Assignment 8
Implementing Concurrent ML

Karl Crary
15-312: Foundations of Programming Languages

Out: Sunday, April 17, 2005
Due: Friday, April 29, 2005 (11:59 pm)

1 Introduction

In this project, you will implement a fragment of Concurrent ML, including threads, channels, and first-class synchronization. You are provided code implementing continuations, timer interrupts, and thread-safe printing. You are also provided with signatures for the code that you write. Figure 1 lists the files you are provided and their contents.

You may work together in teams of at most two. The deliverables for this project include (1) the code, (2) a brief report, and (3) a demo for the instructor and/or TA.

To assist you in the development of this project, it is broken into several pieces and each piece is given a signature. Each piece is designed to provide the necessary tools for later pieces. Do not change the signatures! If you have any questions about the interfaces, consult the instructor or TAs.

This document will walk you through the provided code, and the interfaces you are to implement. A full listing of all signatures appears at the end of this document.

Be sure to read the *entire* assignment carefully before you begin!

File	Sig./Struct.	Description
`cont.sig`	`CONT`	Interface for continuations
`timer.sig`	`TIMER`	Interface for timer interrupts
`print.sig`	`PRINT`	Interface for thread-safe print
`queue.sig`	`QUEUE`	Interface for ephemeral queues
`thread.sig`	`THREAD`	Interface for threads
`condition.sig`	`CONDITION`	Interface for condition signals
`event.sig`	`EVENT`	Interface for events
`channel.sig`	`CHANNEL`	Interface for channels
`cml.sig`	`CML`	Interface for CML as a whole
`cont.sml`	`Cont`	Implementation of continuations
`timer.sml`	`Timer`	Implementation of timer interrupts
`print.sml`	`Print`	Implementation of thread-safe print
`cml.sml`	`CML`	Glue combining your implementation
`queue.sml`	`Queue`	Template for queues
`thread.sml`	`Thread`	Template for threads
`condition.sml`	`Condition`	Template for condition signals
`event.sml`	`Event`	Template for events
`channel.sml`	`Channel`	Template for channels
`demo.sml`	`Demo`	Template for demonstration code
`sources.cm`		SML/NJ configuration management file

Figure 1: Provided files

2 Provided Code

You are provided four pieces of code:

- `Cont : CONT` provides first-class continuations.
- `Timer : TIMER` provides operations for managing timer interrupts and atomic regions. These are discussed in the next two sections.
- `Print : PRINT` provides a thread-safe printing function. Do not use the ordinary print function while timer interrupts are active.

- `CML : CML` provides code that glues your implementation pieces together into a CML implementation. You can use this to implement concurrent applications, and we will use it to test your implementation.

  For your convenience, `CML` also provides a function `activate` that you can use to start up concurrent programs. Its two arguments are the size of the timeslice and a function to call.

Important Note: You must use Standard ML of New Jersey for `Cont`, `Timer`, or `Print` to work. It is not possible to use other SML compilers with this assignment.

3 Timer Interrupts

To assist you with the implementation of preemptive scheduling, we have provided a package for managing timer interrupts. There are two main functions, `startTimer` and `stopTimer`, which start and stop the timer.

The `startTimer` function takes three arguments that you provide, packaged in a record:

- `timeslice` gives the size of the timeslice, in milliseconds.
- `tickHandler` gives the system a function for handing timer interrupts, which are raised once per timeslice.
- `interruptHandler` gives the system a function for handling interruption interrupts (that is, for handling control-C). This is useful because the ordinary SML/NJ handler for control-C does not work very gracefully while timer interrupts are active so you will wish to install a new control-C handler.

Each interrupt handler you provide has the type `unit cont -> unit cont`. Each time the timeslice expires, the interrupt handler is called with a continuation representing the current thread of control. The interrupt handler processes that continuation then returns another continuation representing

the new thread of control. SML/NJ then continues the program using the new continuation.

