Threading and GUI Issues for R

Luke Tierney
School of Statistics
University of Minnesota

March 5, 2001

Contents

1 Introduction

This document collects some random thoughts on runtime issues relating to concurrency, threads, GUI's and the like. Some of this is extracted from recent R-core email threads. I've tried to provide lots of references that might be of use. If anyone has suggestions of things to add please let me know.

While intended mainly for R, I am hopeful that some of this will carry over to XLISP-STAT as well.

2 Concurrency and Parallelism

Butenhof [7, pages 4,5] gives definitions of the terms *concurrent* and *parallel*. Here is a brief version:

Concurrency: Things appear to happen at the same time but may happen serially.

Parallelism: Concurrent sequences that proceed simultaneously.

UNIX processes on a single processor machine are concurrent but not parallel. UNIX processes running on separate workstations run in parallel. To be able to use parallel algorithms of some sort to obtain speed improvements by using more than one processor requires parallelism. To be able to program conceptually concurrent behavior, such as running a long simulation while doing some plot, requires only concurrency.

Thread systems are mainly intended to allow the management of concurrent computations. They are not the only way to do this: Many single threaded systems provide some sort of event loop combined with asynchronous IO facilities that can be used to manage input on several channels concurrently. Tcl's `fileevent` command is an example.

R currently has a small amount of concurrency; how much depends on the platform. On all platforms, since the memory management changes in 1.2, there is now the possibility that finalizers will be run concurrently with other code. On Windows `Rgui` and on Classic MacOS some event handling has to occur in the midst of interpreted calculation in order to allow the delivery of user interrupt actions via the GUI. With the Windows menu facilities, and with `tcltk`, this means that event processing and evaluation of R commands occur concurrently.

The concurrency that already exists raises some problems, in particular with respect to dynamic state and synchronization. One example of dynamic state is the output connection. A sequence like

```
sink("simout")
runsim()
sink()
```

is intended to change the output connection for the evaluation of the call to `runsim` between the two `sink` calls. If a user had added a menu item in `Rgui` to print the elapsed execution time and selected that menu item while `runsim` is executing, then one would probably want the the execution time printed in the listener window, not the `simout` file. Currently the time would go to the file.

Adding the concept of threads to R will allow us to think more clearly about these issue. We could for example make specifications like the following:

- Every thread of execution has its own input and output channels.
- There are three separate threads of execution in an R process:
  - a thread running the read-eval-print loop (REPL), called the REPL thread or the listener thread
  - a thread running the GUI event loop
  - a thread running finalization
- asynchronous IO actions can be registered for connections; they will be executed in the event loop thread.

This description makes it clear what will happen: If a listener command uses sink then this will affect only the output channel for the listener thread; if an IO action is triggered and runs for a long time then it will block the event loop thread (whether it blocks other threads as well depends on scheduling and other things). I'm not suggesting that this specification is necessarily the way we want to do things, but adding the *concept* of threads allows us to talk clearly about what it is we want to do; it provides the language for discussing concurrency issues.

I believe that explicit concurrency management tools (i.e. a threads toolkit) are what we really need in R at this point. Parallelism of some form would be nice, but it need not come from the R-level threads mechanism. It could, for example, come from using OS worker threads for handling vectorized arithmetic operations and perhaps a vectorized BLAS [27, 60]. As an aside, I did a little experiment with this on a dual processor Linux box. For a `daxpy`-like calculation the break-even point for splitting the work onto two worker threads is around a length of 200. Below that the overhead of thread management outweighs the benefit. Obviously this is very much dependent on everything.

3 Concurrency and Dynamic State

By dynamic state I mean certain global values that affect a computation and may be temporarily changed for the duration of a computation; I also include global settings that are manipulated by the runtime system and reflect the state of the system. Some examples:

- `options` settings
- the current device
- possibly `par` settings
- the standard IO connections
- the current (internal) evaluation context

3.1 Options Settings

Options can be used for several different purposes. Different uses are likely to have different intentions in terms of whether they should apply only to the current among several concurrent contexts or the entire process.

- The browser to use can be changed with `options(browser="mozilla")`. The intent is most likely to change the browser for the entire process.

- A package could add a package-specific option by executing `options(foo=xyx)` in its initialization function. This ought to remain available at the process level as long as the package is loaded. (Assuming package loading remains at the process level, as it probably should.)
- An expression `options(show.error=FALSE)` might be used in conjunction with `try`. This is likely to be meant only to apply to the `try` expression and to be undone after the `try` completes.
- Setting `options(na.action=...)` or `options(contrasts=...)` might occur during an analysis. Usually the intent will be to change these setting for the particular analysis only, not for other analyses that might be going on in separate listener windows (if at some point multiple listeners within the same R process were to be allowed).

The `show.error` example would often be done something like this:

```
olderr <- options(show.error=FALSE)
try(...)
options(show.error=olderr)
```

Unfortunately this may not be quite right since errors may not be the only non-local exit that can be taken. For example, in Splus 5.1

```
for (i in 1) {try(break); print(1)}
```

does not print, so the `break` is jumping out of the loop. In R that currently does not happen but probably should.

To deal with this possibility we would need to use `on.exit`. Since `on.exit` is quite coarse-grained, as it applies only to the whole function, we might want to localize it with

```
dotry <- function() {
    olderr <- options(show.error=FALSE)
    on.exit(options(show.error=olderr))
    try(...)
}
dotry()
```

But now we would need to worry that the `try` gets evaluated in the right environment.

