# Computer Music Systems and Information Processing

## Week 2: Discrete Event Simulation

Roger B. Dannenberg, instructor
Spring, 2019

---

# Discrete Event Simulation

- Why DES?
- Overview
- Approaches
- Details of Event Scheduling Systems

Copyright (c) 2018 Roger B. Dannenberg

## Why Discrete Event Simulation

- Most CS is concerned with computing answers at some time in the future (hopefully soon)
- Discrete event simulation models *time* as well as processes
- DES techniques turn out to be very relevant to real-time, interactive music systems.

## Overview of Discrete Event Simulation

- Model behavior …
- … consisting of discrete state changes
- State: all information describing system at a given point in time
- Event: a trigger that initiates an action that changes the state
- Process: a sequence of events, usually associated with some simulated entity

# Simulation Structure

- Executive: manages events and runs actions, keeps track of time
- Utilities: random number generation, statistics, etc.
- Model (program): code that models the specific behavior of the simulated system.
  - Note the design strategy: separate system-specific code from generic, reusable code.

Copyright (c) 2018 Roger B. Dannenberg

---

# Time in Simulation

- Usually, simulation time is not real time.
- Results of simulation available faster than real time (e.g. weather, climate simulation)
- Simulation time
- Run time
- Event time

Copyright (c) 2018 Roger B. Dannenberg

## Time in Simulation

- Usually, simulation time is not real time.
- Results of simulation available faster than real time (e.g. weather, climate simulation)
- Simulation time: the time in the model
  - Also logical time or virtual time
- Run time: the real (cpu) time of the simulation
- Event time: the simulation time at which an event should happen

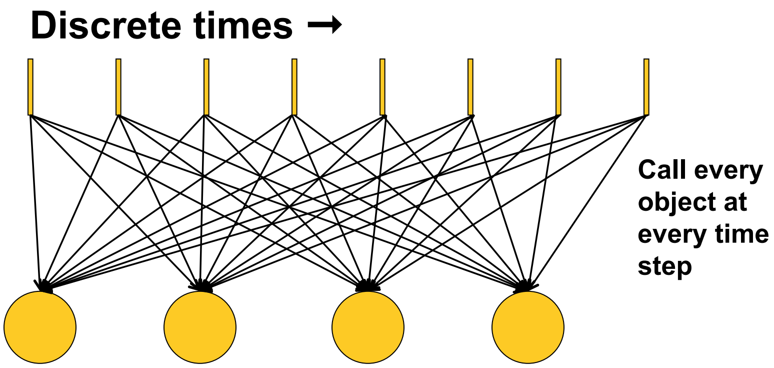
Copyright (c) 2018 Roger B. Dannenberg

## Scheduling Events

- (1) Synchronous simulation
  - What it does:
    - Advance virtual time by small fixed interval
    - Run all events in that interval
  - Timing not precise
  - Wastes computation when no events enabled
  - Similar to frame-by-frame animation engine
- (2) Event-scanning simulation:
  - Advance time to next event time
  - Run the event