To make this more concrete, suppose SML is about to execute the expression `..expr..` when an interrupt occurs. In the place of `..expr..` we could execute the equivalent expression:

```
( () ; ..expr.. )
```

Now suppose the interrupt handler is `handlerfn : unit cont -> unit cont`. Then instead of the above expression, SML executes:

```
( callcc (fn k => throw (handlerfn k) ()) ; ..expr.. )
```

If `handlerfn` is the identity function, then the `throw` expression simply throws `()` to the continuation `k`, resulting in the entire `callcc` expression simply returning `()`. Thus, an interrupt has no effect if its handler is the identity function.

However, a handler can also switch to a new thread by storing the old continuation and returning a new one. Thus, we can implement preemptive threads by putting code very similar to the code for `Thread.yield` (which you will implement) into the timer interrupt handler.

4 Atomic Sections

A standard problem with preemptive multitasking is that many (or perhaps most) data structures must be accessed exclusively by at most one thread at a time. For example, if multiple threads attempt to access a queue at once, it is likely that they will simultaneously attempt to perform inconsistent writes to the queue's state, thereby corrupting the queue.

To prevent this problem, one ordinarily implements mutual exclusion primitives such as semaphores, mutexes, or monitors. These ensure that only one thread is dealing with the data structure at one time.

However, in a single-processor concurrent setting, there is a simpler solution. A thread that requires exclusive access to a data structure may enter an

atomic section, during which is it impossible for that thread to be preempted. When it is done accessing the data structure, it leaves the atomic section, allowing preemption once more.

> **Note:** atomic sections are *not* a good solution to the general problem of mutual exclusions. There are (at least!) two reasons why not: First, atomic sections are not usually possible to begin with on a multi-processor platform. Second, even on a single processor, if very much time is spent in atomic sections, the concurrent performance of the system is seriously degraded.
>
> Therefore, atomic sections are a reasonable means to ensure mutual exclusion to the data structures that implement the concurrency system, since all concurrency primitives should execute quickly and therefore little time will be spent in atomic sections. However, atomic sections should *never* be used in user code. Consequently, no atomic section utilities are provided in the `CML` signature.

Support for atomic sections is provided by two functions in `TIMER`: `beginAtomic` enters an atomic section, and `endAtomic` leaves an atomic section. *Interrupt handlers are automatically executed atomically; calls to `beginAtomic` and `endAtomic` are not necessary and should not be used.*

Note: calls to `beginAtomic` and `endAtomic` must be matched properly. For example, the following code fragments may be correct:

```
beginAtomic ();
..expr 1..;
endAtomic ()
```

```
beginAtomic ();
if ..expr 2.. then
   (
   ..expr 3..;
   endAtomic ()
   )
else
   (
   endAtomic ();
   ..expr 4..;
   )
```

These are correct provided the only one of the elided expressions that can raise an exception is `..expr 4..` If, for example, `..expr 1..` can raise an exception, one must instead write something like:

```
beginAtomic ();
(..expr 1.. handle ex => (endAtomic (); raise ex));
endAtomic ()
```

To assist in diagnosing unmatched atomic section entries and departures, the atomic section functions will raise the exception `Atomic` if one attempts either to enter an atomic section while already in one, or to leave an atomic section while not in one, or to enter or leave an atomic section while in an interrupt handler. (However, you may find this to be of limited helpfulness if your implementation of threads allows threads that raise uncaught exceptions to disappear silently.)

4.1 Atomicity Assumptions

Many of your functions will need to make assumptions regarding whether or not they are in an atomic section when called. For example, many functions will need to enter an atomic section to ensure exclusive access to their data structures, and so they must assume that they are not called in an atomic section.

Similarly, in order to write correct code, we must know whether we are in an

atomic section when a function returns. For example, some functions that begin in an atomic section will end still in an atomic section, whereas others will terminate the atomic section before returning. Finally, many functions do not care about atomicity. They can be called in or out of an atomic section and return the same way.

4.2 Documenting Atomicity Assumptions

Making invalid assumptions about atomicity is a very likely source of errors, so you must be very careful to document all our atomicity assumptions. In each of the signatures you are provided, all of the functions are grouped under comments that indicate their atomicity assumptions, such as "`begins atomic, ends non-atomic`". Functions (such as those in `QUEUE`) that make no assumptions about atomicity are not labelled. (A few functions that must be called with preemption inactive, or that disable preemption are simply marked "`special`".)

