Timers Timeouts

Implementation and Tradeoffs

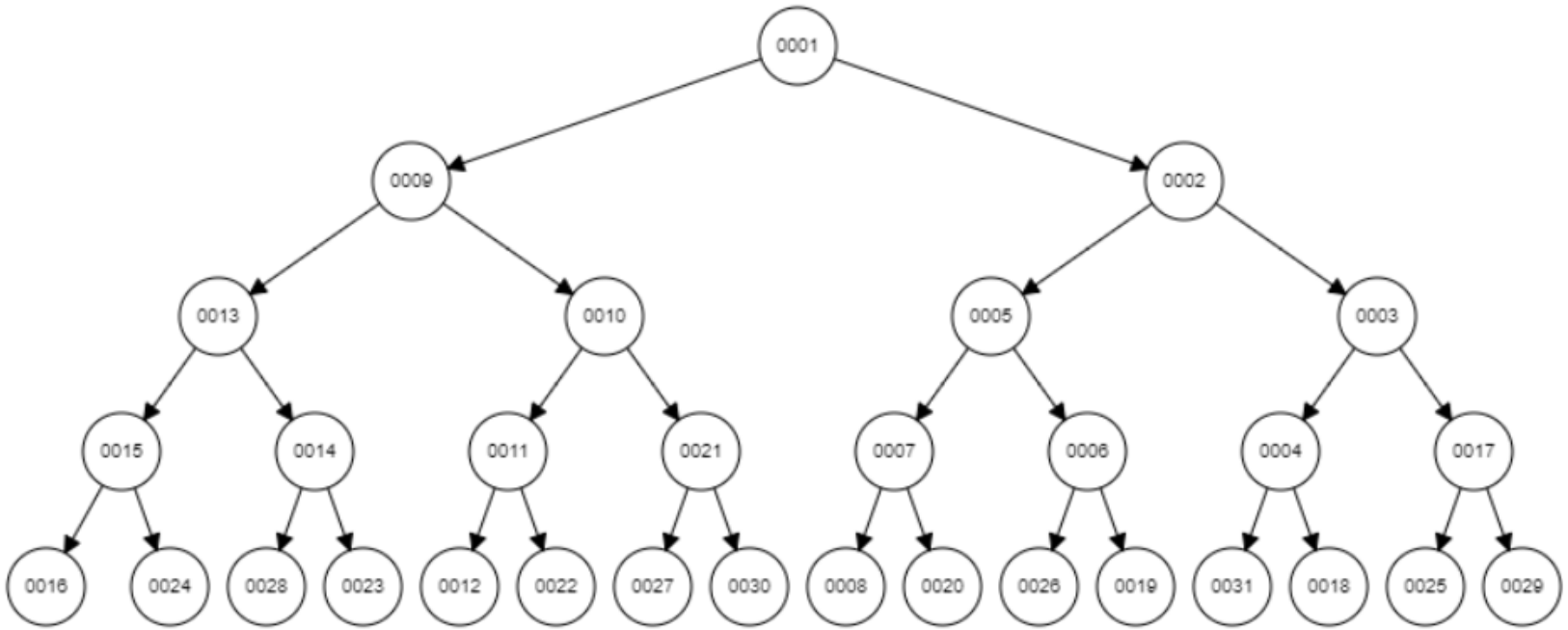
A bit of background…

Classic Timer implementation using Min Binary heap

isaiah perumalla

Min Binary heap

-INF	0001	0009	0002	0013	0010	0005	0003	0015	0014	0011	0021	0007	0006	0004	0017	0016	0024	0028	0023	0012	0022	0027	0030	0008	0020	0026	0019	0031	0018	0025	0029
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31



Binary heap based timers cost

Insert	Update	Remove	Tick/cycle
• Insert cost Log(n)	• Update cost log(n)	• Remove cancel cost log(n)	• Per tick/cycle check O(1)

Timer wheels an alternative approach

- Based of this paper from 1987
- sosp87-timing-wheels.pdf (columbia.edu)
- Will email link worth reading

Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility

George Varghese and Tony Lauck
Digital Equipment Corporation
Littleton, MA 01460

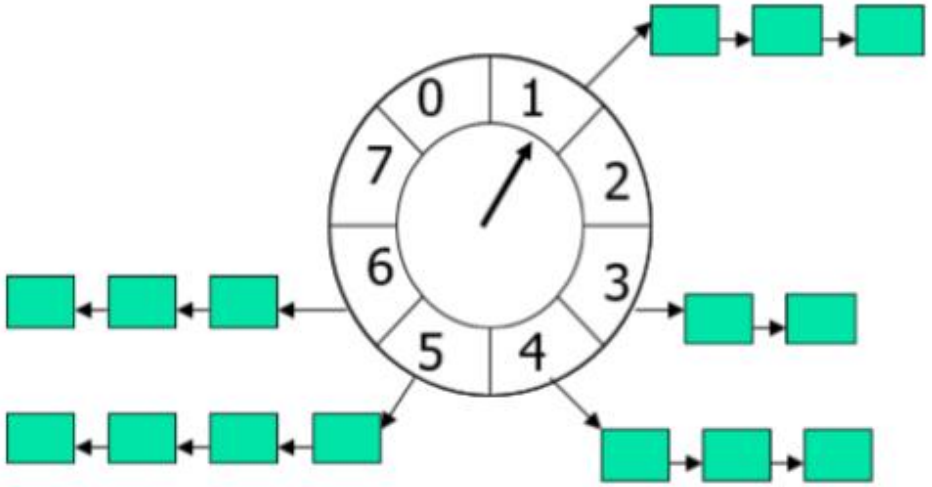
Abstract

Conventional algorithms to implement an Operating System timer module take $O(n)$ time to start or maintain a timer, where n is the number of outstanding timers: this is expensive for large n. This paper begins by exploring the relationship between timer algo-

be detected by periodic checking (e.g. memory corruption) and such timers always expire. Other failures can be only be inferred by the lack of some positive action (e.g. message acknowledgment) within a specified period. If failures are infrequent these timers rarely expire.

