Week 3 – Accurate Timing and Logical Time Systems

Roger B. Dannenberg
Professor of Computer Science and Art
Carnegie Mellon University

Reading Assignment

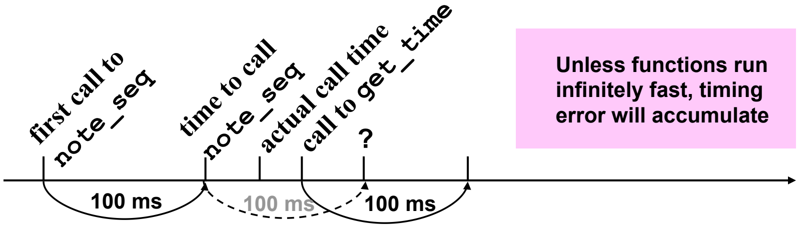
- Anderson, D. P. and Kuivila, R. 1990. A system for computer music performance. *ACM Trans. Comput. Syst.* 8, 1 (Feb. 1990), 56-82.
- David is a computer scientist
- Ron is a composer

(In)accurate Timing

- Consider this function to play a sequence of notes:

```
def note_seq()
    play_a_note_via_midi()
    schedule(get_time() + 0.1, nil,
                 'note_seq')
```

- Possible outcome:



Unless functions run infinitely fast, timing error will accumulate

Accurate Timing With Timestamps

- Scheduler records "ideal" time

```
rtsched_time = scheduled_wakeup_time;
apply(event.fn, event.parameters)
```

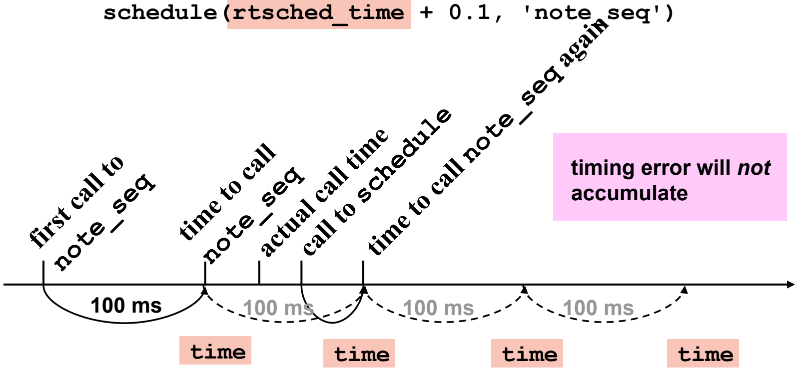
- Future scheduling in terms of "ideal" time, not real time.

```
def note_seq()
    play_a_note_via_midi()
    schedule(rtsched_time + 0.1,
                 'note_seq')
```

Note: `schedule` is pseudo code that takes an absolute time rather than relative time as in `sched_cause`

Example

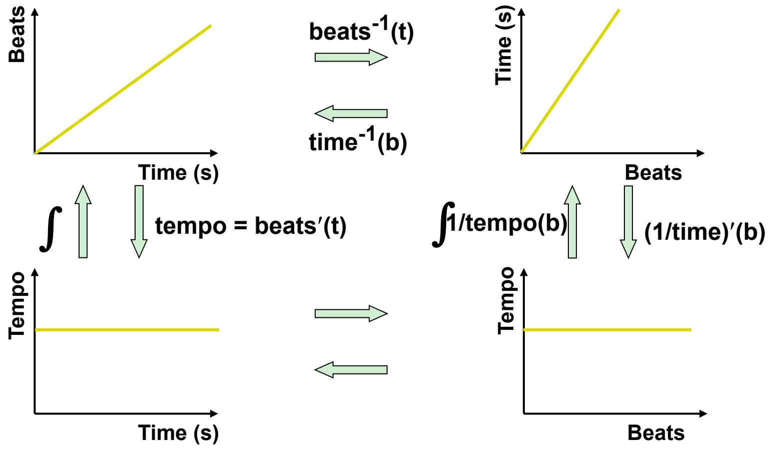
```
def note_seq()
    play_a_note_via_midi()
    schedule(rtsched_time + 0.1, 'note_seq')
```



