15-323/15-623 Spring 2019
Homework 3
Due Feb 21

1. Event and Scheduling Patterns

Complete the following code (for wxserpent64) according to the specifications. Assume that each example uses this code template (just below) that initializes rtsched, vtsched, midi_out, and runs the schedulers:

```
require "debug"
require "wxserpent"
require "sched"
require "midi-io"
require "prefs"
require "mididevice"

<INSERT CODE FRAGMENTS HERE>

sched_init()
prefs = Prefs("homework3")
midi_devices = Midi_devices(prefs)
// the next line only for 1A, 1B, 1C, 1F, not for 1D or 1E
sched_cause(1, nil, 'activity', 0)
```

1.A. Generate Sequence (Example with Solution)

Modify the code to print

```
activity: i = 0
activity: i = 1
…
```

Here is the code:

```
# modify and insert the following:
def activity(i)
    sched_cause(0.5, nil, 'activity', i + 1)
```

Here is a solution:

```
def activity(i)
    display "activity", i
    sched_cause(0.5, nil, 'activity', i + 1)
```

1.B. Using Display (easy)

Start by adding this code to your program:

```
middle_c = 60
E = 2.718282
```

Add one line of code to print the following line of text (do not use the `print` command). Hint: you can quickly test this by running serpent64 and typing commands to the console.

```
debugging: middle_c + 12 = 72, log(E) = 1
```

1.C. Sequence of 10 events

Modify the answer code from problem 1.A. to print

```
activity: i = 0
activity: i = 5
…
activity: i = 20
```

A new line should be printed every 0.5 s, and note that exactly 5 lines are printed.

1.D. Stopping the Sequence

Modify the answer code from problem 1.A. (not 1.B.) to start and stop the activity when a button is pressed. Assume the following code is included to create buttons:

```
start = Button(0, "Start", 5, 5, 75, 20)
start.method = 'startit'
stop = Button(0, "Stop", 5, 30, 75, 20)
stop.method = 'stopit'
```

Also assume the last line "`sched_cause(1, nil, 'activity', 0)`" from 1.A. is *not* included.

Your program should be "robust" in the sense that Start should immediately restart the `activity` sequence (from 0) even if the sequence has not stopped yet (e.g. you press Stop and then immediately press Start, there should be no race condition), and even if Stop was not pressed at all. If you press Start when the sequence is running, the sequence should restart immediately from 0.

Use the "stopping a thread" pattern (see class notes).

1.E. Note-off and Transpose

Using the code from 1.A., inserting the following code plays a note when you push the Play button. Assume the last line "`sched_cause(1, nil, 'activity', 0)`" from 1.A. is *not* included.

The pitch is set by the pitch slider. There is a bug: If you change the pitch slider while the note is playing, the note-off message will have the wrong pitch and the note will not be turned off.

Fix the code. Hint: You can save the pitch that you used to start the note and use that as the pitch parameter for the note-off command. There are a few ways to do this.

```
button = Button(0, "Play", 5, 5, 100, 20)
button.method = 'play'
pitch = Slider(0, 20, 100, 60, 5, 30, 200, 20)

def play(rest ignore):
    var p = int(pitch.value())
    sched_cause(0, midi_out, 'note_on', 0, p, 100)
    sched_select(rtsched)
    sched_cause(1, nil, 'note_off')

def note_off()
    var p = int(pitch.value())
    midi_out.note_on(0, p, 0)
```

1.F. Throttling an Event Sequence to Limit CPU Load

This example models a repeated activity that takes a lot of computation, for example, rendering a frame of animation can take a long time if there are hundreds or thousands of things to draw. Our precise timing approach to scheduling behaves very predictably when the CPU is not fast enough to keep up with real time: it simply allows the logical time to fall behind real time. All events following a schedule will slow down.

In the code (below), rather than actually using a lot of CPU cycles, we call `time_sleep` for a random time from 0 to 0.2s (expected sleep time is 0.1s), but we schedule `activity` every 0.05s (20 frames per second). Therefore, `activity` will fall behind schedule with very high probability.

Your job here is to change the program to run `activity` as often as possible, but without getting further and further behind schedule. Specifically, *always reschedule* `activity` *to run at the current* real *time.* For example, if the real time 20 but logical time is 10 (behind schedule), the next event should be scheduled at 20. Thus, the program is no longer behind (by much).

Hint: `sched_cause` normally takes a *relative* time as the first parameter, i.e. how far in the future should the next event be scheduled? You can provide an *absolute* time using the function `absolute`. E.g. `sched_cause(absolute(60.0), stdout, 'write', "Time is 60s.")` will print "Time is 60s." at the *absolute* time 60.0.

Finally, here's the code to modify:

```
    def activity(i)
        time_sleep(random() * 0.2)
        // show how far behind schedule we are now:
        display "behind by", time_get() – rtsched.time
        sched_cause(0.05, nil, 'activity', i)
```

Discussion: In a "real" implementation, you would do "real" work instead of calling `time_sleep()`. Before you modified the code, if you could not keep up with the schedule, you would use 100% of the CPU and compute as fast as possible. After fixing the code, that's still the case: You always schedule the next activity for the current time, so there's always work to do, and you use 100% of the CPU to compute as fast as possible. The advantage of the "fix" is that since the scheduler keeps up with real time, anything *else* running under the scheduler (e.g. MIDI generation) will also keep up with real time and will not slow down when the frame rate drops. This style of using available compute power to maximize the frame rate is common in video games and animation.

