Processes in real-time computer music

Miller Puckette
Department of Music
University of California, San Diego
msp@ucsd.edu

Abstract

The historical origin of currently used programming models for doing real-time computer music is examined, with an eye toward a critical re-thinking given today's computing environment, which is much different from what prevailed when some major design decisions were made. In particular, why are we tempted to use a process or thread model? We can provide no simple answer, despite their wide use in real-time software.

Keywords

real time, computer music, processes, parallelism

1 Introduction

The language of real-time computer music borrows from three antecedents that were fairly well in place by 1985, before the field of real-time computer music took its current form. Classical computer music models, starting with Max Mathews's MUSIC program (1957), were well studied by that time. The field of computer science, particularly operating system design, was also taking shape; perhaps it may be said to have matured by 1980 with the widespread adoption of Unix. Meanwhile, a loosely connected network of electronic music studios arose in the 1950s whose design is directly reflected in the patching paradigm that is nearly universal in modern computer music environments.

Both computer science and music practice relied on a notion of parallelism, albeit in very different forms and terms. In computer science, abstractions such as *process* and *thread* arose from the desire to allocate computing resources efficiently to users. In music, thousand-year old terms like *voice* and *instrument* imply parallelism, both on written scores as multi-part music were notated for both practical and dogmatic reasons, and in real time as live performers sang or played the music in ensembles.

In both computer science and computer music language, abstractions modeled on processes or threads are used to try to describe the passage of time and also to express, and/or take advantage of, parallelism. But the aims in computer science (efficiency) are different from those in computer music (as an aid to organizing musical computation).

In the sections that follow I will try to trace these developments historically to see why we treat processes and related concepts in the way that we now do in real-time computer music systems. I hope to help clarify why the current practice is what it is, and perhaps contribute to thinking about future computer music programming environments..

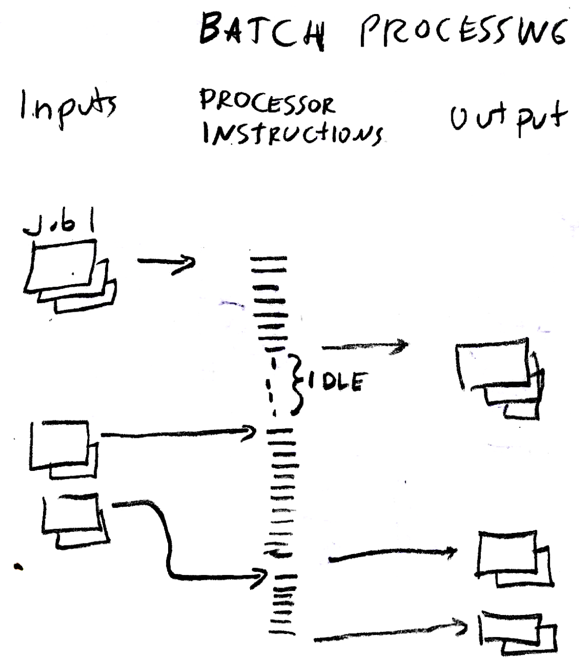

Figure 1: Submitting jobs to a computer circa 1960.

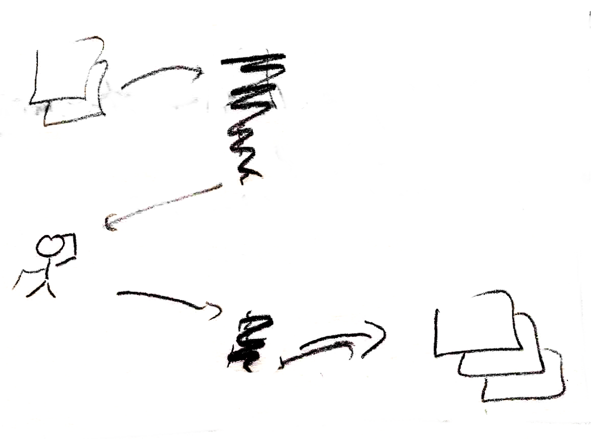

Figure 2: Interactive jobs may stall in mid-computation to ask the operator for more information.

2 Computer science terminology

In classical operating system design theory, the tasks set before a computer were organized into *jobs*. A prototypical computer (fig. 1) sat in a room waiting for jobs to be submitted to it, perhaps in the form of stacks of punched cards. The computer would execute each job in turn, hopefully producing output which could also have been stacks of punched cards. At least three problems arise in this scenario:

- **Idleness.** The computer sometimes had nothing to do and would be idle; idle time reduced the total amount of computation the computer could carry out.
- **Latency.** Sometimes a job would be submitted while another job was running (as in job number 2 in the figure); in this case the job would join a queue of waiting jobs. This meant that the submitter of job 2 had to wait longer to harvest the output.
- **Unanticipated data needed.** For many types of jobs you might not be able to predict at the outset what data will be needed during the computation. The "job" model doesn't offer a way for the computer to ask the operator for additional information it might need.