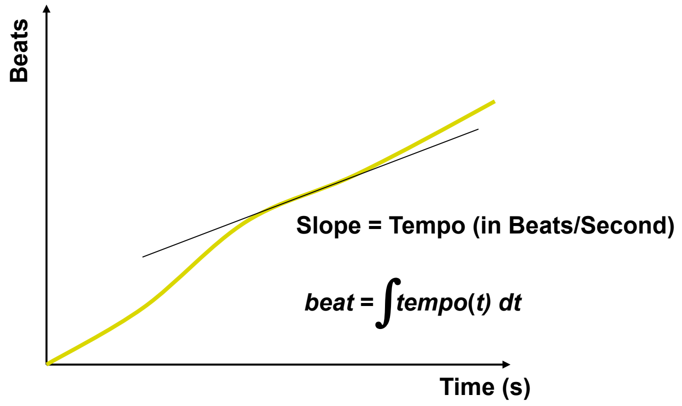
timing error will *not* accumulate

LOGICAL OR VIRTUAL TIME

Tempo, Time, Beats



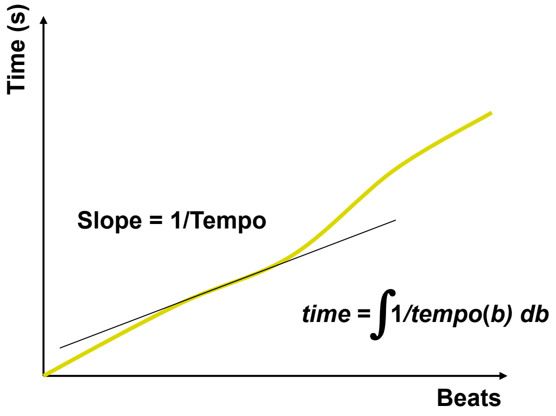
Tempo Curve



Slope = Tempo (in Beats/Second)

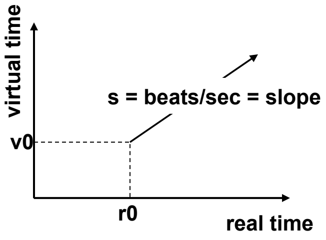
$$beat = \int tempo(t)\ dt$$

From Beats to Time



Logical Time (or Virtual Time)

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



$$v(r) = v0 + (r - r0) \cdot s$$

$$r(v) = r0 + (v - v0)/s$$

Using Logical (Virtual) Time

- If tempo is fixed and known in advance:
  - Scheduling is no problem: just map beats to seconds or seconds to beats as needed
- Interesting case:
  - You want to schedule according to beats
    - E.g. "play these notes on the next *beat*"
  - But *after* you schedule events, the *time map might change*
  - In particular, what happens if the tempo *speeds up?*

A Naïve Approach

- Schedule events as usual:
  - Map beats to seconds
  - Schedule according to the predicted time
- If the tempo changes:
  - Reschedule everything
  - Is this a good idea?
- What alternatives do we have?

Implementing Logical (Virtual) Time System

- 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
- Key idea:
  - If we sort events according to logical time (beats),
  - we only have to map the next event to real time.
  - When tempo changes, only one event needs to be remapped and rescheduled.

LTS Implementation

Invariants:

nxtlt == logi time of next event

a wakeup is scheduled at nxtlt

```
class Lts_event (Event)
  def run()
    lts_sched.wakeup(
              timestamp)

class Lts_sched
  var nxtlt
  var queue = Heap()
  def schedule(event)
    queue.add(event)
    // get next logi time
    lt = queue.peek().
              timestamp
    if nxtlt > lt
      reschedule(lt)
```

(These are also members of Lts_sched)

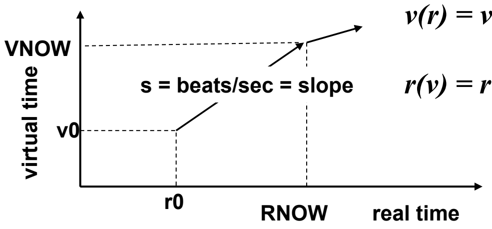
```
def reschedule(lt)
  nxtlt = lt
  // new wakeup event
  e = Lts_event(r(lt))
  RT_sched.schedule(e)

def wakeup(now)
  lt = v(now)
  if lt < nxtlt:
    return
  while lt >= nxtlt
    e = queue.get_next()
    nxtlt = queue.peek().
                 timestamp
    VNOW = e.timestamp
    e.run()
  reschedule(nxtlt)
```

LTS Change Tempo

```
// change tempo to bps beats per second
def lts_set_tempo(bps)
    r0 = r(VNOW)
    v0 = VNOW
    s = bps
    v = queue.peek().timestamp
    reschedule(v)
```

Reschedule because mapping changed



$v(r) = v0 + (r - r0) \cdot s$

$r(v) = r0 + (v - v0)/s$

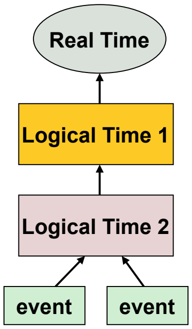
Should we cancel wakeups?

