Interprocess Communication and Timing in Real-time Computer Music Performance

Miller Puckette
MIT, IRCAM

More and more frequently we hear that real-time computer music performers would like to be able to do "high-level" real-time programming. What this means, I think, is that they want to be able to run algorithms which might require so much CPU time that they can't be run uninterruptibly; higher-priority things need to be able to steal the processor. Just doing these things in hardware interrupt routines is not sufficient; that polarizes the world into the "urgent" and the "not urgent", whereas the latency/CPU-time combinations of tasks in a real application often cover a much wider spectrum. Moreover, few computer musicians want to debug interrupt routines. I don't.

An important limitation in real-time systems in general has been the lack of effective ways to combine small systems into larger ones reliably. The problem of resource allocation becomes hard when there are conflicting real-time constraints on the objects requesting the resources. The need for intercommunication places yet other restrictions on the execution of tasks. Large real-time systems which require a high level of interaction among tasks, especially through communication or resource-sharing, are hard to develop and verify. And this is just the sort of system that computer musicians now want to build.

It is time to build a general real-time scheduler which makes these things easy to do. In fact, several systems exist which solve some of the problems; the main missing link now is context-switching. The smartest systems are machine-independently written (i.e. they make no reference to clocks, system calls, memory segments, and the like) and are based on message-passing to allow the most complete modularity we know how to put in a software system so far.

Central to current efforts to support more highly developed real-time software environments is to find an interobject real-time protocol to handle communication and resource contention, in a way which allows confidence that software will act as specified without unpleasant surprises in situations the designers did not forsee. The challenge is to find a set of intercommunication and scheduling primitives simple enough to make the verification problem tractable, yet powerful enough not to place great restrictions on the range of possible applications.

I would regard anything else, for example an overall script describing the performance, or a score language, as being optional and selectable by the user. Naturally we ought to provide such things and even make them capable of doing complicated things; but this complexity must never appear in the dealings between objects, only within them. Three other features currently in vogue seem unnecessary. First, there is no point in having a built-in notion of hierarchy; it is usually a hindrance. Second, I would drop the idea of continuously-running processes; they create overhead and anything they do can be done better through I/O - related timing. Third, there should be few defaults. Rather than hide complexity I would keep it visible as an incentive to avoid it altogether.

The MAX real-time system.

Research on real-time systems is ongoing at many computer music centers and a new real-time system is no longer a rarity, but nontheless I would like to mention another one which has been in development, under various names, for about five years and is nearly stable now. MAX (named for Max Mathews) is designed for real-time computer music performances involving many simultaneous digital signal-processing control tasks. It is currently a soft real-time system, making no hard guarantees about meeting its deadlines; there are nonetheless hooks in it which might someday allow making it a "hard" real-time system. It currently runs on the 4X at IRCAM and with MIDI equipment at MIT.

In MAX the tasks are carried out by active objects which communicate through message-passing. I/O is also carried out through message-passing; hence real sensors and actuators appear as other active objects. A message sent from one object to another has an associated start time and deadline. The message causes its destination object to carry out some calculation; this calculation is stipulated to occur no sooner than the start time and no later than the deadline. This calculation is called the receiving object's *method* for the message. It may in turn pass messages to other objects, which have their own start times and deadlines. A message is not required to be delivered immediately when it is sent, but only when the scheduler decides to deliver it, subject to its time constraints. Objects are allowed to have any amount of internal state, but not to share it with other objects.

MAX allows no context-switching from one method to another, whether within the same object or from one to another. Nonetheless, it could easily allow interobject context switching since no resources are shared between objects. This is seen to form the basis of a simple exclusion system: an object can only be doing one thing at a time but there are no exclusion restrictions between objects. We will develop this idea further below; for now, we observe that this is a simple paradigm for describing exclusion constraints which allows a reasonable amount of flexibility.

