Safe-for-Space Threads in Standard ML

Edoardo Biagioni Ken Cline Peter Lee
Chris Okasaki Chris Stone

Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3891

Abstract

Threads can easily be implemented using first-class continuations, but the straightforward approaches for doing so lead to space leaks, especially in a language with exceptions like Standard ML. We show how these space leaks arise, and give a new implementation for threads that is safe-for-space.

1 Introduction

The ability to provide simple implementations of lightweight, concurrent threads is often cited as one of the great attractions of first-class continuations. We show that this task is not nearly as simple as previously thought, at least if one is concerned about space safety.

Programming with threads is common in domains such as networking, operating systems, and user interfaces. Threads are not strictly necessary for such applications, but designing these systems with threads leads to an overall system structure that is much easier to understand and modify. Principles for programming with threads can be found in any undergraduate textbook on operating-system design. An excellent source for advice is Nelson's book on Modula-3 [10].

This research was sponsored in part by the Advanced Research Projects Agency ITO under the title "The Fox Project: Advanced Languages for Systems Software," DARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

```
signature COROUTINE = sig
   exception NoReadyThread

   val fork : (unit -> unit) -> unit
   val yield: unit -> unit
   val exit : unit -> 'a
end
```

Figure 1: An interface for a simple threads package.

As is well-known, threads can be implemented elegantly in a language with first-class continuations, such as Scheme [13, 7] or Standard ML extended with `callcc` [3, 11]. However, naive implementations are likely to suffer from two potential space leaks, one involving continuations and one involving exceptions. The space leak involving continuations can easily be optimized away by a good compiler, but the space leak involving exceptions cannot. In this paper, we describe the contortions necessary to implement safe-for-space threads in Standard ML [9], using first-class continuations as provided by the Standard ML of New Jersey system (SML/NJ) [1].

We begin by developing a simple threads package, and then point out and fix some bizarre behavior that can be caused by exceptions. Next, we describe the two potential space leaks, and show how they can be avoided with a clever use of `callcc`. We then sketch some of the difficulties in achieving a comparable implementation using other control operators, such as Felleisen's `control/prompt` [5], Danvy and Filinski's `shift/reset` [4], or Gunter, Rémy, and Riecke's `set/cupto` [6]. Finally, we draw some conclusions.

2 A Simple Threads Package

We begin by considering the simple threads interface shown in Figure 1. Three operations are specified in the signature `COROUTINE`: `fork`, `yield`, and `exit`. The `fork` procedure takes a function `f` as an argument and then evaluates the expression `f ()` in a newly created thread. The new process is called the "child" process, whereas the process that called `fork` is the "parent" process. There is also a notion of a "main" thread, which is the one thread that was not created by a call to `fork`. The main thread is the only thread whose return value is significant; its result is the result of the entire program.

Concurrency amongst threads is obtained by having individual threads voluntarily suspend themselves, thereby giving other threads a chance to execute. In this sense, our threads are cooperative coroutines rather than parallel or pre-emptable (time-sliced) processes.[1] A thread calls `yield` to place itself on a queue of "ready" processes and activate the next thread. The ready processes are typically executed in first-in-first-out order, although it is considered bad programming style to depend on this ordering.

The `exit` procedure terminates the current thread and activates the next ready thread. Unlike `yield`, a call to `exit` never returns. Instead, it either transfers control directly to a waiting thread or raises the `NoReadyThread` exception if there are no other threads remaining in the queue. A child thread implicitly exits if the expression it is evaluating returns.

There are some subtleties involving what should happen when the main thread returns. Should it implicitly wait for all the other threads to also finish? From the point of view of the implementor, the simplest approach is to make all the other threads silently disappear when the main thread returns. This is the approach we will describe. There are also questions about what should happen if the main thread calls `exit` rather than returning, or even if it should be allowed to do so. We shall return to this matter in Section 4.

3 A First Implementation

We use first-class continuations as provided by SML/NJ [1], with the built-in type `'a cont` and primitive operators `callcc` and `throw`.

Following the by-now standard approach, first advocated by Wand [13], we represent a thread as a continuation.

```
type thread = unit cont
```

A sleeping thread is activated by throwing to its continuation.

