Kill-Safe Synchronization Abstractions

"Well, it just so happens that your friend here is only mostly dead.
There's a big difference between mostly dead and all dead."
– Miracle Max in *The Princess Bride*

DRAFT: March 29, 2003

Matthew Flatt
University of Utah

Robert Bruce Findler
University of Chicago

Abstract

When an individual task can be terminated at any time, cooperating tasks must communicate carefully. For example, if two tasks share an object, and if one task is terminated while it manipulates the object, the object may remain in an inconsistent or frozen state that incapacitates the other task. To support communication among terminable tasks, language run-time systems (and operating systems) provide kill-safe abstractions for inter-task communication. No kill-safe guarantee is available, however, for abstractions that are implemented outside the kernel.

In this paper, we show how a run-time system can support new kill-safe abstractions without requiring modification to the kernel, and without requiring the kernel to trust any new code. Our design thus frees the kernel implementor to provide only a modest set of synchronization primitives in the trusted computing base, and applications can still communicate using sophisticated abstractions.

1 Introduction

Most modern programming languages offer support for multiple tasks in the form of threads. Support for task *termination* is less widely implemented and generally less understood, but no less useful to programmers. The designers of Java, for example, understood the need for termination, and they included `Thread.stop` and `Thread.destroy` in the language. Flaws in the specification of `Thread.stop` forced its withdrawal, however, and `Thread.destroy` has never been implemented. Meanwhile, various extensions of Java have provided termination in a more controlled form [2, 3, 10, 11, 14], and termination of Java tasks is a driving goal of the new JSR-121 standard [23].

Termination in any language becomes troublesome when tasks share objects. Two tasks may share a queue, for example, and they may require that terminating one task does not corrupt or permanently freeze the queue for the other task. Java extensions such as JSR-121 solve the problem in the same way as conventional operating systems: they restrict sharing among terminable tasks to objects that are managed by the run-time kernel. Thus, if one task is terminated while manipulating a kernel-maintained queue, the kernel delays termination until it can leave the queue in a consistent state for the other task.

The conventional design places full responsibility on the kernel implementor to manage objects and termination correctly. On the one hand, when the kernel implementor's work is complete, application programmers need not worry about termination when using shared objects. On the other hand, application programmers are restricted to the kernel's abstractions for reliable communication; new communication abstractions require extensions to the kernel.

Our latest version of MzScheme [7] avoids this restriction through a novel set of primitives for managing tasks. Using the primitives, a programmer can implement a new kill-safe abstraction without modifying MzScheme's kernel. Crucially, the set of cooperating tasks that share an object need not be defined in advance, and the cooperating tasks need not trust each other. (They must trust only the implementation of the shared object.)

The new MzScheme primitives build on the observation that thread termination is merely indefinite suspension plus garbage collection. By creating a thread to manage a particular synchronization abstraction, a programmer can ensure that the abstraction instance is suspended when it might otherwise be killed. In short, when the instance object is "killed" as part of a task termination, it turns out to be "only mostly dead." A surviving task that shares the object can resurrect it. This technique succeeds because MzScheme's primitives allow a manager thread to preserve its object's consistency across suspends and resumes.

MzScheme also builds on the primitives of Concurrent ML [20], which enable a programmer to construct synchronization abstractions that have the same first-class status as built-in abstractions. By starting with Concurrent ML's primitives, we ensure that our model covers a large class of abstractions. In general, MzScheme can express any abstraction that is expressible in Concurrent ML, and we believe that any such abstraction can be made kill-safe with small adjustments.

Section 2 describes our model of task management and kill-safe abstractions. Section 3 presents MzScheme's task-control mechanisms, and Section 4 sketches the implementation of a kill-safe queue. Section 5 reviews MzScheme's embedding of Concurrent ML primitives, mainly for readers who are not familiar with Concurrent ML. Section 6 presents a complete and realistic queue im-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

plementation using the Concurrent ML primitives. This full implementation motivates a remaining detail of our design that is covered in Section 7. Section 8 compares some of our design choices to some alternatives. Section 9 summarizes related work.

2 Termination and Cooperation

Conventional OSes govern a realm of intense distrust. Each task trusts only the kernel, and the kernel trusts no one. To maintain order, the kernel strictly isolates tasks, as illustrated in Figure 1. Each task occasionally communicates across the "red line" (drawn as a thick gray line in the figure) to the kernel task, but a tasks do not communicate directly to other tasks or cross into another task's space (as indicated in the figure by thick black lines). As a result, a haywire task at worst corrupts its own space (imagine a task in the figure growing uncontrollably to fill its own space), and the task is therefore terminated easily. The cost of easy termination is a restriction on communication. If two tasks need to communicate, they must use the primitives provided by the kernel (depicted in the figure by a telecast from the left task to the right task via the kernel task).

In contrast, the run-time kernel of a safe programming language presides over a realm that is free from bit-bashing vandals. In the same way that a task relies on abstract datatypes for internal protection, the task can rely on abstract datatypes for protection from other tasks. The kernel therefore allows task boundaries to fade away, as illustrated in Figure 2. Compared to a conventional OS, tasks are free to set up new communication abstractions that better match their needs (as depicted in the figure by a close-range telecast in a task's space). However, a haywire task can now bring down the entire system, not by mangling data structures, but by consuming too many resources (again, imagine a task growing uncontrollably, but this time it fills the entire system, squeezing out other tasks).

2.1 The Best of Both Worlds

The architectures in Figure 1 and 2 demonstrate a trade-off between ease of communication versus ease of termination, but they are merely the extremes. A variation of the OS-style architecture can improve communication by enriching the kernel's set of primitives, possibly exploiting language safety in the design of the new primitives. Indeed, many language-based systems take this approach, including Alta [3], SPIN [4], J-Kernel [10], Nemesis [14], KaffeOS [2], and Luna [11]. In each case, however, the system offers a fixed set of primitives to applications. Inevitably, some set of tasks would benefit from a new communication abstraction.