Objects in MAX are reminiscent of the well-known "monitor" abstraction. The main difference is that instead of waiting for access to a monitor, with its implications for process synchronization, objects send each other messages entirely asynchronously. Later we will see that in certain cases we can allow synchronous message-passing, but not to the level of generality

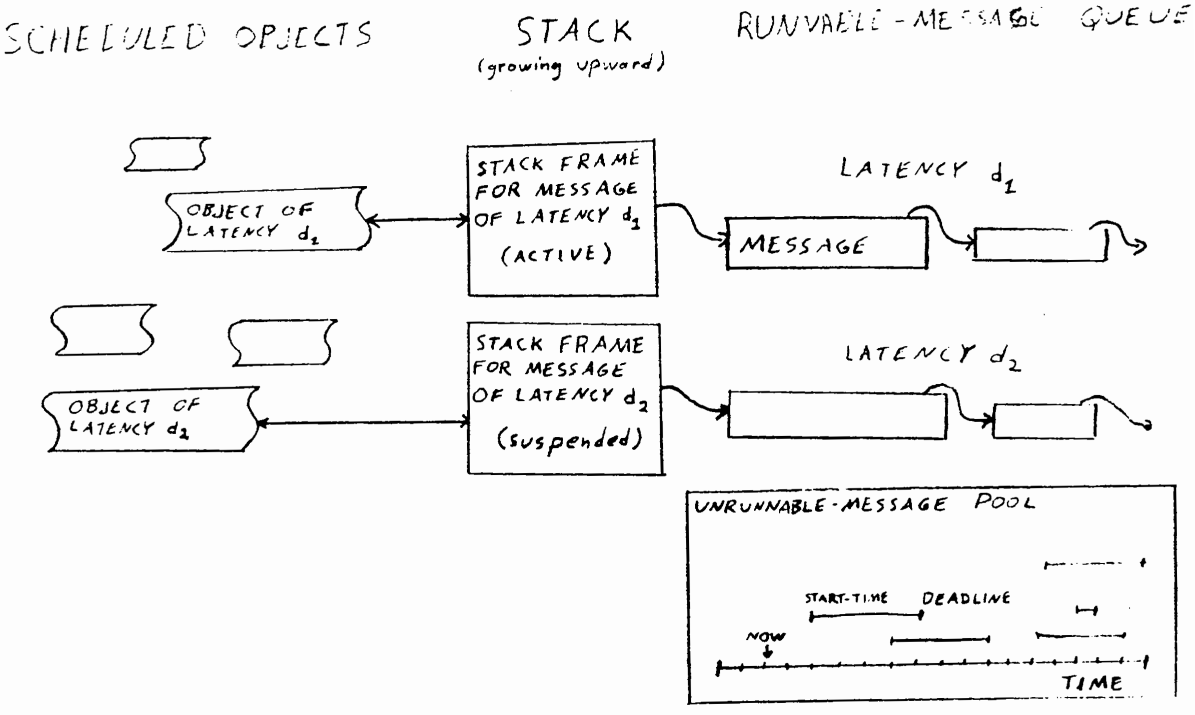

FIGURE.

offered through monitors. This asynchrony is crucial for the design and verification of feasible real-time schedulers.

The power of the MAX paradigm is in its ability to specify more complicated actions from simpler ones. In musical applications, long streams of actions are required which can not only start and stop on command but adjust their tempo and other properties as a result of other stimuli. This kind of scheduling is beyond the scope of stateless graph-based models for real-time computation, and yet process-based implementations of them quickly become very hard to schedule reliably because of the *exclusion problem*, which is that processes which share resources must be able to lock each other out, creating an often intractable system of constraints on the scheduler. I see in MAX a demonstration that the message-passing model offers a good way to approach the exclusion problem in real-time applications.

What a real-time system should be.

MAX in its current state has one major limitation in its ability to carry out tasks according to real-time constraints, which is its lack of context switching. Every time we send a message to an object in MAX the method for that message is carried out from beginning to end, regardless of whether more urgent tasks appear in the meantime. Thus any task, no matter how urgent, faces a possible latency equal to the longest any message could take to be handled. This can be fixed in a way described in the following paragraphs; I will present the ideas in the form of a loose specification for a proposed real-time system I call "X".

Like MAX, the X real-time system would consist of a kernel and a collection of objects which carry out specific real-time tasks. The kernel does nothing but handle I/O requests from objects (sending messages to them on I/O completion) and schedule deferred message-passing between them.

The user starts up X by allocating all the objects he wants (which in turn might allocate others.) At startup the objects do as much memory allocation and file I/O as can be forseen. Whatever I/O requests an object will make of the kernel should be known to the kernel in advance, since they will probably require memory allocation.

