# Design Patterns for Real-Time Computer Music Systems

Roger B. Dannenberg and Ross Bencina

4 September 2005

This document contains a set of “design patterns” for real time systems, particularly for computer music systems. We see these patterns often because the problems that they solve come up again and again. Hopefully, these patterns will serve a more than just a set of canned solutions. It is perhaps even more important to understand the underlying problems, which often have subtle aspects and ramifications. By describing these patterns, we have tried to capture the problems, solutions, and a way of thinking about real-time systems design. We welcome your comments and questions.

# Static Priority Scheduling

Also known as Deadline Monotonic Scheduling.

## *Context*

Real-time systems often have a mix of tasks. Tasks may have hard deadlines, as in sample buffer computation, or strongly desirable upper limits on latency, as in interactive MIDI processing. If this allowable time-to-completion for some task is less than the execution time of some other task, then the long-running task must be *preempted* so that the low-latency task can meet its deadline.

## *Problem*

Organize the program so that deadlines are met and latencies are acceptable, using relatively few processes or threads.

## *Forces*

When there is coordination and communication between tasks (e.g. shared variables, memory, and data structures), it is *much* easier for the programmer if these tasks do not preempt one another – they can then run in a single thread. Design is a tradeoff between letting tasks share threads, to simplify coordination, and allocating tasks to separate threads, to allow fine-grain scheduling.

## *Solution*

Minimize the number of threads, typically using 2 or 3. Divide tasks according to their *latency* requirements, i.e. the time from when the task *can* run to its completion deadline. Low-latency tasks are assigned to the highest-priority thread, and the highest-latency threads are assigned to the lowest-priority thread, etc. The threads are scheduled according to fixed priorities.

Analysis can sometimes be used to evaluate solutions before they are implemented. Essentially, the worst-case latency at the highest priority thread is just the sum of all task run times (assuming tasks will not run again until other tasks have finished). The latency of the thread at the next lower priority level is also the sum of the run times for tasks running in this thread, but in addition, the high priority thread "steals" time and this must be accounted for. Continuing until worst-case latencies are estimated for all threads, one can then decide whether the latencies are small enough to satisfy all tasks.

## *Further Reading*

Ken Tindell, "Deadline Monotonic Analysis," in http://www.embedded.com/2000/0006/0006feat1.htm.

J. Y. T. Leung, J. Whitehead, "On the Complexity of Fixed Priority Scheduling of Periodic, Real-Time Tasks", Performance Evaluation, 2(4), pages 237-250, 1989.

 

# Real Time Memory Management

Also known as non-blocking memory management or deterministic memory management.

## *Context*

When programming "real-time" code to run on a general purpose operating system, such as to generate audio in a high-priority callback, it is not advisable (or sometimes not even possible) to call operating system functions, including those which allocate memory, because it is impossible to predict the time which may be taken by such calls (which could acquire locks, cause priority inversion, or page faults). Yet often in real-time music software objects need to be allocated dynamically.

## *Problem*

Allocate memory for program objects using methods which avoid calling the operating system to allocated memory in real-time code.

## *Forces*

How well is the allocation behavior of these objects understood? How many different types (or size classes) of objects are needed? Is a fully generalised allocation mechanism necessary, or does the allocation behavior allow a simpler solution? How are allocated objects shared between threads of execution?

## *Solutions*

Depending on the type of allocation patterns which need to be supported there are a number of practical options, many can be combined or used in different parts of the same application.

## *Fixed Up-front allocations with a free list*

If the maximum number of required objects is known, arrays of these objects can be preallocated, and a linked list of free objects can be maintained for fast allocation and deallocation.

## *Size-segregated free lists*

An extension of the previous pattern is to keep free lists for separate size classes, instead of distinct object types.

## *Per-thread memory pools*

One way to avoid locks is to only use memory within a specific thread context and write a custom allocator which allocates memory from a large preallocated block. This avoids calling the OS, but the memory allocation algorithm should be chosen carefully if deterministic timing is important.