The first two of these only impact the efficiency of computation, but the third requires that we go back and amend the job model altogether; so we will consider that first. Figure 2 shows an amended model of computation allowing *interactive* jobs that may stop execution part way through and ask the operator for more information. When this happens the job is considered *stalled* and the computer sits idle.

Computer science's answer to the problems of idleness and latency have been to introduce *time-sharing* and *multiprocessing*. Time-sharing is the practice of keeping several jobs in progress at the same time, so that when one job stalls or finishes, the processor's time can then be devoted to some other job that needs running. Perhaps this second job will later stall or finish but meanwhile, too, the first job may have become runnable again (having received a new dose of data it had stalled waiting for). The computer would then return to job 1. One could also fill idle time by keeping low-priority jobs waiting in the background (ones whose latency requirements were less strict) that would run whenever all higher-priority jobs were stalled.

The advent of multiprocessors made it possible to further improve throughput in the same way that having several short-order cooks in a diner can speed orders. As the number of jobs and the number of available processors increases, there should be fewer wild swings in the availability of processing power to satisfy the needs of submitted jobs.

The chief tool for time-sharing and multiprocessing is an abstraction called a *process*, which can be thought of as a virtual computer. When a job is submitted, one creates a brand new (virtual) computer to carry it out, and once the job is finished, the virtual computer, or process, is scrapped. Each job may run in ignorance of all other jobs on the system. Each process gets its own memory and program to run, and its own *program counter*, abbreviated *PC*, that records where in the program the computer is now running. When the computer switches from running one process to another one, the memory and PC (and other context) of the first process are retained so that they are available again when the first process is again run in the future.

Although at the outset we could consider all processes to operate in complete ignorance of each other, at some point the need will certainly arise for processes to intercommunicate. Computer science offers at least two paradigms that we will want to consider: *message passing* and *shared memory* (see fig. 3). Of these, the message passing paradigm is less general but easier to analyze and make robust. In message passing, one process can simply send another a packet or a stream of data, that the second one may read at any later time. This is similar

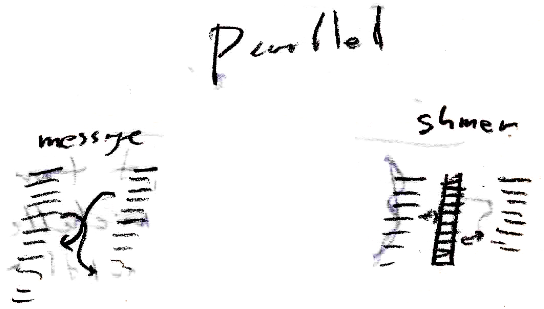

Figure 3: Process intercommunication using messages and shared memory.

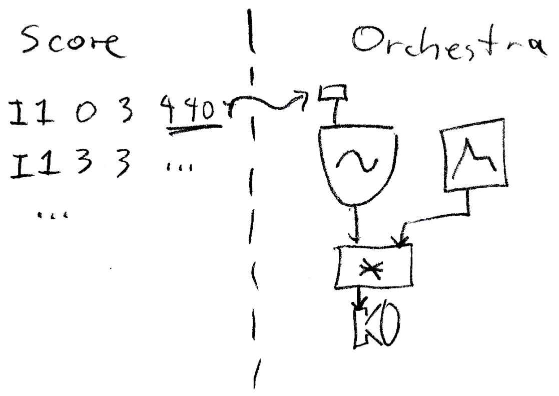

Figure 4: The Music N paradigm

conceptually to how people intercommunicate. The chief difficulty using this paradigm is that it does not allow a process to interrogate another directly, except by sending a message and then stalling until a return message is received. This might greatly increase the latency of computations, and worse yet, if we adopted this strategy for interrogation, two processes could conceivably interrogate each other at the same time, so that both end up deadlocked.

In the shared-memory paradigm two processes communicate by reading and writing to a shared area of memory. We can then arrange for one process to be able to interrogate another one simply by looking in the appropriate location in its memory (which, by prior arrangement, we had arranged to share). But now we have to work hard to make sure that our two processes will carry out their computations deterministically, because the order in which the two access the shared memory is not controlled. We would need to set up some convention to manage this. (One such convention could be to format the shared memory into message queues, thus returning us to the situation described in the previous paragraph.) In general, there is no final answer here; any paradigm will either be onerously restrictive or dangerously permissive, or both, and to make good choices will require careful attention to the particulars of the task at hand.