- Currently, we schedule a wakeup for
    - Any event that becomes the next event
    - The next event any time there is a tempo change
- Alternatives:
    - Cancel wakeups when virtual time changes
        - Avoids lots of event allocations
        - But scheduling an event is lightweight and fast – could be constant time if it matters
        - Cancellation requires a lot more bookkeeping – and cannot be faster than constant time
        - Depends on the scheduling algorithm

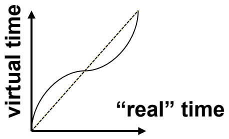
Cancelling wakeups (2)

- That was an argument against
- Imagine this:
  - Tempo is controlled by a Kinect controller, with tempo updates at 30Hz
  - Some events are scheduled far apart, e.g. 10s to next event
  - 300 events will fire around the same time if tempo is fairly steady, just to dispatch one “real” event
- Does this matter?

Composing Logical Time Systems



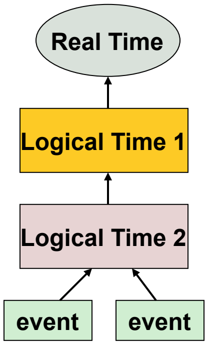
- Your logical time becomes my “real” time, e.g. my reference
- Clock synchronization
  - “Real time” according to local clock is shifted and stretched to match a remote clock
- Rubato, Expressive Timing
  - Anticipate the beat or “lay back”
  - Linger on certain note, rush others:



Composing Logical Time Systems

- $r(v) = r_1(r_2(v))$
- $v(r) = v_2(v_1(r))$

- $lts.r(v) = lts.parent.r(lts.r0 + (v - lts.v0)/lts.s)$
- $lts.v(r) = lts.v0 + (lts.parent.v(\mathrm{r}) - lts.r0)*lts.s$



Concepts

- Explicit timing is key
  - Specify *exactly* when things should run
  - Program order of execution is (largely) independent of real execution times
    - Makes debugging easier: more deterministic
    - In some systems, can run *out of real time,* e.g. for audio and graphics rendering
    - … or *faster than real time,* e.g. to generate and save MIDI file

Concepts (2)

- “System” (e.g. scheduler) and “Client” (e.g. objects) cooperate to specify timing
  - Client tells system:
    - how long things take,
    - time to next thing
    - i.e. the client implements the *model*
  - System tells client:
    - What is the time *within the model*
    - Delays client execution by not dispatching events when event time > real time
    - Runs as fast as possible while event time < real time

Concepts (3)

- Virtual or Logical Time
  - Model for:
    - Variable speed, variable tempo
    - Clock synchronization
    - Anticipating events to compensate for latency
    - Rubato and expressive timing
  - Possible to compose logical time systems hierarchically

FORMULA

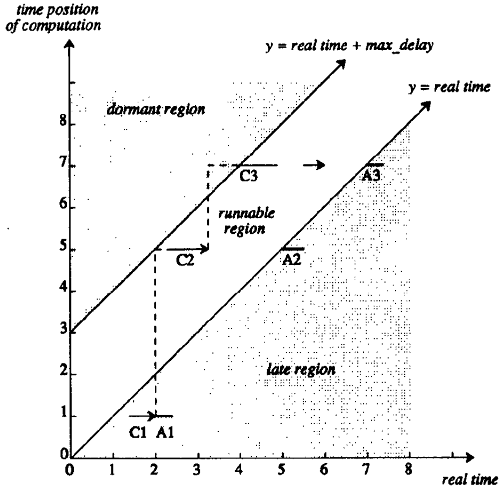
Why FORMULA?

- Formula was one of the first computer music languages to deal carefully with timing issues
- Formula is described in detail in a journal article
- For more recent and related work, see papers on ChucK (Ge Wang's PhD work at Princeton)
- Also my NIME paper in 2011 with Dawen Liang and Gus Xia

The Basics

- `create_process(procedure, arguments)`
- `time_advance(delay)`
- *real time* – based on clock interrupts
- *system time* – scaled by `global_tempo`, may stop to allow system to catch up
- *action computation* vs. *action routine*
  - Compute what to do in advance of real time (on the assumption that computation can be expensive, but can run in advance)
  - Perform the action at a precise time (on the assumption that outputting pre-computed data is *not* expensive)
  - `schedule_action(proc, args)`
  - `schedule_future_action(delay, proc, args)`