 

### *Allocate in a non-real-time context*

Don't allocate any memory in the real-time context, instead allocated memory in a non-real-time thread and send it using an asynchronous message queue.

### *Real-time Garbage Collection*

Write a garbage collector for objects which are used in your real-time thread. An incremental collector can be coded to provide deterministic timing behavior by interrupting collection if its timing allocation is exceeded.

### *Examples*

SuperCollider Language and Serpent both use real-time garbage collectors to manage dynamic memory. Aura and AudioMulch use per-thread size-segregated memory pools. AudioMulch allocates in a non-real-time context for many types of dynamic objects, especially large ones. Fixed up-front allocation is often used for things like synthesizer voices, where the maximum polyphony is known up-front.

### *Further Reading*

See the Memory Patterns chapter of “Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems” by Bruce Powel Douglass. Addison Wesley Professional, 2002. or the online extract at:
http://www.awprofessional.com/articles/article.asp?p=30309

 

# Communication and Synchronization with Lock-Free Queues

Also known as asynchronous message passing.

## *Context*

On general purpose operating systems it is not usually possible to depend on synchronization primitives such as mutexes (locks) to provide deterministic timing behavior since these systems are often optimized for high-throughput rather than low-latency.

## *Problem*

Exchange data or events safely between multiple threads of execution without using operating system synchronization primitives such as mutexes or semaphores.

## *Forces*

Polling can (sometimes) add latency and overhead compared to implementations based on synchronization primitives (but suitable real-time synchronization primitives may not exist and tend to be less portable). Single-reader, single-writer queues are simple, but may be more wasteful of resources than more sophisticated lock-free schemes.

## *Solution*

Use lock free queues to queue messages or data between threads. The queues can contain things such as data samples, message records, variable length messages, or pointers to other data structures. Such queues need to be polled periodically to ensure that they don't overflow. If a queue has data in it a semaphore (pthreads condition variable) may be used to signal this state to a non-real-time thread, however a real-time thread should not normally block waiting for a queue. By virtue of the algorithms used, access to these queues is lock-free.

## *Examples*

Supercollider 3 server uses lock-free queues for communication between the real-time audio callback and non-real-time threads such as the OSC network communications thread. AudioMulch uses lock-free queues for most communication between real-time audio and other threads such as those for the GUI, incoming MIDI callbacks, disk i/o and sample loading. Jsyn uses a lock free ring-buffer for communications to the audio thread, and a lock-free linked list queue for communications from the audio thread. Aura uses lock-free queues between zones. PortMIDI includes an example using a lock-free queue to communicate between the main thread and a high-priority MIDI thread.

## *Further Reading*

Some Notes on Lock-Free and Wait-Free Algorithms:
http://www.audiomulch.com/~rossb/code/lockfree/

 

# Accurate Timing with Timestamps

This pattern is closely tied to discrete event simulation.

## *Context*

Many musical sequences and computations give rise to a set of events with precise timings. The simple timing approach, which is something like

*A(); sleep(5); B(); sleep(3); C(); sleep(7); …*

will accumulate error due to finite computation speed and system latencies. We assume that tasks run when methods or functions are called by a *scheduler* and that tasks do not suspend or sleep.

## *Problem*

To deliver accurately timed outputs. Momentarily high CPU load should at most cause momentary degradation of the output, but no long-term errors.

## *Forces*

Sometimes, it is desirable to let time "slip" when systems encounter too much load. In other words, once a system falls behind, it may be better to remain behind than to catch up.

## *Solution*

Tasks always compute the "ideal" time at which they should run. The scheduler remembers the time at which the task requested to wake up. When the task wakes up, it operates as if this "ideal" time is the true real time. If future events are scheduled relative to this one, they will be scheduled relative to the "ideal" time rather than real time, insuring accurate timing. For example, if every time a task runs, it schedules itself to run again in 0.1 seconds, then it will compute the "ideal" sequence of wake-up times 0.1, 0.2, 0.3, etc. with no error due to finite CPU speed.