A hybrid architecture can avoid the fixed set of primitives, and its implementation may consist simply of running one system inside another. By running, say, KaffeOS under Unix, KaffeOS programs can take advantage of KaffeOS's primitives, while Unix-level tasks continue to communicate with Unix primitives. This layering of kernels, illustrated in Figure 3, effectively allows a programmer to extend the kernel's set of communication primitives by implementing a new kernel. Depending on the base, implementing a new kernel may be difficult, but our previous work on MzScheme/MrEd [8] shows how a language can help. Indeed, the DrScheme programming environment [6] depends on such support from MzScheme, since DrScheme acts as a kernel to the programs that it executes for debugging.

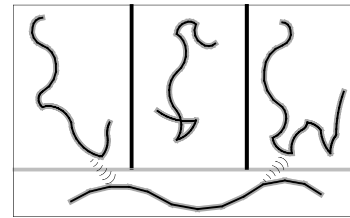

Figure 1. Conventional OS (e.g., Unix)

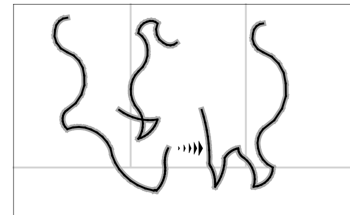

Figure 2. Safe language run-time (e.g., JVM)

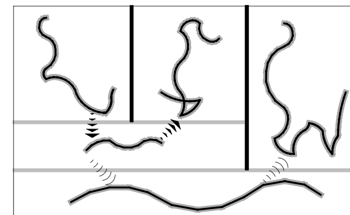

Figure 3. Layers (e.g., KaffeOS on Unix)

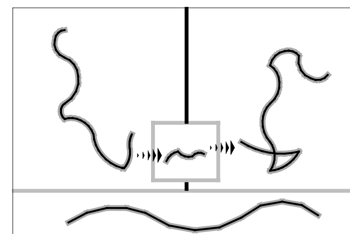

Figure 4. Kill-safe abstractions (MzScheme)

2.2 Remaining Problem: Discovering Cooperation

The drawback of the kernel-layers approach is that the layers must be defined in advance, and tasks must be associated with layers in advance. In practice, two already-running tasks may discover each other and wish to communicate through an abstraction that is not provided by their shared kernel.

For example, servlets for two different sessions might discover each other and wish to share a cache whose implementation is specific to the servlets. Although the servlet tasks can trust the cache implementation, they cannot trust each other to survive, because the server might terminate one or the other at any time.

This configuration is illustrated in Figure 4. In the figure, the left and right tasks do not trust each other, but they trust the small gray

box in the middle (in the same way that they trust the kernel). The kernel, on the other hand, does not trust the gray box—just as in the layered model of Figure 3, the outer kernel (e.g., Unix) does not trust the inner kernel (e.g,. KaffeOS).

The communication pattern suggested by Figure 4 cannot be implemented in existing systems. The problem is that the gray-box task must either be a sub-task of the left or right task (and therefore subject to termination along with its parent task), or it must be at the same level as the two tasks, and therefore no more trustworthy than either task. This problem cannot be solved by allowing the two tasks to enter the gray box and execute atomically, since the two tasks could then collude to starve the rest of the system. For the same reason, the termination of a task cannot be delayed until it leaves the gray box.

2.3 Solution in MzScheme

Our solution requires either the left or right task to create the gray-box task initially as a sub-task. Later, when the other task gains access to the gray box, it "promotes" the box as its own sub-task. As a result, the gray-box task becomes more resistant to termination than either the left or right task alone.

For example, if the left task is terminated, the box task is merely suspended. When the right task later accesses the box, it resumes the box task, and safely continues. If both the left and right tasks are terminated, the box task becomes both suspended and inaccessible, and therefore effectively terminated. Thus, the system as a whole can protect itself against malicious (or buggy) collaborations by terminating both collaborators. Meanwhile, communication channels that are provided by the box protect the left and right tasks from each other, much as kernel-supplied channels protect tasks from each other.

This solution builds on our earlier work for layered kernels in MzScheme. It also relies on primitives in the base kernel for constructing boxes; the primitives of Concurrent ML serve that purpose. Finally, our solution relies on new primitives for modifying the task hierarchy without threatening bystander tasks.

2.4 Application

The new MzScheme solves a problem within the implementation of DrScheme's help system. Help pages are written in HTML, and the help system works by running the PLT web server [9] plus a browser that is connected to the server. Although the server and browser could execute as different OS tasks and communicate through TCP sockets (as web servers and browsers normally do), production OSes make this architecture surprisingly fragile. Therefore, DrScheme implements its own browser, and it runs a web server and a browser directly in its own MzScheme virtual machine. The two parties communicate through a socket-like abstraction.

The core of the socket-like abstraction abstraction is an asynchronous buffered queue. The MzScheme kernel, however, provides no such abstraction, and adding an intermediate kernel layer for the web browser and server would be prohibitively difficult. Furthermore, both the server and browser take advantage of termination for internal tasks (e.g., to cancel a browser click), and those tasks are involved in communication. Such terminations then wreak havoc with the queue implementation.

In the new MzScheme, small adjustments to the queue implementation make it kill-safe as well as thread-safe. The help system now works reliably with no additional changes to DrScheme, the server, or the browser. At the same time, the kill-safe abstraction does not (and cannot) compromise task control. In particular, when testing DrScheme within DrScheme, we can terminate the inner DrScheme, and it reliably terminates the associated help system, including any queue-manager threads.

Kill-safe buffered queues are merely the tip of the abstraction iceberg, but we use this example in the following sections to illustrate essential techniques for kill-safe abstractions. See Reppy's book [20] for many other example abstractions that can be made kill-safe using our technique.