Unfortunately, the long frame computation could still add considerable timing jitter to MIDI output, so even if MIDI generation does not slow down, it could be erratic because all the MIDI events are going to run between frames while the scheduler is "catching up." To avoid jitter, you should use forward synchronous scheduling (e.g. set PortMidi latency to the maximum frame computation time and send timestamped data to PortMidi).

Formal Grammars

A Grammar describes productions that produce strings of terminals (denoted by lower case letters here). Each string is called a sentence, and the set of all possible sentences is called a language (for a particular grammar).

1. Consider this formal Grammar:

   S → ASBC
   S → ABC
   A → a
   B → b
   C → c

   The language generated by this grammar is infinite, but there is at most one sentence in the language of any given length. Write all the sentences in the grammar of length less than 10.

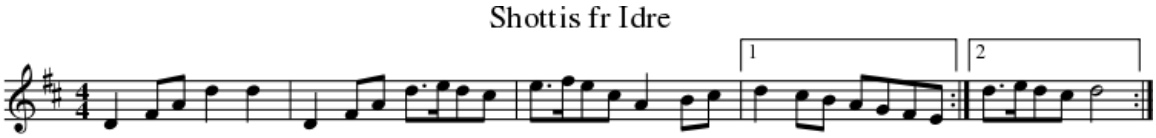
2. Consider this formal Grammar:

   S → ASC
   S → B
   A → a
   B → Bb
   B → b
   C → c

   Describe the language (set of strings) generated by this grammar. You can describe the language in formal terms, e.g. $x^i y^i$, or in English. Please be precise ("the sentences have a's, b's, and c's" is actually true, but it is not a good answer).

2. Music Theory and Reading

Here is a tune from http://richardrobinson.tunebook.org.uk/Tune/1968



1. How many sharps are in the key signature?
2. What is the time signature?
3. How many beats are in a measure?
4. What is the most frequently used duration?
5. What is the longest duration used?
6. What is the shortest duration used?
7. Write the first 5 pitches including octaves (hint: start with D4).
8. If the tempo is 120 BPM, what is the total duration of a performance of the music shown here in seconds?
9. Write the interval sizes in half-steps between each pair of adjacent notes in the 2nd and 3rd measures. Use negative numbers for descending intervals.

4 Algorithmic Composition

Using "Shottis fr Idre" from the previous question:

1. Use the pitch *classes* of this piece to train a first-order Markov chain. *Include the transition from the final E of the first ending to the D at the beginning of the piece. Also count a transition from the final D of the piece back to the beginning D.* Represent the transitions as a list of next states with (non-zero only) counts. Three of the lines are:

   ```
   D -> D:5, E:3, F#:4, C#:4
   E -> D:4, F#:2, C#:2
   F# -> E:3, A:4
   ```
   the last line says "F# is followed by E with weight/count 3 or A with weight/count 4"

2. Use your transition table to create a pitch sequence of length 5 starting on F#. Take the *most likely transition* at each step. If there is a tie, take the highest pitch class (from C to B).

3. Use your transition table to create a pitch sequence of length 5 starting on D. Choose next states randomly according to weights in your transition table.

4. One interesting algorithmic composition technique for tonal music is to generate random pitches (only some of which will be in a particular scale), then modify the out-of-scale pitches to be in the key. Assume your target is the scale for C-major, with pitch classes 0, 2, 4, 5, 7, 9, 11, and you generate a random pitch class sequence as follows:
   ```
   0 3 5 1 9 6 2 8 11 4 7 10
   ```
   Modify these pitch class numbers by subtracting one (mod 12) as necessary so that all pitch classes will be members of the C-Major scale. Notice that if a note is out of the (diatonic) scale, subtracting one will *always* put it in the scale, since diatonic scale intervals are at most of size 2 (2 half steps in music terminology). You can never be more than one half step away from a scale tone.

5. Modify the same set of pitches to fall within the D-Major scale (which has 2 sharps).

5 Elowsson and Friberg Reading

1. Figure 2 suggests that (pick the best answer):
   a) Interval size and pitches are independent.
   b) Interval size and pitch intervals are independent.
   c) Interval direction is more likely to be upward as durations get longer.
   d) Interval size tends to be longer as durations get longer.
   e) Interval size and duration are independent.

2. Figure 3 shows that (pick the best answer):
   a) Most tunes span a range of 7 or 8 scale steps.
   b) Most pitches occur in the middle of the range.
   c) Intervals of 7 scale steps are more common than intervals of 4 scale steps.
   d) Most German folksongs have a length of 7 or 8 notes.

3. According to Figure 4, out of 100 popular songs, how many would expect to be slower than 80 BPM?
   a) Less than 5
   b) About 10
   c) About half
   d) More than half