3 Electronic music terminology

The first widely used model for computer music performance was what is now called *Music N*, developed over a series of programs written by Max Mathews starting in 1957[Mathews, 1969]; by 1959 his Music 3 program essentially put the idea in its modern form, as exemplified in Barry Vercoe's Csound program. These programs all act as "music compilers" or "renderers", taking a fixed text input and creating a soundfile as a batch output. Although Csound has provisions for using real-time inputs as part of its "rendering" process, in essence the programming model is not interactive.

Music N input is in the form of an *orchestra* and a *score*, as shown in fig. 4. The orchestra can be thought of as emulating a 1950s-era *elektronischemusik* studio, in which the hardware is organized in metal boxes with audio inputs and outputs, such as tape recorders, oscillators, pulse generators, filters, ring modulators, and so on. These would be connected by audio cables into a *patch.* Furthermore, the boxes had knobs and switches on them that allowed the user to supply parameters such as the frequency of an oscillator.

In the Music N paradigm, the studio and its patch are represented by the orchestra. Although the actual orchestra file is in a programming language, when publishing algorithms it has traditionally been represented as a block diagram showing a collection of *unit generators* and the audio connections between them.

The score is organized as a list of time-tagged records that are (either nostalgically or deprecatingly) called *score cards.* In addition to one or two time tags (a "note" has two, one for its start and one for its end), a score card has some number of numerical parameters that may be supplied to the unit generators. The score is like a process in that it runs sequentially in time.

Unlike the computer science notion of a process, however, the score advances and waits according to timing information in the score cards. Each score card has an associated *logical time* at which it is run.

Things get interesting when we try to adapt this paradigm to run in real time. We could simply connect the Music N output to a real-time audio output; but presumably our reason for wanting to run in real time is to be able to use live inputs to affect the sound output. Taking the opposite direction, we could require that the user or musician supply all the parameters in real time using knobs and switches, but this quickly reveals itself to be unmanageable for the human. We will need to make intelligent decisions, probably different for any two musical situations, as to how the live inputs will affect the production of sound. More generally, our problem is to design a software environment that will give a musician the freedom to make these choices.

In the early 1980s two influential real-time synthesizers were designed, the Systems Concepts Digital Synthesizer (or "Samson Box") at Stanford[Loy, 1981], and the 4C synthesizer at IRCAM[Moorer et al., 1979][Abbott, 1981]. Both machines ran a fixed computation loop with a fixed number of steps, with one loop finishing at each tick of the sample clock,

Each of these machine designs got some things right for the first time. The Samson box was the first working machine that could do sample-accurate parameter updates in real time. To do this, the fixed program contained an update mechanism in which items were taken off the head of a time-tagged parameter update queue. This queue was filled by the Foonly controlling computer some tenths of seconds, or whole seconds, in advance, so that the Foonly did not have to preform parameter updates synchronously. This approach had one major limitation: it did not take into account the possibility of real-time interaction. It was physically possible to jump the queue for "real-time" parameter updates, but then one lost any ability to determine the timing of such updates accurately.

The 4C machine and its controlling software 4CED were more explicitly designed with real-time interaction in mind, although the timing was less accurate than with the Samson Box. In the 4C parameter updates were effected at interrupt level from the controlling computer;

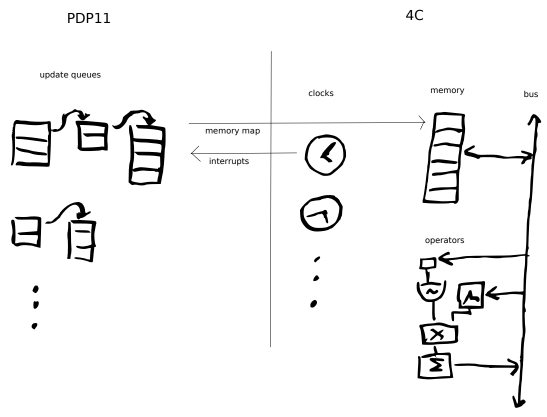

Figure 5: The 4C and the 4CED environment

the computer was interrupted by the 4C when one of a bank of timers ran out.