This is pretty awkward to do and get right, so it might be a good idea to abstract it out into something like

```
with.options <- function(expr, ...) {
     old.opts <- save old options
     on.exit(restore old.opts)
     eval(expr, env=the right place)
}
```

so that we could then use

```
with.options(try(...), show.error=FALSE)
```

It might even make sense to separately pick out the localized `on.exit` issue by defining `unwind.protect` (to use the Common Lisp name) something like

```
unwind.protect<- function(expr, cleanup) {
      on.exit(eval(cleanup, env=the right place))
      eval(expr, env=the right place)
}
```

and then define `with.options` using `unwind.protect`.

In the presence of concurrent threads of evaluation saving and restoring option values around a piece of code is problematic. If we had a `with.options` mechanism, then for code of the form

```
with.options(try(...), show.error=FALSE)
```

is seems reasonable to assume that the intent is to change the option only for the evaluation of the `try(...)` expression. If this code is evaluated in the listener then code evaluated in the event loop should not be affected. Without such a declarative mechanism, the intent of code like

```
olderr <- options(show.error=FALSE)
try(...)
options(show.error=olderr)
```

is harder to interpret. Harder still is

```
options(show.error=FALSE)
try(...)
```

Is the intent really to make a permanent, process-wide change, or did the restore just get forgotten?

We need to think through how this should work in the context of concurrency. One possible approach is to start with a set of options setting for the process main thread. Each time a new thread is started it inherits a copy of the option settings of the creating thread. A variation would be to have a process-wide set of options and then have each new thread inherit a copy (the difference is that the parent thread does not matter). Both make introducing new options difficult. An alternative would be to allow options to be marked somehow as thread or process options. Process options would always be shared by all threads; if one thread changes a process option another would see the change. (This may require reader/writer lock synchronization.) Thread options have process-level values that are copied to the thread on thread creation. Threads could also be allowed to make additional options local to themselves as they run. As Kurt Hornik pointed out in a discussion of this issue on R-core, this is similar to the idea of buffer-local variables in emacs.

The approach of changing an option setting, executing some code with the new option setting, and then restoring the old option setting is essentially a mechanism for dynamically scoping variables implemented using shallow binding. (Shallow binding means using a global location to hold the current value and saving/restoring old values. The alternative, deep binding, would use a stack of values.) Using `on.exit` for restoration is fairly heavy-weight—it might make sense to do something lighter internally.

3.2 User Defined Options

Users or package writers can add new values to `options`. One drawback is that this makes it hard to keep track of options and also raises the possibility of name conflicts. An alternative would be to allow the creation of `options`-like entities (with all the right concurrency handling support), with `options` then being just one example of such a beast. A package for maps might then define a `maps.options` function that manages a process-level option `database` specifying a network location where map data is available and a thread-specific option `projection` specifies the map projection to use.

3.3 Devices and Par Settings

The current device is another candidate for being different for different concurrent threads (different interpreter windows say).

At times it might be nice to be able to create a device that is intended to be private to a set of functions (the `tkdensity` demo for example). At this point I think all you can do is work in terms of the current device index, but other code can use that as well.

On the other hand we may be quite happy for two listeners, say, to share a device, but then we might ask how `par` setting should be treaded. Should changes made the device from one listener be viewed as affecting the device, and therefore affect graphics commands given by the other listener, or not? Having `par` specific to the device, so shared if the device is shared, seems most natural to me, but I could see arguments the other way too.

3.4 Standard Connections

The R process currently has standard input, standard output, and standard error connections (at least internally). If we allow multiple threads to be used, for example, in multiple listener windows, then each window's thread would need its own connections. As already mentioned, redirecting output in one thread using `sink` probably should not affect other threads.

3.5 The Context Stack

Currently there is one internal context stack that lives on the C stack. It is of course used by the currently executing code but also by profiling; it also needs to be available to the garbage collector since heap pointers are stored there. Concurrency, even of the limited form used not, raises some issues that need thinking through. For example, under no circumstances should an error in an event handler that happens to execute during evaluation of a listener command be allowed to abort that command.

3.5.1 Synchronization

With any approach to concurrency, even the current informal one, there is some need for synchronization. One particular point is promises. Currently when a promise evaluation starts the promise is marked as under evaluation. This is to detect cycles where promises depend on themselves—with a single thread of execution, attempting to evaluate a promise that is already being evaluated is an error. With several threads though, it is only an error for the thread doing the evaluation to attempt to evaluate the promise again. If a second thread attempts to evaluate the promise while the first one is evaluating it, then the second thread should block and the first thread should signal it to resume when the evaluation is complete. The Haskell folks call this creating a black hole.

This is just one of the more esoteric things we will need to deal with; there are certainly others.

4 GUI Events And Blocking IO

We would like to be able to simultaneously

- work with a range of GUI systems (R's graphics device, `tcltk`, etc).
- allow simultaneous management of multiple slow IO connections (such as sockets)
- allow long computations and IO operations to be interrupted

An we would like to be able to provide this in a way that is as common as possible across UNIX, Win32 and MacOS.

4.1 UNIX Issues

On UNIX pretty much everything in the way of IO we care about goes through file descriptors. (In principle there may be other things also, such as message queues and semaphores but I'll ignore those.) Multiple IO channels can be handled by waiting using `select` or using one OS thread per blocking call. Interrupts can be handled by `longjmp` calls. I am a little nervous about allowing `longjump`s quite as liberally as we do now, but with selective enabling around blocking calls I think we are OK.

Unfortunately not all blocking calls make a descriptor openly available. For example, `gethostbyname` is a very convenient wrapper around the DNS protocol. It can block for quite a while waiting to get a response, and there is no way we can do anything else within a single thread that is waiting on DNS because we don't have the connection `fd` to use in a `select`. To be able to make this play nice with event loops on UNIX we would need an open implementation of the DNS interface that we can hook our socket layer into. I don't know off hand what Tcl/Python/Perl have done about this, but I think they may include such an interface. For DNS there may be a solution in the GNU `adns` library [1].

