15-323/15-623 Spring 2019
Homework 5
Due April 18

This problem set is about concurrency. It is based on the lecture slides, but you may need supplemental material. An excellent paper (with more information than you need for this homework) is "An Introduction to Programming with Threads" (1989) at (`http://birrell.org/andrew/papers/035-Threads.pdf`). Another excellent tutorial is "What Every Dev Must Know About Multithreaded Apps" by Vance Morrison, MSDN Magazine, August 2005 (`https://blogs.msdn.microsoft.com/vancem/2016/02/27/encore-presentation-what-every-dev-must-know-about-multithreaded-apps/`).

1. Consider concurrent code running on two threads. The right thread launches 4 sixteenth notes and then 2 eighth notes, where the tempo is based on `per`, the beat period in seconds. The left thread changes `per` to 0.4 from its initial value of 1.0.

①`per = 0.4`	②`for i = 0 to 4:` *// play 4 sixteenth notes*
	`    sched_cause(i*per/4, nil, 'note', per/4)`
	③`for i = 0 to 2:` *// play 2 eighth notes*
	`    sched_cause(i*per/2, nil, 'note', per/2)`

Two possible execution orderings are:

①②③, resulting in sixteenth notes at offsets of:
0, 0.1, 0.2, 0.3
and eighth notes at offsets of:
0, 0.2

②③①, resulting in sixteenth notes at offsets of:
0, 0.25, 0.5, 0.75
and eighth notes at offsets of:
0, 0.5

These are considered "correct" executions because in both of these cases, sixteenths and eights are synchronized (e.g. the 1st and 3rd sixteenths occur at the same time as the 1st and 2nd eighths).

1.1. Write an execution order in which eighths and sixteenths are not synchronized. (Assume ①,②, and ③ are atomic operations, i.e. they occur in a strict order – in reality, ① could occur at any time during the execution of ② or ③).

1.2. Assume that Serpent offers locks as follows:

`x = lock_create()` – create a lock (object) and assign to variable x

`lock(x)` – acquire a lock, if lock is already held, block until it is free

`unlock(x)` – release a lock if it is held; only the holding thread can unlock

Write code for initialization, the left thread, and the right thread (see table above) using ①, ②,③, `lock_create()`, `lock()`, and `unlock()`. Your goal is to ensure that sixteenth (②), and eighth (③) notes are always "correct" and synchronized. Rather than copying code, you can simply write ①, ②, or ③, e.g. a (wrong) solution could be:

```
def initialization():
    x = nil

def left_thread():
    unlock(x)
    ②
    y = lock_create()

def right_thread():
    ①
```

2. Similar to Question 1, here is a function that decrements a variable if it is greater than zero:

```
def decrement():
    if ①x > 0:
       x ③= ②x – 1
```

Here, the circled numbers represent just the operations that touch shared memory: ① and ② are memory reads, and ③ is a memory write.

2.1. Assume that `decrement` is run in two threads that share access to `x`. List a possible execution order such that the final value of `x` is -1. The form of your solution should be a table such as the following (incorrect) solution:

Thread 1	Thread 2	Value of x after the operation in this row
		1
①		1
②		1
③		0
	①	0

2.2. Rewrite `decrement` using `lock` operations and add an `initialization` function so that whenever `decrement` returns, the value of `x` will be non-negative.

3. Consider this algorithmic composition system. It uses three single-note synthesizers and multiple "voices," each of which runs on a thread. Each voice selects one or two synthesizers, acquires locks for them, plays a note (or two, if `play_two` is true), then ends the notes and releases the locks. When a voice tries to lock a synth that is busy, the voice simply waits until the synth is released (unlocked) and proceeds.

```
// there are 3 resources that can play notes; to play a note, lock the resource, play,
// and unlock. Here are the locks:
synth_locks = [lock_create(), lock_create(), lock_create()]

// Multiple threads can call voice() to generate tones on the synths. If a synth is unavailable,
// the voice will just wait until the selected synth unblocks.
 1 def voice(i):
 2    var candidates = [0, 1, 2]
 3    var play_two = (random() > 0.5) // true with 50% probability
 4    // select a synth at random
 5    var synth_A = irandom(3) // returns 0, 1, or 2
 6    lock(synth_locks[synth_A]) // wait for availability of synth_A
 7    if play_two: // play a 2nd note
 8        candidates.remove(synth_A) // don't pick same voice twice
 9        var synth_B = irandom(2)    // select index of a remaining synth
10        lock(synth_locks[synth_B]) // wait for availability of synth_B
11        start_a_note_on_synth(synth_B)
12    start_a_note_on_synth(synth_A)
13    time_sleep(1.0) // play notes for awhile
14    // turn off the notes and release (unlock) the synths
15    stop_a_note_on_synth(synth_A)
16    unlock(synth_locks(synth_A))
17    if play_two:
18        stop_a_note_on_synth(synth_B)
19        unlock(synth_locks(synth_B)

// each thread simply calls voice() repeatedly to play notes
def thread(i):
    while true:
        voice(i)
```

3.1. Describe an execution sequence with 2 threads that leads to deadlock. You can refer to variable values, line numbers, and "thread 1" and "thread 2" to describe the execution sequence.

3.2. Describe how to fix this program so that deadlock does not occur.

4. Assume that you have a real-time program that implements a musical instrument with the following tasks. For simplicity, assume there is a single CPU and single core.

