LRU and Clock Replacement Algorithms

Xiaodong Zhang

The Ohio State University

Numbers Everyone Should Know (Jeff Dean, Google)

- **L1 cache reference: 0.5 ns**
- **Branch mis-predict: 5 ns**
- **L2 cache reference: 7 ns**
- **Mutex lock/unlock: 25 ns**
- **Main memory reference 100 ns**
- **Compress 1K Bytes with Zippy 3000 ns**
- **Send 2K Bytes over 1 GBPS network 20000 ns**
- **Read 1 MB sequentially from memory 250000 ns**
- **Round trip within data center 500000 ns**
- **Disk seek 1000000 ns**
- **Read 1MB sequentially from disk 2000000 ns**
- **Send one packet from CA to Europe 15000000 ns**
- **300 millions time difference between fastest and slowest**

Replacement Algorithms in Data Storage Management

- **A replacement algorithm decides**
  - Which data entry to be evicted when the data storage is full.
  - *Objective*: keep to-be-reused data, replace ones not to-be-reused
  - Making a critical decision: a miss means an increasingly long delay
- **Widely used in all memory-capable digital systems**
  - Small buffers: cell phone, Web browsers, e-mail boxes …
  - Large buffers: virtual memory, I/O buffer, databases …
- **A simple concept, but hard to optimize**
  - More than 40 years tireless algorithmic and system efforts
  - LRU-like algorithms/implementations have serious limitations.

Additional Software/Hardware Support

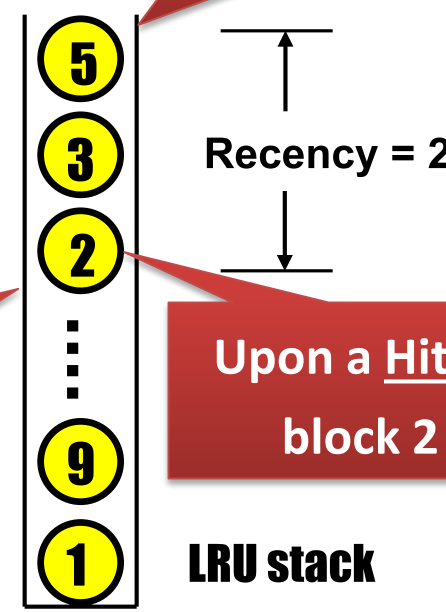
- **Access status of each block is dynamically recorded**
  - This is done by getting both global and local information
  - Local information of each block: a hardware reference bit for each block (fast but low accuracy)
  - Global information of access ranking of all blocks: a software stack is used (relative slow but more accurate)

Least Recent Used (LRU) Replacement

- LRU is most commonly used replacement for data management.
- Blocks are ordered by an LRU order (from bottom to top)
- Blocks enter from the top (MRU) and leave from bottom (LRU)

The stack is long, the bottom is the only exit.

Move block 2 to the top of stack



- **Recency** – the distance from a block to the top of the LRU stack
- **Upon a hit**

Recency of Block 2 is its distance to the top of stack

Upon a Hit to block 2

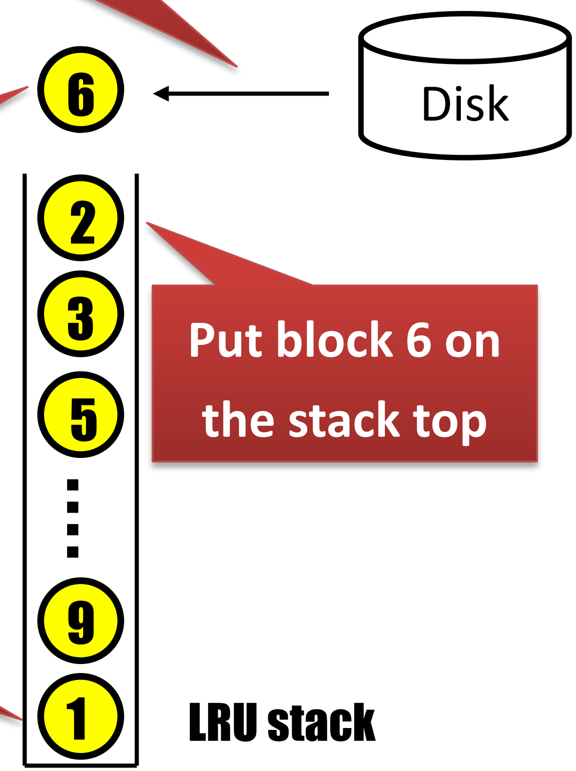
Least Recent Used (LRU) Replacement

- LRU is most commonly used replacement for data management.
- Blocks are ordered by an LRU order (from bottom to top)
- Blocks enter from the top, and leave fro