3 Task Control in MzScheme

MzScheme's support for tasks encompasses threads of execution, task-specific state, per-task resource control, per-task GUI modalities, and more. Instead of supplying a monolithic "process" construct, however, MzScheme supports the many different facets of a process through many specific constructs [8]. With respect to kill-safe abstractions, the only relevant facets are threads of execution, resource control as it relates to thread termination, and thread-specific state as it relates to determining the resource controller.

3.1 Threads

The `spawn` procedure takes a function of no arguments and calls the function in a new thread of execution. The thread terminates when the function returns. Meanwhile, `spawn` returns a thread descriptor to its caller.

```
(define t1 (spawn (lambda () (printf "Hello"))))
(define t2 (spawn (lambda () (printf "Nihao"))))
;; prints "Hello" and "Nihao", possibly interleaved
```

3.2 Resource Control

A *custodian* is a resource controller in MzScheme. Whenever a thread is created, a network socket is opened, a GUI windows is created, or any other primitive resource is allocated, it is placed under the control of the *current custodian*. A thread can create and install a new custodian, but the newly created custodian is a *sub-custodian* that is controlled by the current custodian.

The only operation on a custodian is `custodian-shutdown-all`, which suspends all threads, closes all sockets, destroys all GUI windows, etc. that are controlled by the custodian. The shut-down command also propagates to any controlled sub-custodians. After a custodian is shut down, it drops references to primitive resources such as sockets, GUI windows, etc., and the memory for such objects can be reclaimed by the garbage collector (when they become otherwise unreachable).[1] The custodian itself remains available to control new objects. It also retains a weak reference to each suspend thread, in case the thread is later resumed. If a thread is resumed, a future `custodian-shutdown-all` once again suspends the thread.

The `make-custodian` procedure creates a new custodian. The `parameterize` form with `current-custodian` sets the current custodian during the evaluation of an expression. In particular, `parameterize` can be used to install a custodian while creating a new thread, and the new thread is then controlled by the installed custodian.

[1] In practice, shutting down a resource removes internal references, which frees most of the memory associated with the resource.

```
(define cust (make-custodian))
(parameterize ([current-custodian cust])
  (spawn lots-of-work))
(custodian-shutdown-all cust) ; stops lots-of-work
```

When a thread is created, it inherits the current custodian from its creating thread. Thus, assuming that *lots-of-work* is not closed over the original custodian, (custodian-shutdown-all *cust*) reliably terminates the task that executes *lots-of-work*—no matter how many threads that it spawns, sockets that it opens, GUI windows that it creates, or sub-custodians that it generates.

The current custodian for a particular thread is not necessarily the same as the thread's controller. The thread's controlling custodian is determined when the thread is spawned, but the current custodian (for controlling newly allocated resources) can be changed by the thread at any time through parameterize.

3.3 Thread Control

The custodian-shutdown-all procedure provides one way to suspend a thread. MzScheme also provides thread-suspend for suspending a specific thread.

Suspending a thread stops a computation uncooperatively, but such heavy-handed termination is not always necessary. For cooperative termination, MzScheme provides the thread-break function, which takes a thread descriptor and sends the thread a break signal. This signal is analogous to a Unix process signal, but the break signal is manifest in the target thread as a asynchronous exception, much as in Concurrent Haskell [16].

```
(define cust (make-custodian))
(define t (parameterize ([current-custodian cust])
            (spawn lots-of-work)))
(thread-break t) ; possibly interrupts lots-of-work
```

Asynchronous breaks add only a minor challenge to the implementation of kill-safe abstractions, and we address this challenge in Section 7. For completeness, we offer a few additional details concerning breaks in MzScheme, but the disinterested reader may safely skip to the next subsection.

Break exceptions can be enabled and disabled by using parameterize with break-enabled. Breaks are always implicitly disabled during exception handling and during the evaluation of dynamic-wind pre- and post-actions. If a break signal is delivered to a thread that has disabled breaks, the signal is delayed until breaks are re-enabled in the thread. A break signal has no effect if the target thread already has a delayed break.

Unlike Concurrent Haskell's takeMVar, a blocking sync does not implicitly enable breaks while it blocks. Such implicit enabling is unnecessary to encourage interruption when thread-suspend is available is a back-up, and in our experience, implicit enabling of breaks makes a system fragile. A separate sync/enable-breaks function enables breaks such that either a break execution is raised or an event is selected, but not both. (Merely wrapping a sync with a parameterize to enable breaks does not achieve this exclusive-or behavior, because the break may occur after an event is selected but before breaks are re-disabled.)

3.4 Thread Resumption

Given a single thread argument, MzScheme's thread-resume function resumes the thread if it is suspended.

```
(define cust (make-custodian))
(define t (parameterize ([current-custodian cust])
            (spawn lots-of-work)))
(custodian-shutdown-all cust) ; stops lots-of-work
(resume-thread t) ; resumes lots-of-work
```

MzScheme provides no operation for resuming all controlled objects in a custodian. Indeed, for most kinds of controlled objects, a shut-down action is terminal. Thus, a thread such as *t* above may resume and discover that its resources have been closed. For kill-safe abstractions, the implication is that an abstraction cannot rely on controlled resources other than threads.

The thread-resume function accepts an optional second argument. The second argument, which also must be a thread descriptor, plays two roles:

- It yokes the first thread to second for resumes, in that the first thread is resumed whenever the second thread is resumed.
- It potentially "strengthens" the first thread by changing the thread's controlling custodian to a more senior custodian. The more senior custodian is determined by comparing the two thread's controlling custodians, and finding the least-senior common super-custodian.

The overall effect of (thread-resume *t t2*) is to ensure that *t* is suspended only when *t2* is also suspended—assuming that *t* is suspended only indirectly via custodian-shutdown-all. This effect holds because an indirect suspension of *t* will also suspend *t2*, and if *t2* is resumed later, so is *t*.

The two-argument thread-resume allows two threads *t1* and *t2* share an object that embeds a thread *t*. In that case, (thread-resume *t t1*) plus (thread-resume *t t2*) makes the embedded thread *t* act as though it has no controlling custodian, at least as far as *t1* and *t2* can tell. This combination was the key to implementing kill-safe versions of the queue abstractions.

