INSTITUTE OF SONOLOGY

Royal Conservatoire The Hague

Clocks

Programming and Music
http://www.bjarni-gunnarsson.net

Clocks

Music happens over time and a key element of control is to know when things happen. In SuperCollider, this is done by scheduling things using **clocks**.

A **clock** in SuperCollider has two major functions. It knows what time it is, and it knows when things are supposed to happen. When the time comes that things are scheduled to run the clock executes them.

There are **3 clocks** in SuperCollider:

- *SystemClock*
- *TempoClock (same as above but counts in tempo)*
- *AppClock (musically unreliable, used with GUI's)*

Clocks

Musical sequencing will usually use **TempoClock**, because you can change its tempo and it is also aware of meter changes.

There is only one **SystemClock**, there can be many **TempoClocks** all running at different speeds

AppClock, which also runs in seconds but has a lower system priority (so it is better for graphic updates and other activities that are not time critical).

Streams

Routine and **Task** are subclasses of Stream.

A **stream** uses a **lazy evaluation** where an expression is only evaluated if the result is required.

Using a **stream** means obtaining one value at a time with a lazy evaluation using the .**next** message.

A stream sequence can be finite but also infinite.

Scheduling

Routine is a function which can pause during execution (return in the middle) and later continue.

The message **yield** or **wait** (interchangeable) is used to **wait** some number of seconds (using *SystemClock*) or number of beats (using *TempoClock*).

By waiting for a specific amount of time event **scheduling** can take place.

Routines

A **Routine** runs a **Function** and allows it to be suspended in the middle and be resumed again where it left off. This functionality is supported by the Routine's superclass Thread.

A **Routine** is started the first time -next is called, which will run the Function from the beginning. It is suspended when it "yields".

A **Routine** can be stopped a before its Function returns using -stop.

Routine inherits from **Stream**, and thus shares its ability to be combined using math operations and "filtered".

Tasks

Task is a pauseable process. It is implemented by wrapping a **PauseStream** around a **Routine**.

Most of its methods (start, stop, reset) are inherited from **PauseStream**.

Tasks are not 100% interchangeable with **Routines**. **Condition** does not work properly inside of a Task.

Stopping a task and restarting it quickly may yield surprising results (see example below), but this is necessary to prevent tasks from becoming unstable if they are started and/or stopped in rapid succession.

Exercises

```
(
NF(\iop, {|freq=78, mul=1.0, add=0.0|
    var noise = LFNoise1.ar(0.001).range(freq, freq + (freq * 0.1));
    var osc = SinOsc.ar([noise, noise * 1.04, noise * 1.02, noise * 1.08],0,0.2);
    var out = DFM1.ar(osc,freq*4,SinOsc.kr(0.01).range(0.92,1.05),1,0,0.005,0.7);
    HPF.ar(out, 40)
}).play;
)

(
NF(\dsc, {|freq = 1080|
    HPF.ar(
        BBandStop.ar(Saw.ar(LFNoise1.ar([19,12]).range(freq,freq*2), 0.2).excess(
        SinOsc.ar( [freq + 6, freq + 4, freq + 2, freq + 8])),
        LFNoise1.ar([12,14,10]).range(100,900),
        SinOsc.ar(20).range(9,11)
    ), 80)
    ;
}).play;
)
```

```
var <>pindex, <>cindex;

initialize {
    if(pindex.isNil, { pindex = 1000 });
    if(cindex.isNil, { cindex = 2000 });
}

clearProcessSlots {
    pindex = 1000;
    (this.pindex - 1000).do{|i| this[this.pindex+i] = nil; }
}

clearOrInit {|clear=true|
    if(clear == true, { this.clearProcessSlots() }, { this.initialize() });
}

transform {|process, index|
    if(index.isNil && pindex.isNil, {
        this.initialize();
    });

    pindex = pindex + 1;
    this[pindex] = \filter -> process;
}

control {|process, index|
    var i = index;

    if(i.isNil, {
        this.initialize();
        cindex = cindex + 1;
        i = cindex;
    });

    this[i] = \pset -> process;
}
```

```
(
NF(\depfm, {|freqMin=5, freqMax=20, mul=20, add=80, rate=0.5, modFreq=2100, index=0.3, amp=0.2|
    var trig, seq, freq;
    trig = Dust.kr(rate);
    seq = Diwhite(freqMin, freqMax, inf).midicps;
    freq = Demand.kr(trig, 0, seq);
    HPF.ar(PMOsc.ar(LFCub.kr([freq, freq/2, freq/3, freq/4], 0, mul, add),
    LFNoise1.ar(0.3).range(modFreq,modFreq*2), index) * amp, 50)
}).play;
)
```

Exercises

Exercise 1: Basic Routine Sequence

Create a Routine that plays 5 notes in sequence with 0.3 second waits between them.

Exercise 2: Loop with Counter

Create a Routine that plays 8 notes using a loop. Each note should be 2 semitones higher than the previous one. Use 0.2 second waits.

Exercise 3: Conditional Rhythm

Create a Routine that loops 16 times. On every 4th iteration, play a low note (40) and wait 0.4 seconds. On other iterations, play a high note (80) and wait 0.1 seconds.

Exercise 4: External Control

Create a Routine that loops infinitely, using global variables ~pitch and ~tempo to control the note and timing. Start it, then change the variables while it runs.

Exercise 5: SystemClock vs TempoClock

Create the same Routine twice: once using SystemClock and once using TempoClock at 120 BPM. The routine should wait 0.5 (seconds or beats) between notes.

Exercises

Exercise 6: Task Control

Create a Task that counts from 1 to 20 with 0.2 second waits. Start it, pause it after a few counts, then resume it. Observe that it continues from where it paused.

Exercise 7: Multiple Routines

Create 3 different Routines that each play notes at different intervals:

- Routine A: plays a note every 0.5 seconds
- Routine B: plays a note every 0.75 seconds
- Routine C: plays a note every 1.0 seconds

Start them all at the same time and listen to the polyrhythm.

Exercise 8: Probabilistic Stopping

Create a Routine that uses a while loop with .coin to continue. Each iteration should have a 70% chance to continue (0.7.coin). Post the iteration number each time.
Run it several times to see different lengths.

Exercise 9: Scheduled Start Times

Create two Routines and schedule them to start at different times using clock.sched().
The first should start after 2 beats, the second after 5 beats. Use a TempoClock.

Exercises

Exercise 10: Tempo Changes

Create a TempoClock and a Routine that plays notes on that clock. While it's running, change the tempo of the clock to make the routine speed up or slow down.