Real-Time Systems Concepts for Computer Music

Roger B. Dannenberg
Carnegie Mellon University
and
Ross Bencina
Universitat Pompeu Fabra, MTG

Introduction

- Goals
  - Give practical know-how
  - Present some reusable design patterns for real-time, interactive music systems
  - Review "best practices" for common problems
- Divided into a number of topic areas

Basic Real-Time Concepts

- Objectives of Computing
- Why We Need/Use Concurrency
- Preemption
- Scheduling Basics
- Latency
- Design Pattern: threads with static priority
- Locks and Critical Sections
- Interaction with Priority

Objectives of Computing

- Get the right answer (program correctness)
- Get it fast (algorithm complexity theory)
- Be on time (real-time computing)
  - Faster is not always better
  - Sensitive to worst case, average doesn't matter
- Security, Reliability, Availability, Low-power, …

Why We Need/Use Concurrency

- Real-Time systems have a mix of tasks
  - Compute audio
  - Respond to MIDI
  - Manage Graphical User Interface
  - Read files from disk
  - …
- Maximum response time allowed for audio might be 1ms
- Maximum computation time for screen update may be 200 ms
- Maximum latency in the operating system to open a file may be 100ms
- How can we respond to audio input quickly if we are in the middle of a long graphics update or file access?

Preemption

- When two or more programs are “running”
- and there is only one CPU,
- one program can be halted,
  - its registers are saved
  - all other program state is saved or retained
- another program can continue
  - by restoring all registers and any other state
- How do we decide what to run when?

Scheduling Basics

- Standard OS tries to be “fair” and responsive
  - Give each process an equal “slice” of time
  - May detect compute-bound processes and run them in the background (when other processes are not ready to run)
- Real-time OS may try to be “on time”
  - Admission schemes only let a new process run if resources are available
  - Earliest Deadline First – optimal if all deadlines can be met
  - Static Priority – run the process with the highest priority of all ready-to-run processes

Latency

- How long does it take to deliver results?
- Sources:
  - Hardware (usually very small), e.g. audio anti-aliasing filters, sample buffers
  - Interrupt latency
    - System may be processing higher-priority device
    - System my have interrupts disabled for a time
  - Kernel latency, deferred procedure calls
    - Systems often defer processing from the hardware interrupt to a software level (interrupts become more responsive, actual response time may suffer)
  - Process-scheduling latency
    - How long before a ready-to-run process actually runs
  - Application latency
    - How long before the application computes the result

Latency (2)

- Latency can vary widely among systems
- Modern systems are being tuned to deliver about 1ms latency (worst case) to highest priority process.

9 ICMC 2005 Workshop on Real-Time Systems Concepts for Computer Music

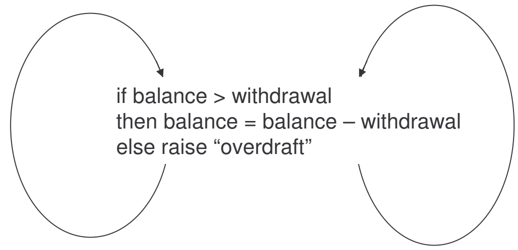
Design Pattern: Threads With Static Priority

- Description:
  - Multiple tasks.
    - Some must be completed quickly (with low latency)
    - Some take long to compute
  - Computation time is small compared to allowable latency
- Design Solution:
  - Divide tasks into a small number of latency classes (low latency, medium latency, etc.)
  - Create one thread for each latency class
  - Schedule threads with static (real-time) priorities: lowest latency class gets highest priority

10 ICMC 2005 Workshop on Real-Time Systems Concepts for Computer Music

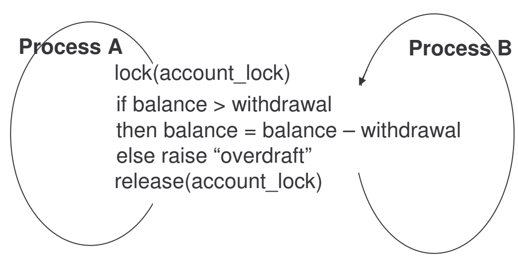
Locks and Critical Sections

- A classic problem is the critical section:
- Solution:



Locks and Critical Sections

- A classic problem is the critical section:
- Solution:



Interaction with Priority

- Suppose a low-priority thread L has the lock.
- A medium-priority thread M starts to run.
- A high-priority thread H starts to run and tries to acquire the lock.
- H blocks, so M resumes.
- H blocked as long as M runs! (*Priority Inversion*)
- One solution: *Priority Inheritance*
  - Modern Real-Time Operating systems implement it
  - Does WinXP, Linux, Mac OS X?
- Another solution: no locks! (discussed later)

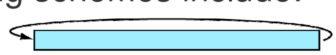
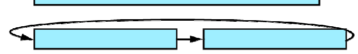
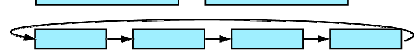
Basic Digital Audio Concepts

- PC Audio Systems, DMA
- Buffering Schemes
- Userspace Audio APIs: Synchronous/blocking vs. Asynchronous/callback APIs
- PortAudio: an abstraction of audio APIs
- PortAudio example: playing a sine wave

PC Audio Systems & DMA

- Handling a CPU interrupt for each sample isn't practical (context switching overhead...)
- Typical solution:
  - Audio Hardware exchanges data with main memory using DMA
  - CPU gets interrupts when buffers are full/empty
  - These interrupts can lead to user-space code being executed (eventually)

Buffering Schemes

- Hardware buffering schemes include:
  - Circular Buffer 
  - Double Buffer 
  - Buffer Queues 
- these may be reflected in the user level API
- Poll for buffer position, or get interrupts when buffers complete
- Typically audio code generates samples into a buffer, it doesn't care about the buffering scheme.
- Exception: when buffer lengths don't factor well

User space Audio APIs:
Synchronous/blocking vs Asynchronous/callback APIs

- Blocking APIs
  - Typically provide primitives like read() and write()
  - Can be used with select() to interleave with other operations
  - Users manage their own threads for concurrency
  - Great if your OS threading services can provide real-time guarantees (e.g. SGI)
- Callback APIs
  - User provides a function pointer to be called when samples are available/needed
  - Concurrency is implicit, using locks or blocking functions may not be possible or desirable
  - You can assume the API is doing its best to be real-time

17 ICMC 2005 Workshop on Real-Time Systems Concepts for Computer Music

PortAudio: an abstraction of audio APIs

- PortAudio wraps multiple Host APIs providing a unified and portable interface for writing real-time audio applications
- Main entities:
  - Host API – a particular user-space audio API (ie JACK, DirectSound, ASIO, ALSA, WMME, CoreAudio, etc.)
    - `PaHostApiInfo, Pa_GetHostApiCount(), Pa_GetHostApiInfo()`
  - Device – a particular device, usually maps directly to a host API device. Can be full or half duplex depending on the host
    - `PaDeviceInfo, Pa_GetDeviceCount(), PaGetDeviceInfo()`
  - Stream – an interface for sending and/or receiving samples to an opened Device
    - `PaStream, Pa_OpenStream(), Pa_StartStream()`
- See http://www.portaudio.com

18 ICMC 2005 Workshop on Real-Time Systems Concepts for Computer Music

PortAudio example: generating a sine wave

```
struct TestData{
    float sine[TABLE_SIZE];
    int phase;
};

static int TestCallback( const void *inputBuffer, void
   *outputBuffer,
        unsigned long framesPerBuffer, const
   PaStreamCallbackTimeInfo* timeInfo,
        PaStreamCallbackFlags statusFlags, void *userData ) {
    TestData *data = (TestData*)userData;
    float *out = (float*)outputBuffer;

    for( int i=0; i<framesPerBuffer; i++ ) {
        float sample = data->sine[ data->phase++ ];
        *out++ = sample;  /* left */
        *out++ = sample;  /* right */
        if( data->phase >= TABLE_SIZE ) data->phase -= TABLE_SIZE;
    }
    return paContinue;
}
```