The queue of ready processes is then easily implemented as a queue of continuations. Using the standard structure `Queue` (an implementation of imperative queues provided by the SML/NJ library), we have the following definition of the ready queue:

[1] Extending the system for pre-emption is a straightforward exercise, given suitable primitives for interrupting programs at regular intervals. To see how this is done in SML/NJ, the interested reader can consult the source code for ML-Threads or CML, which are both part of the SML/NJ standard distribution, available from `ftp.research.att.com`. The space-safety considerations discussed in this paper are applicable to both coroutining and pre-emptable threads.

```
val readyQueue : thread Queue.queue = Queue.mkQueue ()
```

A couple of auxiliary functions turn out to be useful. The first one, called `dispatch`, activates the next thread on the ready queue. If no threads are waiting on the ready queue, the `NoReadyThread` exception is raised.

```
exception NoReadyThread

fun dispatch () =
   let val t = Queue.dequeue readyQueue
               handle Queue.Dequeue => raise NoReadyThread
   in throw t () end
```

The second auxiliary function is simply a shorthand for enqueuing a continuation on the ready queue:

```
fun enqueue t = Queue.enqueue (readyQueue, t)
```

With these helper functions in hand, we can now make simple definitions of the main thread routines. The first is `fork`:

```
fun fork f =
   callcc (fn parent =>
     (enqueue parent;
      f ();
      exit ()))
```

To start a new process, we first capture the continuation of the parent process and enqueue it. We then activate the child function `f`. The return value of `f` is irrelevant, so when and if `f` returns, we end this thread by calling `exit`. Note that there are several other ways to schedule the parent and child threads during a `fork`. For instance, the following code enqueues the child process and continues with the parent.

```
fun fork' f =
   let val child =
          callcc (fn return =>
            (callcc (fn child => throw return child);
             f ();
             exit ()))
   in enqueue child end
```

Alternatively, we could enqueue both threads and then call `dispatch` to run the next thread on the ready queue.

For `yield`, we have the following definition:

```
fun yield () =
   callcc (fn parent =>
     (enqueue parent;
      dispatch ()))
```

We first capture and enqueue the current continuation, and then call `dispatch` to run the next thread.

Finally, we have `exit`, which immediately starts the next thread.

```
fun exit () = dispatch ()
```

Note that `exit` does not capture the current continuation before calling `dispatch`. Therefore, the current thread is lost.

This implementation is essentially similar to Wand's implementation of threads in Scheme [13]. Existing implementations of threads in Standard ML, such as ML-Threads [3] and CML [11, 12], differ mainly in their treatment of exceptions, as discussed in the following section.

4 The Problem With Exceptions

Although appealingly simple, the above implementation exhibits bizarre behavior in the presence of exceptions. The key question is: when a thread raises an exception that is not caught within that thread, where does that exception go? This is not always obvious when `callcc` is involved, but keep in mind that the exception handler stack is part of the context that is captured by `callcc` and restored by `throw`.

Armed with this knowledge, we can easily see that `yield` has no effect on the exception handlers of a particular process — the same handlers are active after the `yield` as before the `yield`.

```
fun yield () =
   callcc (fn parent =>
     (enqueue parent;
      dispatch ()))
```

The current handlers are captured by the `callcc`. Later, when this process reaches the head of the ready queue, the handlers are restored by the `throw` in `dispatch`.

By the same reasoning, `fork` has no effect on the exception handlers of the parent process. However, note that although `callcc` saves the current handlers, it does not change them. Thus, inspecting the code for `fork`

```
fun fork f =
   callcc (fn parent =>
     (enqueue parent;
      f ();
      exit ()))
```

we see that `f` is executed in an exception context inherited from its parent process. So, an exception that escapes a child thread may be caught within its parent thread. For example, the following program will print `Surprise!`[2]

```
let fun child () = (1 div 0; ())
    fun parent () = fork child handle Div => print "Surprise!"
in
    fork parent;
    ...
end
```

At first this may seem reasonable, but with only a small change we begin to see why this behavior is undesirable.

```
let fun child () = (1 div 0; ())
    fun parent () = (fork child handle Div => ();
                     x := !x + 1)
in
    fork parent;
    ...
end
```

Now, when `child` raises the `Div` exception, `parent` catches the exception and increments `x`. But note that the parent thread is still in the ready queue. When it eventually reaches the head of the ready queue and is dispatched, it will increment `x` a second time!