Another complication is that the interaction of signals with some of the socket functions is a bit complex. For instance, if connect is interrupted according to Stevens [58] you should select on the socket before trying to reconnect.

GUI systems for UNIX/X11 usually each make their separate X11 connection, which deep down means each one has its own file descriptor. Getting access to that file descriptor may be tricky, but if we can get it and if the GUI library provides a way to process a single event at a time, as they usually do, then we can handle a GUI just like any other IO source. Alternatively we can place each GUI in a separate thread or even process and make the connection to R through IO channels we create (see Section 4.6).

4.2 Win32 Issues

On Win32 there are two sources for input: kernel objects (which include files, sockets, pipes, semaphores, and lots more) and message queues. Each thread has its own message queue. A thread can wait for either a message or one of several kernel objects by calling `MsgWaitForMultipleObjects`.

A window that is created in a thread will always have its messages sent to the creating thread. If we want to run several GUI libraries this creates an issue of which one is in charge of managing the message queue. Fortunately this is not a very big issue. The Windows libraries require that a callback be registered with Win32 for each window. The standard event loop in Win32 looks something like

```
while(GetMessage(&msg, NULL, 0, 0)) {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
```

That is, the code never needs to look directly at the message. It just gets it off the queue and passes it on to two system functions; `DispatchMessage` calls the registered callback. As a result, it really doesn't matter which GUI library runs the event loop as long as one of them does.

Windows does not have proper signal handling. Some level of fake signals is available, but I do not believe it is sufficient at this point. It is possible to use messages much like a signal but this requires cooperation of the code that wants to allow interrupts. That is, instead of a blocking read on a socket you can do a `MsgWaitForObject`, which will wake either when the socket becomes readable or when a message is received. I believe the WinSock libraries are written more or less along these lines. They allow processing of messages during blocking IO operations.

4.3 Classic MacOS Issues

A fundamental problem with Classic MacOS it that there is a single event queue per process. The process code is responsible for looking at each event and deciding how to handle it. This means that coexistence of multiple GUI libraries is difficult. Each library would have to get a turn to look at the event queue, would have to be able to reliably recognize and process its own events, and be willing and able to pass on other events for handling by other libraries. Very few libraries are written like this.

There is quite a nice sockets library called GUSI (Grand Unified Sockets Interface) [40] that now also supports a variant of pthreads. The author is also the person responsible for the Mac port of Perl. GUSI allows for a GUI to insert event polling into blocking IO calls. But I have not been able to figure out a clean way of using this mechanism together with Stefano's event handling and Tcl's (never mind wxWindows).

MacOS also does not have proper signal handling, but again interrupts can be translated into events and these are processed within blocking GUSI calls.

One tricky issue on the Mac relating to the use of multiple GUI'S is that Classic MacOS has a single menu bar that is owned by the application in foreground. Each GUI toolkit tends to think it is the only one and hence it controls the menu bar. This alone makes having multiple GUI's in the same process difficult.

4.4 Implementations To Consider

In terms of places to look, I think Tcl/Python/Perl are all worth a look. Tcl in particular has dealt with getting Tk events and async IO to play together on UNIX at least; I'm not as sure about the other two.

Another possibility is wxWindows 2.0 [66]. This has socket support and thread support; I believe they also have at least http protocol support. I believe their Windows and UNIX ports are in fairly good shape; the Mac one I'm not so sure about. Don't know about the license.

Another system I plan to take a look at is the Rice scheme system [16]. This system uses (a modified version of) wxWindows, is supported on UNIX/Windows/MacOS, has socket support, has user-level thread support, has a defined interface for integrating (fake) blocking calls with the event loop and its threads.

4.5 A Note On Java

Just in case anyone thinks all would be simple if we just used Java: Java has its own little issues. There is a discussion of this in Oaks and Wong [41, pages 82–85]. A brief summary: Java has no asynchronous IO – you are supposed to use threads. Suppose thread A does a blocking IO call. Java does provide in thread A an interrupt message that thread B can call. But the specs don't specify how this interacts with blocking IO calls. As a result, on some VM's it terminates the IO call and throws an exception (which one also varies among 1.1 VMs but was standardized to `InterruptedIOException` on Java 2), but on other VMs it does not. To be safe, it is recommended that you close the stream being read or written. Exactly how you do this depends on whether you want to work around a bug in Solaris 2.6 or not. Have a look at Oaks and Wong if you have it handy–its a sobering read. And keep in mind that this business of closing the stream only works if you can get your hands on it: if you have a nice friendly interface that lets you do something like

```
  sock.open("http://fred.frog")
```

and the DNS lookup under the hood hangs....

At least with real multi-threading in Java you should have thread B continuing to run even if thread A is blocked indefinitely.

4.6 A Strategy for GUI/IO Management

At this point the best approach so far for something that makes sense across all three platforms seems to be this:

- Have R and the GUI(s) run in separate threads of execution or processes.
- Have these processes communicate through a message-passing inter-process communication (IPC) mechanism.
- The IPC mechanism works on top of an IO mechanism that can be integrated with other IO channels.

The threads of execution can be OS threads within a single OS process where that makes sense or they can be separate OS processes. On UNIX/pthreads using threads should be OK (but raises challenges of figuring out how to configure use of the right libraries); on Win32 it should also be OK; on Classic MacOS the thread system combined with the single event queue is not adequate–I think you have to use separate processes there.

The system dependency of the IPC mechanism will be isolated in the transport layer it uses and how multiple IO sources are registered. On UNIX, communication will go over file descriptors, say using pipes, and IO multiplexing goes through select. On Win32, pipes or other kernel objects can be used along with MsgWaitForMultipleObjects. On MacOS we can use either Apple events, GUSI PPC sockets, or a combination.