Copyright (c) 2018 Roger B. Dannenberg

# (1) Synchronous Simulation



**Objects being simulated**

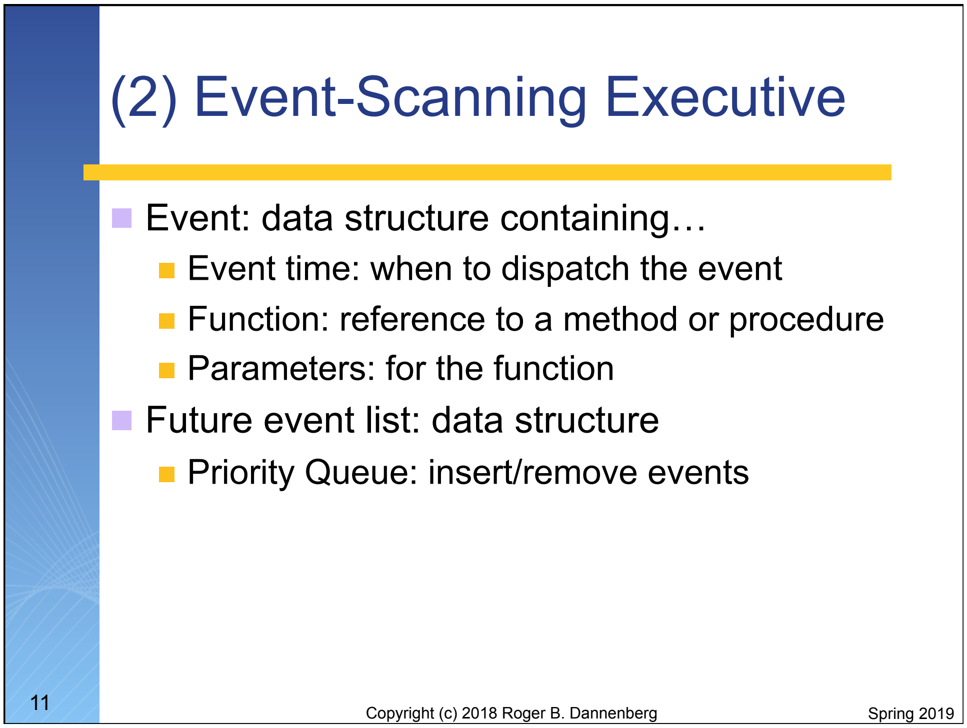
Copyright (c) 2018 Roger B. Dannenberg

---

# Question:

- What are some examples where synchronous simulation makes sense?
  - Computer animation – everything changes frame-by-frame; maybe drawing dominates cost (so testing for behavior is insignificant)
  - Physical simulation, difference equations, everything changes each time step
  - Continuous music control, e.g. objects generate envelopes, vibrato, etc. and change “continuously” (i.e. at each time step)

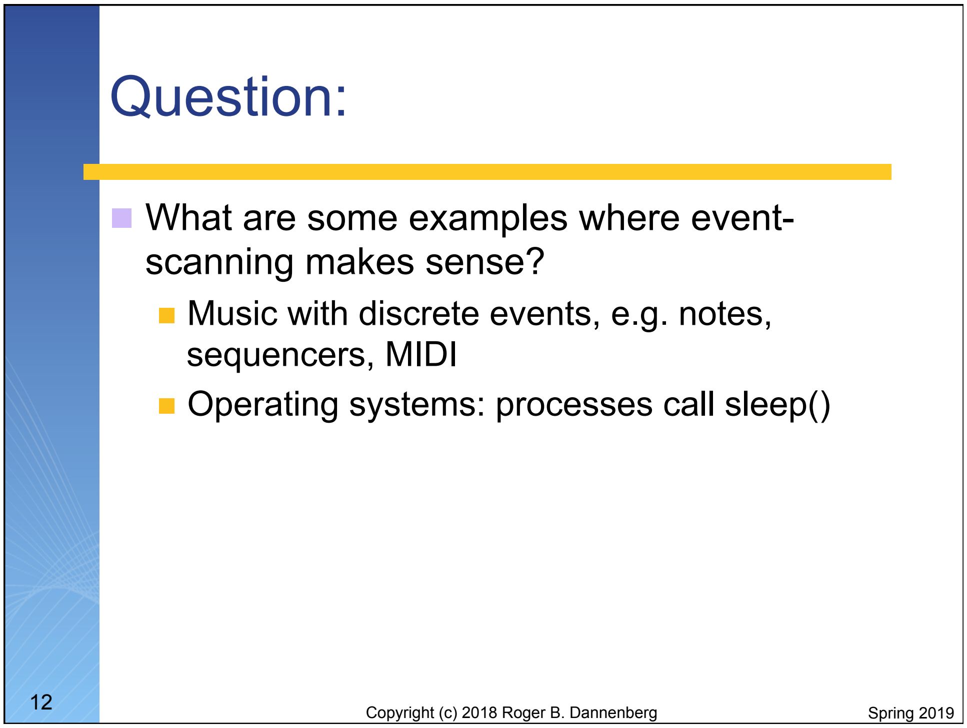
Copyright (c) 2018 Roger B. Dannenberg



# (2) Event-Scanning Executive

- Event: data structure containing…
  - Event time: when to dispatch the event
  - Function: reference to a method or procedure
  - Parameters: for the function
- Future event list: data structure
  - Priority Queue: insert/remove events

Copyright (c) 2018 Roger B. Dannenberg



# Question:

- What are some examples where event-scanning makes sense?
  - Music with discrete events, e.g. notes, sequencers, MIDI
  - Operating systems: processes call sleep()

Copyright (c) 2018 Roger B. Dannenberg

# Simulation Organization

- Activity Scanning
- Event Scanning/Scheduling
- Process Interaction

Copyright (c) 2018 Roger B. Dannenberg

# Activity Scanning

- Organized around the activity
- Activities start when conditions are met
- Executive scans for an activity that is enabled

- Makes sense when enabling conditions are complex, e.g. particle systems, crowds, physics with collision detection
- Amounts to synchronous simulation with polling to decide when to perform events

Copyright (c) 2018 Roger B. Dannenberg

# Event Scheduling

- Organized around the event
- Activities change the state, figure out time of next event and schedule events accordingly
- Appropriate when
  - Interactions are limited
  - Precise timing is important
  - Timed state changes

Copyright (c) 2018 Roger B. Dannenberg

# Process Interaction

- Organized around the Process
- Modeled entities represented by processes
- Processes can wait to simulate passing of time
- Processes can use synchronization, e.g. semaphores or condition variables to represent interdependencies.