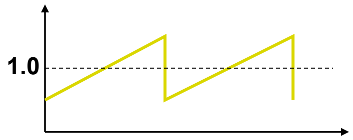
Timing in FORMULA



Time Deformation

- Per-process virtual time
- Time deformation defined by coroutine
  - Procedural programming makes a sequence of calls to `td_segment(from, to, duration)`
  - System runs coroutine as far as necessary

```
for (i = 0; i < 2; i++) {
    td_segment(0.5, 1.5, 1.0);
}
```



- Product td and serial td

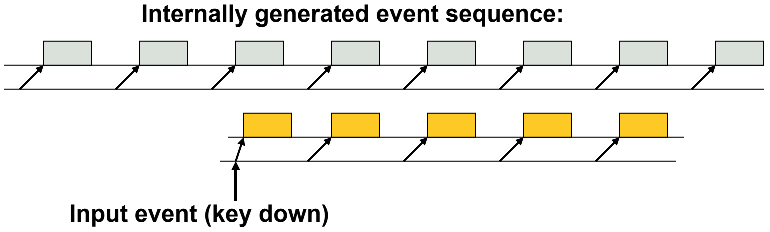
Control Structures

- maxtime(n) *statement*
- mintime(n) *statement*
- minloop(n) *statement*

- Question: how does the control construct take control of the inner *statement*?

Input Handling

- Set process time position to time of the event
- Let the process run until it is ahead of ST + max_delay
- Example:

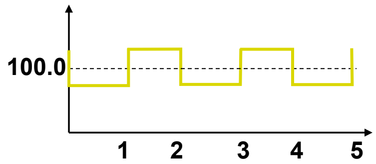


Internally generated event sequence:

Input event (key down)

"Continuous" Control – not in paper

- Just as time deformation is specified procedurally,
- FORMULA allows procedural specification of things like volume control, pitch bend, etc.
- Done with co-routines
- E.g. accent 2 and 4:



```
while (true) {
    control_segment(VOL, 80, 80, 1);
    control_segment(VOL, 120, 120, 1);
}
```

Wrapping Up

- Calculate “ideal” time to perform action as well as the action itself
- Use scheduling so that “ideal” time is approximately real time
- Cumulative timing errors should only be limited by numerical accuracy
- Virtual/Logical time allows for tempo, clock synchronization, and speed control. Same principle: compute “ideal” time and scheduling accordingly.
- FORMULA:
  - action buffering for more precise timing
  - procedural specification of time deformation

Week 3 – Day 2
Event Buffering, Forward Synchronous

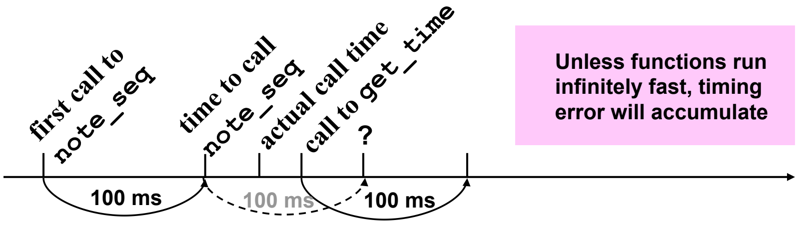
Roger B. Dannenberg
Associate Research Professor of
Computer Science and Art
Carnegie Mellon University

Review: (In)accurate Timing

- Consider this function to play a sequence of notes:

```
def note_seq()
    play_a_note_via_midi()
    schedule(get_time() + 0.1, 'note_seq')
```

- Possible outcome:



