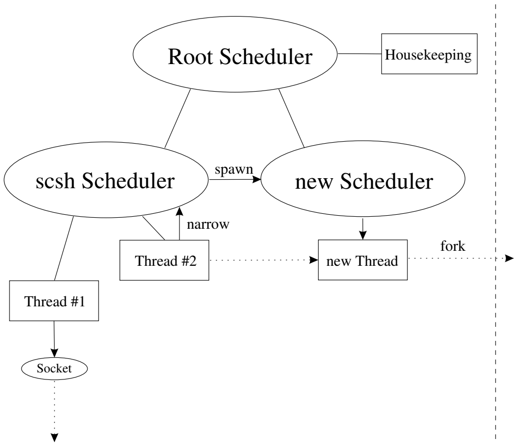

Figure 2. The `narrow` operation

```
(define (fork thunk)
  (let ((proc #f))
    (narrow
     (preserve-ports
      (preserve-thread-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 thread fluids via `preserve-thread-fluids`. Moreover, `preserve-ports` preserves the regular fluids holding the `current-{input,output,error}` ports.

This implementation of `fork` avoids the various semantic pitfalls: All threads are still present after 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 User-level threads and the C libraries

In addition to an interface to the Unix system calls, scsh also provides bindings for standard libraries. Two library facilities cause problems: DNS queries via `gethostbyname`/`gethostbyaddr` eventually block the process. The Syslog connections are an additional process resource. This section explains how scsh tackles these issues.

7.1 DNS queries

A user-level thread implementation must never call functions which might block the process and thereby stops all threads. All POSIX system calls can operate in non-blocking mode. Unfortunately, the same is not true for the standard C library: `gethostbyaddr` and `gethostbyname` 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`... blocks for up to several minutes[8]. To prevent scsh from blocking, we have written a library for DNS queries directly in Scheme; it is part of the upcoming version of the Scheme Underground networking package[17].

7.2 Syslog

Another problem is the standard C library's interface to the system message logger: The `openlog` function opens 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 last `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 thread 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 thread fluid. Thus, every thread has its own virtual connection to the syslog daemon.

7.3 FFI Coding Guidelines

Generally, threads complicate FFI issues because the language substrates on both sides of the FFI barrier are currently likely to be using different thread systems. The work on scsh indicates that coding guidelines should impose certain restrictions on foreign code called via the FFI:

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

[8] Internet applications such as Netscape [12] and the Squid web cache [16] work around this problem by launching a second process to perform DNS queries. This allows the normal process to continue asynchronously or block on a pipe to the helper process using `select`.

8 Related work

The POSIX manpage [13] specifies that `fork` replicate only the calling thread. The manpage also mentions a proposed `forkall` function that replicates all running threads in the child. However, `forkall` was rejected for inclusion in the standard. The manpage 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.

The GNU adns C library [9] also provides an implementation of asynchronous DNS lookups.

9 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 and scsh's support for the full POSIX API. The solutions have led to the design of the thread-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.

Acknowledgements

The implementation of thread-local process resources was developed in collaboration with Olin Shivers during the first author's visit at MIT. An email discussion with Richard Kelsey eventually led to the design of thread cells and thread fluids. He specifically proposed separating fluids from thread-local cells. Marcus Crestani implemented the DNS library for the Scheme Underground networking package. We also would like to thank all the users of scsh for constant feedback and valuable bug reports.

10 References

[1] Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, 11(2):209–225, December 1998.

[2] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, September 1995.

[3] CGI: Common gateway interface. `http://www.w3.org/CGI/`.

[4] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[5] Marc Feeley. Parameters in Gambit-C. Personal communication, September 2001.

[6] Matthew Flatt. *PLT MzScheme: Language Manual*. Rice University, University of Utah, August 2000. Version 103.

[7] Christopher T. Haynes and Daniel P. Friedman. Engines build process abstractions. In *ACM Conference on Lisp and Functional Programming*, pages 18–24, 1984.

[8] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):109–121, 1987.

[9] Ian Jackson and Tony Finch. GNU adns. `http://www.chiark.greenend.org.uk/~ian/adns/`, 2000.

[10] Richard Kelsey and Jonathan Rees. *Scheme 48 Reference Manual*, 2002. Part of the Scheme 48 distribution at `http://www.s48.org/`.

[11] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[12] Netscape. Netscape browser central. `http://browsers.netscape.com/browsers/main.tmpl`, 2002.

[13] The Open Group Base Specifications Issue 6 IEEE Std 1003.1-2001. `http://www.opengroup.org/onlinepubs/007904975/`, 2001.

[14] Olin Shivers. A Scheme Shell. Technical Report TR-635, Massachusetts Institute of Technology, Laboratory for Computer Science, April 1994.

[15] Olin Shivers. Automatic management of operating system resources. In Mads Tofte, editor, *International Conference on Functional Programming*, pages 274–279, Amsterdam, The Netherlands, June 1997. ACM Press, New York.

[16] Team Squid. Squid web proxy cache. `http://www.squid-cache.org/`, 2002.

[17] The Scheme Underground networking package. `http://www.scsh.net/sunet/`.