The 4CED user conceptualized the 4C as a collection of 32 independent processes (Abbott's simile was a collection of 32 music boxes that the user could start at any time). The guiding idea seems to have been that a performer could play a keyboard with 32 keys on it, but each key, rather than being restricted to playing a single note, could in turn set off a whole sequence of actions. This would seem to greatly magnify what the keyboard player could do.

It seems also to have been on people's minds that the playing of sequences could usefully be unified with the business of scheduling breakpoints to a pitch or amplitude envelope consisting of many segments. Both the sequencing of collections of notes and the sequencing of envelope breakpoints led many computer music researchers to think that a process model, as would appear a time-sharing operating system, was a perfect metaphor to reuse in the design of real-time computer music control systems.

Both the Samson box and the 4C maintained lists of parameter updates that resemble Music N scores in that they have sequences of numeric updates for synthesis parameters. In the case of the 4C, the scheduling of the updates could depend on real-time inputs. Both these systems, but particularly 4CED, modeled musical sequences in ways that resembled processes in the computer science sense.

4 Processes in modern computer music environments

The four most widely-used computer music environments are probably Csound, Max, Supercollider, and Pd. (Since this is a linux conference, we won't consider Max here, but only the closely related Pd.) Of all these, Supercollider is unique in that it explicitly adopts a process-like model, which offers at least two advantages. First, it allows the user to "think" in processes in order to express the parallelism that is desirable for polyphony, for instance in voice banks or collections of sinusoids in additive synthesis. Second, it allows parallelism in the signal processing engine so that multiprocessors can be exploited.

A newer environment, ChucK[Wang and Cook, 2003], also uses a thread model and aims in part to make the creation and destruction of processes (named "shreds") as lightweight as possible. This language is still under active development and may lead to new ideas for adapting the concept of process to interactive computer music environments.

The process models in both Supercollider and ChucK both lend themselves well to generative applications, where processes may need to quickly and efficiently create new voices or instances of computational algorithms. In both environments the creation and destruction of processes is highly optimized so that large numbers of them may be created and managed dynamically.

Pure data, on the other hand, offers no model of a process and therefore is badly adapted both to expressing polyphony (although this is fixable using a voice bank management object available in Pd extended but not yet in "vanilla"). It is even less well adapted to expressing generative algorithms in which data may fork and recombine in ways that Supercollider and ChucK make easy. It is perhaps the most distinguishing feature of Max and later Pd that they both radically did away with the notion of process altogether.

Pd offers no easy way to manage parallelism either; the facility provided is the "`pd~`" object which can be considered a throwback to the Max/FTS solution from 1990. There are advantages gained by this trade-off. Most importantly (in my view at least), the fluency and ease by which Pd patches can react to input from the outside world is much greater. This is partly because of the absence of a process model, because one never has to consider how different processes must be synchronized in order to react consistently to new and possibly unpredictable inputs.

5 Conclusion

The notions of "process" and "thread" seem eternally attractive to designers of real-time computer music programming environments such as the ones discussed here. The attraction sees to be for both expressive reasons (as a way to describe polyphony, particularly in generative situations) and for efficiency reasons (as ways to efficiently exploit parallelism in general-purpose processors). Yet the difficulties of managing coordination between processes or threads still make it appear impossible to adapt them easily to an environment like Pd. This is a hard problem that is worthy of future work.

Meanwhile, the most powerful arithmetic processors in modern devices are their graphics processors. We don't yet have a good understanding of how to exploit these architectures for computer audio, and indeed this seems so far from today's programming models that it is hard to see where we could start on this.

It seems that the state of the art in programming environments for doing interactive computer music is out of sync with current developments in computing. Past efforts to make music out of computer hardware and operating systems that were often ill suited to the task have often resulted in advances that had implications not only for computer musicians but for computer science as well. Attacking the current situation in a similar way might similarly give rise to useful new ideas.

6 Acknowledgements

Thanks as always to the hard-working organizers of LAC for making this publication possible. This paper is based in part on material from a manuscript for a planned future book.

References

C. Abbott. 1981. The 4ced program. *Computer Music Journal*, pages 13–33.

D. Gareth Loy. 1981. Notes on the implementation of musbox: A compiler for the systems concepts digital synthesizer. pages 333–349.

Max V. Mathews. 1969. *The Technology of Computer Music*. MIT Press, Cambridge, Massachusetts.

J.A. Moorer, A. Chauveau, C. Abbott, P. Eastty, and J. Lawson. 1979. The 4c machine. *Computer Music Journal*, pages 16–24.

Ge Wang and Perry R. Cook. 2003. Chuck: A concurrent, on-the-fly audio programming language. pages 217–225, Ann Arbor. International Computer Music Association.