4 Sketch for a Kill-Safe Queue

The primitive synchronization abstraction in MzScheme is a CSP-style synchronous channel [12], which allows two tasks to rendezvous and exchange a single value. This built-in abstraction is kill-safe, in that the termination of a task on one end of the channel does not endanger the task on the other end of the channel—though, obviously, no further communication will take place.

In this section, we consider the implementation of a kill-safe queue (a.k.a. asynchronous buffered channel). Values sent into the queue are parceled out one-by-one to receivers. A send to a queue never blocks, except to synchronize access to the internal list of queued items. A receive blocks only when the queue is empty, or to synchronize internal access.

```
(define q (queue))
(queue-send q "Hello")
(queue-send q "Bye")
(queue-recv q) ; ⇒ "Hello"
(queue-recv q) ; ⇒ "Bye"
```

Figure 5 sketches an implementation of queues. Each queue consists of a channel *in-ch* for putting items into the queue, a channel *out-ch* for getting items out of the queue, and a manager thread running *serve* to pipe items from *in-ch* to *out-ch*.

The implementation is not yet kill-safe. For example, suppose that

```
;; queue : → α-queue
;; queue-send : α-queue α → void
;; queue-recv : α-queue → α

;; Declare an opaque q record; this declaration introduces the
;; private functions make-q, q-in-ch, and q-out-ch
(define-struct q (in-ch out-ch))

(define (queue)
  (define in-ch (channel)) ; to accept sends into queue
  (define out-ch (channel)) ; to supply recvs from queue
  ;; A manager thread loops with serve
  (define (serve items)
     ;; Handle sends and recvs
     ....
     ;; Loop with new queue items:
     (serve new-items))
  ;; Create the manager thread
  (spawn (lambda () (serve (list))))
  ;; Return a queue as an opaque q record
  (make-q in-ch out-ch))

(define (queue-send q v)
  ;; Send v to (q-in-ch q)
  ....)

(define (queue-recv q)
  ;; Receive from (q-out-ch q)
  ....)
```

Figure 5. Implementation sketch for a queue

a thread `t1` creates `q` by calling `(queue)`. Suppose further that `t1` is controlled by custodian `c1`, and that `q` is made available to a thread `t2` controlled by custodian `c2`:

`;; t1, controlled by c1`
`(define q (queue))`
`(send-to-other q)`
`;; suspend threads of c1`
	`;; t2, controlled by c2`
	
	`(define q (get-from-other))`
	
	`;; stuck since q suspended by c1`
	`(queue-send q 10)`

Since `t1` creates `q`, the queue's internal thread `t` is controlled by `c1`. Suspending all threads of `c1` suspends both `t1` and `t`. As a result, the *send* in `t2` gets stuck—and a *send* into a buffered queue should never get stuck.

We might attempt to fix the problem with a resume of the queue's thread before each queue operation. A simple resume is not enough, however; between the time that `t2` resumes `t` and the time that it performs its action on the queue, another thread might re-suspend all threads of `c1`, thus re-suspending `t`.

The solution is `(thread-resume t t2)`, which not only resumes the queue thread, but also changes the controlling custodian of `t` to a super-custodian of both `c1` and `c2`. Afterward, a mere suspension of `c1`'s threads does not suspend `t`.

Since `t` is not accessible outside the queue implementation, it can be suspended only through its custodian. Furthermore, since `t`'s custodian is a common ancestor of `t1` and `t2`, both of those threads will be suspended as well, and it does not matter that `t` is suspended. Indeed, `t` *must* be suspended, because the suspension may be intended to terminate the set of tasks `t1` and `t2`.

If the suspension is not intended to terminate the task, and if `t2` is later resumed, then `t` is also resumed, due to the chaining installed by `(thread-resume t t2)`. More generally, by guarding each queue operation with `(thread-resume t (current-thread))` we ensure that `t` runs whenever a queue-using thread runs. Figure 6 shows revisions of the queue implementation with these guards, which make it kill-safe.

```
(define-struct q (in-ch out-ch mgr-t))

(define (queue)
  ....
  (define mgr-t (spawn (lambda () (serve (list)))))
  ;; The q record now refers to the manager thread
  (make-q in-ch out-ch mgr-t))

(define (queue-send q v)
  (resume-thread (q-mgr-t q) (current-thread))
  ;; Send v to (q-in-ch q)
  ....)

(define (queue-recv q)
  (resume-thread (q-mgr-t q) (current-thread))
  ;; Receive from (q-out-ch q)
  ....)
```

Figure 6. A kill-safe queue, revises Figure 5

This example demonstrates both how kill-safe abstractions are possible, and how abstractions can be made kill-safe with relative ease. Nevertheless, it does not demonstrate the full power of MzScheme's primitives for defining kill-safe abstractions. For such a demonstration, we must introduce MzScheme's embedding of the Concurrent ML primitives, so that we can build more flexible abstractions.

5 Review of Concurrent ML

This section provides a brief tutorial on MzScheme's embedding[2] of the Concurrent ML [20] primitives that are listed in Figure 7. The tutorial is intended mainly for readers who are unfamiliar with Concurrent ML.

The primitives support synchronization among tasks via first-class events. A few kinds of events are built in, such as events for sending or receiving values through a channel. More importantly, the primitives enable the construction of entirely new kinds of events that have the same first-class status as the built-in events.