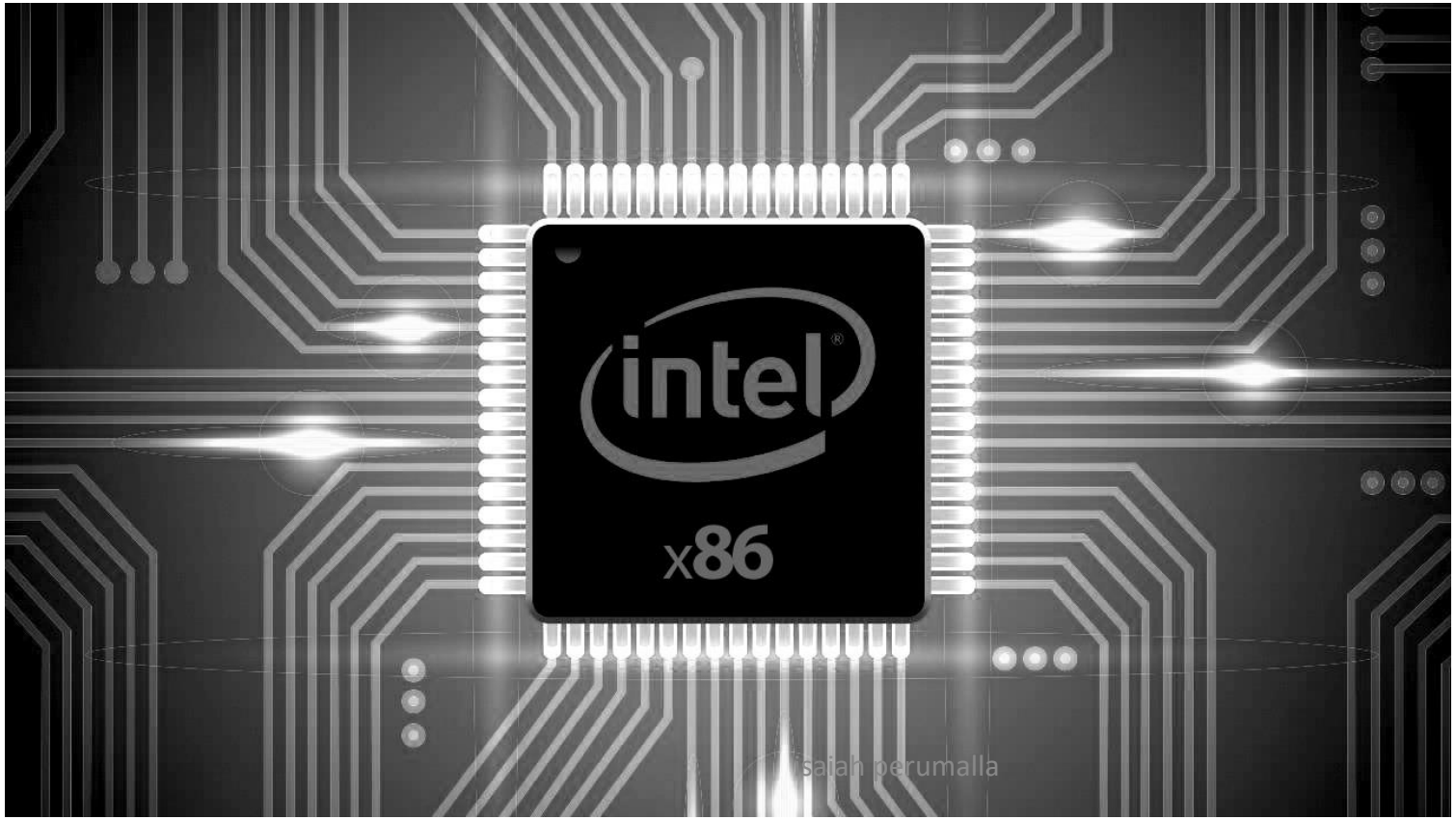
Timer wheel basic idea

- Each tick cursor move one location forward
- Similar to seconds hand on a clock
- Each slot has list of timer expiring at that time (*think bucket sort*)
- Can vary granularity of tick depending on context