PortAudio example: running a stream (1)

```
int main(void)
{
    TestData data;
    for( int i=0; i < TABLE_SIZE; ++i )
        data.sine[i] = sin( M_PI * 2 *
   ((double)i/(double)TABLE_SIZE) );
    data.phase = 0;

    Pa_Initialize();

    PaStreamParameters outputParameters;
    outputParameters.device = Pa_GetDefaultOutputDevice();
    outputParameters.channelCount = 2;
    outputParameters.sampleFormat = paFloat32;
    outputParameters.suggestedLatency =
            Pa_GetDeviceInfo( outputParameters.device )-
   >defaultLowOutputLatency;
    outputParameters.hostApiSpecificStreamInfo = NULL;

    ...
```

PortAudio example: running a stream (2)

```
    ...

    PaStream *stream;
    Pa_OpenStream( &stream, NULL /* no input */,
   &outputParameters,
              SAMPLE_RATE, FRAMES_PER_BUFFER, paClipOff /*flags*/,
              TestCallback, &data );

    Pa_StartStream( stream );

    printf("Play for %d seconds.\n", NUM_SECONDS );
    sleep( NUM_SECONDS );

    Pa_StopStream( stream );
    Pa_CloseStream( stream );
    Pa_Terminate();
}
```


Real-Time Memory Management

- Conventional Memory Management
- Real-Time Memory Management Strategies
- How Does malloc() Work?
- Memory Allocation in Aura
- Design Pattern: Memory Allocation
- Reference Counting
- Real-Time Garbage Collection
- Other Memory Issues

Conventional Memory Management Issues

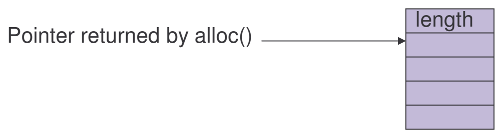
- *alloc*(*n*): return address of n contiguous bytes
- *free*(*ptr*): free a block previously *alloc*ated
- *external fragmentation* – wasted space between allocated blocks of memory
- *internal fragmentation* – wasted space when allocated block is bigger than request (e.g. power of 2)
- Is the memory pool shared by threads?
  - Is memory allocation in a critical section?
- Are freed blocks consolidated? At what cost?
- Does *alloc* search for a good block? At what cost?
- Can compaction operation eliminate fragmentation?

Real-Time Memory Management Strategies

- Static allocation – allocate what you need when program initializes.
  - Example: Aura copies and converts floating point samples from multiple buffers into interleaved 16-bit samples before playing them. Rather than allocating temporary space before each write, Aura pre-allocates a big buffer and reuses it.
- Allocate but do not free – allocate from a big free memory block. Do not free anything.
- Allocate only in non-real-time thread and send pointers to real-time thread.
- Traditional alloc(n) and free(p) operations.
- Reference counting to replace free(p)
- Garbage collection to replace free(p)

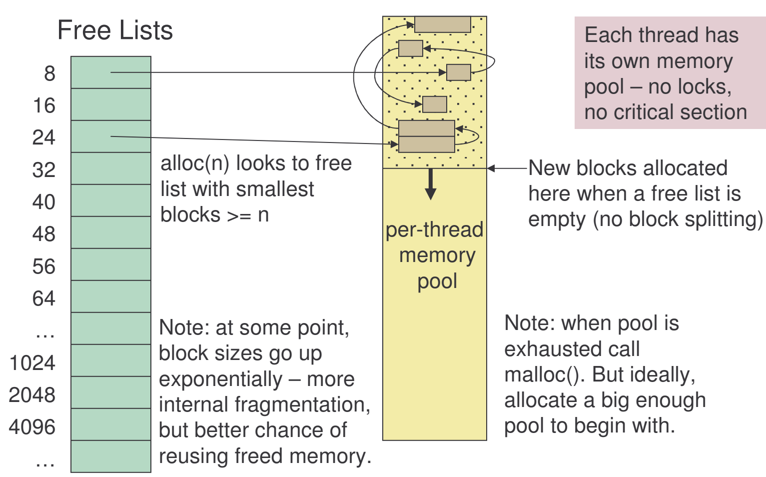
How Does malloc() Work?