For any code that we provide, you should assume that the stated atomicity assumptions are satisfied. For any code that you are to provide, you must ensure that the stated assumptions are accurate. For example, in a function with no stated assumption (which therefore makes no assumption), one should not call `beginAtomic` or `endAtomic`. As another example, in a function that begins and ends non-atomic, one may call `beginAtomic`, provided one ensures a matching `endAtomic` is called. (Note that a function that begins and ends atomic is not *required* to enter an atomic section, but it may.)

Note: Since atomic sections are not available to user code, the `CML` signature does not list any atomicity assumptions. Implicitly, all `CML` functions are "`begins and ends non-atomic`" (except for `activate` which would be "`special`").

5 Your Implementation

Your implementation is broken into five pieces: queues, threads, condition signals, pure events, and channels. You will also provide a small demonstration program.

5.1 Queues

First you should implement ephemeral FIFO queues, with the signature `QUEUE`. The `reset` operation empties a queue. The `front` operation returns the front element of the queue without deleting it; `remove` returns it and deletes it. The other functions are self-explanatory.

5.2 Threads

Next you should implement a thread package. Providing the following functions:

- `spawn`: spawns a new thread and makes it ready.
- `yield`: causes the current thread to yield to another ready thread.
- `exit`: causes the current thread to exit immediately. If no ready threads remain, it throws to the top-level continuation (see below).
- `shutdown`: shuts down the thread system and throws to the top-level continuation (see below).
- `spawnAtomic`: like `spawn`, but for use within an atomic section.
- `beginPreemption`: initializes the thread system and activates timer interrupts. The first argument is the size of the timeslice in milliseconds. The second argument is a *top-level continuation* to throw to in the event that either all threads die or there is an interruption interrupt (*i.e.,* control-C). You may assume that the top-level continuation calls `endPreemption`. (The function `CML.activate`, which we provide, supplies an appropriate top-level continuation when it calls `beginPreemption`.)
- `endPreemption`: does any necessary final cleanup when shutting down the thread system, such as stopping the timer. The top-level continuation must call `endPreemption`.

Suggestions:

- Your `Thread` implementation does not need to know anything about blocked threads. As far as `Thread` is concerned, blocked threads have exited.
- Try implementing non-preemptive threads first. Then adapt your `yield` code to an appropriate timer interrupt handler.
- When threads that raise uncaught exceptions die silently, it can be difficult to diagnose bugs. Rather than allowing them to die silently, have your `spawn` function cause threads to print a message before they die. The built-in functions `exnName` or `exnMessage` may be helpful.

5.3 Condition Signals

The key synchronization mechanism you will use internally when implementing events and channels is *condition signals.* (Note that condition signals are for internal use only, and are not made available to user code.) For those who are familiar with condition variables, condition signals are like condition variables, except that only a single thread can ever wait on any particular condition signal.

Condition signals are created by the `wait` function. When `wait` is called with an argument function `f`, `wait` (1) creates a new condition signal, (2) passes the condition signal to `f`, which typically places it into various queues, then, when `f` has returned, (3) blocks until the condition is signalled. Once the condition is signalled, `wait`'s return value is a value passed to `signal` (see below).

A thread is awakened using the `signal` function. If the thread is still blocked (that is, if the condition has not already been signalled) then the thread is awakened and `signal` returns true. If the thread has already been awakened, then `signal` returns false. When `signal` awakens a waiting thread, `signal`'s second argument becomes the awakened thread's return value for `wait`. The function `query` determines whether a thread is still waiting (returning true if so), without awakening it.

Finally, the function `wrapCond` composes a function with a condition signal.

Thus, if `wrapCond (c,f)` is signalled with value `v`, the waiting thread receives `f(v)`. The argument function is assumed never to raise an exception.