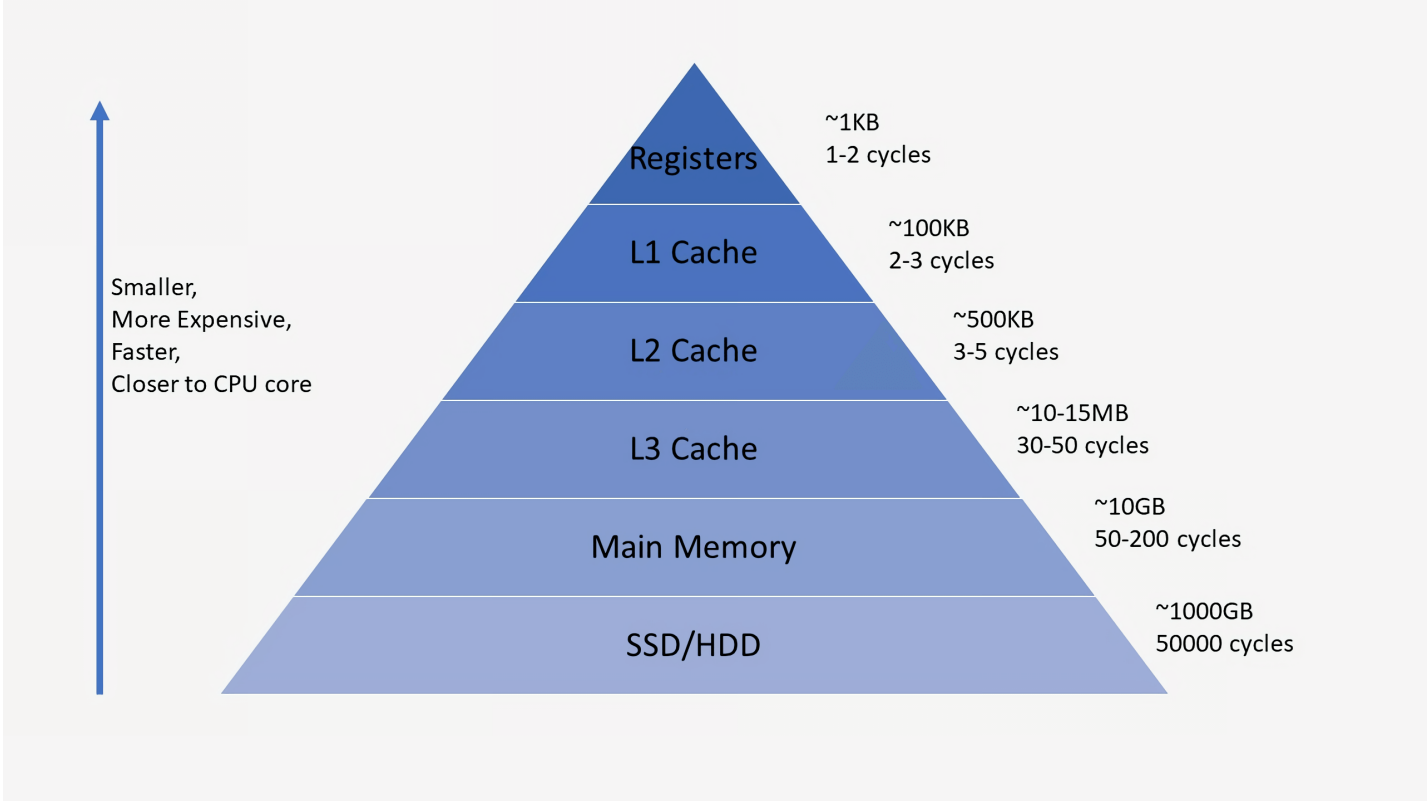


Implementation matters

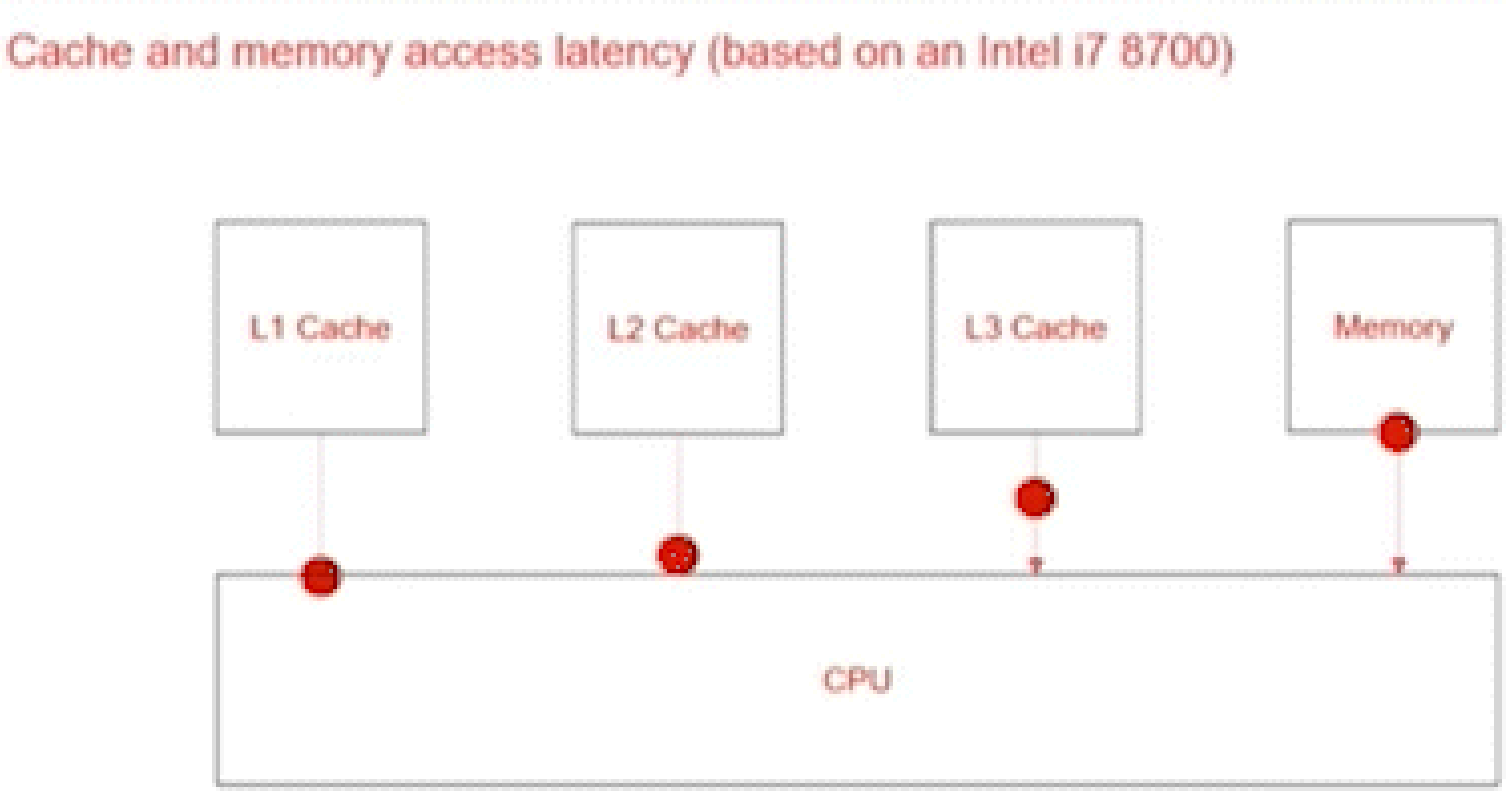
(JVM is not the platform)