- Single free memory pool protected by locks.
- Size of allocated block is stored before block:



- Once allocated, blocks are not moved
- Allocation and free algorithms are part of C run-time library

Memory Allocation in Aura

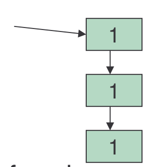


Design Pattern: Memory Allocation

- Description:
  - Real-time demands constant time allocate and free operations
  - Memory efficiency is not critical
  - Most allocations likely to be from a relatively small set of different sizes
- Solution:
  - Linked lists of free memory blocks
  - Each list contains one size of block
  - If there are multiple threads, keep memory pools separate to avoid lock overhead and possible priority inversion

Reference Counting

- Every memory object keeps track of the number of incoming pointers
- When count goes to zero, free the block
  - When assigning to a pointer: decrement ref count of old value and increment ref count of new value
- Can be good when objects are shared
- Problems:
  - Costly to assign pointers to new values
  - Free operation can have unbounded cost
    - Because many dependent objects can be freed



Real-Time Garbage Collection

- Mark all reachable objects
- Scan all objects: any object unmarked is moved to free list
- GC can be performed incrementally
- Marking must be very carefully coordinated with the application (the “mutator”)
  - Usually, writes to pointers must run some code to maintain consistency
- Some variants “mark” objects by copying them from one half of address space to other
- Getting this right and debugging is a BIG job.
- Some real-time garbage collectors for C++ may be available.
- Serpent and Supercollider are two examples with GC integrated into real-time scripting languages.

Other Memory Issues

- Virtual memory – it may be expensive to touch newly allocated memory because it may not be mapped to physical memory.
- Mapping to physical memory may require zeroing memory for security reasons.

Timed Events

- Computer music computation has mix of:
  - Very heavy but periodic audio computation
  - Very light but non-periodic event computation
    - (MIDI, envelope breakpoints, start, stop, sequenced events and updates, etc.)
  - Perhaps some high-latency activities:
    - File I/O, Network I/O
- Let's focus on non-periodic events

Concurrency without locks?

- Lock-based designs aren't a good solution for real-time applications unless the OS supports real-time thread scheduling.
- How can we communicate data between threads safely without locks?
  - Atomic values
    - Limited applicability, easy to misuse
  - Lock-free queues

Queue Topics

- Queue Usage
- Applications
- Simple single-reader, single-writer lock-free queue
- Variations
- Other Considerations
- Multiple CPU issues (memory ordering)

Queue Usage

- Communicate between asynchronous processes
- Producer pushes items, consumer polls for items “sometime later”
- Queues can contain:
  - Audio samples
  - Fixed size data blocks e.g. MIDI messages, Message records (message id, params), pointers to messages
  - Variable length messages
  - Bundles of messages to execute atomically
- Lock-free implementations exist

Some Applications

- Send samples to another thread so it can perform blocking operations with them (write to disk/network)
- Send MIDI messages for interpretation by an audio callback
- Send commands to another thread for execution
  - (see SC server for a good example of this)
- Send VU meter data to a GUI thread for display

Simple single-reader single-writer lock-free queue

- Ring buffer with one read pointer and one write pointer:

- Data is available when read pointer != write pointer
- Queue is full when read pointer == write pointer – 1

Variations

- Linked lists
  - Pro: Variable size queues
  - Con: Need to allocate links somehow
- Semaphores to signal full/empty state for blocking readers and/or writers
- Connecting more than one reader/writer:
  - Combine locks with srsw queues
  - Use one srsw queue for each writer-reader pair
  - Use multiple reader multiple writer queues