The stack is long, the bottom is the only exit.

Load block 6 from disk

Upon a Miss to block 6



Put block 6 on the stack top

- **Recency** – the distance from a block to the top of the LRU stack
- **Upon a hit** – move block to the top
- **Upon a m**

Replacement – the block 1 at the stack bottom is evicted

LRU is a Classical Problem in Theory and Systems

- **First LRU paper**
  - L. Belady, IBM System Journal, 1966
- **Analysis of LRU algorithms**
  - Aho, Denning & Ulman, JACM, 1971
  - **Rivest**, CACM, 1976
  - Sleator & **Tarjan**, CACM, 1985
  - **Knuth**, J. Algorithm, 1985
  - **Karp**, et. al, J. Algorithms, 1991
- **Many papers in systems and databases**
  - ASPLOS, ISCA, SIGMETRICS, SIGMOD, VLDB, USENIX…

The Problem of LRU: Inability to Deal with Certain Access Patterns

- **File Scanning**
  - One-time accessed data evict to-be-reused data (cache pollution)
  - A common data access pattern (50% data in NCAR accessed once)
  - LRU stack holds them until they reach to the bottom.

- **Loop-like accesses**
  - A loop size $k+1$ will miss k times for a LRU stack of k

- **Different accessing frequencies**
  - For B-tree, the index structure is accessed much more frequently than records. Infrequently used records can evict frequently used indices, degrading the data access performance

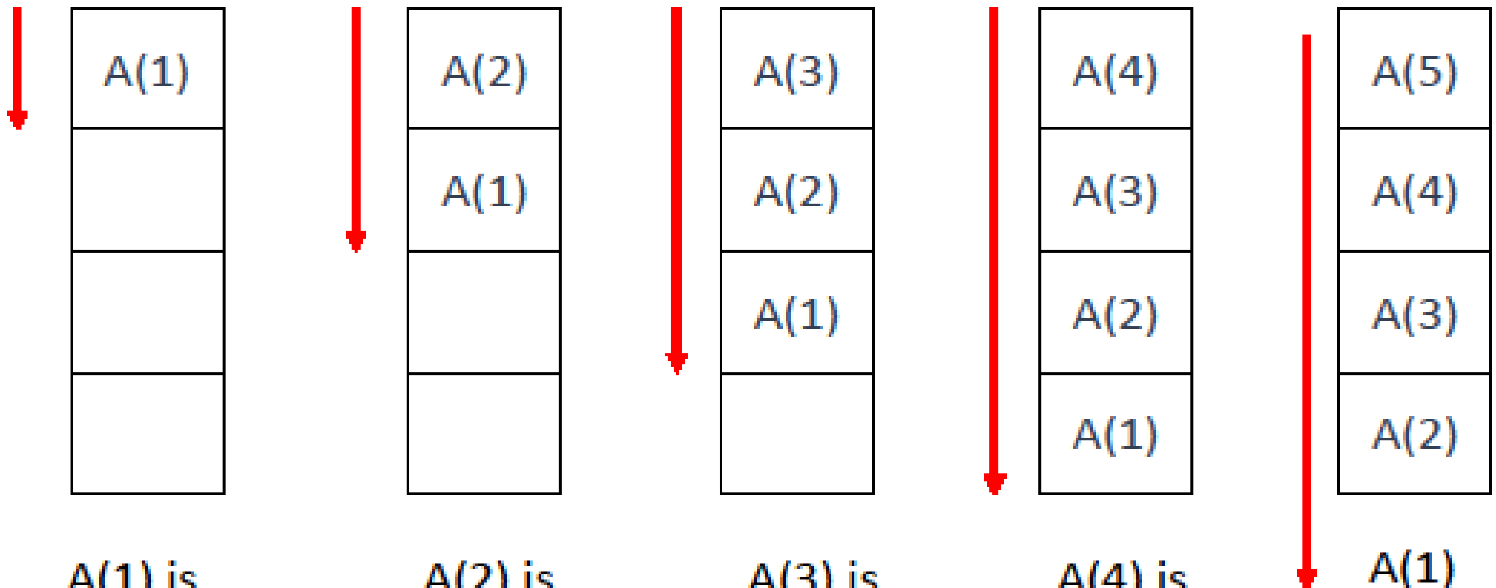
A loop example managed by LRU

```
for (int i=1; i<=n; i++){
    for (int j=1; j<=5; j++){
        A[j] = A[j] + 1;
    }
}
```