Very quick Primer on memory

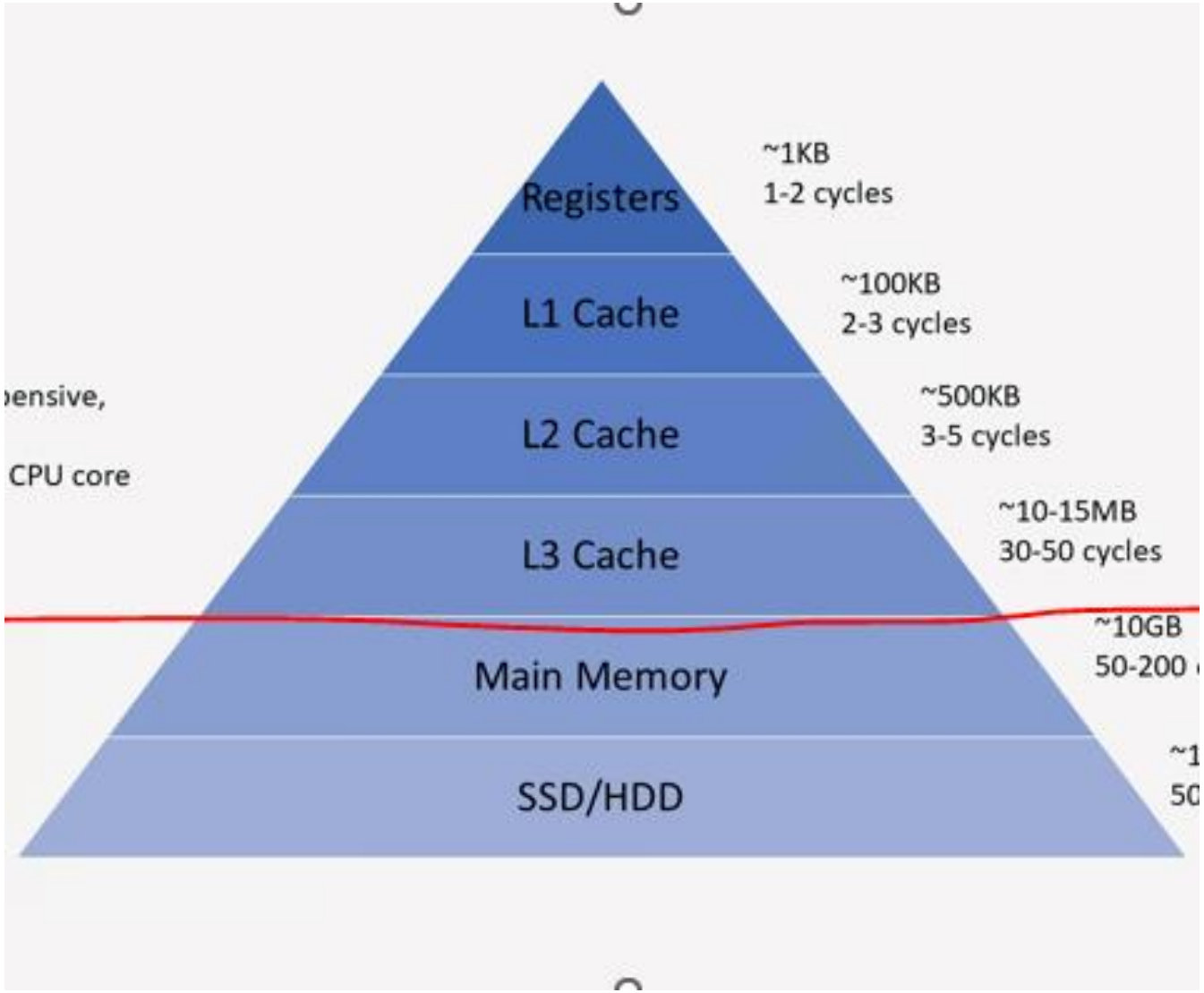


Intuition for memory access latency



Keep data in cache

- Main memory latency up to 500cycles



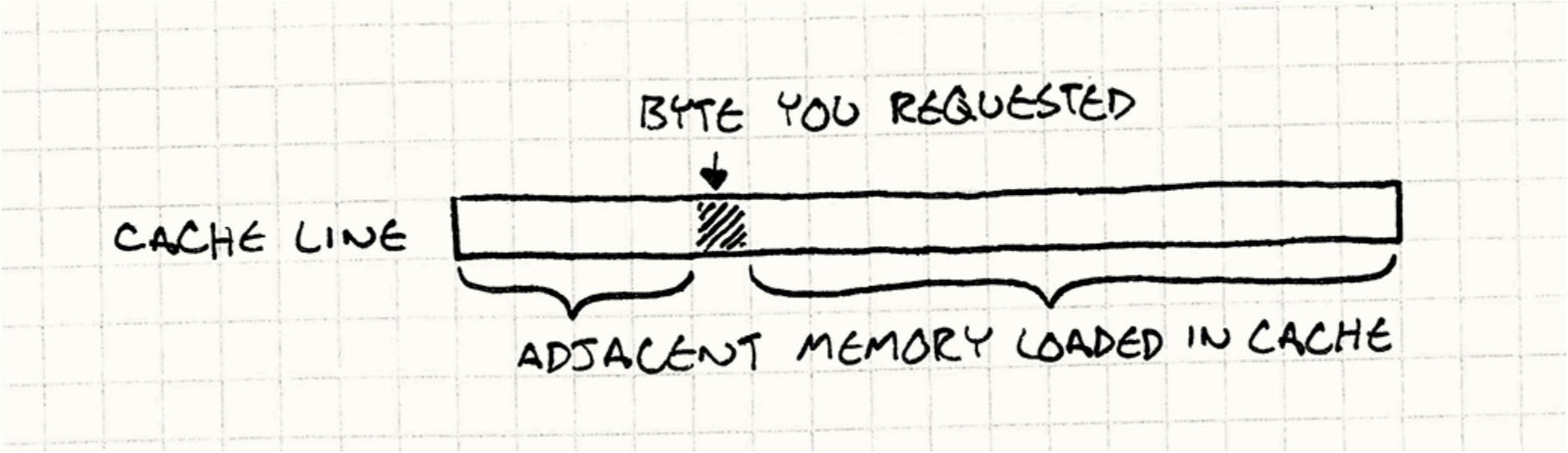
Intuition for memory access latency

```
While(doWork) {
        sleep500Cycles();  //get rid / minimize this for biggest win
        a = array[i].load();
    }
```

Cache-line

- Smallest unit of memory transfer 64 bytes usually on x86
- Accessing a single byte, you get whole cache line

Don’t waste the cache-line



Java / JVM objects memory in-efficient

- Every object has at-least 16 byte header
- Poor information density a lot of waste
- Array of objects with long could be 32 bytes each ! Or more

Main take away

- Main memory is slow
- Cache Line smallest unit of memory 64 bytes (usually)
- Fast caches are very small
- Data layout ***is*** Performance
- Compilers cannot help with memory layout
- Java objects avoid for perf critical data structures

Different problems require different solutions.