Copyright (c) 2018 Roger B. Dannenberg

# Example

- Manufacturing

**Step 1** → **Step 2** → **Step 3**

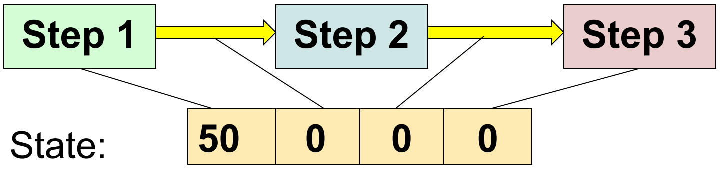
- Four ways to approach this (at least):
  - Activity scanning model
  - Event scheduling model
  - Process interaction model
    - Process could be a manufactured article
    - Process could be a manufacturing station
- (discuss each of these four approaches)

Copyright (c) 2018 Roger B. Dannenberg

---

# Event Model in More Detail

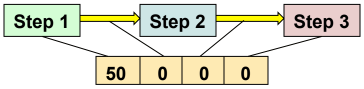
- Why focus on event model? Because we're going to be using the event model for music generation.



- State:
  The number of objects at each position
- Event: Begin or end a step

Copyright (c) 2018 Roger B. Dannenberg

# Events



Take something out of the input queue

```
def start_step(i):
    if s[i] > 0:
        s[i] = s[i] – 1
        schedule(now + dur(i), ‘event_done’, i)
```

Add to output and process next input

```
def event_done(i):
    s[i+1] = s[i+1] + 1 // increment output tray
    start_step(i) // begin work on next input
```

- Bugs:
  - Need range check on `i`, there’s no step 3, 4, 5, …
  - If step is inactive and item is added to input tray, need to start the step. How do we implement this?

Copyright (c) 2018 Roger B. Dannenberg

# Events and Time

- Virtual time *does not advance* while events are “running” – being computed.
- Virtual time *only advances* when the next event is in the future.
- If there are multiple events at the same virtual time, *time does not advance* until all are computed: Arbitrary amounts of computation in zero logical time.
- Event “duration” is zero, but we can model process begin, middle, end as multiple events

Copyright (c) 2018 Roger B. Dannenberg

## Processes/Threads

- A “natural” way to model behaviors
- But processes can be “heavy”
  - Stacks
  - Context switch must swap all registers
- Processes must coordinate updates to the state. Consider this in parallel:
  - `s[i] = s[i] + 1`
  - `s[i] = s[i] - 1`

Copyright (c) 2018 Roger B. Dannenberg

---

## Processes and Time

- How can we model time with Processes?
- *Virtual time should only advance when processes block or sleep.*
- Blocking and sleeping are typical OS primitives, but not designed for simulation
- Coordinating multiple CPUs to order computation with respect to virtual time is tricky and beyond our (current) discussion
- Processes are not recommended for DES

Copyright (c) 2018 Roger B. Dannenberg

# Coroutines

- Similar to processes, but
- “synchronous” in that threads must explicitly yield:
  - sleep() – run other threads for some amount of virtual time
  - semaphores and condition variables – block until ready to run, run other threads in the meantime
  - yield() – just pick another thread and run
- Consider:
  - `s[i] = s[i] + 1`
  - `s[i] = s[i] - 1`

*Each of these is an atomic operation: it runs to completion with no possibility of another thread seeing intermediate results.*

Copyright (c) 2018 Roger B. Dannenberg

---

# Coroutines and Logical Time

- How can we model time with coroutines?
- Sleep(): inserts an event in a priority queue
  - Advance virtual time to next event in queue (virtual time does not necessarily change)
  - Event wakes up the sleeping coroutine
- Yield(): checks and switch to ready-to-run coroutines.
- Assessment:
  - Similar to Event Model
  - Ability to suspend (sleep) within a procedure

Copyright (c) 2018 Roger B. Dannenberg

# Coroutines, Threads, Serpent

- Until last year, this discussion of coroutines was mainly for completeness, since few languages support them.
- Serpent has a new feature: non-preemptive threads!
- Note: Python can implement coroutines by building on generators; semantics are different
- Coroutines in Serpent are called “threads” (sorry, but “non-preemptive threads” was too verbose)
- Serpent scheduler works with threads

Copyright (c) 2018 Roger B. Dannenberg

---

# Simulating a Sequence with Serpent Threads

```
def myseq():
    if fork(): return
    // computation in STEP1
    sched_wait(DUR1)
    // computation in STEP2
    sched_wait(DUR2)
    // computation in STEP3
    sched_wait(DUR3)
    …
```

Copyright (c) 2018 Roger B. Dannenberg

# Simulating a Sequence with Events

```
def myseq():
    if state == START:
        state = STEP1
        // computation in STEP1
        schedule(now + DUR1, ‘myseq’)
    elif state == STEP1:
        state = STEP2
        // computation in STEP2
        schedule(now + DUR2, ‘myseq’)
    elif state == STEP2:
        state = STEP3
        // computation in STEP3
        ...
```

What if we want multiple instances of myseq?