4.7 A Sample Implementation

As a proof of concept I put together an alternate version of the `tcltk` package. The package is called `ipctcltk`[1]. It's `.First.lib` does

```
attach(NULL, "package:tcltk")
```

to make it look like the real thing is loaded so that the `tkdens` and `tkcanvas` demos will work with this version.

Installing the package installs a separate server program[2] linked to Tcl/Tk and my IPC code and a shared library for the R side, also linked to the IPC code. Loading the package loads the shared library and fires up the server process. The R process and the tcltk server then communicate over pipes. This sort of thing is vaguely similar to the way Mathematica used to work, at least conceptually, with a front end and a kernel (I don't know if it still does or if it ever really did). It also is similar to S little brown book extension functions. If/when we settle on a single GUI it may make sense to have the GUI start R rather than the other way around, but this isn't a major difference—except for startup and shutdown the relation is fairly symmetric.

The IPC stuff is isolated into a module that links up two processes and allows each to post messages to the other (synchronously or asynchronously; at the moment I am just using synchronous posting). The idea is to to isolate most system dependencies here. I have only done the implementation for UNIX/pipes but am fairly sure it can be done on Win32 and Classic MacOS too. The only other system dependency is how to register the IPC mechanism for event multiplexing. On the GUI side, each system provides some mechanism for registering an event source, and that will need to be used. On the R side I currently just do what the standard tcltk package does. Eventually we should probably provide something like Tcl's `Tcl_CreateFileHandler` that allows a procedure to be registered for an event source. The representation of the event source will differ from one OS to another, but it should be possible to manage an interface something like

[1] See URL `http://www.stat.umn.edu/~luke/R/tcltk`.

[2] I used a fairly grizzly hack in `Makevars` to get the server executable built and installed. With this hack I was not able to get the `SHLIB_EXT` to be set from config info so I hard wired it to so–on HPUX you're out of luck.

```
R_CreateEventSource(R_EventSource s, R_EventCallback p);
#if defined(UNIX)
typedef int R_EventSource; /* a file descriptor */
#elif defined(Win32)
typedef HANDLE R_EventSource; /* a kernel object */
#elif defined(Macintosh)
typedef int R_EventSource; /* a (GUSI) file descriptor */
#endif
```

At the moment this package is only intended as a proof of concept. I am sure it is still riddled with bugs. (At a minimum I have not bothered to turn off some debugging output that gets printed at times.)

5 Threads and GUI's

Conceptually, when we arrange to process GUI events while at the same time processing a command line we are creating two concurrent threads: one that runs the GUI event loop,

```
while(GetEvent(&event)) ProcessEvent(event);
```

and one that runs the Read-Eval-Print loop (REPL).

```
Print(Eval(ReadAndParse())))
```

We may not use a thread mechanism to implement this, but that is in effect what we get. Treating them explicitly as threads helps clarify issues like how option setting and other dynamic state affect the different computations.

Up to a point these threads can operate completely independently. But suppose a command is given in the REPL that asks for a new window. There are two ways this can work:

I The code to create the new window is executed in the R REPL thread. To insure safety this will require acquiring a lock on the GUI (or at least on parts of it). Figure 1 (a) illustrates this.

II The code to create the new window is executed by the GUI thread. This requires some sort of message passing mechanism. This is shown in Figure 1 (b).

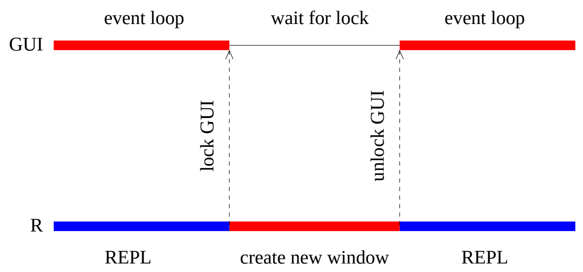
In terms of the details of how things run, the two approaches are not really very different if the threads are light weight and run on a single OS thread. But they are quite different if two separate OS threads are involved.

Approach I seems simpler; locking is a bit of an issue but can be hidden for the most part. However some OS's and some GUI libraries create problems if these conceptual threads actually run on separate OS threads. On Win32 when a new window is created on a particular OS thread, then the creating thread is recorded in the window and all Windows messages are posted to that thread's message queue. This means that I just will not work: the windows is created in the REPL thread, so its messages go to the REPL thread. But the message loop is running on the GUI thread and will never see them.

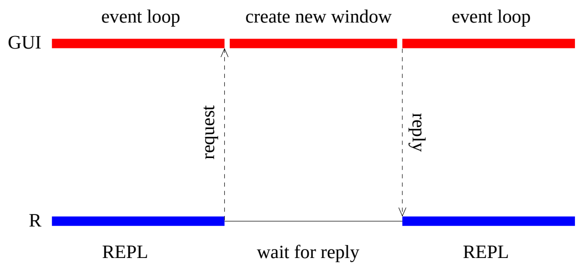
Another case is Tcl. Tcl has support for threads but, according to the `Thread` man page:

> An important constraint of the Tcl threads implementation is that *only the thread that created a Tcl interpreter can use that interpreter*. In other words, multiple threads can not access the same Tcl interpreter. (However, as was the case in previous releases, a single thread can safely create and use multiple interpreters.)

Again this means that II would need to be used with Tcl/Tk.



(a) Case I: Creation in the REPL thread with a lock on the GUI



(b) Case II: Creation in the GUI thread requested with a message

Figure 1: Two approaches to handling creation of a new window requested in the R Read-Eval-Print loop (REPL) when the REPL and GUI run in separate threads.

6 Threading Design Space

There seem to be three dichotomies:

1. Do we use one OS thread (S) or multiple OS threads (M)? Here OS can mean something like Linux or Win32 or it can mean a virtual OS like the JVM or the .NET framework.