Timeouts are different

- Timeout are normally used to notify exceptional or error conditions

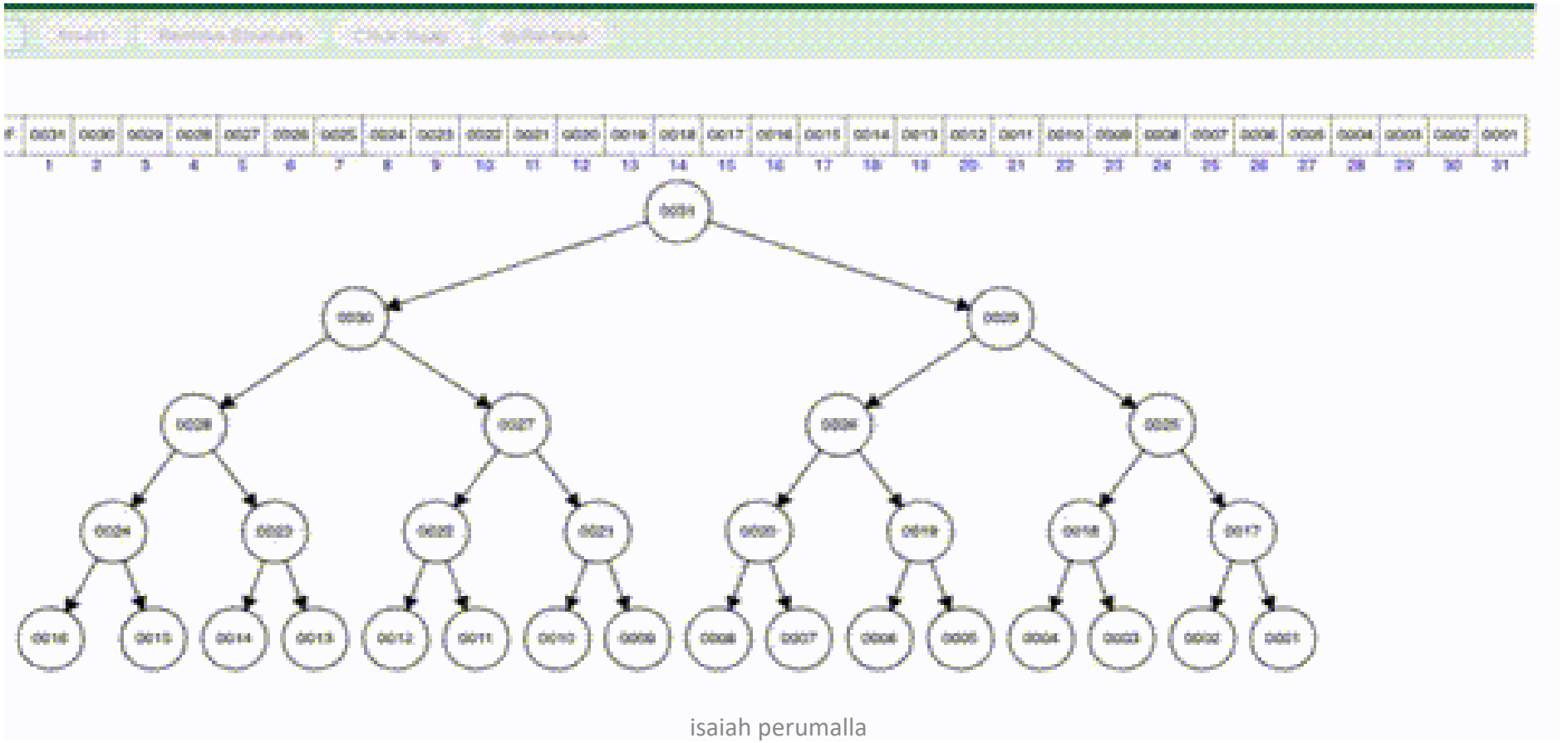
(example expected event failed to arrive on time)

- In most common case timeout are scheduled and cancelled and rarely materialize (***lets optimize for this !***)
- Precision requirement for timeouts don’t need to precise

(does it matter if a market data symbol time out after 5000ms or 5064ms ?)

- We usually know what the max timeout range is ahead of time

Memory access of Heap for time-outs



Solve for most **common** case *not* most generic case

- JVM is ***not*** the platform
- Everything runs on hardware

Designing a Timeout tracker (efficient cache usage)

- Main use case track time-outs
- We know max time-out value ahead of time

Agrona Timer Wheel

- Generic , supports large ranges of timers
- not optimized for timeouts
- Stores entire 64bit timestamp for each timer (not needed if we know max time-out range ahead of time)
- For our use-case we can do better