Copyright (c) 2018 Roger B. Dannenberg

# Events and Instances

```
def myseq(state):
    if state == START:
        state = STEP1
        schedule(now + DUR1, ‘myseq’, state)
    elif state == STEP1:
        state = STEP2
        schedule(now + DUR2, ‘myseq’, state)
    elif state == STEP2:
        state = STEP3
        ...
```

Now we can launch many instances of `myseq(START)`

But, what if we need local state for each instance?

Copyright (c) 2018 Roger B. Dannenberg

# Events and Instances (2)

```
def myseq(state, per_inst):
    if state == START:
        state = STEP1
        schedule(now + DUR1, ‘myseq’,
                 state, per_inst)
    elif state == STEP1:
        state = STEP2
        schedule(now + DUR2, ‘myseq’,
                 state, per_inst)
    elif state == STEP2:
        state = STEP3
        ...
```

Now we can launch many instances of `myseq(START, x)` where each instance has its own local state (`x`).

Copyright (c) 2018 Roger B. Dannenberg

---

# Simulating a Sequence with Events – A Variation

```
def myseq_step1():
    schedule(now + DUR1, ‘myseq_step2’)

def myseq_step2():
    schedule(now + DUR2, ‘myseq_step3’)

def myseq_step3():
    schedule(now + DUR3, ‘myseq_step4’)


...
```

Copyright (c) 2018 Roger B. Dannenberg

# Object Oriented Simulation

- In previous slide, what if we wanted more state?
  - Events solution requires us to pass state or at least a reference to state through parameters.
- Let's look at Object Oriented simulation as another way to deal with instances and state.

Copyright (c) 2018 Roger B. Dannenberg

# Object Oriented Approach

- Scheduling an event effectively calls a procedure in the future
  - Procedures are not the best model for entities in the real world (that have state)
  - A previous example shows how we can pass state through parameters
- What if we extend events to activate objects?
  - Object can hold state – no need to encode into parameters
  - Objects can model entities in the real world

Copyright (c) 2018 Roger B. Dannenberg

# Object Oriented Approach

```
class My_thing (Event):
   var state // an instance variable
   def init():
      state = START
   def run():
      switch (state):
        START:
          state = STEP1
          schedule(now + DUR1)  // inherited method
        STEP1:
          state = STEP2
          schedule(now + DUR2)
        STEP2:
          state = STEP3
        ...
```

Copyright (c) 2018 Roger B. Dannenberg

# Object Oriented Approach (2)

- Notes on the previous slide:
  - Everything can be statically type checked
    - schedule() takes an object of type Event rather than a procedure reference
    - no dynamically typed parameters to pass to an event
  - State is encapsulated in objects
  - Slightly clumsy that every schedule results in a call to `run()`
    - Some languages allow you to pass pointers to methods, similar to passing function pointers in the Event Model
    - OO languages allow you to pass objects that could then invoke the desired method, but this could be clumsy too

Copyright (c) 2018 Roger B. Dannenberg

# Summary

- Discrete Event Simulation computes (virtual) time as well as state changes
- Event Scheduling can compute timing with high precision (no rounding to discrete intervals or system clock)
- Various approaches:
  - Processes – heavy and hard to manage time
  - Coroutines – stacks, context switch, relatively easy to incorporate virtual time
  - Objects – lighter weight, popular for models
  - Events – very lightweight, simple