In the above example, the exception prematurely woke a sleeping process. With another slight change, the exception can actually resurrect a process that has already exited.

[2]In each of the following examples, we rely on the fact that `fork` suspends the parent thread and immediately executes the child thread. Similar examples can be devised for different scheduling policies.

```
let fun child () = (yield (); 1 div 0; ())
    fun parent () = (fork child handle Div => ();
                     x := !x + 1)
in
    fork parent;
    ...
end
```

In this example, the child thread voluntarily yields control and the parent thread executes to completion. Later, when the child thread resumes, it raises an exception that resurrects and re-executes the parent process.

The situation becomes even worse given only a slightly different implementation strategy for threads. For example, threads are sometimes represented as functions of type `unit -> unit` rather than continuations. Then, the function passed to `fork` can be transferred directly to the ready queue (possibly with some minor modifications involving exception handlers). Later, when a thread is suspended, its continuation is coerced into a function by partially applying `throw`. In this representation, a child thread may be executed not in the exception context of its parent, but rather in the exception context of whatever thread happens to be active when the child thread first reaches the head of the ready queue. Then, any exceptions that escape the child thread are caught, not by its parent, but rather by this unrelated thread.

Returning to our implementation of threads as continuations, we take a first step towards solving these kinds of problems by guaranteeing that no exception escapes its thread. We accomplish this by installing around each new thread a universal handler that will catch any errant exceptions.

```
fun fork f =
   callcc (fn parent =>
     (enqueue parent;
      f () handle _ => print "Uncaught exception.";
      exit ()))
```

However, this is not quite enough. What if `exit` itself was the culprit? In other words, what if `f` raised the `NoReadyThread` exception by calling `exit` when the ready queue was empty (i.e., when all other threads had already exited)? The handler installed by `fork` will catch the exception, but the exception will be immediately re-raised by the subsequent `exit`. This exception will be caught either by an internal handler of `f`'s parent, resurrecting `f`'s parent from the dead, or by the handler wrapped around `f`'s parent by `fork`. In the latter case, the exception will again be immediately re-raised by the subsequent `exit`. In this fashion, the `NoReadyThread` exception can propagate through each of `f`'s ancestors all the way out to the main thread.

This problem can only arise if a child thread calls `exit` after the main thread has already exited. However, since the result of the main thread is the result of the entire program, it is reasonable to forbid the main thread from calling `exit`. This is easily accomplished by keeping track of whether the current thread is the main thread or a child thread, and raising a new exception `MainThreadCantExit` if the main thread attempts to call `exit`. With this approach, there will always be at least one thread in the ready queue whenever we are executing a child thread — namely, the main thread. Therefore, `dispatch` can never fail and there is no longer any need for the `NoReadyThread` exception.

Recall that when the main thread returns (as opposed to exiting), any sleeping threads silently disappear. Therefore, if we forbid the main thread from exiting, then we should also provide a way for the main thread to find out when it is safe for it to return (i.e., when all the other threads have exited). It is not difficult to provide a primitive to allow the main thread to sleep until the other threads have all exited. For the details of this primitive, as well as the changes necessary to prevent the main thread from calling `exit`, see the full implementation in Appendix A.

5 Space-Safety of the Threads Package

The term "space-safety" refers informally to the notion that the implementation of some feature or mechanism will not, through normal use, leak heap or stack storage. This notion is almost always informal because it often depends on the intricate details of a particular implementation and what constitutes "normal use." Still, there are common practices in areas such as automatic garbage collection that allow one to make useful conclusions about the space-safety of mechanisms such as threads.