Information Density

Timestamps millis

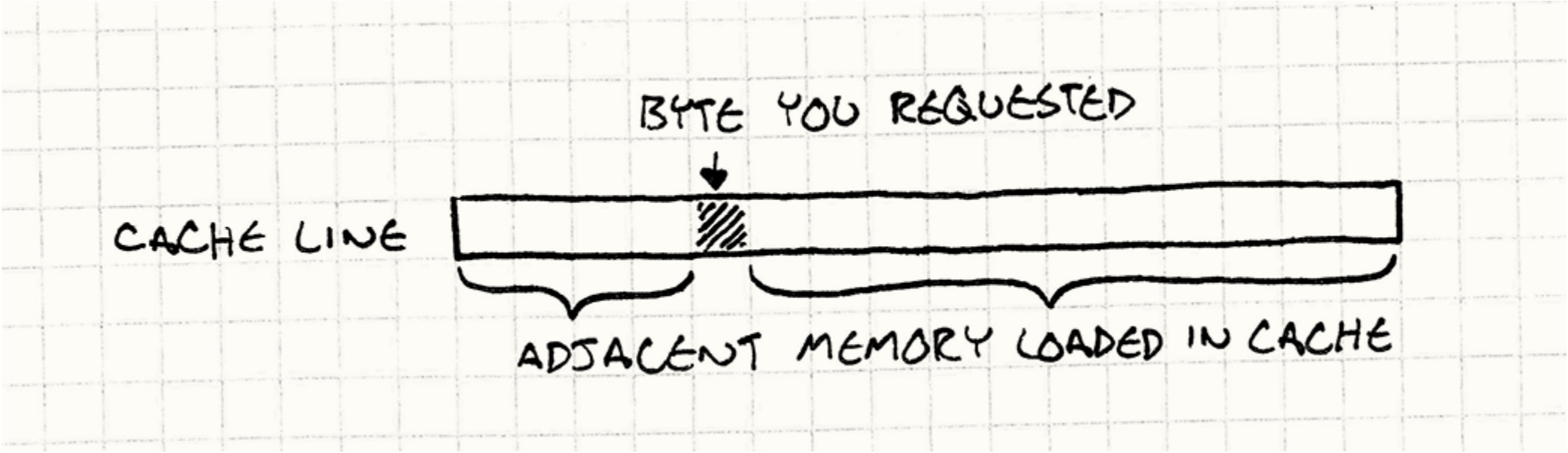
- 1663797265490 -> 0b11000001101100010000010111000010001010010
- 1663797265494 -> 0b11000001101100010000010111000010001010110
- 1663797265900 -> 0b11000001101100010000010111000010111101100
- 1663797266000 -> 0b11000001101100010000010111000011001010000

Half a cache line used up with mostly repeated information

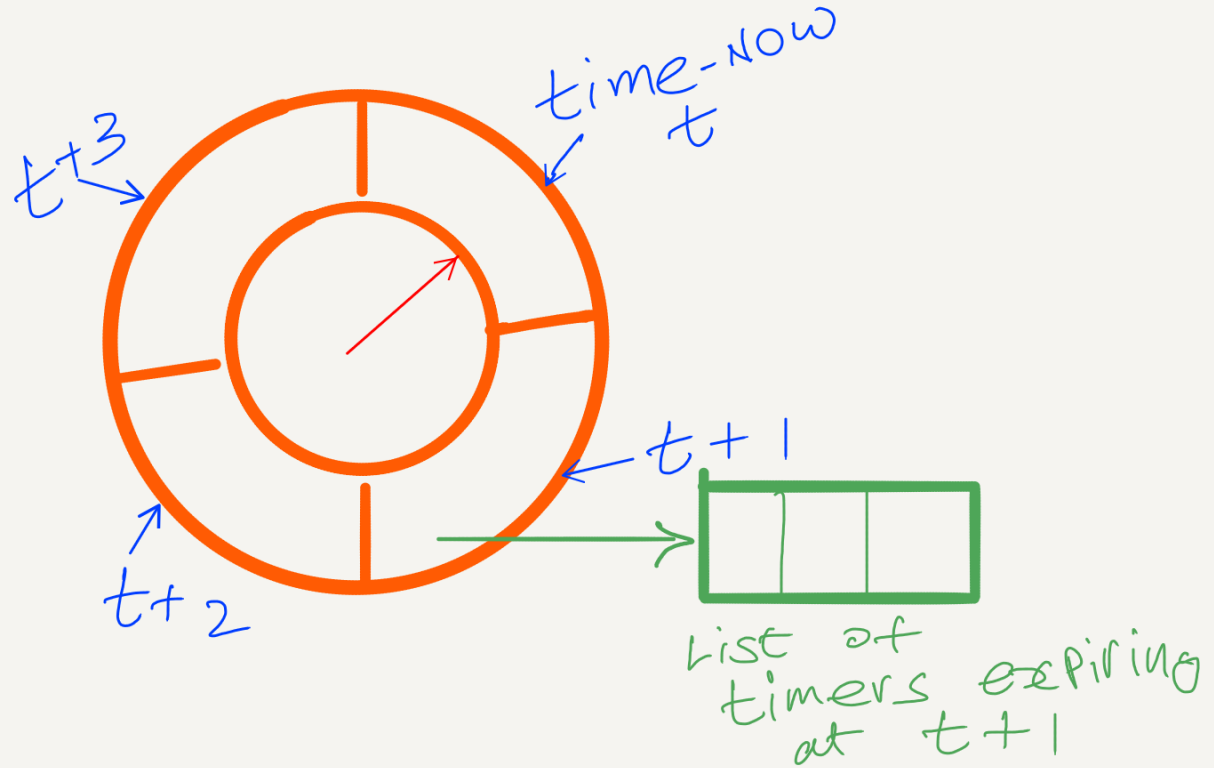
11000001101100010000010111000010001010010 11000001101100010000010111000010001010110
11000001101100010000010111000010111101100 11000001101100010000010111000011001010000

- Wasted cache-line space
- All the redundant higher order bits are using valuable space