All the accesses to array A will be missed in a buffer of 4 elements

The buffer contains 4 elements (LRU stack = 4)

The first outer loop, i = 1



A(1) is brought to the buffer

A(2) is brought to the buffer

A(3) is brought to the buffer

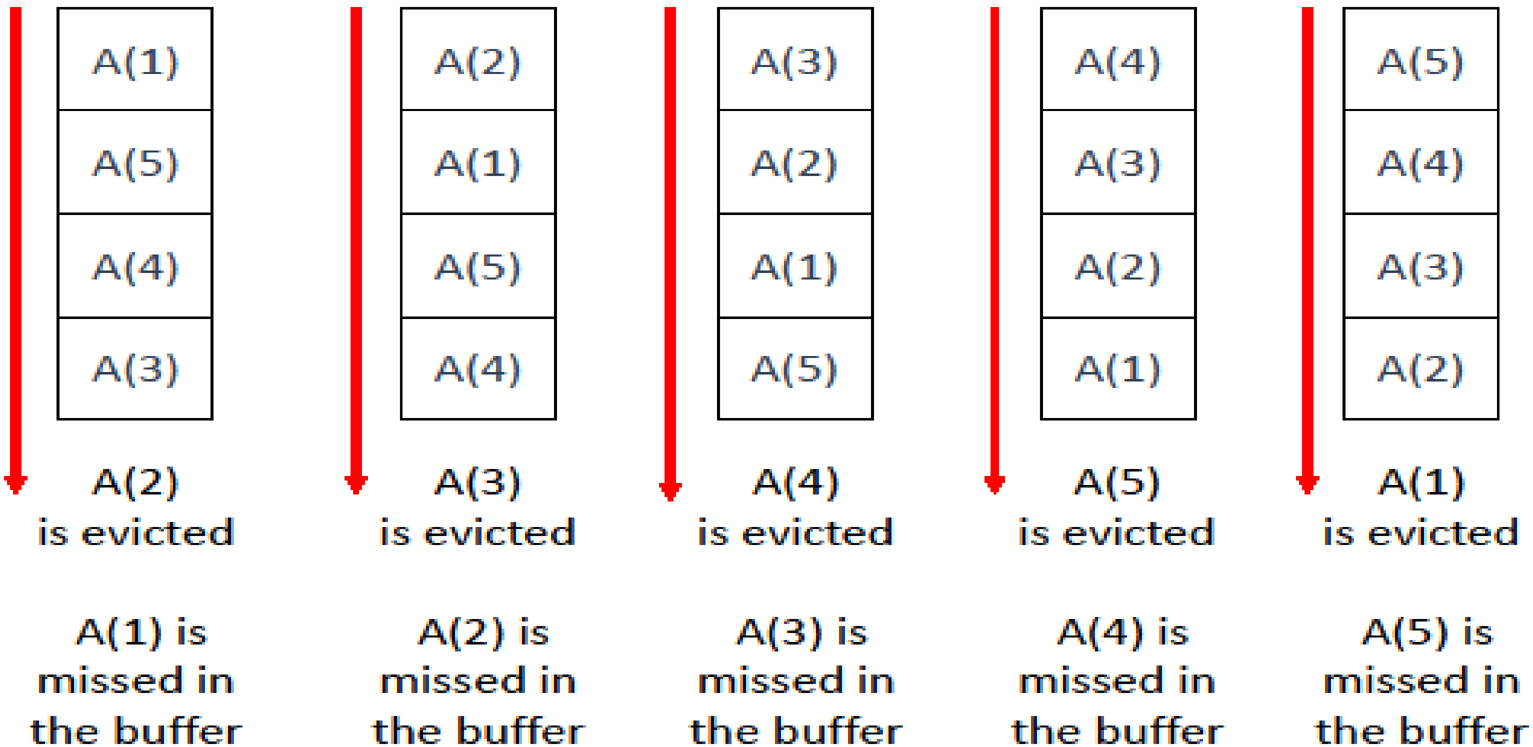
A(4) is brought to the buffer

A(1) is evicted

A(5) is brought to the buffer

The buffer contains 4 elements (LRU stack = 4)

The second outer loop, i = 2



Continuing the outer loop for another n-2 times, all access to array A will be missed for the same reason

Why Flawed LRU is so Powerful in Practice

- **What is the major flaw?**
  - The assumption of “recently used will be reused” is not always right
  - This prediction is based on a simple metrics of “recency”
  - Some are cached too long; some are evicted too early.

- **Why is it so widely used?**
  - Works well for data accesses following LRU assumption
  - A simple data structure to implement

Challenges of Addressing the LRU Problem

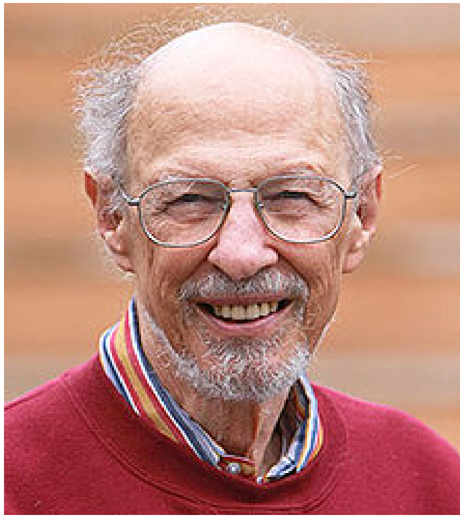
- **Two types of Efforts have been made**
  - Detect specific access patterns: handle it case by case
  - Learn insights into accesses with complex algorithms
  - Most published papers could not be turned into reality
- **Two Critical Goals**
  - Fundamentally address the LRU problem
  - Retain LRU merits: low overhead and its assumption
- **The goals are achieved by a set of three papers**
  - The LIRS algorithm (SIGMETRICS'02)
  - Clock-pro: a system implementation (USENIX'05)
  - BP-Wrapper: lock-contention free assurance (ICDE'09)

Two Technical Issues to Turn it into Reality

- **High overhead in implementations**
  - For each data access, a set of operations defined in replacement algorithms (e.g., LRU or LIRS) are performed
  - This is not affordable to any systems, e.g., OS, buffer caches …
  - An approximation with reduced operations is required in practice
- **High lock contention cost**
  - For concurrent accesses, the stack(s) need to be locked for each operation
  - Lock contention limits the scalability of the system
- **Clock-pro and BP-Wrapper addressed these two issues**

Only Approximations can be Implemented in OS

- The dynamic changes in LRU and LIRS cause some computing overhead, thus OS kernels cannot directly adopt them.
- An approximation reduce overhead at the cost of lower accuracy.
- The clock algorithm for LRU approximation was first implemented in the Multics system in 1968 at MIT by
  - Fernando Corbato (1990 Turing Award Winner)
  - The inventor of the Time-Sharing concept