Other Considerations

- Don't forget overflow (fixed size queues) or node allocation (variable length queues)
- Programs designed around asynchronous messaging tend to be organised differently from those using synchronous execution – plenty has been done in this field, it's worth reading about it.
- Some languages are built around asynchronous message passing with no shared-state e.g. Erlang

Multiple CPU lock-free issues

- Lock free algorithms assume in-order memory access
- Compilers don't guarantee in-order access (volatile is not enough!)
- Hardware can reorder memory access: OK for 1 cpu, leads to inconsistent view of memory on multiprocessor systems.
- Therefore, use memory barriers, or atomic access APIs which use them (e.g. Interlocked* API on windows)

Logical Clock Systems

- Timed Events
- Scheduling and Dispatching
- Accurate Timing With No Accumulated Error
- Scheduler/Dispatcher
- Logical Time

Scheduling and Dispatching

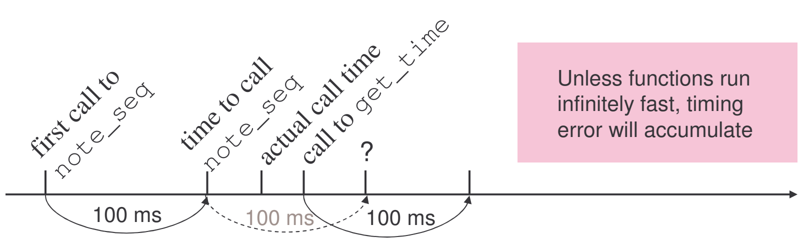
- schedule(time_stamp, function_pointer,
  parameter_1, parameter_2, …)
  - Call on *schedule* should return immediately
  - In the future, at *time_stamp*, there should be a call to (**function_pointer*)(parameter_1, parameter_2, …)
- Terminology:
  - The *scheduler* is a software module
  - The function and parameters are an *event*
  - Calling the function is *dispatching* the *event*

(In)accurate Timing

- Consider this function to play a sequence of notes:

```
void note_seq() {
    play_a_note_via_midi();
    schedule(get_time() + 100, // in ms
             &note_seq);
}
```

- Possible outcome:



Unless functions run infinitely fast, timing error will accumulate

Design Pattern: Accurate Timing With Timestamps

- Scheduler records "ideal" time

```
NOW = scheduled_wakeup_time;
(*event->fn_ptr)(event->p1, event->p2,
…);
```

- Future scheduling in terms of "ideal" time, not real time.

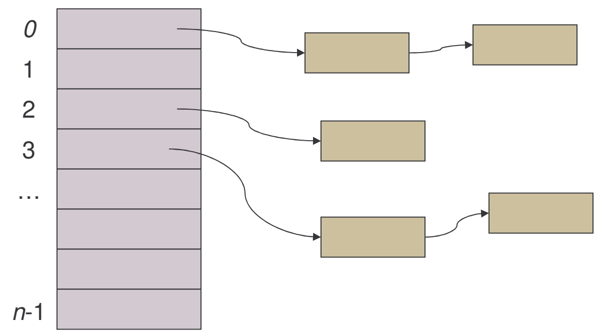
```
void note_seq() {
    play_a_note_via_midi();
    schedule(NOW + 100, // in ms
             &note_seq);
}
```


Extension for Accurate MIDI Timing

- Problem: you may not see MIDI data immediately, JIJO: (timing) jitter in, jitter out
- Solution:
  - Get timestamps from MIDI device driver (e.g. use PortMidi and use incoming timestamps)
  - Treat (accurate) MIDI timestamps as "NOW"
  - If response to MIDI is immediate
    - E.g. MIDI controls audio synthesis
  - Then one option is to delay the response a few milliseconds.
  - PortMidi output can automatically add a time offset and schedule MIDI output in the driver to reduce output jitter
  - Tradeoff between Jitter and Latency

Fast Scheduling and Dispatching

- Calendar Queue



- Expected case is O(1), worst case is O(n)