Don’t waste the cache-line



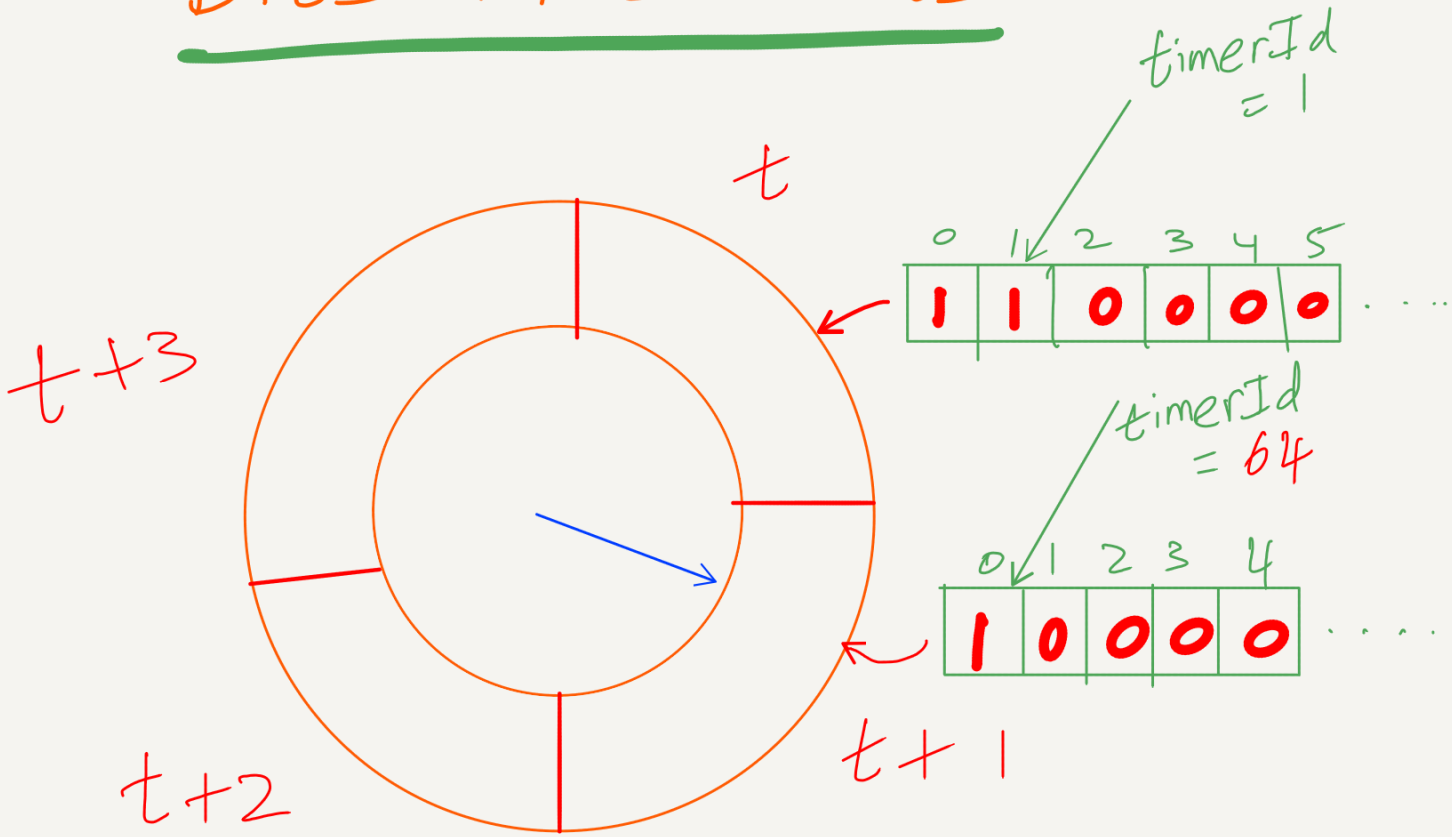
Simple time wheel



Constrains and goals

- Focus on Timeouts and do it really well
- Fast efficient schedule/cancel (handle 1000’s in a few millseconds)
- Low memory , fit in cache-lines