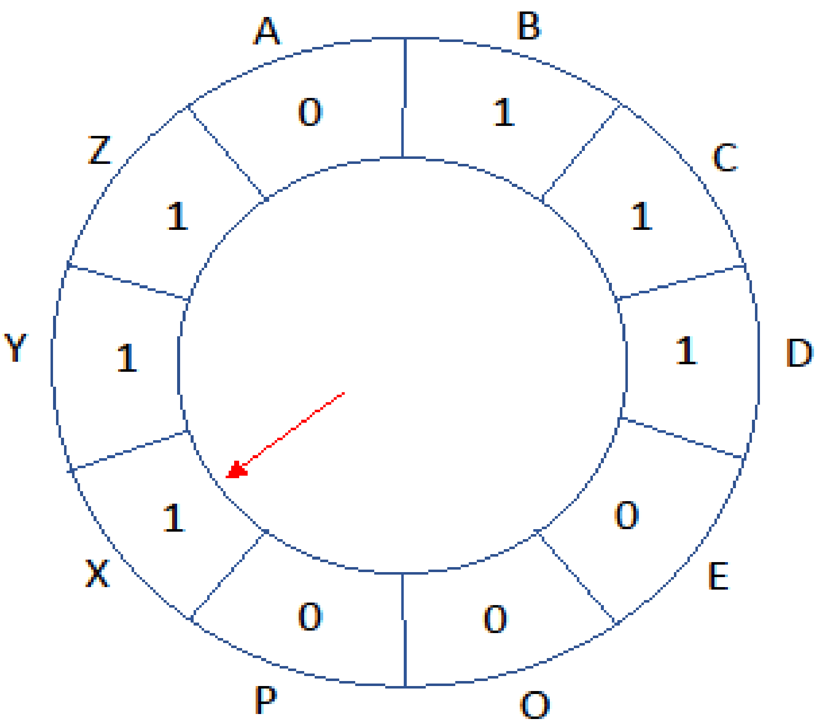


Fernando Corbato

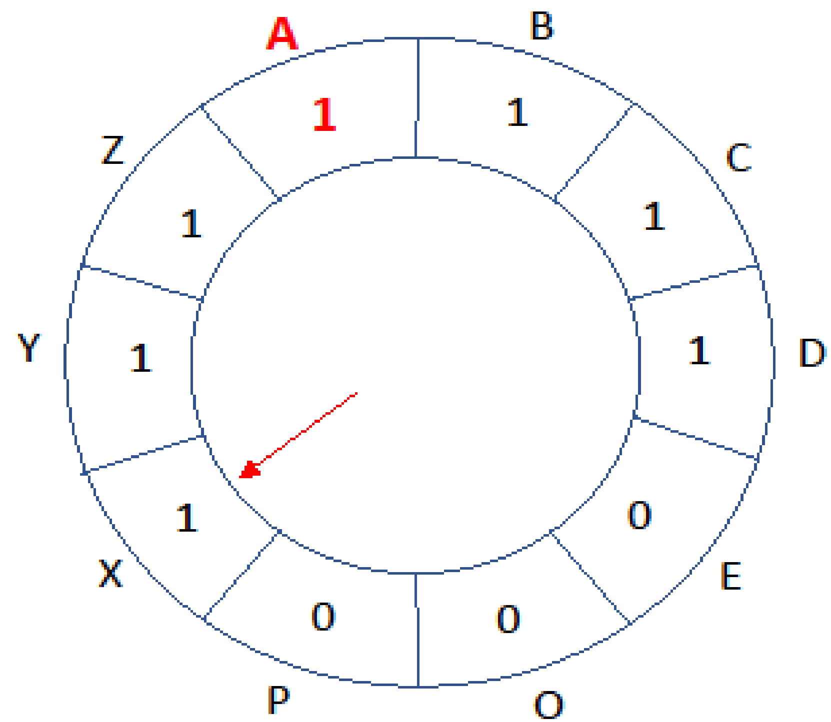
1926-2019

Basic Operations of Clock Replacement

- The cached blocks are recorded in a circular list, like a clock
- Each block has a reference bit (accessed =1, not-accessed =0)