• `sync` : α-`event` → α

The `sync` procedure takes an *event* and blocks until the event is ready to supply a value. Some primitives provide a source of events. For example, `thread-done-evt` takes a thread descriptor and returns an event that is ready (with a void value) when the thread has terminated.

```
;; thread-done-evt : thread → void-event

(define t1 (spawn (lambda () (printf "Hello"))))
(define t2 (spawn (lambda () (printf "Nihao"))))
(sync (thread-done-evt t1)) ; waits until t1 is done
(sync (thread-done-evt t2)) ; waits until t2 is done
(printf "Bye")
;; prints "Hello" and "Nihao" interleaved, then "Bye"
```

[2] The functions are defined in the `(lib "cml.ss")` module.

```
sync             : α-event → α
channel          : → α-channel

channel-recv-evt : α-channel → α-event
channel-send-evt : α-channel α → void-event
choice-evt       : α-event ... α-event → α-event

wrap-evt         : α-event (α → β) → β-event
guard-evt        : (→ α-event) → α-event
nack-guard-evt   : (void-event → α-event) → α-event
```

Figure 7. Concurrent ML primitives in MzScheme

• `channel` : → α-channel
`channel-recv-evt` : α-channel → α-event
`channel-send-evt` : α-channel α → void-event

The `channel` procedure takes no arguments and returns a channel descriptor. A channel's only purpose is to generate events; the `channel-recv-evt` and `channel-send-evt` procedures create events for receiving values from the channel and sending values into the channel, respectively. The result of a receive event is a value sent through the channel, and the result of a send event is void. A send event is created with a specific value to put into the channel, and the event is ready only when a receive event can accept the value simultaneously. Similarly, a receive event is ready only when a send event can provide a value simultaneously.

```
(define c (channel))
(spawn (lambda () (sync (channel-send-evt c "Hello"))))
(sync (channel-recv-evt c)) ; ⇒ "Hello"
```

Multiple threads can attempt to send or receive through a particular channel concurrently. In that case, the system selects threads arbitrarily but fairly to form a send–receive pair.

```
(define c (channel))
(spawn (lambda () (sync (channel-send-evt c "Hello"))))
(spawn (lambda () (sync (channel-send-evt c "Nihao"))))
(sync (channel-recv-evt c)) ; ⇒ "Hello" or "Nihao"
(sync (channel-recv-evt c)) ; ⇒ "Nihao" or "Hello"
```

• `choice-evt` : α-event ... α-event → α-event

The `choice-evt` procedure takes any number of events and composes them into a single event. The composite event is ready when one of the composed events is ready. If multiple composed events are ready, one is chosen arbitrarily but fairly, and the value produced by the composite event is the value produced by the chosen event.

```
(define c1 (channel))
(define c2 (channel))
(spawn (lambda () (sync (channel-send-evt c1 "Hello"))))
(spawn (lambda () (sync (channel-send-evt c2 "Nihao"))))
(define cc (choice-evt (channel-recv-evt c1)
                       (channel-recv-evt c2)))
(sync cc) ; ⇒ "Hello" or "Nihao"
(sync cc) ; ⇒ "Nihao" or "Hello"
```

In the above example, even if both sending threads are ready when the main thread first calls `sync`, only one receive event in *cc* is chosen, and so it is matched with only one sending thread. The other sending thread remains blocked until the second `(sync cc)`.

• `wrap-evt` : α-event (α → β) → β-event

The `wrap-evt` function takes an event and a transformer procedure of one argument, and it produces a new event. The new event is ready when the given event is ready, and its value is the result of the transformer procedure applied to the original event's value.

```
(define c1 (channel))
(define c2 (channel))
(spawn (lambda () (sync (channel-send-evt c1 "Hello"))))
(spawn (lambda () (sync (channel-send-evt c2 "Nihao"))))
(sync (choice-evt
       (wrap-evt (channel-recv-evt c1)
                 (lambda (x) (list x "from 1")))
       (wrap-evt (channel-recv-evt c2)
                 (lambda (x) (list x "from 2")))))
;; ⇒ (list "Hello" "from 1") or (list "Nihao" "from 2")
```

• `guard-evt` : (→ α-event) → α-event

The `guard-evt` function is the dual of `wrap-evt`. Whereas `wrap-evt` supports post-processing on the result of an event, `guard-evt` supports pre-processing to generate an event for synchronization. For example, assume that `current-time` produces the current time, and that `time-evt` produces an event that is ready at a given absolute time. Then, `guard-evt` can be used to construct a timeout event.

```
;; current-time : → num
;; time-evt : num → event

(define one-sec-timeout
  (guard-evt (lambda ()
               (time-evt (+ 1 (current-time))))))
(sync one-sec-timeout) ; ⇒ void, one second later
(sync one-sec-timeout) ; ⇒ void, another second later
```

The result from `guard-evt` function might be best described as "event generator" instead of an immediate event, but this generator can be used anywhere than an event can be used. Event generation is important for *one-sec-timeout*, which must construct an alarm time based on the time that *one-sec-timeout* is used, not when *one-sec-timeout* is created.

• `nack-guard-evt` : (void-event → α-event) → α-event

The `nack-guard-evt` function generalizes `guard-evt`. For `nack-guard-evt`, the given guard procedure must accept a single argument. The argument is a "Negative ACKnowledgement" event that becomes ready if the guard-generated event is not chosen by `sync`.

```
(define c (channel))
(spawn (lambda () (sync (channel-send-evt c "Hello"))))
(sync (choice-evt
       (nack-guard-evt
         (lambda (nack)
           ;; Start a thread to watch NACK
           (spawn (lambda ()
                    (sync nack) (printf "not first")))
           ;; This event is never ready
           (channel-recv-evt (channel))))
       ;; This event is ready eventually
       (channel-recv-evt c))) ; ⇒ "Hello"
;; Meanwhile, "not first" is printed
```

Each time `sync` is applied to a NACK-guarded event, the guard procedure is called with a newly generated NACK event. Thus, a NACK event becomes ready only when a specific guard-generated event is not chosen in a specific `sync` call.

We defer a complete definition of "not chosen" to Section 7, following a motivating example.

6 Queue: Complete and Improved

Having reviewed the Concurrent ML primitives, we are ready to complete the implementation sketch of queues from Section 4. First, however, we refine the queue abstraction to better match the programming idioms of Concurrent ML. This refinement helps demonstrate that our strategy for kill-safety applies to other Concurrent ML abstractions. After showing the implementation of the first improved queue abstraction, we improve the abstraction one step further to demonstrate an additional key idiom.