Neither of the threads implementations presented so far is safe-for-space. Both suffer from two kinds of potential space leaks. In the first kind of leak, a child thread unnecessarily retains its parent's continuation. This potential leak is not too worrisome because it is likely to be optimized away by a good compiler. The second kind of leak, however, is more serious. In this leak, a child thread unnecessarily retains its parent's exception handlers. Unfortunately, this second leak is unlikely to be optimized away without enormous advances in compiler technology.

The fact that these implementations leak storage is surprising, because implementations like these (e.g., ML-Threads [3] and CML [11, 12]) have been in use for many years. We can only speculate that the kinds of idioms for which these space leaks prove problematic have not arisen until now, or that these leaks have been hidden by other, more obvious leaks.

First, consider why a child thread might retain its parent's continuation. In the `fork` function,

```
fun fork f =
   callcc (fn parent =>
     (enqueue parent;
      f ();
      exit ()))
```

the code sequence `f (); exit ()` comprises the child thread. We know that `exit` never returns, but the compiler may not realize this fact. Therefore, the system may hang on to `exit`'s continuation, which is just `parent`, in case `exit` ever returns. Even if the parent thread exits, it cannot be garbage collected until the child thread releases this pointer. In fact, the system may retain the parent thread's memory until not only the child thread, but all of the child thread's descendants, have exited.

Fortunately, it is reasonable to hope that a good compiler would recognize that `exit` ends in a call to `throw`, and therefore never returns. In that case, the system can release these extraneous pointers to the parent continuations and avoid the space leak. Strangely, SML/NJ[3] does not appear to optimize throws in this fashion.

However, the picture is not so rosy when we turn to the second space leak, in which a child thread retains its parent's exception handlers. Recall the cascade of exceptions that arose in the previous section when `exit` raised the `NoReadyThread` exception. The system must allow for such a cascade, and thus must retain each of the handlers in this chain. Once we forbid the main thread from calling `exit`, we can prove that `exit` will never raise an exception within a child thread, and hence that such a cascade will never occur. But, *it is unreasonable to expect the compiler or run-time system to be able to prove this fact.* Therefore, exception handlers for a given thread cannot be garbage collected until all of its descendants have exited. This problem is particularly severe for our second implementation of threads, in

[3] Version 109.

which we install a universal handler around each thread, but it can arise even in the first implementation if a thread installs its own exception handlers.

Because a thread may retain context from each of its ancestors, these space leaks are most noticeable when there are deeply nested threads. For example, imagine a server architecture in which each request is processed in its own thread. When a thread receives a request, it immediately forks off a new thread to wait for the next request. In this architecture, the dth request is processed in a thread of depth d. Obviously, such a server cannot afford a space leak that grows with each new thread. Exactly this kind of architecture appears in the Fox Project's FoxNet system [2].

6 Safe-for-Space Threads

In both implementations so far, problems arise because a child thread hangs on to some of its parent's context. We can avoid these problems by arranging that every thread is executed in a top-level context rather than in its parent's context. We do this by using first-class continuations to capture a "thread-activating" context at the top level.

```
val threadActivator : (unit -> unit) cont =
   callcc (fn return =>
      let val f = callcc (fn fc => throw return fc)
      in
          f () handle _ => print "Uncaught exception.";
          exit ()
      end)
```

Then, `fork` can start each new thread in this context.

```
fun fork f =
   callcc (fn parent =>
     (enqueue parent;
      throw threadActivator f))
```

Now each child thread may retain some of the top-level context, but none of its parent's context. More importantly, all children retain the *same* top-level context, so the size of the retained context does not grow with every new thread.

This is enough to avoid the space leak, but in fact we can do slightly better with only a small change. Each child thread now creates its own exception handler. Since these exception handlers are all the same, we would prefer for all children to share the same handler. This can be accomplished by capturing the thread-activating context *inside* the exception handler that checks for uncaught exceptions.

```
val threadActivator : (unit -> unit) cont =
   callcc (fn return =>
     (let val f = callcc (fn fc => throw return fc)
      in f () end
      handle _ => print "Uncaught exception.";
      exit ()))
```