Copyright (c) 2018 Roger B. Dannenberg

---

Copyright (c) 2018 Roger B. Dannenberg

# Computer Music Systems and Information Processing

## Week 2, Day 2: Discrete Event Simulation, Scheduling

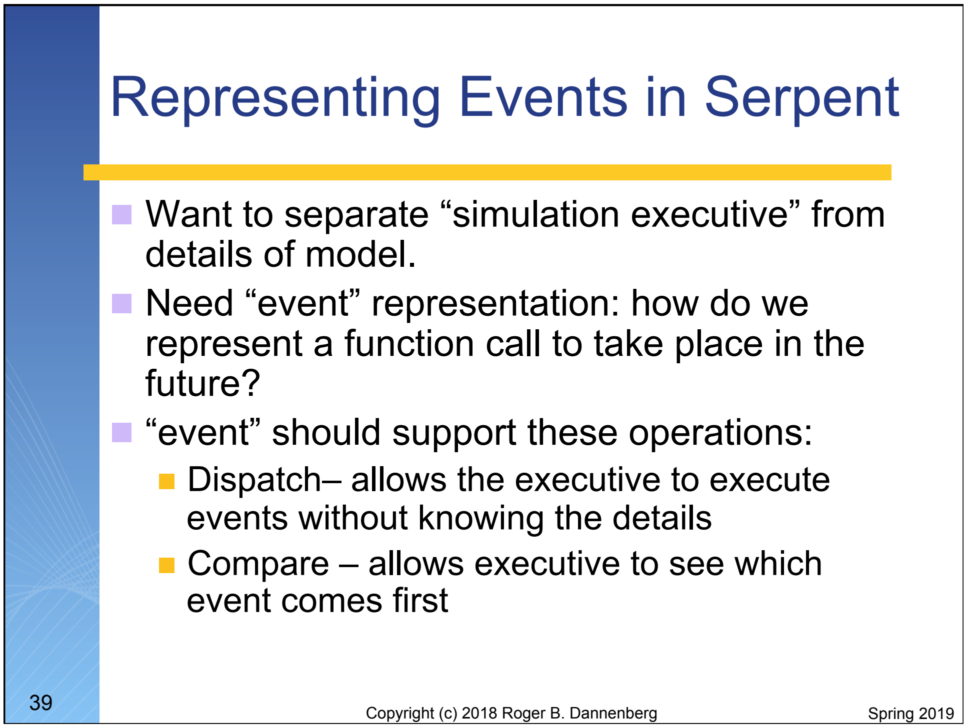
Roger B. Dannenberg, instructor
Spring, 2019

---

# Review

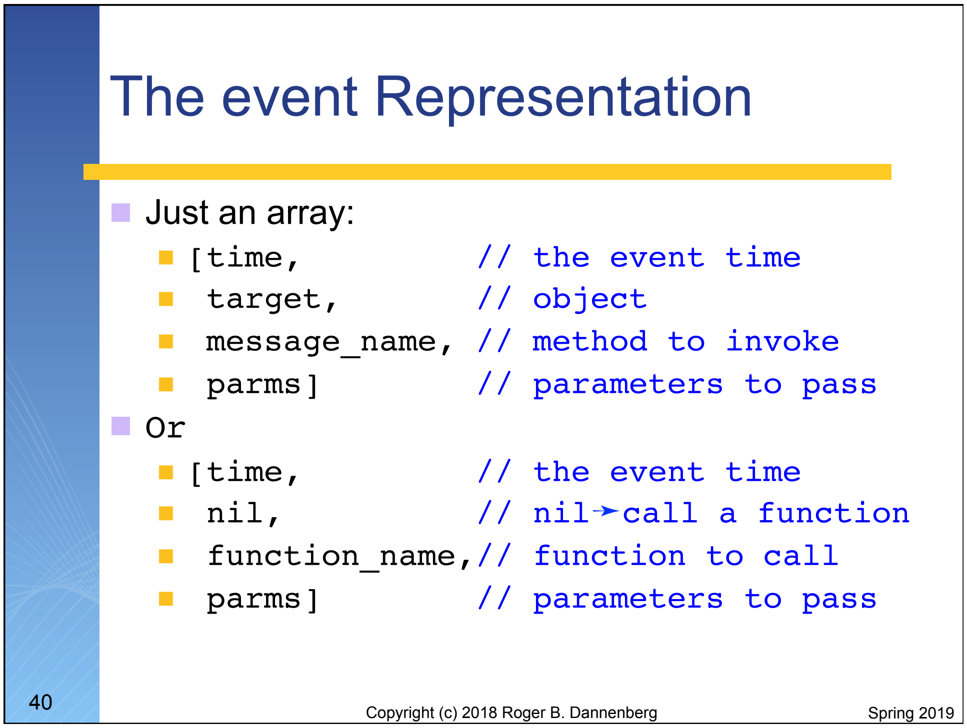
- Discrete – at points in time
- Event – state changing action
- Simulation – a model
- Big ideas:
  - All behavior modeled as instantaneous events
  - Compute precise times of events
  - Future events in a priority queue (sort by time)
  - Perform events in time order
  - Always keep track of virtual (simulated) time

Copyright (c) 2018 Roger B. Dannenberg



## Representing Events in Serpent

- Want to separate "simulation executive" from details of model.
- Need "event" representation: how do we represent a function call to take place in the future?
- "event" should support these operations:
  - Dispatch– allows the executive to execute events without knowing the details
  - Compare – allows executive to see which event comes first



## The event Representation

- Just an array:
  - `[time,          // the event time`
  - `target,         // object`
  - `message_name,   // method to invoke`
  - `parms]          // parameters to pass`
- `Or`
  - `[time,          // the event time`
  - `nil,            // nil➝call a function`
  - `function_name,  // function to call`
  - `parms]          // parameters to pass`