6.1 Queue Actions as Events

Our original queue sketch provided *queue-send* and *queue-recv* functions that block until the corresponding action completes. We should instead provide *queue-send-evt* and *queue-recv-evt* functions that generate events. With events, a programmer can incorporate queues in future synchronization abstractions, which may need to select among multiple blocking actions.

```
(define q (queue))
(sync (queue-send-evt q "Hello"))
(sync (queue-send-evt q "Bye"))
(sync (queue-recv-evt q)) ; ⇒ "Hello"
(sync (queue-recv-evt q)) ; ⇒ "Bye"
```

Figure 8 shows the complete implementation of improved, kill-safe queues:

- The *queue* function creates a thread to manage the internal list of values. Access to the internal list is thus implicitly single-threaded, avoiding race conditions.
- When the queue is neither empty nor full, the queue-managing thread uses `choice-evt` to select among the send and receive actions. If both actions become enabled at once, one or the other is chosen atomically and fairly.
- The `wrap-evt` function meshes with `choice-evt` to implement a dispatch for the ready action within the manager thread.
- The *queue-send-evt* function guards its result event with a use of `thread-resume`. The guard ensures that the manager thread runs to service the send. The *queue-recv-evt* similarly guards its result.
- If a queue becomes unreachable, its manager thread is garbage collected. More generally, when a thread becomes permanently blocked because all objects that can unblock it become unreachable, the thread itself becomes unreachable, and its resources can be reclaimed by the garbage collector.

To a consumer of the abstraction, the values produced by *queue*, *queue-recv-evt*, and *queue-send-evt* have the same first-class status as values produced by `channel`, `channel-recv-evt`, and `channel-send-evt`. For example, queue *send* and *recv* events can be multiplexed with other events (using `choice-evt`) in building additional abstractions.

6.2 Selective Dequeue

In DrScheme's help system, a *queue* is used in place of a socket that listens for connections. The *queue* abstraction might also be useful for handling messages to GUI objects, such as a mouse-click messages and refresh messages. A GUI message queue, however, must support a selective dequeuing. For example, a task might wish to handle only refresh messages posted to the queue, leaving mouse-click messages intact.

Unfortunately, selective dequeue cannot be implemented by dequeuing a message, applying a predicate, then re-posting the message if the predicate fails; re-posting the unwanted message changes its order in the queue with respect to other messages.

To support selective dequeue, we must modify the server so that it accepts dequeue requests with a corresponding predicate, and then satisfies a request only when an item in the queue matches the predicate. On the client side, the selective receive event must be guarded so that it sends a request to the server, then accepts a result through a newly created channel. The new channel ties together the request and the result, so that a result is sent to the correct receiver.

Figure 9 shows a revision of the queue implementation to support selective dequeue. The manager thread still accepts *send*s through *in-ch*, but it no longer supplies queued items to a fixed

```
;; queue : → α-queue
;; queue-send-evt : α-queue α → void-event
;; queue-recv-evt : α-queue → α-event

(define-struct q (in-ch out-ch mgr-t))

(define (queue)
  (define in-ch (channel)) ; to accept sends into queue
  (define out-ch (channel)) ; to supply recvs from queue
  ;; A manager thread loops with serve
  (define (serve items)
    (if (null? items)
        ;; Nothing to supply a recv until we accept a send
        (serve (list (sync (channel-recv-evt in-ch))))
        ;; Accept a send or supply a recv, whichever is ready
        (sync (choice-evt
               (wrap-evt
                (channel-recv-evt in-ch)
                (lambda (v)
                  ;; Accepted a send; enqueue it
                  (serve (append items (list v)))))
               (wrap-evt
                (channel-send-evt out-ch (car items))
                (lambda (void)
                  ;; Supplied a recv; dequeue it
                  (serve (cdr items))))))))
  ;; Create the manager thread
  (define mgr-t (spawn (lambda () (serve (list)))))
  ;; Return a queue as an opaque q record
  (make-q in-ch out-ch mgr-t))

(define (queue-send-evt q v)
  (guard-evt
   (lambda ()
     ;; Make sure the manager thread runs
     (thread-resume (q-mgr-t q) (current-thread))
     ;; Channel send, as before
     (channel-send-evt (q-in-ch q) v))))

(define (queue-recv-evt q v)
  (guard-evt
   (lambda ()
     ;; Make sure the manager thread runs
     (thread-resume (q-mgr-t q) (current-thread))
     ;; Channel receive, as before
     (channel-recv-evt (q-out-ch q)))))
```

Figure 8. Implementation of a kill-safe queue

```
;; msg-queue : → α-msg-queue
;; msg-queue-send-evt : α-msg-queue α → void-event
;; msg-queue-recv-evt : α-msg-queue (α → bool) → α-event

(define-struct q (in-ch req-ch mgr-t))

(define (msg-queue)
  (define in-ch (channel))
  (define req-ch (channel))
  (define never-evt (channel-recv-evt (channel)))
  (define (serve items reqs)
    (sync (apply
           choice-evt
           ;; Maybe accept a send
           (wrap-evt
            (channel-recv-evt in-ch)
            (lambda (v)
              ;; Accepted a send; enqueue it
              (serve (append items (list v)) reqs)))
           ;; Maybe accept a recv request
           (wrap-evt
            (channel-recv-evt req-ch)
            (lambda (req)
              ;; Accepted a recv request; add it
              (serve items (cons req reqs))))
           ;; Maybe service a recv request in reqs
           (map (make-service-event items reqs)
                reqs))))
  (define (make-service-event items reqs)
    (lambda (req)
      (define pred (car req))
      (define out-ch (cadr req))
      ;; Search queue items using pred
      (find-first-item pred items
       (lambda (item)
         ;; Found an item; try to service req
         (wrap-evt
          (channel-send-evt out-ch item)
          (lambda (void)
            ;; Serviced, so remove item and request
            (serve (remove item items)
                   (remove req reqs)))))
       (lambda ()
         ;; No matching item to service req
         never-evt))))
  (define mgr-t
    (spawn (lambda () (serve (list) (list)))))
  (make-q in-ch req-ch mgr-t))

(define (msg-queue-send-evt q v)
  ;; Same as queue-send-evt in Figure 8
  ....)

(define (msg-queue-recv-evt q pred)
  (guard-evt
   (lambda ()
     (define out-ch (channel))
     ;; Make sure the manager thread runs
     (thread-resume (q-mgr-t q) (current-thread))
     ;; Send the server a request for an item matching
     ;;  pred, and request reply to out-ch
     (sync (channel-send-evt (q-req-ch q)
                             (list pred out-ch)))
     ;; Result arrives on out-ch
     (channel-recv-evt out-ch))))
```