Unless functions run infinitely fast, timing error will accumulate

---

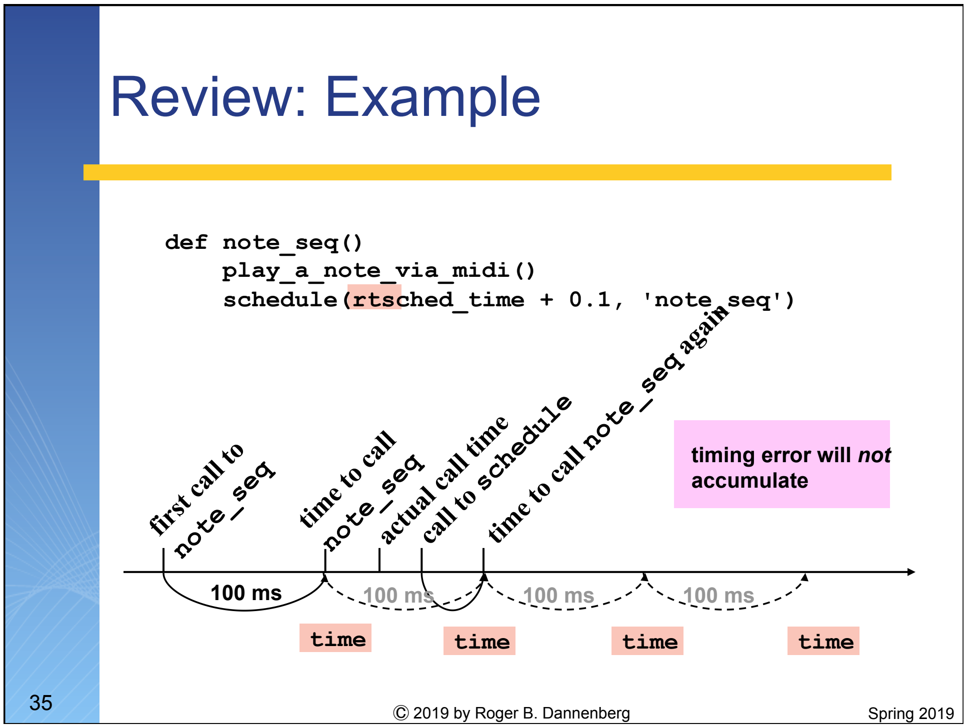
Review: Accurate Timing With Timestamps

- Scheduler records "ideal" time

```
rtsched_time= scheduled_wakeup_time;
apply(event.fn, event.parameters)
```

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

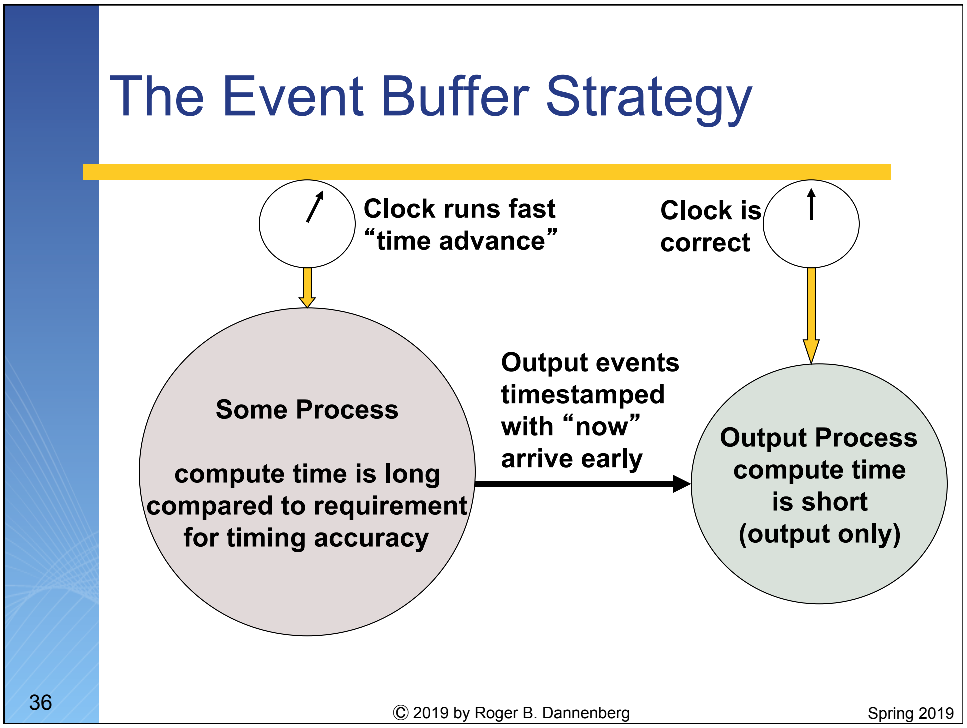
```
def note_seq()
    play_a_note_via_midi()
    schedule(rtsched_time + 0.1,
             'note_seq')
```



Review: Example

```
def note_seq()
    play_a_note_via_midi()
    schedule(rtsched_time + 0.1, 'note_seq')
```

timing error will *not* accumulate



The Event Buffer Strategy

Clock runs fast "time advance"