# Dispatching an event

```
def general_apply(target, message, parms)
    if target
        sendapply(target, message, parms)
    else
        apply(message, parms)

// note: parms is an array, e.g.
//
//   general_apply(synth, 'play', [60, 100])
//   is same as: synth.play(60, 100)
//
//   general_apply(nil, 'foo', [10, "hi"])
//   is same as: foo(10, "hi")
```

# Implementing a scheduler (a “simulation executive”)

- Need two operations:
  - Schedule (“cause”) an event: the event is remembered and dispatched at the specified event time
  - Poll:
    - advance virtual time to the earliest event time,
    - if real time >= virtual time
    -     dispatch the earliest event
- Design choice:
  - Simulation executive can be a process that polls
  - It can be our responsibility to call poll() frequently

# Implementation 1: linked list

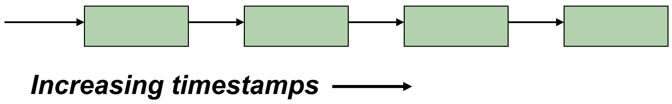
- To *schedule*, insert *event* at the head of a list

```
while len(list) > 0: // run until done
    for each r in list
        if r[0] < now
            list.remove(r)
            dispatch(r)
    now += interval
```

- Example of Event Scanning
- Problems:
  - Could run events out of order
  - Searches entire list very often
  - Bug: changing list while iterating over list is not allowed

# Implementation 2: priority queue

- Like before, but linked list is sorted:

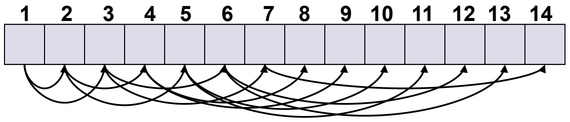


```
while len(list) > 0: // run until done
    var r = list[0]
    list.remove(r)
    now = r[0] // get the event time
    dispatch(r)
```

- Problems:
  - Scheduling (insertion) is linear in size of list
  - In Serpent, list.remove(r) is linear as well

# Implementation 3: heapsort

- Trick: embed complete binary tree in array.



  - $parent(n) = floor(n/2)$
  - $left\_subtree(n) = 2 * n$
  - $right\_subtree(n) = 2 * n + 1$
- Heap invariant:

  No parent is greater than its children

  It follows that the root is the minimal element

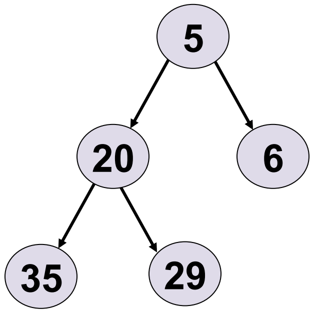
Copyright (c) 2018 Roger B. Dannenberg

# Heapsort (2)

- After removing least element (array[1]), move last array element to first. Then, “bubble” down the tree by swapping new element with least of two children (iteratively) until no child is smaller.
- To add an element, insert at end of array. Then “bubble” up the tree by swapping new element with parent until parent is smaller.
- Log(n) insert and delete.