2. Do we support only one high-level language (HL) thread per OS thread (S) or do we allow for multiple HL threads per OS thread (M) (these are often called something like light-weight threads or micro threads).

3. Do we allow only one OS thread at a time to execute HL code (S) or can multiple OS threads execute HL code simultaneously (M)? The M option is only available if the answer to the first question is M (multiple OS threads). Only the M option here leads to parallelism.

This gives the following possible combinations:

SSS This is the single-threaded option, no concurrency.

SMS A single OS thread runs multiple HL threads. This is the design used by DrScheme [16], concurrent Haskell [28], concurrent ML [49], Java with green threads, and many others.

MSS Each HL thread corresponds to an OS thread. But only one HL thread may run at a time. This is the Python model. Python [45]uses a global interpreter lock; only the thread that holds the lock may run Python code. This may well be the easiest approach to get working (which is probably why the Python folks chose it).

MMS Multiple OS threads are used and each can run multiple HL threads, but only one OS thread at a time may run HL code. This is what Python will have if micro threads [64] are made a standard part of the system. This allows for light weight threads while at the same time simplifying the handling of blocking.

MSM Multiple OS threads that may run in parallel. Only one HL thread per OS thread. This is the simpler of the two combinations that provide parallelism on multi-processor. In theory this is what the Sun JVM does, though the synchronization required by the memory manager may limit the amount of real parallelism that is achieved.

MMM Multiple OS threads that may run in parallel, each running multiple HL threads. Parallelism with light weight thread support.

6.1 Parallelism Through HL Threads: The MXM Options

Parallelism through HL threads is only achievable by choosing MSM or MMM. In the longer run this is probably a desirable goal, but I don't think it is realistic in the short run. The reason is the need for synchronization. Here are some of the things that need to be synchronized:

- The memory manager. This would not be a our responsibility with an implementation for the JVM or .NET but is for a C implementation.
- Accessing of environments. This would require reader/writer locks to insure integrity of the environment structures except in cases where it is known that the environment cannot be accessed in multiple threads.
- Promises. Promises can be accessible to several threads. Locking is needed to insure they are only forced once.
- Reference counts. In R each object has a `NAMED` field. These are two bit reference counts that are used to limit the amount of copying that is done to preserve the call-by-value illusion. It *may* be possible to insure that all objects that might be accessible to more than one thread will have `NAMED` values of 2, in which case locking is not needed, but I'm not sure this is possible. In addition, the fact that the `NAMED` field is a bit field in a larger structure means certain assumptions of atomicity need to be made for the contents of the field always to be reliable without locking. This may all be OK, but I doubt it.
- Various other data structures like graphics devices, the options list, and so on.

To allow parallel threads to work we would need to correctly identify and protect all things that need synchronization protection. This in itself is a very tall order. And when we are done, unless we can justify not protecting environments and `NAMED` fields, we are quite likely to end up with something that, because of the synchronization overhead, is significantly slower than the current sequential version even when there is only one running thread.

Perhaps I am being overly pessimistic, but I don't think this is a realistic approach. Others seem to agree, as reflected by the comments in the concurrent Haskell paper, for example. Java hype seems to suggest that JVM's are now able to run in parallel, but researchers are only now starting to publish results on GC methods designed for Java in parallel thread environments [44, 15], so this should probably be taken with a grain of salt.

6.2 Light-Weight Threads: The XMX Options

Operating system threads tend to be expensive resources. Creating one is time consuming, they take up a significant amount of memory, and sometimes also use up other important resources. For example, on Linux each thread takes up an entry in the process table, and the process table usually has something on the order of 1000 slots.

Some approaches to the use of concurrency require the efficient creation of possibly very large numbers of threads. There are languages where each *active object* that is allocated is given its own thread. Some approaches to GUI design attach an event loop thread to each GUI object that can respond to interaction instead of using a callback [23, 52].

Even if we do not go so far as to separate the event loop into separate threads for each interacting device, running a single thread for the event loop has some problems. For example long-running callback can block the entire GUI. When this is known to be an issue one can spin off a separate thread. But it might be useful to do this automatically. This isn't entirely trivial since some things do have to be processed in order and some, such as a sequence of mouse motion events, should be compressed if the event handler falls behind. but this is something that could be handled under the hood with multiple light-weight threads and suitable synchronization.

Another issue is GUI debugging. If you enter a debugger from an error in a GUI callback and that debugger itself needs access to the GUI, then a deadlock may occur. Having light-weight threads would allow a new thread to be created for handling the debugger's GUI needs.

None of these examples are 100% compelling to me at this point, but it would be nice to be able to play with these ideas.

There is of course one setting where light weight threads are essential: If we want concurrency and do not have OS threads available or do not want to use them for some reason. Most OS's have threads of one form or another (even Classic MacOS has a cooperative thread system, which is the best you can hope for since the whole OS is based on cooperative multi-tasking). However, getting the right configuration setting for using the right thread-safe libraries is likely to be a bit of a platform-dependent nightmare even within UNIX systems. So there is at least some reason to think hard about relying on multiple OS threads.

An issue with light-weight threads is handling blocking. Any call to a C function will have to block (to some degree at least; there may be the option of running some threads from within cooperative C code or C code that calls back to the HL). Blocking IO operations will also block unless these are turned into asynchronous IO operations that are then handled internally by `select` calls. All the systems I mentioned that use the SMS approach deal with this in more or less the same way. Here is the corresponding bit[3] from the Rice Scheme system documentation:

> **Blocking the Current Thread.** Embedding or extension code sometimes needs to block, but blocking should allow other MzScheme threads to execute. To allow other threads to run, block using `scheme_block_until`. This procedure takes two functions: a polling function that tests whether the blocking operation can be completed, and a prepare-to-sleep function that sets bits in `fd_sets` when MzScheme decides to sleep (because all MzScheme threads are blocked). Under Windows and BeOS, an `fd_set` can also accommodate OS-level semaphores or other handles via `scheme_add_fd_handle`.
>
> Since the functions passed to `scheme_block_until` are called by the Scheme thread scheduler, they must never raise exceptions, call `scheme_apply`, or trigger the evaluation of Scheme code in any way. The `scheme_block_until` function itself may call the current exception handler, however, in reaction to a break (if breaks are enabled).

