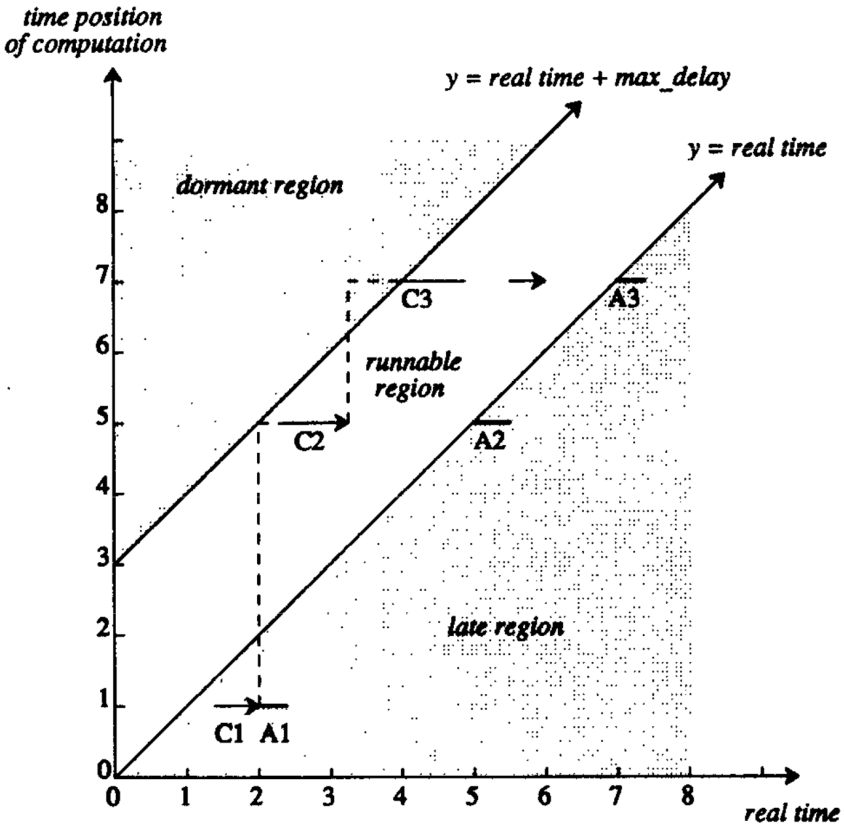
15-323/623 Spring 2019
Homework 2
Due Feb 14

Scheduler implementation:

1. Compare an ordered linked list, a heap, and a timing wheel implementation of a scheduler. Fill in a table like the following with characteristics of these approaches. Assume N means the number of currently scheduled events.

Algorithm	Expected Insertion Time per event, e.g. O(N), O(log N)	Expected Dispatch Time per event, e.g. O(N) etc.	Worst-case Insertion Time	Worst-case Dispatch Time	Clarify what would cause the worst case situation(s) and/or assumptions you are making
Ordered Linked List					
Heap Priority Queue					
Timing Wheel					



In this **Timing in FORMULA** graph (Figure 6 of Anderson and Kuivila from the readings; also week 3, slide 26), we see three scheduled and executed events. Show your understanding of this important graph by answering the following questions:

2. Assume that C1 takes 0.5 time units. What time would C1 have to begin in order to compute results on time?
3. Assuming C1 started early enough and A1 also takes 0.5 time units, what time would A1 produce output?
4. There is separation in time between C2 and A2. Why is there no time separation between C1 and A1?
5. C2 becomes runnable at time 2, but it starts later at about 2.5. Why?
6. C3 follows C2, but we do *not* immediately start computation C3 after C2. Why not?
7. Suppose that the CPU is fast enough so that all computations are meeting deadlines with time to spare, but the system is not very responsive because of pre-computation and event buffering. How would you change the parameters and how would the graph change to make the system more responsive?

More questions…

8. Suppose you had Serpent (or another language) with preemptive threads. To get more computation done, you propose to:
    - use a thread for each task (e.g. each drum in a drum machine could be on a different thread),

- use a single scheduler: threads will sleep by waiting on an synchronization event, but first the thread schedules a function call that signals the event. Thus, the thread blocks on the synchronization event until the correct logical time, after which the thread will run again,
- you use locks around the scheduler to avoid concurrent scheduling by multiple threads (which would undoubtedly have race conditions), but since scheduling is fast, you can assume that the scheduler is not a bottleneck.

What other problems would you have to deal with? There are at least 2 problems. Try to answer with one problem having to do with concurrency and one having to do with timing.

9. The "Global Drum Circle" aims to allow drummers across the globe to play together. But internet delays are high (over 100 ms for intercontinental connections) and there is considerable variability (jitter) in transmission times. Describe briefly what you would do to transmit drumming events to a remote location over the Internet. Assume that network messages are delivered reliably but with a latency that varies from 25 to 500 ms. Assume further that delays of up to 1000 ms are acceptable (i.e. the receiver can hear each drumming event exactly 1000 ms after the performer plays), but you need to reproduce drum timing to within 5ms to achieve musically acceptable results.

10. Here is a naïve Serpent program for playing a sequence of notes (don't worry about the details of `play_note()`):

```
for i = 0 to 10:
    play_note()
    time_sleep(0.5)
```

Assuming that your program will have a scheduler `rtsched` initialized and running, finish the program below by defining `play_sequence` to use the scheduler to achieve the same general effect (i.e. rewrite the naïve version to make it correct). Since this program is not concerned with the details of `play_note()`, you should not worry about timestamps or forward-synchronous scheduling:

```
sched_select(rtsched)  // prepare to use rtsched
play_sequence(0)
```