Now each child retains no more extra context than is already captured by `threadActivator`. This includes the handler inside `threadActivator` and any other handlers that were active when `threadActivator` was created. If we are careful, this should only be the default handler that catches uncaught exceptions at the top level. In addition, if the compiler is not smart enough to realize that `exit` can never return, `threadActivator` may accidentally retain `exit`'s continuation, `return`, which could be very large. We hope that this reference will be optimized away, but if not, we can sometimes provide hints to the compiler that this continuation can be deallocated. For example, we can add a superfluous `raise` after the `exit` to make it perfectly clear that the child thread will never return.

```
val threadActivator : (unit -> unit) cont =
   callcc (fn return =>
     (let val f = callcc (fn fc => throw return fc)
      in f () end
      handle _ => print "Uncaught exception.";
      exit ();
      raise SomeException))
```

This `raise` will never be executed, but is just enough to convince current versions of SML/NJ to release the unnecessary continuation. Of course, we would not recommend such an ugly hack in general, but it may be appropriate for library code that is expected to be as efficient as possible.

7 Other Control Operators

Researchers have studied many control operators besides `callcc` and `throw`. In this section, we briefly summarize some of the difficulties one encounters when trying to implement safe-for-space threads using a few of these alternative control operators.

First, we consider Felleisen's `control/prompt` [5] and Danvy and Filinski's `shift/reset` [4]. These seem like natural choices for implementing safe-for-space threads since they essentially allow one to run a thread in a top-level context, much like the `threadActivator` continuation does. However, if the user also has access to these operators, then it becomes difficult to give *any* reasonable implementation of threads, much less one that is safe-for-space. The problem is that a prompt set by the user might mask a prompt set by the threads package, or vice versa.

Gunter, Rémy, and Riecke's `set/cupto` [6] solve this problem by allowing one to name prompts. Then, we can prevent the user from interfering with our prompt simply by hiding its name. In fact, Gunter, Rémy, and Riecke present as an example an implementation of threads similar in spirit to those described in this paper. Unfortunately, their implementation of threads suffers from several space leaks. One leak stems from the use of exception handlers. Fixing this leak is complicated by the fact that their implementation of `set/cupto` is buggy with respect to exceptions. Still, assuming a correct implementation of `set/cupto`, it should be possible to avoid this leak with careful programming. A second, more serious leak involves the stack of control points maintained by the control operators. Every time a thread yields and resumes, it pushes an extra control point on the stack, so that a thread that has yielded and resumed n times has a control stack of at least depth n. It is unclear whether this leak is inherent in any implementation of these control operators, or an artifact of the particular implementation of `set/cupto` presented in [6].

Finally, Reppy has proposed variants of `callcc` and `throw`, called `capture` and `escape`, that do not save and restore the exception handler stack [12, page 136]. On the surface, these operators sound like they might help prevent the retention of unnecessary exception handlers. In fact, however, just the opposite is true. Suppose that we replace `callcc` and `throw` in our threads package with `capture` and `escape`, and consider the following program fragment:

```
(fork (fn () => exit ()); raise E) handle E => ...
```

First, we install the `E` handler, and then fork the child thread, which installs a new universal handler. The child thread then exits, and the parent thread resumes, with both handlers still active. Thus, when we raise `E`, it will be caught by the universal handler of the now defunct thread. Not only can this lead to faster space leaks by retaining too many exception handlers, it also causes the kinds of bizarre behavior about which we complained in Section 4, only worse. Now, a thread's exceptions may be caught by another thread even when we specifically attempt to handle the exception in the current thread.

8 Conclusions

This work was motivated by a space leak discovered in the FoxNet HTTP server [2]. We have experimentally observed such leaks for many different implementations of threads, but these leaks are difficult to quantify because of the lack of adequate profiling tools. However, the example of the FoxNet HTTP server, which is expected to run for months at a time, shows that even slow leaks can be intolerable. We have also modeled these space leaks in a semantics combining both exceptions and first-class continuations.

Threads are an important structuring tool for real-world systems, and the ability to implement light-weight threads is often cited as one of the major benefits of first-class continuations. However, as we have shown, implementations of threads that are both safe-for-space and predictable in the presence of exceptions can be quite subtle and somewhat complicated. This suggests, possibly, that threads ought to be a primitive notion in the language, instead of constructed out of continuations. But until functional languages such as Standard ML incorporate threads as primitive features, we can build on the techniques shown in this paper to build threads that are fast and behave well even in long-running programs.