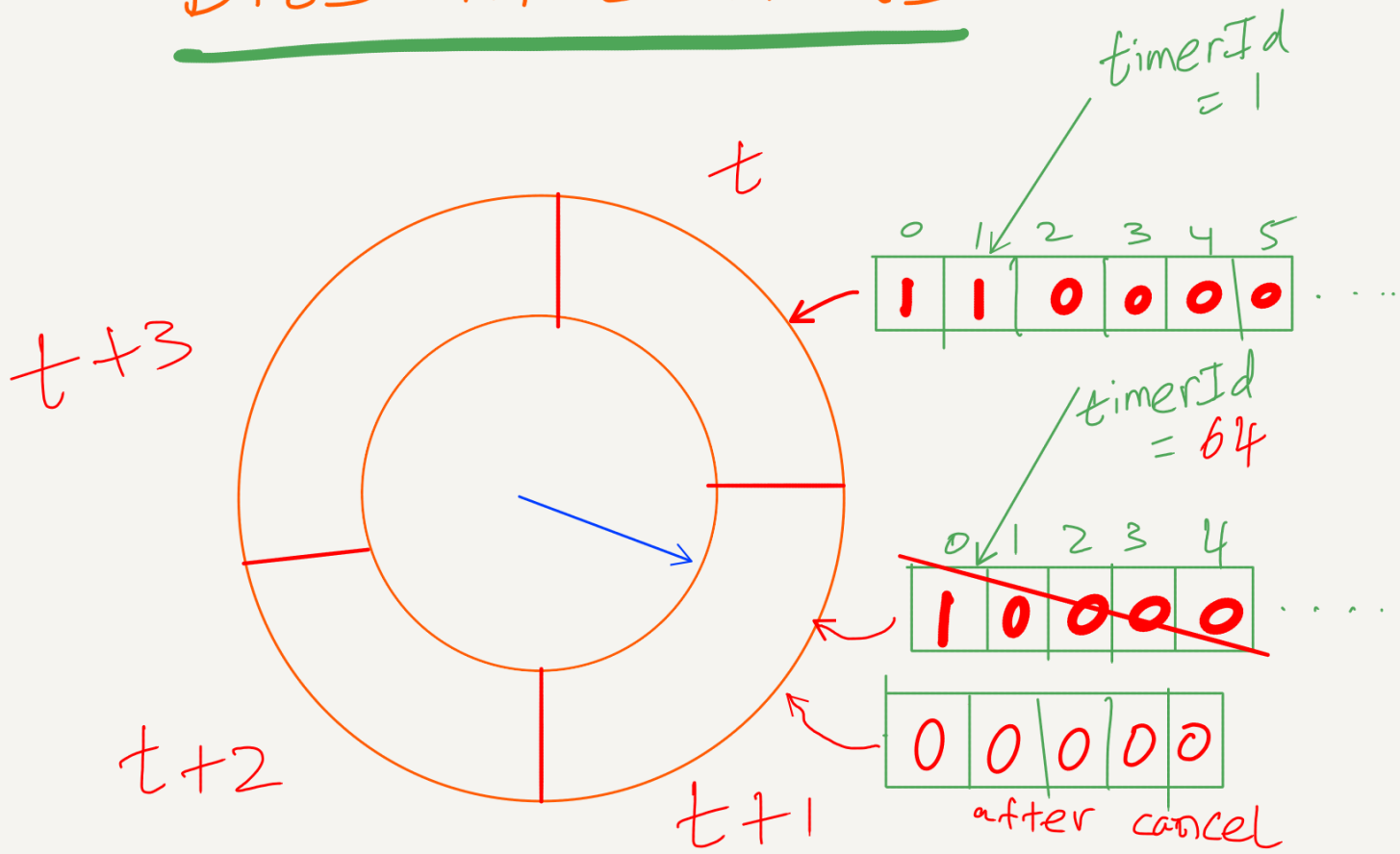
Bits in Buckets



timeout.schedule("EURUSD", t+1) -> 64

"EURUSD" <-> 64

Bits in Buckets

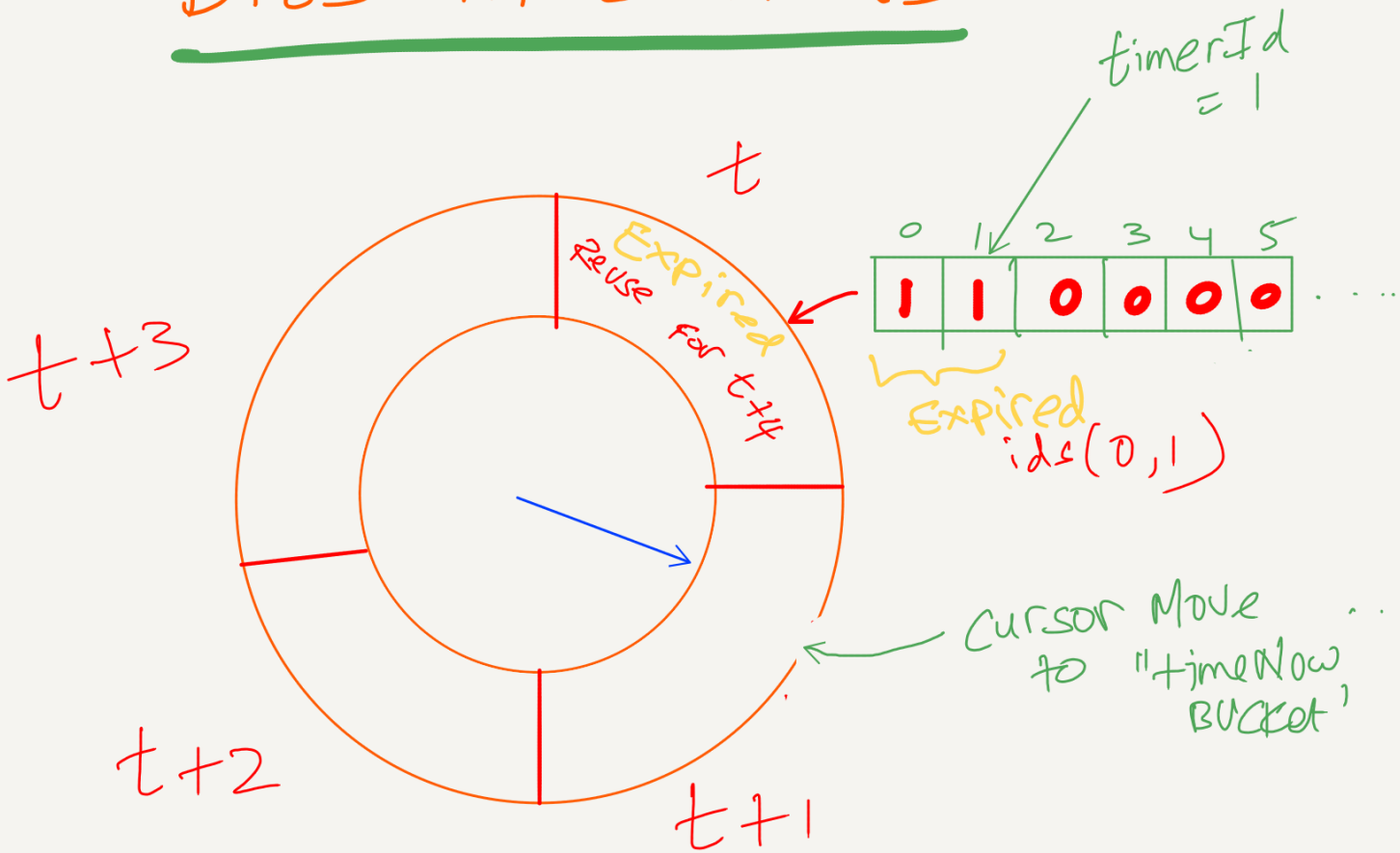


timeout.cancel("EURUSD")

"EURUSD" → 64

clearBit 64

Bits in Buckets



timerMgr.pollTimeouts(timeNow, callBack)

Trade offs

- Trading off precision for performance and efficiency
- For timeout precision doesn’t matter so much
- Some timer can expire up to (tick granularity later)

Example if granularity of bucket is 256 millseconds, some timeout can expire up to 256 millis later than scheduled

This doesn’t matter for timeout in ***most*** cases atleast not for the HA Symbol allspark case

- Questions ?