A deferred message in X would have a start-time and a deadline as in MAX, but only the start-time would be specified by the sending process; the deadline would be obtained by adding the *latency* of the receiving process. This is in order to ensure that all messages sent to an object have the same latency; why we want this to be true is explained in the next paragraph.

Suppose a method is running whose associated deadline is a second away, when I/O completes and generates a message whose deadline is 1 msec from now. We need to suspend the slow method while the urgent one runs. This urgent method might send other deferred messages in turn, causing a whole tree of method routine calls all of higher urgency than that of the original method. We can do all these context-switches without fear as long as we never pass a message to the original object; its data may be in an inconsistent state since it is already busy with another method. But if we hold to the constant-latency rule any message sent to that object will have a deadline later than the deadline it already has; so we will never want to carry out the method for that method before finishing the suspended processing.

It would be desirable to pass some messages directly, without pooling and scheduling them. This would not only reduce overhead but would allow the receiving object to return information to the sending one. This can be done without violating the mutual exclusion constraint as long as an object never passes a message to another object of higher latency (because such an object might be in the middle of another method which has been interrupted, and passing a new message to it would violate the exclusion constraint), and as long as the scheduler is informed so that it can maintain a software priority which would prevent I/O from causing messages to be delivered to the lower-latency object to which we are sending the "direct" message. We expect that this will be a worthwhile provision, since many messages can be treated in this way, and verification of real-time operation is not complicated much by it.

An implemented system should contain some form of input logging in order to recreate situations in which applications do something unexpected. Input logging is simplified by the discreteness of this scheduler; we need only keep a count of the number of messages the scheduler has delivered before a new input arrives in order to be able to recreate the sequence of events exactly. This and the ease of examining the single stack should make applications easier to debug.

Example.

The "schedule" object would replace the "play" CP in MAX. It takes a stream of "tempo" messages to convert "score-time" to "real-time" to decide when to send the desired messages. It is presented here in non- language.

```
definition of "schedule" object
instance variables:
        score   ;gettable-settable list of beat-tagged "notes";
                ;i.e. tasks to carry out.  These tasks are general
                ;function calls and thus are capable of anything.
        notes-left-to-play
                ;pointer to next item in score
        score-time, real-time, score-per-real
                ;current relationship of score-time to real-time
                ;as the equation of a line in point-slope form
        is-playing
                ;flag to indicate that playback is turned on

method: start (beat-to-start-at score-per-real-to-start-at)
        set notes-left-to-play according to given beat
        set tempo (i.e. score-time, etc)
        turn on is-playing
        send self a "timeout" message.

method: timeout ()
        if is-playing is off ignore the timeout, otherwise:
        play all notes whose play-times (as estimated via
                current tempo) have arrived (i.e. send
                the associated message given in the score.)
        send self a deferred "timer" message to arrive at
                the earlier of 50 milliseconds and the
                calculated play-time of the next note

method: stop ()
        turn off is-playing

method: set-tempo (new-score-time, new-real-time,
                new-score-per-real)
        set the appropriate instance variables.
```

We could get the object above to play midi-style scores as follows. In the score we write,

```
((4 2 send-message piano-player note-on 74 50)
(5.5 2.75 send-message piano-player note-off 74)
(5.5 2.75 send-message piano-player note-on 74 50)
... )
```

where the first number is beat, the second is score-time (i.e. a beat value which has been warped to reflect any tempo and micro-timing information which might be available before the performance). The remainder of the message is what to do. The simplest way to manage this would be to apply a function to a list of arguments, in this case "send-message" to the object "piano-player."

Obviously, there is no restriction to sending the equivalent of "midi" messages; the score could be a bunch of parameter updates or messages to 4x-instrument-driving "unit generators." If you want the utmost generality, just use "eval" and you can do anything you want.

Example at the user level.

Suppose a "user" of X wanted to play a score with an instrument called "piano-player" as above. His instructions might look like:

```
(setq bag (allocate-object play))
        ;allocate a play object
(setq piano-player (allocate-object ...)
        ;set up a piano-player, which converts MIDI messages
        ;to sound
(send-message 'bag 'set-score (read 'piano-file))
        ;read score into play object
(send-message 'bag 'start 0 1)
        ;start playback at beat 0, default tempo
```

Note that there is no "go" instruction as in 4xy, nor a "startup" and "run" phase differentiation as in MAX. We are in fact running in real-time during the whole session; while "bag" is playing along we would be able to allocate and start other playback object, although there would be a gap in the first one while the second was reading its score in from disk. The more forsightful user would do all the allocations first and then start the music. To implement a real-time lisp reader interface as in the example above, we need a read-eval-print loop running during the performance. In fact this would be an object which gets messages when tty I/O completes, whereupon it sends data to the reader.

