A Real-Time Operating System for Computer Music

Eric Viara, Miller Puckette

A real-time operating system for computer music

Eric Viara, Miller Puckette

ICMC 90, Glasgow (Ecosse) 1990

The programming environment of the IRCAM Musical Workstation (IMW) is described. The IMW is based on a shared-memory multiprocessor. A real-time operating system, CPOS, provides UNlX-like system calls, as well as specific ones to facilitate real-time interprocessor communication. A distributed program, FTS, running under CPOS, provides real-time message-based scheduling of objects whose functionality (data structure and methods) can be changed dynamically.

An operating system, CPOS and a real-time distributed program, FTS, form the software platform for real-time musical programming on the IRCAM. Station de Travail Musicale (IMW). Graphical applications, such as Animal[1], the sound editor [2], the universal recorder [3], and MAX[4], use CPOS and FTS to create real-time synthesis or control objects and communicate with them.

The IMW archiecture consists of a NeXT host and between two and 24 INTEL i860-based coprocessors (CPs). Each CP has a local memory and direct access to all the other CPs' memories (of course local and non-local access times are different.) The hardware architecture of IMW is more fully described in [5].

CPOS (Co-Processor Operating System) supports general purpose, scientific and real-time applications for Digital Signal Processing, taking advantage of IMW's specific architectural features. CPOS is mainly composed of a kernel running on the CPs and a driver on the NeXT host machine for host/CP communication.

For general-purpose applications, CPOS offers a set of system calls based on a subset of standard UNIX. The system calls include filesystem access (open(), read(...) and memory/task management (sbrk(), fork()...). Using a C or FORTRAN program development environment (compiler, assembler, loader, libraries) for the i860, any portable application can run in the CPOS environment The file system calls are done by a NeXT server via the driver; there is no local file system.

For memory and process management, CPOS offers a set of system calls adapted to the (multiprocessors and local memory) architecture. We can chose which CPU to run a process on and specify a physical zone for memory allocation. For communication between processes, CPOS offers an Inter-Process Communication package based on formatted messages and a package for shared memory management between processes.

The response time in a typical operating system to an external interrupt is about l millisecond (100 to 200 microseconds on a i860) In real-time audio processing where the computation of N samples cannot be more than N/F_0 = N*22 microseconds (for 44KHz), this would clearly not be fast enough. CPOS response time to a an external interrupt is about 30 to 40 microseconds, consisting only of context saving and restoring.

CPOS introduces a real-time mode that can give a certain task priority over all others (many operating systems allow that much), but also forbids the processing of 11 exernal interrupts except the ADC/DAC clock - used only for time reference - and those set by processes running in superpreemptive mode on other processors. Testing for the origin of an external interrupt and processing a clock interrupt take only 0.6 microseconds each. FTS ("Faster Than Sound") consists of an object system together with a collection of classes, defined in C, which can be instantiated from a CP or from the host. A protocol is defined for passing messages beween objects in FTS, either locally or between CPs. FTS also supplies support for

MIDI and sound I/O and reschedulable delayed callbacks.

The FTS object system was designed specifically for real-time music applications. In many respects it is much simpler than most object systems, but it provides a combination of services unique among c-language message systems, that is needed in our context. The most unusual point, among C message systems, is that messages are objects which can be copied and stored, whose arguments are typed. The FTS message system can check the argument types of the message against the types then by the receiving object's method for the message. The typing of message arguments also facilitates transmission across machine boundaries; byte swapping is necessary when passing messages between the NeXT host and a CP. The FTS object system is also unusual (among C object systems) in that one can dynamically install new classes, and (with some care) change a class's instance data structure and/or mehods.

An FTS message consists of a selector, which is a pointer o a symbol, and zero or more yped arguments. The fundamental operation defined for a message is to pass it to an object (figure 1). The caller assembles the arguments for the method into a contiguous data structure and calls FTS's message routine. This routine looks up the receiving object's entry for the message, in a table that the first slot of the receiving object's data structure points to. This entry gives a pointer to the method and an argument type template. The message passing rouine checks the argument types and calls the method.

The FTS task and its relation to the rest of the world is diagrammed in figure 2. The task's inputs all appear as time-tagged queues. Except for the serial input queue and the timeout queue, they all share the same structure. This general queue structure treats messages and sound differently. In each queue slot (the contents of a queue for a specific tick) there is a subqueue of messages, of undetermined length, and a predetermined number of signal buffers. In the case of sound input, the message part of the queue is empty.

For each tick, the FTS carries out its (message and DSP) duty cycle as follows. The task empties out, in sequence, the message contents of each of its queue slots for that tick, passing each message to its destination. (In the case of the serial port and timeout queues, this is not an FTS message pass but a prearranged function call.) Before it processes the tick, it waits until all the queue slots for that tick have been filled, i.e. that the task or device which fills the queue has promised that no more information will be sent for the given queue slot. The queue slot associated with sound input is processed last; instead of looking in the (empty) message portion of the slot, FTS runs the DSP duty cycle for that tick which handles sound production.

References

[1]
Lindemann, E. "ANIMAL-a Rapid Prototyping Environment for Computer Music Systems", ICMC 90.

[2]
Eckel, G. "A Signal Editor for the IRCAM Musical Workstation ", ICMC 90.

[3]
Smith, B. "A Universal Recorder for the IRCAM Musical Workstation ", ICMC 90. [4]
Miller Puckette, "The Patcher", Proceedings, ICMC 1988 (Cologne).

[5]
Lindemann, E., et al The IRCAM Musical Workstation: Hardware Overview and Signal Processing Features", ICMC 90.

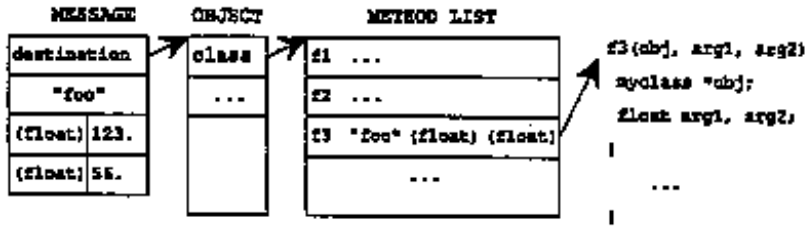

Figure 1. The message system.