Copyright (c) 2018 Roger B. Dannenberg

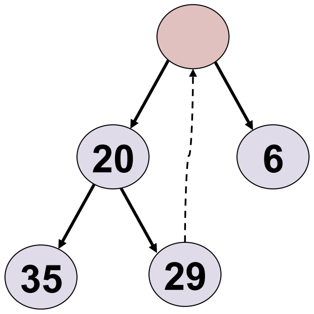
# Heapsort (3): Remove



Copyright (c) 2018 Roger B. Dannenberg

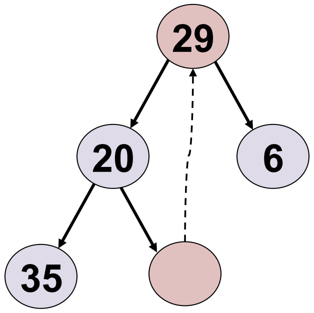
---

# Heapsort (3): Remove



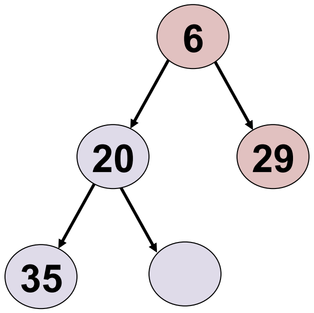
Copyright (c) 2018 Roger B. Dannenberg

# Heapsort (3): Remove



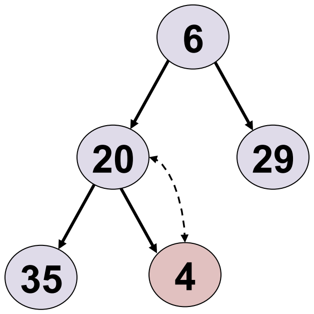
Copyright (c) 2018 Roger B. Dannenberg

# Heapsort (3): Remove



Copyright (c) 2018 Roger B. Dannenberg

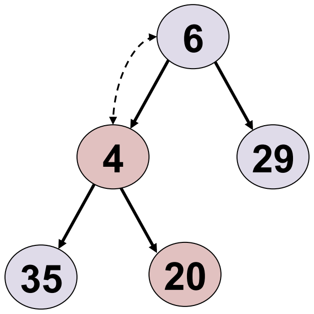
## Heapsort (3): Insert



Copyright (c) 2018 Roger B. Dannenberg

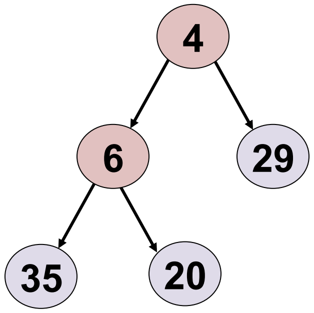
---

## Heapsort (3): Insert



Copyright (c) 2018 Roger B. Dannenberg

# Heapsort (3): Insert



Copyright (c) 2018 Roger B. Dannenberg

---

# From DES to Real-Time Systems

- DES computes and simulates precise timing
- We want precise timing in music systems too
- Example:
  - Thread1():

```
Thread1():
   loop:
      play bass_drum
      sleep(1.0)
Thread2():
   loop:
      play snare
      sleep(1.0)
```

```
main():
   new Thread1()
   sleep(0.5)
   new Thread2()
```

Copyright (c) 2018 Roger B. Dannenberg

## How Much Precision Do We Need?

- Suppose we compute things to nearest frame of a video game:
  - Frame rate = 60fps
  - Frame period = 1/60 = 17ms
  - Quantization error is perceptible
- What if system always responds within 1ms?
  - 100 beats per minute * 0.5ms error = 50ms error per minute (!)
- Recommendation: compute time with doubles

Copyright (c) 2018 Roger B. Dannenberg

## What can we hear?

- 0.1s jitter
- 20ms jitter
- 5ms jitter
- 1ms jitter

- 10ms is typical Just-Noticeable Difference (JND) for (almost) equally spaced taps
- 10ms jitter in a drum roll is clearly audible though, so 1ms is a much better goal

Copyright (c) 2018 Roger B. Dannenberg

# DES-like Real-Time Scheduling

- In music, usually very small timing errors (~1ms) are OK, but cumulative errors are bad:
  - Otherwise, two musical lines might drift apart
  - Otherwise, MIDI synchronized to audio or video might drift
- By the way, what are synchronization requirements for audio/video?
  - EBU R37: recommends audio at most 40ms early, at most 60ms late

# DES-like Real-Time Scheduling (2)

- Key ideas:
  - DES techniques to compute when things *should* happen – a specification
  - Use clock reference to make things happen as close to specification as possible
- Algorithm:

```
periodically do this:
   if time of first event in queue < get_time()
      remove event from queue
      now = event.time
      event.run()
```

## Where do we get “real time”?

- system clock – every computer has a built-in crystal clock
- audio sample count – to sync to audio
- video frame count or SMPTE – sync to video

Copyright (c) 2018 Roger B. Dannenberg

---

## How to implement “periodically”

- Simplest scheme (for command line – e.g. serpent64 – or embedded programs)
  - **while** true

    ```
    while true
        do periodic computation
        sleep(0.002) // sleep 2ms to reduce CPU load
    ```
- GUI toolkits/libraries usually have a timer callback function, e.g. in Swing:
  - `new Timer(2, periodicComputation).start();`
  - Where periodicComputation implements Action interface

Copyright (c) 2018 Roger B. Dannenberg

# Scheduler in Serpent

```
require "sched”
def sched_poll():
    nil
sched_init()
// put something on the scheduler
def demo(n):
    print "I'm alive!", n
    sched_cause(1, nil, 'demo', n + 1)
sched_select(rtsched)
sched_cause(1, nil, 'demo', 0)
sched_run()
```

Copyright (c) 2018 Roger B. Dannenberg

# Scheduler in wxSerpent

```
require "sched"
sched_init()
def demo(n):
    print "I'm alive!", n
    sched_cause(1, nil, 'demo', n + 1)
sched_select(rtsched)
sched_cause(1, nil, 'demo', 0)
// do not call sched_run()
```