   - Audio synthesis (uses 50% of CPU, runs for 0.7ms every 1.4ms)
   - File I/O (seldom used, runs up to 0.2s to fetch audio data)
   - Graphical User Interface generating animated display (runs up to 5ms every 30ms)
   - Music Control, e.g. receive OSC commands and update synthesizer parameters. (polls every 2ms and runs up to 0.1ms whenever data arrives)

   4.1. Which task is the most time-critical?

   4.2. Using fixed priority scheduling, how would you order the tasks from lowest to highest priority to achieve the best musical results?

   4.3. Assuming the OS can delay up to 1ms before running the highest priority task, what is the longest period of time between when the Audio synthesis task is ready to run and the completion of the Audio synthesis task? (Note 1: this time tells us the minimum number of audio samples there must be in the audio output buffer. Note 2: if the OS delayed 1ms *every* time the Audio task was ready to run, it would not be able to keep up with real time, so we assume that these 1ms delays are rare, e.g. they might happen only when the kernel forks a process, when the kernel writes log data, etc.)

   4.4. What is the maximum delay experienced by the Music Control task? Consider that the task only polls every 2ms, it has some execution time, it may suffer delay (1ms) due to the OS not running it immediately, and it may be preempted due to priority scheduling. Show a timeline for the worst case that indicates all the sources of delay.

5. In the previous example, assume the Audio task runs at high priority and the Graphical User Interface (GUI) runs at low priority. Furthermore, assume that sometimes the GUI calls free() to free previously allocated memory. Similarly, the Audio task also calls free() to free previously allocated memory.

   A deep systems secret will now be revealed to you: When memory is allocated with `malloc()`, `malloc` will search for or even create a memory pool (a heap) for which there is no lock being held, but when `malloc()`'d memory is freed with `free()`, the memory *must* be returned to the same memory pool from which it was allocated, and this requires that `free()` obtain a lock to that memory pool.

   Therefore, let's assume the worst: the GUI and Audio threads both `malloc()` from the same pool (heap), and later call `free()` more-or-less simultaneously. Somewhere else on the system, ProTools is running, and for good performance, ProTools has elevated its file IO thread to medium priority. Let's assume that sometimes, this file IO thread runs compute-bound running audio compression algorithms for up to 0.1 seconds.[1]

   5.1. Show a sequence of execution that will lead to priority inversion and cause an audio underflow (failure of the Audio thread to run promptly and refill the output buffer).

   5.2. How can you avoid this situation? (This is a design/software architecture question. A sentence or two will suffice to give the general direction and key idea(s).)

[1] Of course, I don't really know what ProTools does, but the point is that there is certainly the possibility that a medium-priority task running for a long time. That's all that matters here.

6. Message Passing. Serpent supports programming with a preemptive thread in a very restricted fashion.[2] Read "Procs - Preemptable Thread Interface and Functions" (http://www.cs.cmu.edu/~music/serpent/doc/serpent.htm#procs). As you will read, you can:
   - create a "proc," e.g. `proc_create(10, "mp2.srp", 0)`
   - send a string message from one proc to another, e.g. `proc_send(proc_id, "a message is a string")`
   - receive a string message, e.g. `var msg = proc_receive(proc_id)`

   6.1. Write a Serpent program in `mp1.srp` that creates another proc (defined by the file `mp2.srp`), then loops reading strings from the command line and sending them to the other thread.

   6.2. Write `mp2.srp`, which defines `porttime_callback(n)`, the function that Serpent will call automatically after the thread is started. In the callback, check for a message using `proc_receive(thread_id)`, and if it is a non-empty string, convert it to an integer pitch and send a MIDI note-on message for that pitch. (In this simple example, we will not worry about sending note-off messages.)

   Note: you do not have to submit separate mp1.srp and mp2.srp files – just include them in your PDF. We'll check for the main ideas and to see you did this exercise, but we will not try to test your code by actually running it.

   The point of this problem is to illustrate that procs (preemptive threads) can communicate without shared memory and without many of the locking and synchronization difficulties of other programs. You can actually run and test your code if you like[3], but we will only check that your code gets the main ideas correct.

   6.3. A common complaint (e.g. in OS textbooks) is that polling is a bad practice. Indeed, this program *could* block, waiting for messages, and save a lot of cycles. And in fact, polling is often a sign of poor designs that have complicated time-dependent behavior. However, polling that avoids condition variables, locks, and other complexities *can* result in reliable designs and good performance.

   Either measure your program from 6.2, or write a simple loop that runs `time_sleep(x)` so that serpent wakes up every x seconds. What is the CPU load in percent for $x = 0.1$, $x = 0.01$, $x = 0.001$? (On OS X or Linux, try using `top -o cpu`, on Windows, try the Windows Task Manager.)

   Lesson: Why are we looking at this number? This is the overhead of polling. This is an estimate of the CPU load you could save by causing your program to block rather than poll when there is no work. Some other things to consider:
   - In favor of blocking: Polling adds latency – you can't respond to input until you wake up and poll.
   - In favor of polling: The overhead of polling goes down as the load goes up – if there's work to do, you're going to wake up and do it either way, so there's less advantage to blocking as the load increases.

[2] The proc feature of Serpent only exists when Serpent is compiled with the USE_PROC flag. The currently compiled releases do *not* include this proc feature, but you could implement something similar by running two instances of Serpent that communicate via O2.

[3] Either by building Serpent with USE_PROC, or you can run serpent64 from the version 249 downloads on Sourceforge (except in that older version, procs were called "threads," so the function names start with "`thread_`" and not "`proc_`".