Important Note: Be careful to make sure that your `wait` function can only return once. Later calls to `signal` should return false, they should *not* reawaken or respawn a formerly blocked thread.

Suggestions:

- You will need to use `callcc` and `throw` to implement condition signals.
- You may wish to use the following code (from class) for composing a function with a continuation:

```
fun compose (k : 'a cont, f : 'b -> 'a) : 'b cont =
    callcc
    (fn escape =>
           throw k (f (callcc (fn k' => throw escape k'))))
```

  Recall that this code works only when the function `f` never raises an exception.
- A good way to block a thread is simply to have it exit, and to rely on the signaller to respawn the thread once it is awakened. The thread mechanism should not need to know anything about the condition signal mechanism.
- Be sure to note that all the condition signal functions except `wrapCond` are intended to be called within an atomic section; `signal` and `query` return still in the atomic section, but `wait` ends it.

5.4 Events

Using condition signals, you will implement events. The functions `sync`, `select`, `wrap`, `guard`, `choose`, `never`, and `alwaysEvt` are as described in class.

The `Event` structure must also provide a back door so that other structures (such as `Channel`) can create events. This back door is called `makeEvt`, which builds an event from a base event. A base event contains three functions that are used to synchronize on the event:

- `poll` returns true if the event is currently enabled (that is, it can be synchronized immediately without blocking).
- `perform` immediately synchronizes the event and returns its value.
- `block` takes a condition signal and records it in an appropriate queue to be signalled when the event becomes enabled. If and when the condition is signalled, it is passed the event's value.

All of the three functions are assumed to begin and end in an atomic section, and all are assumed never to raise an exception. You may wish to establish some additional invariants regarding their usage, particularly in conjunction with each other. You may do so, provided that the invariants are properly documented.

Suggestions:

- Think carefully about the best representation for the `event` type. The right representation will make your implementation **much** easier.

5.4.1 Wrap and Pure Events

There are to be two implementations of the `EVENT` signature, one called `PureEvent` and one called `Event`:

- In the `PureEvent` structure, you may assume that the function argument to `wrap` never raises an exception, and never calls any CML utilities. This may be useful, because it means that the function can be called in any context, including within an atomic section (where many CML utilities cannot be called since they may attempt to enter an atomic section), or in a context where exception raises are not permitted.
- In the `Event` structure, the function argument to `wrap` is permitted to be arbitrary code.

You will implement the `PureEvent` structure. The `Event` structure is provided. It uses your implementation of pure events to implement general events.

5.5 Channels

Using the tools you have built so far, you will implement channels. Each of the channel functions are as described in class.

5.6 Demonstration Code

The `CML` structure you are provided combines all of your code into a (hopefully!) working implementation of Concurrent ML. The last thing you will implement is a small concurrent program that demonstrates your system in an interesting manner.

Your demonstration program need not use every single feature of the system, but at a minimum it should illustrate multiple threads, channels, `select` or `choose`, and `wrap`. Of course, the more it uses the more fun it is. (Also keep in mind that, in any case, all the features of the system will be tested by our code.)

To assist you in calibrating how much time to spent on this part of the assignment, the demonstration code will be worth no more than five percent of the full assignment.

6 What to Hand In

You are to hand in two things with this assignment:

1. The code for your implementation. In order to ensure that your code can link with ours, you must be sure:
   - **not** to alter any signatures,
   - **not** to change the names of any files, and
   - **not** to rely on any files other than those listed in Figure 1.
2. A brief report (no more than 2 or 3 pages) documenting your design decisions and invariants. Your report should be named `report.txt` and included with your code.

You will also sign up for a project demo to be held on Monday, May 3. Students for whom this date causes a hardship may make special arrangements with the instructor.

7 Conclusion

Have fun!

A A Resource and a Warning

You may find *Concurrent Programming in ML* by John H. Reppy (Cambridge University Press, 1999) to be a useful reference for Concurrent ML. However you will probably *not* find its chapter on implementing concurrency very helpful, and you may even find it confusing, as the implementation discussed there is quite different from the one you are to undertake.