For R this would involve moving the context stack out of the C stack and onto the heap. This may have some other advantages, for example it might help reduce the need for `setjmp` calls and may also help in cleaning up the exception handling mechanism.

This approach requires dealing with all the conceptual issues of concurrency discussed in Section 3 and also requires the extra effort of stack lifting. For the SMS case it does allow us to avoid dealing with any OS multi-threading issues (which we will wish for at some point if we choose to go the MXX route :-)).

[3] See URL `http://www.cs.rice.edu/CS/PLT/packages/doc/insidemz/node29.htm`.

6.3 Multiple OS Threads Running One At A Time: MSS

This is the Python solution and *may* be the best option for R. It does require dealing with multiple OS threads and some of the headaches that go with that. It does require dealing with all basic conceptual concurrency issues. It also requires eliminating all global variables that are used to maintain dynamic state and either moving them to thread-local state or arranging for them to be saved/restored when a thread releases and acquires the global lock. The context stack is a particular case in point, but there are others.

Running one thread at a time in the HL subsystem means that blocking can occur. On the other hand if we know a call to `foo` may take a long time, then we can do something along the lines of

```
ReleaseGlobalLock();
foo(...);
GrabGlobalLock();
```

and other OS (and hence HL) threads will be able to run while `foo` is running. It doesn't matter whether `foo` runs actively or blocks on IO. If `foo` calls back into the HL layer, then it will need to acquire the lock to do anything useful there.

6.4 Variations on OS Threads

It this discussion I have been thinking of OS threads of the preemptive variety as provided on Win32 and most Pthreads implementations. There are alternatives.

For UNIX systems there are several user-level threads packages; one example is GNU Pth [18, 19]. On Win32, there is a user-level threading support mechanism called Fibers [50, Chapter 12].

6.5 SMS or MXS: Which To Choose?

The choices seem to be SMS, MSS or MMS. The XMS choices require figuring out how to support light weight threads, which will be a bit of a challenge but one that may fit in well with byte code compilation. The advantage is that they allow us to play around with some advanced concurrency models that use lots of small threads. Whether this is necessary is unclear, but it could be an interesting research direction.

The MXS options require us to byte the bullet of OS threads, with all the locking and configuration issues that go with them. We could avoid the configuration and locking issues by using something like GNU Pth on UNIX and Fibers on Win32; MacOS threads are already cooperative. But this would leave us with managing blocking IO via `select` and the like. The chief benefit of multiple real OS threads is that they significantly simplify the handling of blocking and, for MSS they allow us to get concurrency without worrying about light-weight threads.

I think I would like to spend some time working on light weight threads to see if we can make something work. But it probably makes sense to simultaneously explore the MSS option. If both succeed we end up with MMS. If we can't get light weight threads to work, we still have MSS.

In terms of implementing any of these, probably at least 80% of what needs to be done is common to all three. For the remaining 20% we could work on SMS and MSS in parallel, perhaps initially using a user-space thread system like Pth.

7 Light-Weight Thread Implementation

To support multiple HL threads on a single OS threads means we need some way of suspending a thread and saving its execution context. There are a number of different possibilities with different trade-offs, and some can be used in combination.

There are calls `makecontext`, `getcontext`, `setcontext` and `swapcontext` in the Unix 95/98 specification that do just what is needed. Unfortunately they are not yet implemented on many systems (such as current Linux distributions). An alternative is careful use of `setjmp` and `longjmp` together with `sigaltstack`, also Unix 95/98 but more widely implemented (only in Linux from 2.2 on though), or the BSD `sigstack`. This is used in Pth [18, 19], and we could use what is provided there or adapt it if necessary.

Another interesting line of work to watch is C−− [8, 48, 47]. This is an effort to develop a target language for compilers for high-level languages. At this point there do not seem to be any solid implementations available yet, and in any case the design is still evolving.

One approach used by Scheme implementations is to save the current C stack and restore it when the thread is resumed. This cannot be done 100% portably, but it is something most Scheme implementations do, so we can look at their code to see how it is done. The Rice Scheme system [16] is one example; others are referenced at [53]. It does however tend to lead to somewhat expensive context switches and threads that might better be describes a medium-weight rather than light weight. Typical of the sort of issues this raises is the following from the Rice Scheme documentation:[4]

> **Integration with Threads.** MzScheme's threads can break external C code under two circumstances:
>
> – *Pointers to stack-based values can be communicated between threads.* For example, if thread A stores a pointer to a stack-based variable in a global variable, if thread B uses the pointer in the global variable, it may point to data that is not currently on the stack.
>
> – *C functions that can invoke MzScheme (and also be invoked by MzScheme) depend on strict function-call nesting.* For example, suppose a function F uses an internal stack, pushing items on to the stack on entry and popping the same items on exit. Suppose also that F invokes MzScheme to evaluate an expression. If the evaluate on this expression invoked F again in a new thread, but then returns to the first thread before completing the second F, then F's internal stack will be corrupted.
>
> If either of these circumstances occurs, MzScheme will probably crash.