45 ICMC 2005 Workshop on Real-Time Systems Concepts for Computer Music

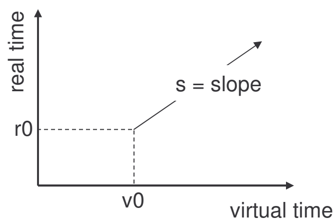
Alternative: Priority Queue

- Various structures allow log(n) insert/delete:
  - Red-Black Trees
  - Heap
- To Schedule: insert into priority queue
- To Dispatch: remove earliest item from queue

46 ICMC 2005 Workshop on Real-Time Systems Concepts for Computer Music

Logical Time (or Virtual Time)

- Used for
  - tempo control
  - clock synchronization
  - speed control/time-scaling
- Mapping from logical/virtual time to real time:



r(v) = r0 + (v – v0)s
v(r) = v0 + (r – r0)/s

set_tempo(new_s, at_v):
r0 = r(at_v)
v0 = at_v
s = new_s

Implementing Virtual Time

- Build on real-time scheduler/dispatcher
- Logical time system represented by object with:
  - priority queue
  - r(v) – virtual time to real time
  - v(r) – real time to virtual time

```
lts::schedule(time, event)
    queue.insert(time, event)
    v = queue.next_time()
    schedule(r(v), lts_wakeup, lts)

lts_wakeup(lts)
    lts->wakeup()
```

```
lts::wakeup()
    v = queue.next_time()
    if (r(v) <= NOW)
        VNOW = v
        dispatch(queue.get())
        v = queue.next_time()
        schedule(r(v), lts_wakeup, lts)
```

Clock Synchronization

- The Problem
- Simple Network Clock Synchronization
- High Resolution vs. High Latency
- Synchronizing to MIDI clocks
- Other Clock Issues

The Problem

- Clocks are based on crystal oscillators
- Machines can have multiple clocks:
  - Time-of-day clock
  - CPU clock (e.g. 3.3 GHz)
  - Audio sample clock
- Crystals are accurate only to about 0.1%
- Crystal clock speed varies with temperature
- 10 minutes x 0.1% x 2 = 600s/500 = 1.2s (!)

Simple Network Clock Synchronization

- On the other hand, drift in 1s = 1/500 = 2ms
- So resynchronize every second or so…
- Simple protocol:
  - Designate a master clock available at "server"
  - Clients adjust their clocks as follows:

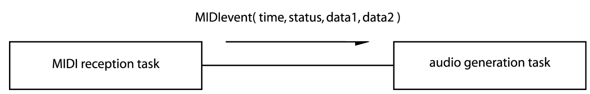
```
t0 = get_time()
tm = get_time_from_master()
t1 = get_time()
if (t1 < t0 + 5ms) {
        tm += (t1 – t0) / 2
        bump_local_time_by(tm – t1)
}
```


High Resolution and High Latency

- Simple protocol can break down due to:
  - Need for high resolution
  - High network latency
- Some solutions (see the literature):
  - Average computed clock skew over multiple queries to the master
  - Estimate the difference in clock *rates* as well as the difference in clock times
  - Estimate network typical network latency to help determine outliers
  - Systems with many clients be based on broadcasts from master – a very different approach

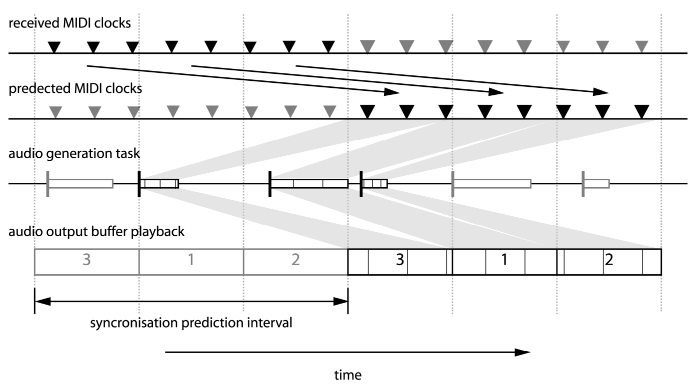
Synchronizing with MIDI Clocks

- Why?
  - Performance involving multiple systems / instruments
- How?