Figure 9. Queue with selective dequeue, first attempt

```
(define (msg-queue)
  ....
  (define (serve queue reqs)
    ....
           ;; Use make-service/abandon-event
           ;; instead of make-service-event
           (map (make-service/abandon-event queue reqs)
                reqs))))
  ....
  (define (make-service/abandon-event queue reqs)
    (lambda (req)
      (choice-evt
       ;; Service event, as before
       ((make-service-event queue reqs) req)
       ;; Add event to detect that the receiver gives up
       (wrap-evt (caddr req) ; gave-up-evt
                 (lambda (void)
                   ;; Receiver gave up; remove request
                   (serve queue (remove req reqs)))))))
....

(define (msg-queue-recv-evt q pred)
  (nack-guard-evt
   (lambda (gave-up-evt)
     (define out-ch (channel))
     ;; As before, but also tell the server that gave-up-evt
     ;; will become ready if we give up
     (thread-resume (q-mgr-t q) (current-thread))
     (sync (channel-send-evt (q-req-ch q)
                             (list pred out-ch
                                   gave-up-evt)))
     ;; Result arrives on out-ch
     (channel-recv-evt out-ch))))
```

Figure 10. Revision to Figure 9

out-ch channel. Instead, the manager thread accepts receive requests through *req-ch*, and it keeps a list of the requests. While the manager waits for sends and additional receive requests, it also services requests for which a matching item is available.

The initial implementation of selective dequeue, however, contains a space leak. The following example illustrates the problem:

```
(define q (msg-queue))
(sync (msg-queue-send-evt q 1))
(sync (msg-queue-send-evt q 2))
(sync (choice-evt
       (msg-queue-recv-evt q odd?)
       (msg-queue-recv-evt q even?)))
....
```

The `sync` call sends two requests to the server. One is serviced, and the program continues. Meanwhile, a leftover request remains with the server. The request will never be successfully serviced, because no `sync` waits on the associated *out-ch*. Still, the request is stuck in the internal *reqs* list, and leftover requests can pile up over time, degrading performance and wasting resources. A similar problem occurs if the thread making a request is terminated.

To avoid this problem, the server needs to know when `sync` has abandoned a receive request. Figure 10 shows how `nack-guard-evt` provides this information. The *msg-queue-recv-evt* function now sends the manager a “gave up” event in addition to a result channel. The manager thread uses the “gave up” event to keep the request list clean.

The `msg-queue` example illustrates a particular Concurrent ML idiom: a client–server protocol where the client sends a request to the server, but may withdraw the request before it can be satisfied. Withdrawal reliably prevents acceptance and vice-versa, due to the rendezvous associated with a channel transfer (i.e., the sender and receiver must simultaneously agree to the transfer of a result).

The request idiom poses an extra challenge for kill-safety. A client can be terminated at any point in the request cycle, so we must define "not chosen" for `nack-guard-evt` so that it handles this possibility. The next section completes our explanation of MzScheme's primitives with this definition of "not chosen".

7 Termination and NACKs

Recall that the event provided to a guard procedure by `nack-guard-evt` becomes ready if the guard-generated event is not chosen. MzScheme extends the Concurrent ML definition of "not chosen" so that it includes all of the following cases, which cover all of the ways that a thread can abandon an event:

- The `sync` call chose an event other than the one returned by the guard.
- Control escaped from the `sync` call through an exception or continuation jump. The exception or jump may have been triggered through a break signal, by another guard involved in the same `sync`, or even by the guard procedure that received the NACK event. (Continuation jumps back into a guard are always blocked, so multiple escapes are not possible.)
- The `sync`ing thread terminated (i.e., it is suspended and unreachable).

MzScheme's `nack-guard-evt` corresponds to Concurrent ML's `withNack`. An earlier version [19] of Concurrent ML offered `wrapAbort`, instead, and a later presentation [20] explains how `withNack` can be implemented with `wrapAbort`. Our definition of "not chosen" does not allow such an implementation, and thus strengthens the argument that `withNack` is the correct primitive.

8 Design Considerations

Before arriving at MzScheme's current primitives for kill-safe abstractions, we explored two main alternatives:

- **Restricted atomic sections:** As mentioned in Section 2.2, the kernel cannot allow a task to execute arbitrary code atomically, otherwise it might starve the rest of the system. The kernel might, however, allow a task to execute atomically for a short period of time, or to execute code that provably terminates in a short time.

  We abandoned this approach, because we could not find a way to define "short time" that made much sense to the programmer. Dynamic measurements in terms of clock ticks or program operations were too sensitive to small program changes, and static methods, based on limiting the code to certain primitive operations, proved insufficiently expressive.
- **Transactions with rollbacks and commit points:** Although a transaction-oriented approach looked promising, and although Rudys and Wallach have made progress in this direction [22], synchronous channels encode directly the kind of transactions that seem most useful for our purposes. We therefore abandoned this direction and embraced the Concurrent ML primitives as our base.

Two aspects of the current design merit further review. The first concerns the usefulness of immediate termination independent of reachability. The second relates to the transitivity of thread operations.