Clock is correct

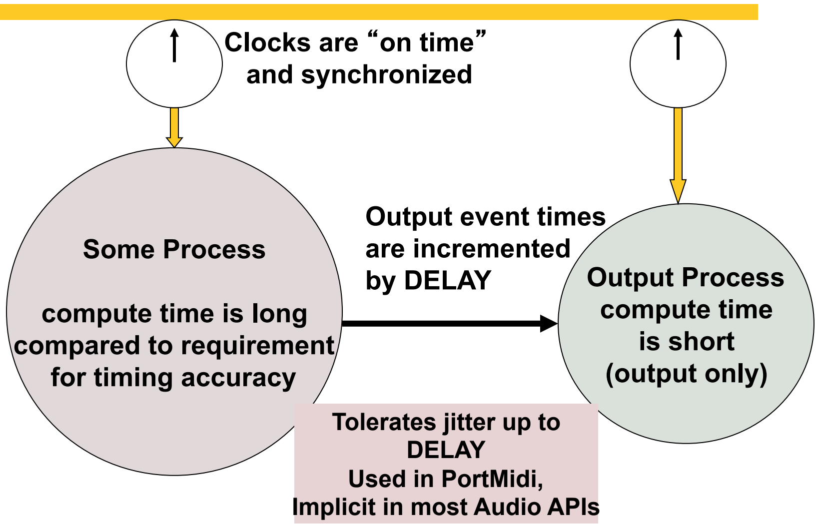
Some Process

compute time is long compared to requirement for timing accuracy

Output events timestamped with "now" arrive early

Output Process compute time is short (output only)

Almost Equivalent: Delayed Output



Clocks are “on time” and synchronized

Some Process

compute time is long compared to requirement for timing accuracy

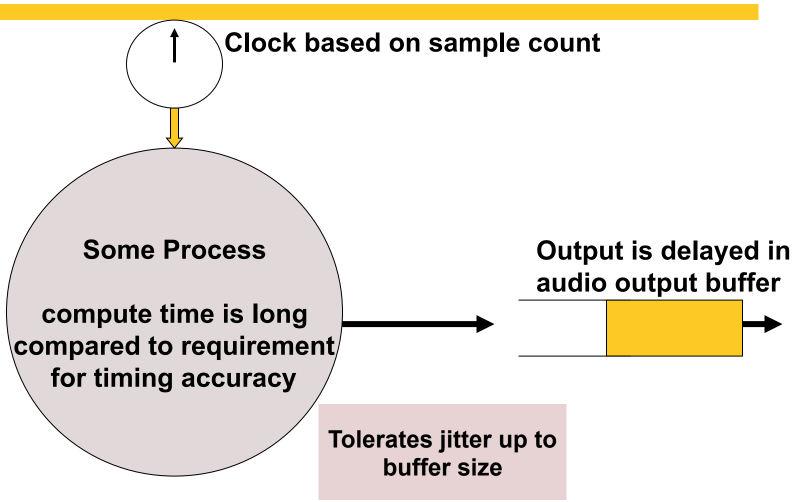
Output event times are incremented by DELAY

Output Process

compute time is short (output only)

Tolerates jitter up to DELAY
Used in PortMidi,
Implicit in most Audio APIs

Delayed Output and Audio Processing



Clock based on sample count

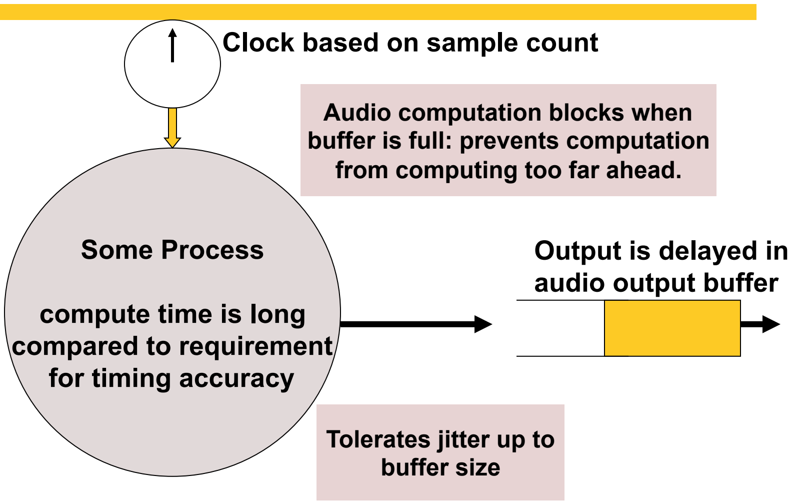
Some Process

compute time is long compared to requirement for timing accuracy

Output is delayed in audio output buffer

Tolerates jitter up to buffer size

Delayed Output and Audio Processing (2)



Clock based on sample count

Audio computation blocks when buffer is full: prevents computation from computing too far ahead.