  - Timestamp MIDI clocks
  - Predict MIDI phase at buffer playback times
  - Generate audio according to predicted phase

MIDI Clock Phase Prediction



See Bencina, R. (2003). “PortAudio and media synchronization. In Proceedings of the 2003 Australasian Computer Music Association (ACMC'03).

Implementation Issues

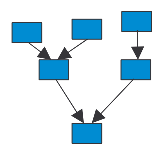
- Multiple time bases, one OS, e.g.
  - Soundcard sample clock
  - API timers of unknown origin
  - CPU cycle counter (high precision, unknown frequency)
  - OS timers (possibly low precision)
- This can lead to skew problems
- Different APIs use different timers, would be good to be interoperable but no good solutions exist.

Modular Audio Processing

- Unit generators
- Graph evaluation
- Evaluation mechanisms
- Block-based processing
- Vector allocation strategies
- Variations

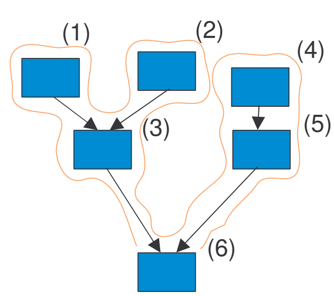
Unit generators

- A sample generating or processing function, and its acompanying state. e.g. Oscillators, filters, etc
  - *f(state, inputs) -> (state, outputs)*
  - Class Ugen{ virtual Update(float*[] ins, float *[] outs); }
- In a dynamic system, the flow between units is explicitly represented by a “synchronous dataflow graph”



Graph evaluation

- Generators which produce signals must be evaluated before the generators which consume those signals, therefore: execute in a depth-first order starting from sinks.



Evaluation mechanisms

- Direct graph traversal
  - Simple, dynamic
  - Can't modify the graph while evaluating
- Execution sequence (list of function pointers, polymorphic object pointers, bytecodes)
  - Possibly more efficient, harder to modify
  - Decouples evaluation from traversal. Graph can be modified during traversal, e.g. different language for graph (e.g. SC synthdefs)

Block-based processing

- Process *arrays* of input samples and produces *arrays* of output samples
- Pros: more efficient (loop unrolling, SIMD etc)
- Cons: latency, feedback loops incur blocksize delay
- Vector size:
  - fixed (c.f. Csound krate)
  - variable (allows sample-accurate scheduling of notes, envelope breakpoints, etc.)

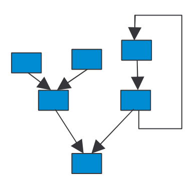
Vector allocation strategies

- One buffer/vector per generated signal, i.e. for every Unit Generator output.
- Reuse buffers once all sinks have consumed them (c.f. Graph colouring register allocation)

61 ICMC 2005 Workshop on Real-Time Systems Concepts for Computer Music

Feedback

- Don't visit a node more than once during graph traversal



- Save output from previous evaluation pass so it can be consumed during next evaluation
- Consider compression/saturation in feedback loops to avoid bad stuff happening

62 ICMC 2005 Workshop on Real-Time Systems Concepts for Computer Music

© 2005, Ross Bencina and Roger Dannenberg

Variations

- Hierarchical block sizes e.g. process subgraphs with smaller blocks to reduce feedback delay
- Synchronous multi-rate: separate evaluation phases using the same or different graphs (e.g. Csound krate/arate passes).
- Combine synchronous dataflow graph for audio with asynchronous message processing for control (e.g. Max/MSP)

Case Study: Audio over Network

Key Idea: Accurately-timed event-based scheduling

Event = computation of 32 sample audio block (every 32/44100 s)

Master clock (at sink) based on audio sample clock (no drift between clock and sample stream)

Assume some worst-case network latency (e.g. 50 ms = 0.05 s)

Schedule audio for time *t* to be computed at source at *t* – 0.05 s

Buffer source audio into 10 block (320 sample = 1280 byte blocks)

Send to sink every 10^{th} block time (every 320/44100 ≈ 7.2 ms)

Audio over Network Discussion

- Note that there is no flow control, acknowledgements, or extra messages.
- Use TCP/IP
  - Pro: reliable protocol
  - Con: lost data recovery is not “real-time”
  - Pro: packets almost never lost
- Limit the message rate
- Numbers are conservative choices – depends on network load, machine load, etc.