8.1 Terminated versus Suspended+Unreachable

The `spawn` function described in this paper has the primitive name `thread/suspend-to-kill` in MzScheme. MzScheme also provides a `thread` primitive, which creates a thread that is immediately terminated when it is shut down by a custodian. Immediate termination acts as hint to the run-time system, so that it can reclaim the thread's resources.

In our implementation, the hint appears to be worthwhile. Indeed, we have used `thread/suspend-to-kill` so far only to implement kill-safe abstractions. Nevertheless, the hint's usefulness most likely reflects weakness in our memory manager. In principle, immediate termination can be implemented in terms of `spawn` by introducing an indirection to the thread descriptor that is severed when the thread is shut down (so that the underlying thread descriptor become inaccessible).

8.2 Transitive Resume and Non-Transitive Promotion

The `thread-resume` function can yoke one thread to another for resumes. If a chain of threads become yoked, a resume for the initial thread propagates through the entire chain. This transitivity does not apply to the promotion of a thread to a more superior custodian.

The lack of transitivity for promotion could create a problem in the following situation. Suppose that a manager thread within an object creates several helper threads when it starts. As the object is used by different tasks, the manager thread is promoted to the control of a superior custodian, but its helpers are not automatically promoted, and therefore they may become suspended when the manager thread keeps running.

This pattern has not appeared in any implementation so far. If it did, the problem could be solved by changing the manager thread's role so that its only job is to resume and promote helper threads; the manager's original work would become the job of a new helper. Without concrete experience, we cannot judge whether this conversion is reasonable, or whether it is so difficult that our primitives should be refined to support transitive promotion.

9 Related Work

Much previous work addresses the interaction between termination and synchronization for specific primitives. Examples include work on monitors in Pilot [17] and remote pointers in Luna [11]. To our knowledge, no previous work addresses the problem of termination with respect to *programmer-defined* synchronization abstractions. Indeed, the problem makes sense only after programmers are given significant abstraction capability, which is why our work depends on Concurrent ML [18, 19, 20].

The idea of managing a resource through an designated thread appears in many contexts, notably in microkernels [5]. Argus's guardians [15] reflect a similar idea in the area of persistent, distributed computing. Our specific use of the thread-manager pattern is typical of Concurrent ML programs, but also reminiscent of the J-Kernel [10] approach, which creates a thread when crossing a trust boundary to defend against termination. We extend this idea by

adding a mechanism to adjust a thread's execution capability relative to other threads.

CSP-style communication [12] is especially popular for purely functional languages such as Erlang [1] and Concurrent Haskell [13]. Our handling of termination applies to a purely functional setting as well as it does for Scheme and ML. (No mutations appear in our examples.)

MzScheme does not provide a way to revoke access to an object, as in Luna [11]. It also provides no way to disable code that is associated with a task, as in Rudys and Wallach's soft termination [21]. Given a mechanism for disabling code, we conjecture that code fragments could be connected to the custodian hierarchy to prevent a shared abstraction's code from being disabled prematurely.

10 Conclusion

MzScheme provides the run-time system for the DrScheme programming environment, the PLT web server, and other applications that naturally consist of cooperating tasks. To better support such applications, we experiment with task-management constructs in MzScheme that ease cooperation without abandoning control. In the long run, we hope to build ever larger applications by composing other applications, and we hope to make this composition as simple as composing libraries.

This paper reports the latest step in our experiment, which allows a set of applications to trust a communication abstraction without trusting all clients of the abstraction, and without requiring the kernel to trust the abstraction. In effect, we have freed application programmers in MzScheme to pursue their own experiments for improving cooperation.

11 References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

[2] G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proc. IEEE Conference on Operating Systems Design and Implementation*, pages 333–346, Oct. 2000.

[3] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. In *Proceedings of the USENIX 2000 Technical Conference*, pages 197–210, San Diego, CA, June 2000.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proc. ACM Symposium on Operating Systems Principles*, pages 267–284, Dec. 1995.

[5] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G.-R. Malan, and D. Bohman. Microkernel operating system architecture and Mach. *Journal of Information Processing*, 14(4):442–453, 1991.

[6] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, Sept. 1997.

[7] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[8] M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming languages as operating systems (*or* revenge of the son of the Lisp machine). In *Proc. ACM International Conference on Functional Programming*, pages 138–147, Sept. 1999.

[9] P. Graunke, S. Krishnamurthi, S. V. D. Hoeven, and M. Felleisen. Programming the Web with high-level programming languages. In *Proc. European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[10] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proc. of USENIX Annual Technical Conference*, pages 259–270, June 1998.

[11] C. Hawblitzel and T. von Eicken. Luna: a flexible Java protection system. In *Proc. IEEE Conference on Operating Systems Design and Implementation*, Oct. 2002.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[13] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 295–308, Jan. 1996.

[14] I. M. Leslie, D. McAuley, R. J. Black, T. Roscoe, P. R. Barham, D. M. Evers, R. Fairburns, and E. A. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, Sept. 1996.

[15] B. Liskov and R. Scheifler. Guardians and actions: Linguistics support for robust, distributed systems. *ACM Transactions on Computing Systems*, 5(3):381–404, 1983.

[16] S. Marlow, S. L. P. Jones, A. Moran, and J. H. Reppy. Asynchronous exceptions in haskell. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 274–285, 2001.

[17] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, Feb. 1980.

[18] J. H. Reppy. Higher–Order Concurrency. Technical Report TR92-1852, Cornell Univ, Ithaca, NY, 1992.

[19] J. H. Reppy. Concurrent ML: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198, 1993.

[20] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[21] A. Rudys, J. Clements, and D. S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security*, 5(3):138–168, 2002.

[22] A. Rudys and D. S. Wallach. Transactional rollback for language-based systems. In *Proc. International Conference on Dependable Systems and Networks*, June 2002.

[23] Soper, P., specification lead. JSR 121: Application isolation API specification, 2003. `http://www.jcp.org/`.