Another approach, that can be used on its own or in combination with some of the ones already mentioned, is to eliminate the use of the C stack by the evaluation mechanism (except where the HL calls C and C calls back to the HL). This may also involve or facilitate conversion of the interpreter to a tail-recursive form. For Python this sort of thing is being done in the Stackless Python project [61]. The concurrent Haskell paper also discusses this. How to handle callbacks from C into the HL is not completely clear. Both [61] and [28] discuss this issue; I need to think this through a bit more to see if I am happy with their solutions. Other papers on the Glasgow Haskell runtime may also be relevant [43, 29].

I will try to add a bit more on this in the next few days (before the DSC meeting).

8 Other Issues

All the discussion so far has been in terms of the low level interface. There of course is also a need to think about the user-level view we want to provide.

[4]See URL `http://www.cs.rice.edu/CS/PLT/packages/doc/insidemz/node29.htm`.

8.1 High-Level GUI Interfaces

Whatever toolkit we end up using for implementing GUI things (Tk, wxWindows, ...) we probably don't want to indefinitely provide just a raw interface like the current `tcltk`. It would be best to provide a higher-level layer on top of that, both to allow us to switch between toolkits if necessary and to provide an easier programming model.

Both Java and Tk provide a good starting point. There are a range of other approaches. One very high level functional one built on Tk is FranTk . This builds to some degree on an idea called functional reactive programming [63] and also on TkGofer [62].

Other functional approaches are Haggis [23, 52] and the GUI libraries for Concurrent Clean [11].

Qt has an interesting way of handling wiring together with slots and signals. the MS .NET framework has events and delegates. In some ways these are intended to help implement Model-View-Controller approaches [31] (on line through CiteSeer [6]), something Swing also uses heavily to allow it pluggable look-and-feel to work,

Concurrent ML also has some user interface ideas related to its concurrency model [49].

There are a number of ideas in the CMU HCI work on Amulet [39]; also a paper on future directions that may be worth a look. [38]

The Self system introduced a user interface approach called Morphic which has also been adopted by the Squeak Smalltalk system [56, 36, 54].

Some Common Lisp systems support the Common Lisp Interface Manager [13]. LispWorks has a layer called CAPI that I think sits under CLIM [10]. Dylan has a similar system called DUIM [17].

8.2 High-Level Thread Interfaces

For thread interfaces, there is Duncan's thesis [33] and R threading notes [32].

Java has a threading API that is worth looking at. Erlang also has some interesting ideas [20, 3]

There is currently a proposal for threading support in Scheme under discussion. [21, 22]. This is part of the Scheme Request for Implementation (SRFI) setup [57].

Languages that use very light weight threads include CML [49] and Concurrent Haskell [28].

Many Lisp systems have some form of threading interface; documentation for the one for Franz Allegro CL is available on-line [2].

Objective Caml has a threads interface [42].

There are lots more, but these are a start.

8.3 High-Level Streams Interfaces

Most Common Lisp systems that provide a flexible and extensible streams system for extending standard CL streams [12] use the Gray streams proposal [25]. The CLOCC [14] project seems to be building on this also.

8.4 Completely Random Stuff

Some discussion of exceptions in lazy functional languages is in [30].

In interesting use of first class continuations to model web browsing [46].

One idea perhaps worth thinking about is a memory profiling tool [55].

If we do need to get into concurrent garbage collection, there has recently been a fair bit of research activity that should be reviewed [9, 59, 5, 34, 4, 37, 35, 26].

References

[1] Advanced, alternative, asynchronous resolver. http://www.chiark.greenend.org.uk/~ian/adns/.

[2] Thread support in franz allegro common lisp. http://www.franz.com/support/documentation/5.0.1/doc/cl/multiprocessing\%.htm.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996.

[4] Alain Azagury, Elliot K. Kolodner, Erez Petrank, and Zvi Yehudai. Combining card marking with remembered sets: How to save scanning time. In *Proceedings of the International Symposium on Memory Managementc*, pages 10–19, Vancouver Canada, October 1999.

[5] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, volume ACM, pages 104–117, Atlanta, GA USA, May 1999.

[6] Kurt Bollacker and Lee Giles. Researchindex (citeseer): The NECI scientific literature digital library. http://citeseer.nj.nec.com/.

[7] David R. Butenhof. *Programming with POSIX Threads*. Addison Wesley, 1997.

[8] C−−. http://www.cminusminus.org.

[9] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 264–273, Vancouver, BC Canada, June 2000. ACM.

[10] Capi user guide. http://www.xanalys.com/software_tools/reference/lwu41/capiuser/CUG_1.HT\%M.

[11] The clean language. http://www.cs.kun.nl/~clean.

[12] Common lisp hyperspec. http://www.xanalys.com/software_tools/reference/HyperSpec/HyperSpec-4-0\%.tar.gz.

[13] Common lisp interface manager (CLIM). http://www.alu.org/table/references.htm\#clim.

[14] The common lisp open code collection. http://clocc.sourceforge.net.

[15] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Eliot E. Salant, Katherine Barabash, Itai Lahan, Yossi Levanoni, Erez Petrank, and Igor Yanorer. Implementing an on-the-fly garbage collector for java. In *International Symposium on Memory Management*, pages 155–166. ACM, 2000.

[16] Rice scheme system. http://www.cs.rice.edu/CS/PLT/packages/drscheme/.

[17] Building applications using duim. http://www.fun-o.com/products/doc/dguide/index.htm.

[18] Ralf S. Engelschall. GNU Pth – the GNU portable threads. http://www.gnu.org/software/pth/.

[19] Ralf S. Engelschall. Portable multithreading. In *USENIX Annual Technical Conference*, June 2000.

[20] Erlang. `http://www.erlang.org/`.

[21] Marc Feeley. Srfi 18: Multithreading support. `http://srfi.schemers.org/srfi-18/srfi-18.html`.

