Real-Time Algorithms for Event Scheduling

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

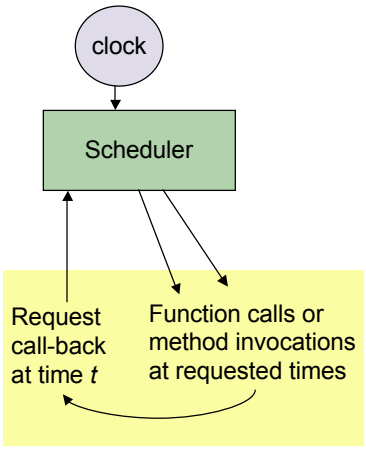
Introduction: Concurrency in Computer Music Applications

- We might want to play lots of notes in parallel
- Each MIDI note-on must have a pending note-off
- We might play multiple “tracks” within MIDI
- We might have an interactive program that listens to input and schedules new output
- BOTTOM LINE: We don’t have a single, sorted list of events to output.
- We might also want to control tempo, so we don’t really know the exact real-time for output.

Why not threads?

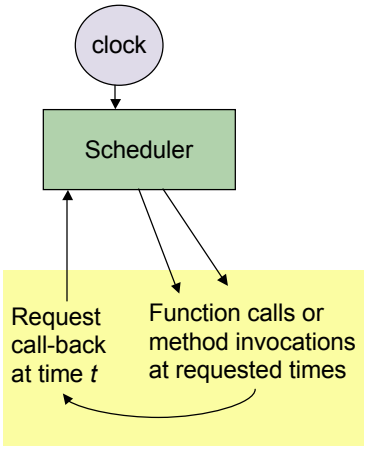
- Threads handle concurrency, and
- System already knows how to schedule threads
- BUT…
  - When threads share data, you must be very careful (use critical sections, etc.)
  - Thread overhead: stack, initialization, context switch time, etc.
  - Later will learn more about working with threads.
  - This only passes the buck to the OS…scheduling has to be solved somewhere

Concurrency Without Threads




```
def echo(pitch, loudness)
    loudness -= 5
    if loudness > 0
        note(pitch, loudness)
        cause(DELAY, 'echo',
              pitch, loudness)
```

Concurrency Without Threads




```
def echo(pitch, loudness)
    loudness -= 5
    if loudness > 0
        note(pitch, loudness)
        cause(DELAY, 'echo',
              pitch, loudness)

def keydown(pitch)
    note(pitch, loudness)
    cause(DELAY, 'echo',
          pitch, 100)
```


Some Comments

- We assume all callbacks are instantaneous
- Timing is explicit – common in real-time programs including music
- Many "threads" can run in parallel
- Processing is interleaved, but
- All procedures run to completion (non-preemptive)
- Non-preemption is a feature: no locks needed for critical sections

Implemention of *cause()*

- Assume *schedule*(*id, t*) exists such that:
  - at time *t, event*(*id, t*) is called
- Implement *cause*(*t, func, p1, p2, p3, p4*):

```
let e = new Event()
e.func = func
e.p1 = p1, e.p2 = p2, e.p3 = p3, e.p4 = p4
let id = &e
schedule(id, t)
```

- Implement *event*(*id, t*):

```
let e = (Event *) id
(e.func)(e.p1, e.p2, e.p3, e.p4)
free e
```

- From now on, we can just worry about *schedule*()

Implementation 1: linked list

- To *schedule*, insert (*id, t*) at the head of a list
- Periodically, do this:
  - **for each** *r* **in** list
    if *r.time* < *gettime*()
    remove *r* from list
    event(*r.id*)
- "Periodically":
  - see Windows MME timer
  - loop in a thread (maybe sleep 1ms or so)
- Problem:
  - Searches entire list very often

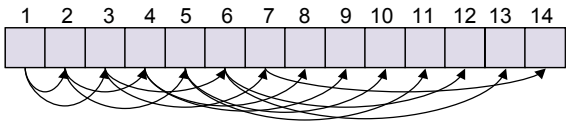
Implementation 2: priority queue

- Like before, but linked list is sorted:

- Periodic task removes and dispatches events until the timestamp at the head of the list is greater than the current time.
- Problem:
  - Scheduling (insertion) is linear in size of list

Implementation 3: heapsort

- Trick: embed complete binary tree in array.



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

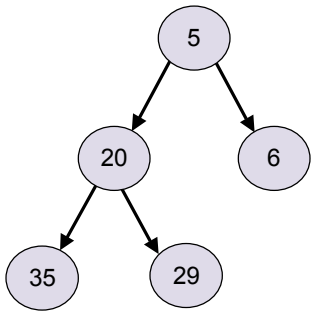
  No parent is greater than its children

  It follows that the root is the minimal element

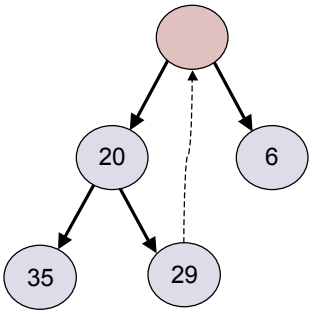
Heapsort (2)

- 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.