Graphics.

Obviously it would be nice to have graphical representations of objects, with mouse-driven configuration and message-passing. This could get complicated fast. I think it is possible to segment this off into a object itself, whose job is to display other object which it knows about. The window manager object would open the mouse and the tty keyboard (through the kernel) and the other objects, when they opened their tty connections, would somehow get pointed at the window manager instead of the kernel. Then whenever tty input comes to the window manager, the manager would send it on to whichever object is "current" (selected via the mouse.)

Objects which wish to have tty or graphical output also send it to the window manager which puts it at the right place in the screen. In the case of graphical output, the object might just see a bitmap which is invisibly transformed to the right place and size. It is important that the graphics NOT be considered an integral part of a system, since it is doubtful that such graphics can be made easily portable to future systems. Nonetheless the graphics will be a powerful tool and are worth developing well.

Implementation.

It is of obvious importance to limit the computation overhead associated with the real-time scheduler itself. This overhead generally consists of the costs of computation devoted to making scheduling decisions, context-switching between processes, and communication between processes. In this section we describe a proposed implementation of an earliest-deadline scheduler for a message-passing real-time system; it will be seen that the major cost associated with the scheduler comes from interprocess communication, a situation which suggests that algorithms which work well under this scheduler will also work well in distributed systems.

Deferred messages can appear when another object creates one (while it itself is handling a message) or as a result of I/O completion. They may or may not be immediately runnable, depending on their start-times. The scheduler thus stores the runnable ones and the nonrunnable ones in separate pools; at clock ticks messages in the nonrunnable-message pool may become runnable and are moved to the runnable-message pool, either at interrupt time or as a scheduled task itself.

Suppose the latencies which arise in the system are $d_1<d_2< \cdots d_n$. A runnable task at latency d_i appears as a message which is to be passed to an object of that latency; hence the scheduler keeps a pool of deliverable messages, which we organize according to latency as shown in the figure. A message is passed to an object by creating a stack frame for the object's method for that message and then executing the method as a subroutine.

The scheduler keeps the runnable-message pool in the form of a separate queue for each latency. The scheduler always sends the first message in the lowest-latency nonempty queue. When the associated method returns the scheduler sends another message and so on. The only situation in which we need to interrupt a method before it is done is when I/O (including the clock) causes a lower-latency message to appear. The figure shows the situation in which a message of latency d_1 has appeared while a method of latency d_2 was running. In this case the scheduler causes a software interrupt to occur by pushing a new stack frame onto the stack and executing the lower-latency method. When this method returns (and after all other messages resulting from it whose latencies are less than d_2 have been serviced) we pop the stack back to the prior frame at latency d_2 and resume the associated method.

Note that we do not have the bother and expense of maintaining several stacks. This will keep context-switching overhead lower and greatly ease debugging. The major cost associated with this implementation will be allocating space for messages and copying them to the runnable-message pool; the scheduling decisions themselves are quite inexpensive, especially if the number of different latencies required is small.

At the lowest level possible I/O should be handled inside the kernel; but we would like to work interchangeably with "true" I/O and with the result of some other processing. For instance, if the object "KX88" is parsing MIDI from a KX88 and you want switch number 5 being pressed to result in a message being sent to object "bob" containing the data "bang", one could say, (send-mess 'KX88 'get-IO 'switchdown 5 'bob 'bang) and when the switch was hit bob would get the message (switch-IO-done bang). The reason for the "bang" is so that "bob" can tell which among all the switches bob might have opened was the one that fired. In the case of other I/O (pot motion, for instance) the message is sent with the pertinent data, as in (pot-IO-done bang 319) to indicate that the pot's value has changed to 319.

It is interesting to notice the number of ideas in common between this presentation and that of [Boynton]. I would like to thank Lee Boynton and David Wessel for their part in many animated and illuminating conversations, to which this paper owes much.

Reference.

Boynton, *et al*, "Adding a Graphical User Interface to FORMES", these *Proceedings*.