Case Study: Effects Processor with Graphical Control

- Separate effect into the part which runs in real-time and the part which exists in the graphics thread
- Keep state in both threads and mirror it (Proxy pattern)
- Message queue with commands to change values
  - messages could be paramId, value pairs, or functor objects (command pattern)
- In VST where the setValue call could come from any thread you need to know which thread you are in to know which methods to call.
- Alternative: use atomic updates to shared synthesis variables (makes it hard to do a group of updates together)

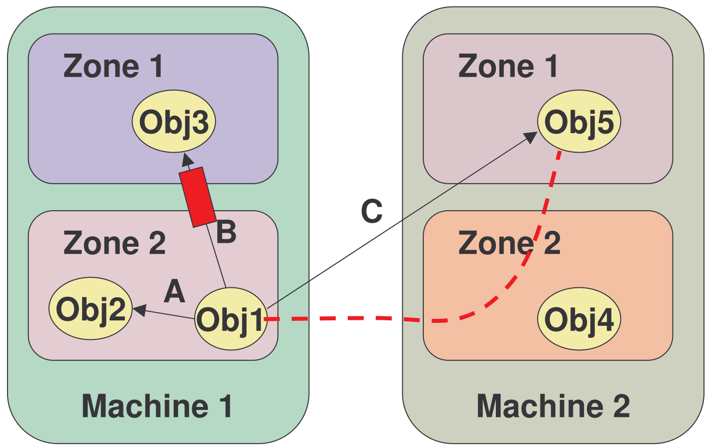
Aura Architecture

- Goal 1: General platform for interactive multimedia
- Goal 2: Open-ended, extensible for video, graphics, networking, software systems.
- Based on Real-Time Distributed Object System
- Objects have globally-unique 64-bit names
- Asynchronous messages
- Location independent

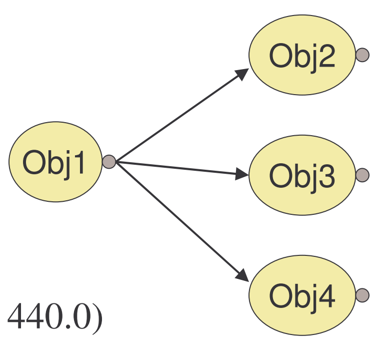
Communication with Aura

- Remote Method Invocation
  - *send_set_hz_to*(*osc*, 440.0)
  - Automatically generated macros to send messages
  - Receiver is indicated by globally unique ID
- Location Transparency
  - Object in same thread – synchronous call
  - Object in same address space – msg queue
  - Object on remote machine – TCP/IP to msg queue

Messages and Location Transparency



Output Ports



send_set_hz_to(*osc*, 440.0)

vs.

send_set_hz(440.0)

Aura Details

- Each Zone (thread + memory + scheduler):
  - Memory pool and real-time allocator
  - Calendar Queue-based scheduler
  - Time (seconds) based on audio sample count
- Pre-processor generates:
  - RPC message handlers
  - Stubs to pack parameters into msgs and send
  - Macros to make them easy to call
- Structure by *latency*, not *function*

Serpent Scripting Language

- Serpent virtual machine (everything the program/programmer sees) is a C++ object
- Multiple instances of Serpent give you multiple independently running systems
- One Serpent virtual machine per Aura zone
- Absolutely no shared variables, so use Aura messages
- Serpent objects can be tied to special Aura objects that relay Aura messages
- Real-time garbage collection limits GC *latency* to a constant time (can be set well below 1ms)

Audio Mulch Architecture

- Ross

Wrap-Up

- Thanks for attending!
- We'll be happy to discuss these and other issues throughout the ICMC and be email afterward.