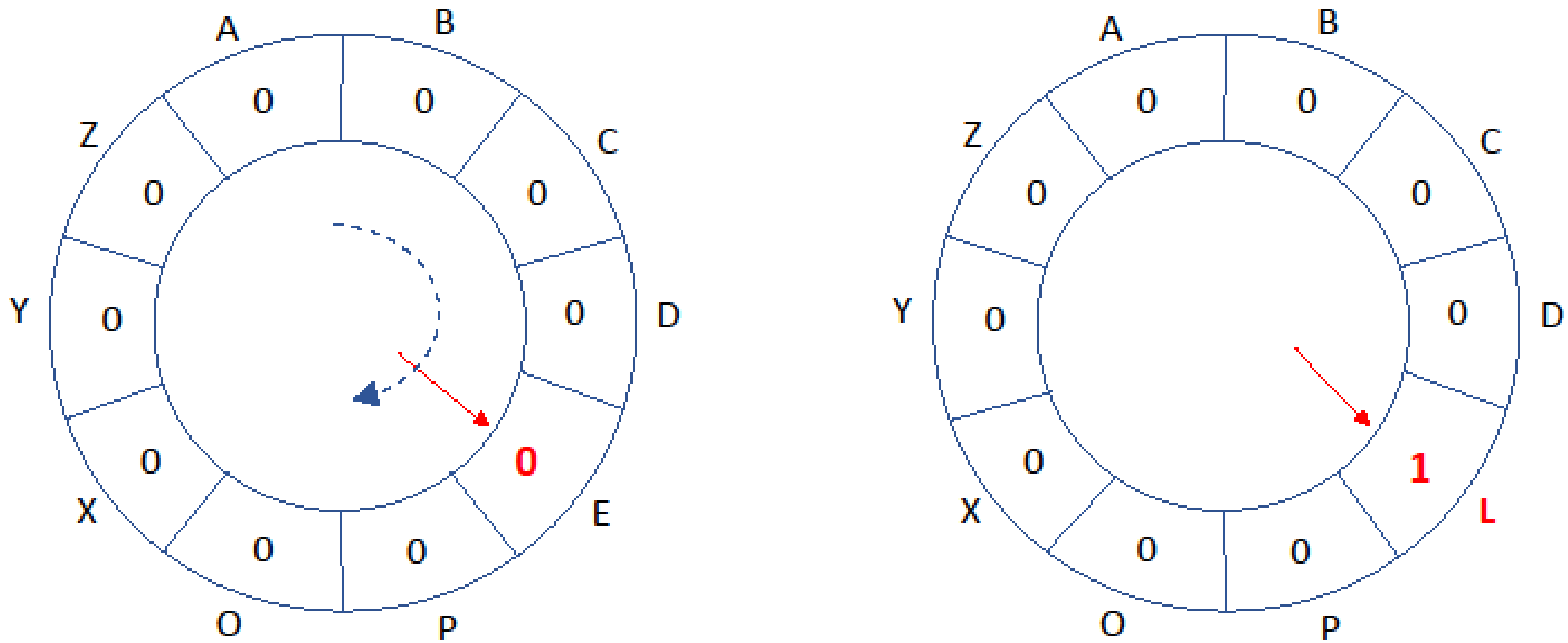
(a) Initial Clock status: Blocks B, C, D, X, Y, Z have been accessed and blocks A, E, O, P have not been accessed. Clock hand points to block X



(b) Accessing block A that is a hit. Access-bit of block A is set to 1 automatedly by hardware without other operations.

Basic Operations of Clock Replacement

- The cached blocks are recorded in a circular list, like a clock
- Each block has a reference bit



(a) Accessing block L that is a miss. The clock hand sweeps to block E that is selected for replacement. On the way of sweeping, the clock hand sets access-bits of blocks X, Y, Z, A, B, C and D to 0s.

(b) Block E is evicted, and block L is in. Access-bit of block L is set to 1.

Some Questions about Clock Replacement

- **Can clock always find a block for replacement?**
  - Yes, even if all the reference bits are 1s, the clock hand sets them all to 0s in the first sweeping loop and get one in the next loop.
- **What does it mean if the clock hand moves slowly?**
  - There are a lot of hits in the buffer. The hand is not actively searching for blocks for replacements
- **What does it mean if the clock hand moves quickly?**
  - There are a lot of accessing misses in the buffer. The hand is actively searching for blocks for replacements
- **How do we characterize the clock replacement?**
  - It may not evict LRU blocks, but evicts not-recently used blocks