## *Further Reading*

Anderson, D. P. and Kuivila, R. 1990. A system for computer music performance. *ACM Trans. Comput. Syst.* 8, 1 (Feb. 1990), 56-82.

 

# Event Buffers

Often discussed in the context of *jitter*, and this could be considered a special case of *Accurate Timing with Timestamps.* Also called Action Buffers.

## Context

Sometimes, data cannot be computed accurately at the desired time. This is usually caused by latency when the operating system fails to run a process immediately. It may also be caused when the application has too much to do, perhaps because work comes in bursts. In some of these cases, data can be computed early. In other words, in some cases, it is better to reduce timing deviations, called *jitter*, than to reduce overall timing delay, or *latency.*

## Problem

*Deliver* output with timing that is more accurate than can be achieved by the task that *computes* the output.

## Forces

This design relies on the ability to pre-compute data, which means that *latency* will suffer in order to reduce *jitter*. Sometimes, it is better to just minimize latency. As latency approaches zero, jitter also approaches zero, so sometimes this pattern has no advantage. This pattern will be most useful when the computation time for output data is significant and exhibits a lot of variation or burstiness.

## Solution

Compute both data and the time at which the data should be delivered. Typically, the data is computed using the *Accurate Timing with Timestamps* pattern, and the "ideal" time of the computation is used as the time at which the data should be delivered. *However*, since the computation will often be somewhat late, a *constant time offset* is added to the "ideal" time to form an *output time stamp.* The data is then transferred through a buffer to another process (or device driver or hardware) running at higher priority and better able to output data accurately according to the timestamp. A trivial example is prefetching a MIDI sequence from disk so that it can be output with lower latency than disk read operations would otherwise allow. Another common example is the use of timestamps on MIDI data which is delivered to a device driver.

## Further Reading

D. P. Anderson and G. Homsy. "A Continuous Media I/O Server and its Synchronization Mechanism". *IEEE Computer*, 24(10):51-57, October 1991.

Roger B. Dannenberg, Thomas P. Neuendorffer, Joseph M. Newcomer, Dean Rubine, David B. Anderson: "Tactus: Toolkit-Level Support for Synchronized Interactive Multimedia". *Multimedia Syst.* 1(2): 77-86 (1993)

 

# Synchronous Dataflow Graph

Also known as synthesis graph, Signal Graph (PD), Filter Graph (DirectShow), Ugen Graph (SuperCollider)

## *Context*

Multiple independent signal generation and processing modules (unit generators) need to be dynamically interconnected at program runtime, such as when loading a new configuration file (i.e. an orchestra) or in response to dynamic interconnection requests from the user interface. The nature of the interconnections is not limited to sequential or parallel topologies.

## *Problem*

Unit generators have data dependencies: The inputs to a unit generator must be computed before the unit generator runs. The many interconnections and their implied dependencies require that special attention be paid to the order of unit generator execution.

## *Forces*

Some systems are static, i.e. the graph does not change at run-time. Other systems are dynamic, allowing the graph to be modified interactively.

## *Solution*

Organise the modules in a directed graph which indicates the signal flow between modules. By traversing backwards through the graph it is possible to establish the data dependencies, and hence which modules need to be allocated first. By applying graph-coloring techniques from the compiler literature it is possible to minimize the number of buffers necessary to communicate intermediate data between modules.

Two major variations of this technique exist: one is to traverse the graph once to determine evaluation order, and cache this ordering in a secondary structure. The other alternative is to traverse the graph directly. Each alternative has its own benefits. For example, although direct graph traversal is simpler, caching the execution order means that the graph may be altered without needing to protect the execution list from concurrent access.

## *Further Reading*

Read the compiler literature on graph coloring.

 