B Signatures

```
signature CONT =
   sig
      type 'a cont

      val callcc : ('a cont -> 'a) -> 'a
      val throw : 'a cont -> 'a -> 'b
   end

structure Cont :> CONT
```

```
signature TIMER =
   sig
      (* special *)
      val startTimer : {
                       timeslice : int,
                       tickHandler
                         : unit Cont.cont -> unit Cont.cont,
                       interruptHandler
                         : unit Cont.cont -> unit Cont.cont
                       } -> unit
      val stopTimer : unit -> unit

      exception Atomic

      (* begins non-atomic, ends atomic (of course) *)
      val beginAtomic : unit -> unit

      (* begins atomic, ends non-atomic (of course) *)
      val endAtomic : unit -> unit
   end

structure Timer :> TIMER


signature PRINT =
   sig
      (* begins and ends non-atomic *)
      val print : string -> unit

      (* begins and ends atomic *)
      val printAtomic : string -> unit
   end

structure Print :> PRINT
```

```
signature QUEUE =
   sig
      type 'a queue

      val queue : unit -> 'a queue
      val reset : 'a queue -> unit

      val is_empty : 'a queue -> bool
      val insert : 'a queue * 'a -> unit

      exception EmptyQueue
      val remove : 'a queue -> 'a
      val front : 'a queue -> 'a
   end


signature THREAD =
   sig
      (* begins and ends non-atomic *)
      val spawn : (unit -> unit) -> unit
      val yield : unit -> unit
      val exit : unit -> 'a
      val shutdown : unit -> 'a

      (* begins and ends atomic *)
      val spawnAtomic : (unit -> unit) -> unit

      (* special *)
      val beginPreemption : int -> unit Cont.cont -> unit
      val endPreemption : unit -> unit
   end
```

```
signature CONDITION =
   sig
      type 'a cond

      (* arg fn must not raise an exception *)
      val wrapCond : 'a cond * ('b -> 'a) -> 'b cond

      (* begins and ends atomic *)
      val signal : 'a cond * 'a -> bool
      val query : 'a cond -> bool

      (* begins atomic, ends non-atomic, arg is called atomically *)
      val wait : ('a cond -> unit) -> 'a
   end
```

```
signature EVENT =
   sig
      type 'a event

      (* begins and ends non-atomic *)
      val sync : 'a event -> 'a
      val select : 'a event list -> 'a
      val wrap : 'a event * ('a -> 'b) -> 'b event
      val guard : (unit -> 'a event) -> 'a event
      val choose : 'a event list -> 'a event
      val never : 'a event
      val alwaysEvt : 'a -> 'a event

      (* for each function belonging to a baseEvt we assume:
         - the function begins and ends atomic
         - the function never raises an exception *)
      type 'a baseEvt = {
                        poll : unit -> bool,
                        perform : unit -> 'a,
                        block : 'a Condition.cond -> unit
                        }

      (* begins and ends non-atomic *)
      val makeEvt : 'a baseEvt -> 'a event
   end


signature CHANNEL =
   sig
      type 'a chan

      val channel : unit -> 'a chan

      (* begins and ends non-atomic *)
      val send : 'a chan * 'a -> unit
      val recv : 'a chan -> 'a
      val sendEvt : 'a chan * 'a -> unit Event.event
      val recvEvt : 'a chan -> 'a Event.event
   end
```

```
signature CML =
   sig
      type 'a event
      type 'a chan

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

      val sync : 'a event -> 'a
      val select : 'a event list -> 'a

      val wrap : 'a event * ('a -> 'b) -> 'b event
      val guard : (unit -> 'a event) -> 'a event
      val choose : 'a event list -> 'a event
      val never : 'a event
      val alwaysEvt : 'a -> 'a event

      val channel : unit -> 'a chan
      val send : 'a chan * 'a -> unit
      val recv : 'a chan -> 'a
      val sendEvt : 'a chan * 'a -> unit event
      val recvEvt : 'a chan -> 'a event

      val shutdown : unit -> 'a

      val activate : int -> (unit -> 'a) -> unit
   end
```