Some Process

compute time is long compared to requirement for timing accuracy

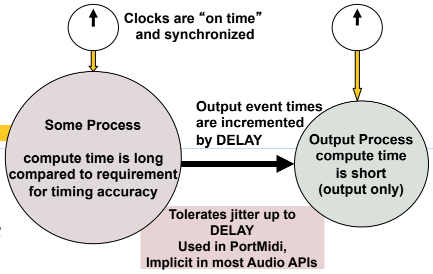
Output is delayed in audio output buffer

Tolerates jitter up to buffer size

Event Buffers Everywhere

- Audio:
  - Disk I/O in audio playback typically runs well ahead of sample output to device
  - Application is called to fill output buffers as soon as they are empty (way before audio is played)
  - Device driver sets up DMA transfer to device before samples are needed
  - Digital-to-Analog Converter loads next sample to internal register ahead of sample clock
  - Ultimately, sample clock gives <1ns jitter
- MIDI
  - Sequencers load sequence data to RAM
  - Typically send time-stamped sequence data to a low-latency output process
- VoIP
  - Network packets (high jitter) are buffered before playback

Delayed Output Example



Clocks are "on time" and synchronized

Some Process

compute time is long compared to requirement for timing accuracy

Output event times are incremented by DELAY

Output Process compute time is short (output only)

Tolerates jitter up to DELAY
Used in PortMidi,
Implicit in most Audio APIs

- Scheduler:

```
now = event.time;
event.run()
```

- Application:

```
midi.send(status, data1, data2);
```

- MIDI Output:

```
def send(…) {
  ShortMessage message = …
  midi_write(message, now);
}
```

An Aside: PortMidi timing

- `midi_open_output(midi, devno, buffer_size, latency)`
- `midi_write(midi, time, msg)`
- latency is the delay in milliseconds applied to timestamps to determine when the output should actually occur.
- If latency is zero, timestamps are ignored and all output is delivered immediately.
- If latency is greater than zero, output is *delayed* until the message timestamp plus the latency.
- So behavior of previous slide is built-in.

Schedulers and Event Buffers

- Recall FORMULA
- Uses scheduler to compute outputs with accurate logical time
- Compute slightly ahead of real time
- Schedule output actions at precise output times
  - When to schedule output? Use the logical time.

Discussion

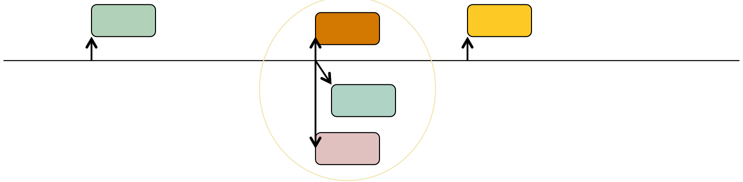
- Provides an *absolute* timestamp to specify MIDI (or other) output time
  - independent of run time and scheduling delays
- Potentially passes accurate timing all the way down to the MIDI device driver
- MIDI will not be output instantly due to timestamp.
  - Is this delay bad?
  - Audio gets buffered too; this might actually *help* to synchronize audio and MIDI
- Aside: Java is vague about how to work with timestamps
  - In particular, *what is the reference time?*
  - *E.g. how do I synchronize to the audio sample clock?*
  - These questions are addressed in PortMidi

Extension for using MIDI *input*

- Problem: you may not see MIDI data immediately
- “jitter in, jitter out”
- Solution:
  - Get timestamps from MIDI device driver
  - 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
- Issue: what if time goes backward?
  - (A timestamped event may set “NOW” to be earlier.)
- No general solutions here.

Concurrency and Precise Timing

- Events are ordered in time
  - Need the results (state changes) of one event before running the next event (usually)
  - Could run *simultaneous* events in parallel
    - Must be *very* careful with shared state updates
  - Are simultaneous events common?
  - No general solutions here.



Concurrency and Precise Timing (2)

- Sometimes you can partition the application into independent synchronized processes:

MIDI file player

Audio file player

Synthesizer

- Each can run a scheduler
- All schedulers share a time source
  - Or else synchronize their clocks – details later
- What if there are dependencies?

Problem 1: Asynchrony

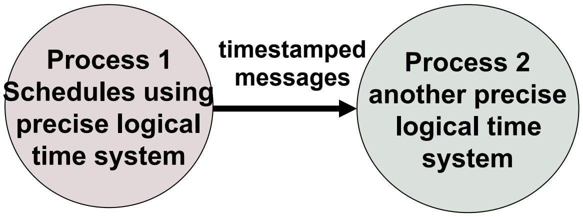
- What could go wrong?
- Process 1 has several events at time *t* that change some state,
- Process 2 runs events at *t* that depend on shared state
- ➔result is a race condition between Process 1 and 2
  - non-atomic updates to shared state could cause problems
  - (could insist on locks around all shared state)
- Why isn't this a problem with a single thread?
- **Partial Solution**:
  - Process 1 sends timestamped events to Process 2 through a FIFO to update non-shared state
  - Process 2's scheduler moves events from FIFO into the future event list
  - Now, events from Process 1 are handled synchronously with respect to every other event in Process 2. Updates happen before or after Process 2 events, but not *during* events.