[22] Marc Feeley. Srfi 21: Real-time multithreading support. `http://srfi.schemers.org/srfi-21/srfi-21.html`.

[23] S. Finne and S. L. Peyton Jones. Composing haggis. In *Proc. 5th Eurographics Workshop on Programming Paradigms in Graphics*, Maastricht, September 1995.

[24] FranTk. `http://haskell.cs.yale.edu/FranTk/`.

[25] Common lisp streams proposal. `ftp://parcftp.xerox.com/pub/cl/cleanup/mail/stream-definition-by-user.m\%ail`.

[26] Lorenz Huelsbergen and Phil Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In *Proceedings of the International Symposium on Memory Managementc*, pages 166–175, Vancouver Canada, October 1999.

[27] Intel. Math kernel library. `http://developer.intel.com/software/products/mkl/`.

[28] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, St Petersburg Beach, Florida, January 1996. ACM.

[29] Simon L. Peyton Jones and Simon Marlow. The new GHC/Hugs runtime system. `http://www.research.microsoft.com/users/simonpj/Papers/papers.html\#comp\%iler`, 1998.

[30] Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, volume ACM, pages 25–36, Atlanta, GA USA, May 1999.

[31] G. Krasner and S. Pope. A cookbook for using the MVC user interface paradigm in smalltalk. *Jourlan of Object-Oriented Programming*, 1(3):26–49, 1988.

[32] Duncan Temple Lang. R threading ideas. `http://franz.stat.wisc.edu/˜rdevel/RThreads`.

[33] Duncan Temple Lang. *A Multi-Threaded Extension to a Hich Level Interactive Statistical Computing Environment*. PhD thesis, University of California, Berkeley, 1997. `http://cm.bell-labs.com/stat/doc/multi-threaded-S.ps`.

[34] Martin Larose and Marc Feeley. A compacting incremental collector and its performance in a production quality compiler. In *Proceedings of the International Symposium on Memory Management*, pages 1–9, Vancouver Canada, October 1999.

[35] Tian F. Lim, P. Pardyak, and Brian N. Bershad. A memory-efficient real-time non-copying garbage collector. In *Proceedings of the International Symposium on Memory Managementc*, pages 118–129, Vancouver Canada, October 1999.

[36] John Maloney. Tutorial: Fun with the morphic graphics system. `http://www.squeak.org/tutorials/morphic-tutorial-1.html`.

[37] Luc Moreau. Hierarchical distributed reference counting. In *Proceedings of the International Symposium on Memory Managementc*, pages 57 – 67, Vancouver Canada, October 1999.

[38] Brad Myers, Scott Hudson, and Randy Pausch. Past, present and future of user interface software tools. `http://www.cs.cmu.edu/~amulet/papers/futureofhci.ps`.

[39] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, 1997.

[40] Matthias Neeracher. Gusi. `ftp://sunsite.cnlab-switch.ch/software/platform/macos/src/mw_c`.

[41] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly, 2nd edition, 1999.

[42] Objective Caml. `http://caml.inria.fr/ocaml/`.

[43] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[44] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collect. In *International Symposium on Memory Management*, pages 143–154. ACM, 2000.

[45] The python language. `www.python.org`.

[46] Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 23–33, Montreal Canada, September 2000.

[47] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 285–298, Vancouver, BC Canada, June 2000. ACM.

[48] Norman Ramsey and Simon Peyton Jones. Featherweight concurrency in a portable assembly language. `http://www.cminusminus.org/abstracts/c--con.html`, 2001. submitted to PLDI 01.

[49] John H. Reppy. *Concurrent Programming in ML*. Cambridge Univ Press, 1999.

[50] Jeffrey Richter. *Programming Applications for Microsoft Windows*. Microsoft Press; I, 1999.

[51] Meurig Sage. FranTk – a declarative gui language for haskell. In *Proceedings of Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000*, pages 106–118, Montreal, Canada, September 2000. ACM, ACM Press.

[52] Meurig Sage and Chris Johnson. Interacting with haggis: Implementing agent based specifications in a functional style. In S Howard, J. Hammond, and G. Lindgaard, editors, *Human Computer Interaction – INTERACT 97*, pages 126–133, Sydney, Australia, July 1997. Chapman and Hall.

[53] Schemers.org. `http://www.schemers.org/`.

[54] The self home page. `http://www.sun.com/research/self/index.html`.

[55] Manuel Serrano and Hans-J. Boehm. Understanding memory allocation of scheme programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 245–256, Montreal Canada, September 2000.

[56] The squeak homepage. `http://www.squeak.org/`.

[57] Scheme requests for implementation. `http://srfi.schemers.org/`.

[58] W. Richard Stevens. *UNIX Network Programming: Networking APIs – Sockets and XTI*, volume 1. Prentice Hall;, 1997.

[59] Elliot K. Kolodner Tamar Domani and Erez Petrank. A generational on-the-fly garbage collector for java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 274–284, Vancouver, BC Canada, June 2000. ACM.

[60] Guignon Thomas. BLASTH, a BLAS library for dual SMP computer. `http://www.usenix.org/publications/library/proceedings/als2000/full_pap\%ers/thomas/thomas_html/`.

[61] Christian Tismer. Stackless Python. `http://www.stackless.com`.

[62] TkGofer. `http://www.informatik.uni-ulm.de/pm/ftp/tkgofer.html`.

[63] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 242–252, Vancouver, BC Canada, June 2000. ACM.

[64] Will Ware, Christian Tismer, Just van Rossum, and Mike Fletcher. Python microthreads. `http://world.std.com/˜wware/uthread.html`.

[65] Limsoon Wong. The functional guts of the Kleisli query system. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 1–10, Montreal Canada, September 2000.

[66] wxWindows. `http://www.wxwindows.org`.