Acknowledgements

This paper grew from a tutorial presented at the Second International Summer School on Advanced Functional Programming Techniques in Olympia, Washington [8]. We have benefitted greatly from discussions with, and comments from, Olin Shivers, Olivier Danvy, Bob Harper, the participants of the summer school, and the referees of this workshop.

References

[1] Appel, A. W., and MacQueen, D. B. A Standard ML compiler. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science* (1987), vol. 274, Springer-Verlag, pp. 301–324.

[2] Biagioni, E., Harper, R., and Lee, P. Signatures for a protocol stack: A systems application of Standard ML. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming, Orlando* (June 1994), pp. 55–64.

[3] Cooper, E. C., and Morrisett, J. G. Adding threads to Standard ML. Tech. Rep. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[4] Danvy, O., and Filinski, A. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (June 1990), pp. 151–160.

[5] Felleisen, M. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages* (Jan. 1988), pp. 180–190.

[6] Gunter, C. A., Rémy, D., and Riecke, J. G. A generalization of exceptions and control in ML-like languages. In *Proceedings of the 1995 Conference on Functional Programming Languages and Computer Architecture* (June 1995), pp. 12–23.

[7] Haynes, C. T., Friedman, D. P., and Wand, M. Obtaining coroutines with continuations. *Computer Languages 11*, 3–4 (1986), 143–153.

[8] Lee, P. Implementing threads in Standard ML. In *Advanced Functional Programming* (Aug. 1996), vol. 1129 of *LNCS*, pp. 115–130.

[9] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.

[10] Nelson, G., Ed. *Systems programming with MODULA-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[11] Reppy, J. H. CML: A higher-order concurrent language. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto* (June 1991), pp. 293–305.

[12] Reppy, J. H. *Higher-Order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, January 1992.

[13] Wand, M. Continuation-based multiprocessing. In *Proceedings of the 1980 LISP Conference* (Aug. 1980), pp. 19–28.

A Final Implementation

```
signature COROUTINE =
sig
   exception MainThreadCantExit
   exception ChildThreadCantSync

   val fork : (unit -> unit) -> unit
   val yield: unit -> unit
   val exit : unit -> 'a

   val sync : unit -> unit
     (* sync () yields until all other threads have completed *)
end

structure Coroutine : COROUTINE =
struct
   exception MainThreadCantExit
   exception ChildThreadCantSync

   val callcc = SMLofNJ.callcc
   val throw  = SMLofNJ.throw

   datatype threadType = Main | Child
   type thread = unit cont * threadType

   val readyQueue : thread Queue.queue = Queue.mkQueue ()
   val syncCont : thread option ref = ref NONE
   val currentThreadType = ref Main

   fun enqueue thread = Queue.enqueue (readyQueue, thread)

   fun dispatch () =
      let val (t,typ) =
             Queue.dequeue readyQueue
             handle Queue.Dequeue =>
                (* syncCont cannot be NONE *)
                case !syncCont of SOME main => main
      in
          currentThreadType := typ;
          throw t ()
      end
```

```
  fun exit () =
     case !currentThreadType of
        Main  => raise MainThreadCantExit
      | Child => dispatch ()

  fun sync () =
     case !currentThreadType of
        Main => callcc (fn t =>
                  (syncCont := SOME (t, Main);
                   dispatch ()))
      | Child => raise ChildThreadCantSync

  fun yield () =
     callcc (fn parent =>
       (enqueue (parent, !currentThreadType);
        dispatch ()))

  val threadActivator : (unit -> unit) cont =
     callcc (fn return =>
       (let val f = callcc (fn fc => throw return fc)
        in f () end
        handle _ => print "Uncaught exception.";
        exit ();
        (* raise dummy exception as hint to compiler *)
        raise MainThreadCantExit))

  fun fork f =
     callcc (fn parent =>
       (enqueue (parent, !currentThreadType);
        currentThreadType := Child;
        throw threadActivator f))
end
```