Problem 2: Ordering in Time

- What could go wrong?
- Process 1 event at time T – ε changes flag to false to disable output
- Process 2 event at time T checks a flag for true and computes output
- If Process 1 runs late by more than ε, Process 2 computes output anyway
- How would this work with a single thread? What if the computation runs late by more than ε?

Ordering in Time (2)

- Suppose Process 2 is like an event buffer.



Process 1
Schedules using precise logical time system

timestamped messages

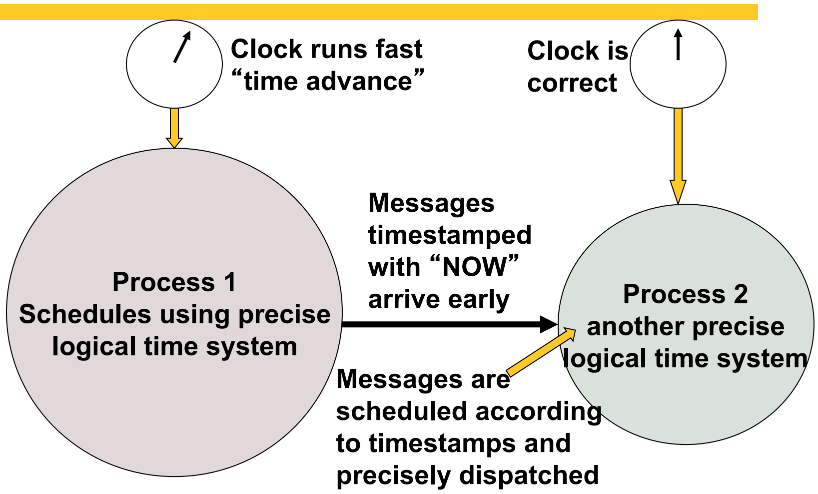
Process 2
another precise logical time system

- Suppose Process 1 runs Δ ahead of real time, where the total delay from Process 1 to Process 2 < Δ
- Output from Process 1 to Process 2 is timestamped
- Any output from Process 1 at logical time T will update Process 2 at logical time T: precise timing + concurrency!

Forward Synchronous

- I coined the term “forward synchronous” for this:
  - “Forward” because it is one-way, e.g. from input to output.
  - “Synchronous” because if you schedule everything as we’ve described (logical time systems, accurate timing), then everything is deterministic and well-ordered.
- Brandt and Dannenberg (1999), “Time in Distributed Real-Time Systems,” in *Proc. ICMC*.

Forward Synchronous (2)



Clock runs fast “time advance”

Clock is correct

Process 1 Schedules using precise logical time system

Messages timestamped with “NOW” arrive early

Process 2 another precise logical time system

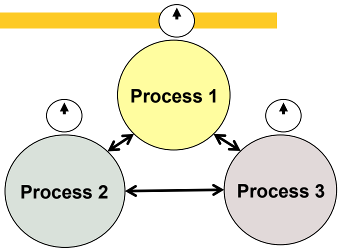
Messages are scheduled according to timestamps and precisely dispatched

Forward Synchronous (3)

- Advantages
  - Works well with separation of control and synthesis
    - E.g. music generation, sequencers, user interface in Process 1
    - … software synthesis in Process 2
  - Output timing can be precise even when connection has high latency, e.g. network
  - Failure mode is reasonable – late messages are handled ASAP, fallback is to asynchronous control (such as MIDI)
- Disadvantages
  - One-way: at best, mutual dependencies require delays or out-of-time-order processing
  - “Time advance” (running on scheduler ahead of real time) can be confusing: you have two logical time systems that are offset from one another

---

Distributed Precisely Timed Systems



- A reasonable compromise in a general distributed system (laptop orchestras?) is timed messages but explicit time advance
- All processes use the same clock (no built-in time advance)
- To get “Forward Synchronous” behavior: add time advance to timestamp when you send a message to another process
- To get asynchronous, ASAP behavior, use current time (or just 0 which implies the message is late) so message will be processed immediately on arrival

Summary

- Discrete Event Simulation showed us how to compute times precisely
  - Why do we care? Avoid drift. Deterministic behavior is easier to debug.
- Real Time Schedulers extend the idea simply by pausing until logical time = real time
  - Gives illusion of infinitely fast CPU with precise scheduling
- Event Buffering and more generally Forward Synchronous systems extend precise timing across otherwise asynchronous processes:
  - Application and device driver
  - Processes separated by networks, etc.