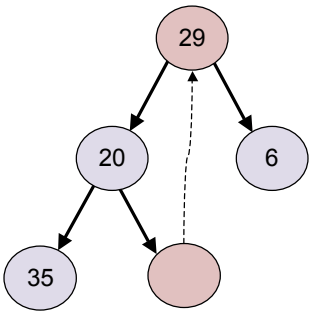
Heapsort (3)



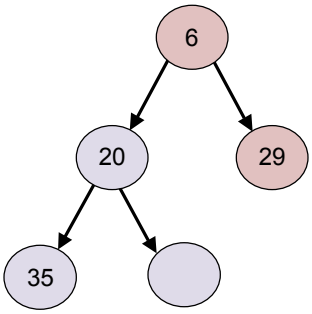
Heapsort (3)



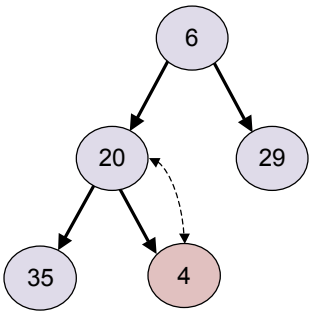
Heapsort (3)



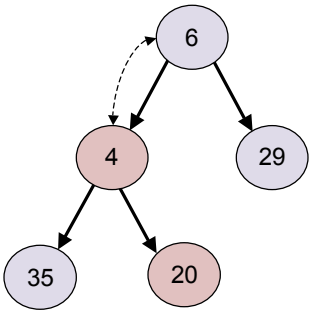
Heapsort (3)



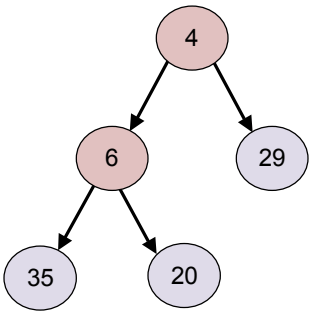
Heapsort (3)



Heapsort (3)



Heapsort (3)

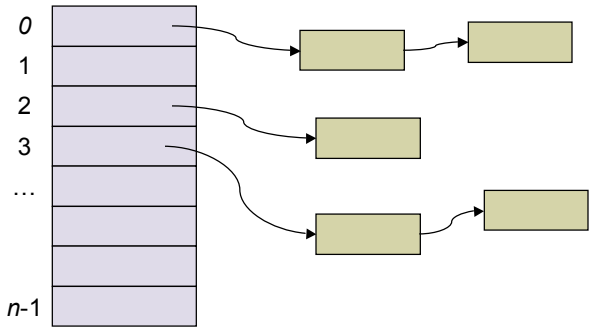


Implementation 4: no polling

- 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

Implementation 5



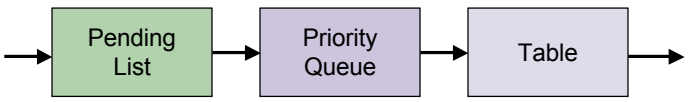
Assuming event times are random, and table size *n* is comparable to number of events, this can have O(1) scheduling and dispatching time.

Implementation 6

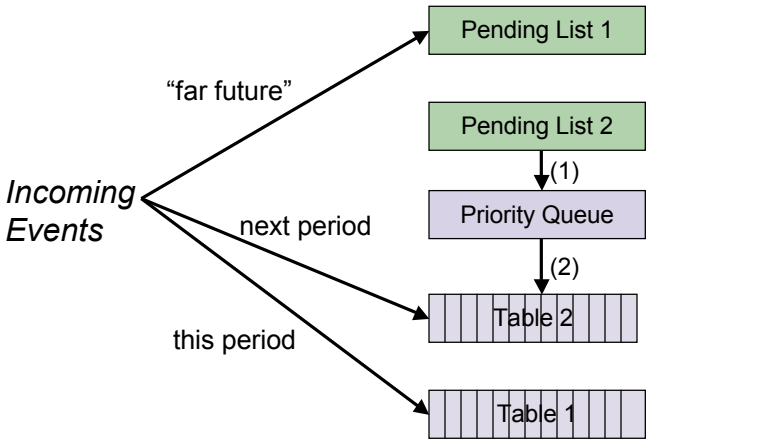
- What happens if events are not randomly distributed but separated by *n*?
  - E.g. table size = 1024 and each slot represents 1ms. Many events are scheduled at times 50+1024*n* ms. Slot 50 gets all events!
- Suppose we use table only for events in the near future?
- Note: reading makes this assumption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

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 Implementation 5:
  - Simple to implement.
  - Works with floating point timestamps.
  - Worst-case performance does not seem to happen in practice.

Project 2

- Implement an efficient time-based scheduler/dispatcher based on Implementation 5.
- See course web for details.
- Hint:
  - create an *event* type or class with:
    - Timestamp
    - Action (e.g. function name or function pointer)
    - Parameters (in C++, use one event type and a fixed set of parameters. If you're really ambitious you can subclass events so every action can have it's own class with its own custom set of parameters.)
    - Link to next event in linked list priority queue
  - Use PortTime callback to poll for and dispatch events from table (In Serpent, just loop reading the time, i.e. busy-wait until the time advances to the time of the next table slot.)
- Use scheduler/dispatcher in your music player.

Conclusions

- You should now understand…
  - How real-time applications can be structured as sequences of short computations scheduled at specific times. (note-on, note-off, respond to user input, etc.)
  - How to implement time-based schedulers efficiently
  - That polling has its advantages in real-time systems.

What's Next?

- Work on your project.
- Next week:
  - We assumed that events have negligible run times. How can we avoid accumulating timing errors due to real-world run-time and latency?
  - We assumed events take place at specific times. What if I want to schedule according to beats instead of seconds and the tempo is changing?