Copyright (c) 2018 Roger B. Dannenberg

# Review

- Now you know how to build an accurate scheduler for music that:
  - Can handle hundreds or thousands of concurrent streams of events (notes, chords, beats, etc.)
  - Is efficient with computer time
  - Does not drift with respect to reference clock
  - Does not introduce critical sections, locks, multiple threads, or the overhead of traditional concurrent programs
- Let’s look at some more scheduling algorithms…

Copyright (c) 2018 Roger B. Dannenberg

---

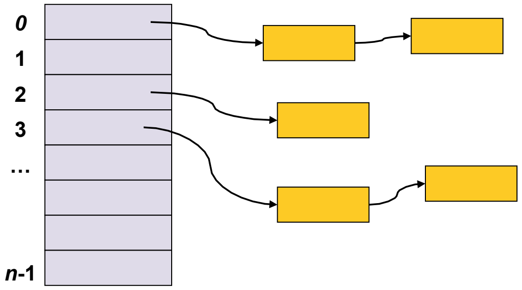
# Implementation 4: no polling

- Implementations 2 and 3 use priority queues
- Time of the next event is easily determined
- Why wake up periodically?
- Instead, *sleep* until the next event time.

- Observations:
  - +Saves time when there nothing to do
  - -Overhead of polling every ms or so is small
  - -Often, you need to poll for other things (audio processing, sensor input, …)

Copyright (c) 2018 Roger B. Dannenberg

# Implementation 5: timing wheel or calendar queue



**To Schedule: insert in table[int(ticks) mod n]**

**To Dispatch: every tick, search in table[tick mod n]**

**Assuming event times are random, and table size *n* is comparable to number of events, this can have O(1) scheduling and dispatching time.**

Copyright (c) 2018 Roger B. Dannenberg

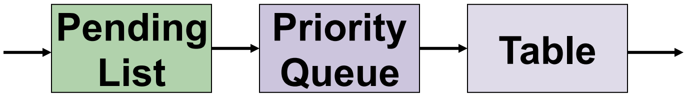
# Implementation 6

- What happens if events are not randomly distributed but separated by *n*?
  - E.g. table size = 1024 and each slot represents 1ms. Many events are scheduled at times 50+1024*n* ms. Slot 50 gets all events!
- Suppose we use table only for events in the near future?
- Note: reading makes this assumption already in Implementation 5.
- What do we do with events too far in future?

Copyright (c) 2018 Roger B. Dannenberg

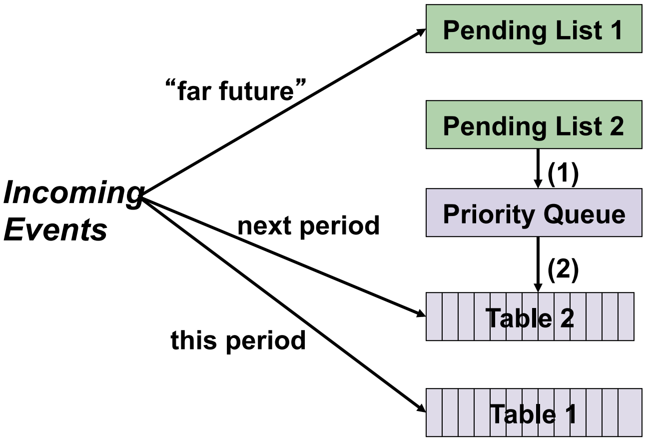
# Implementation 6 (2)

- The answer:
  - Keep far-future events in a heap-based priority queue and deal with them later.
  - But a heap-based priority queue has O(log n) insert time, so…
  - Schedule far-future events by inserting into a list; process the events later.



---

# Implementation 6 (3)



# Implementation 6 – Analysis

- Schedule time is O(1): based on time, just insert into Table 1, Table 2, or Pending List
- Dispatch time is O(1) per event and O(1) per clock tick: dispatch everything in corresponding slot in Table 1.
- Additional background processing time is O(log n) per far-future event.
- Background processing must be completed each period.

# Implementation 6 – Discussion

- How do you schedule background processing? What if it doesn’t finish in time?
- Yann Orlarey has a related scheme using an incremental radix sort instead of the heap – implemented and used in MidiShare system.
- I currently use Serpent’s (linear) resort() method to make priority cues. In C, I use Implementation 5 (timing wheel):
  - Simple to implement.
  - Works with floating point timestamps.
  - Worst-case performance not bad in practice.
  - Determining when to do background processing and coordinating that with foreground processing really needs OS support so it’s hard to do right

# Summary

- Events and Priority Queue
- Adaptable to almost any programming language
  - Function pointers
  - Subclass events
- Accurate timing
- Deterministic execution even in the face of some timing jitter
- Scheduling can be both fast and simple
- Implementation 5 is common, but tricky to implement due to rounding issues

Copyright (c) 2018 Roger B. Dannenberg