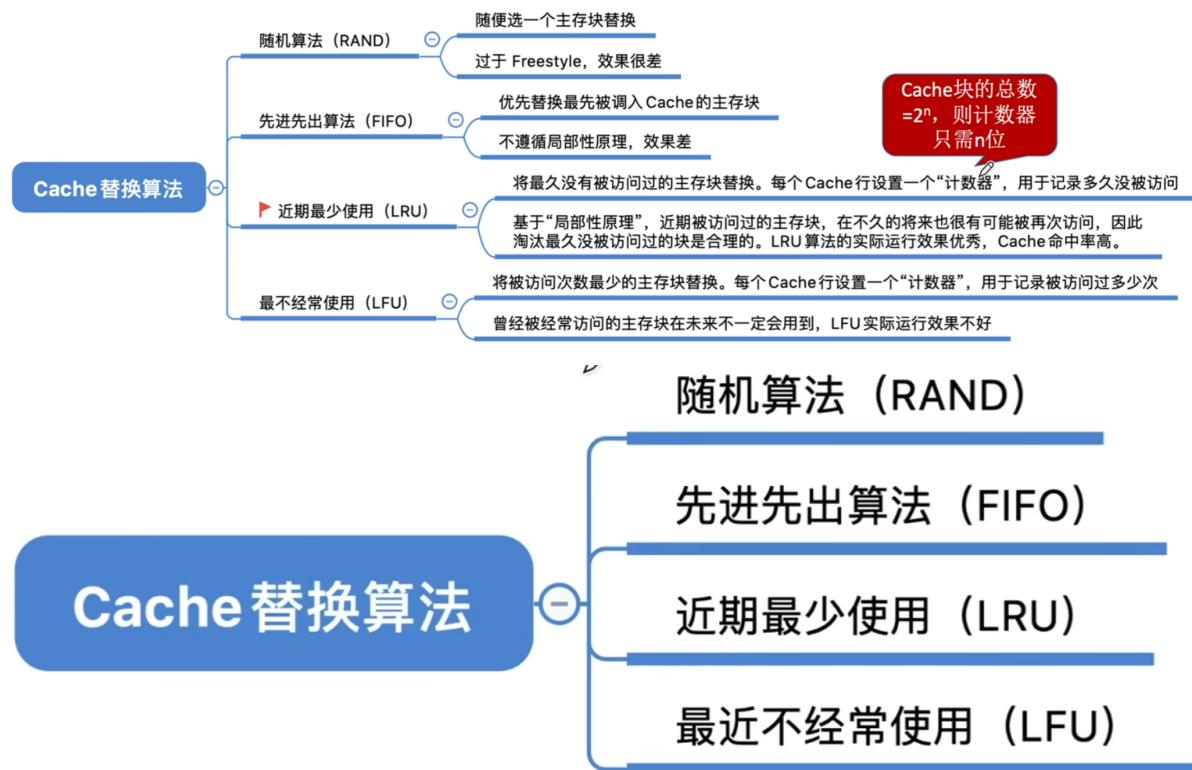
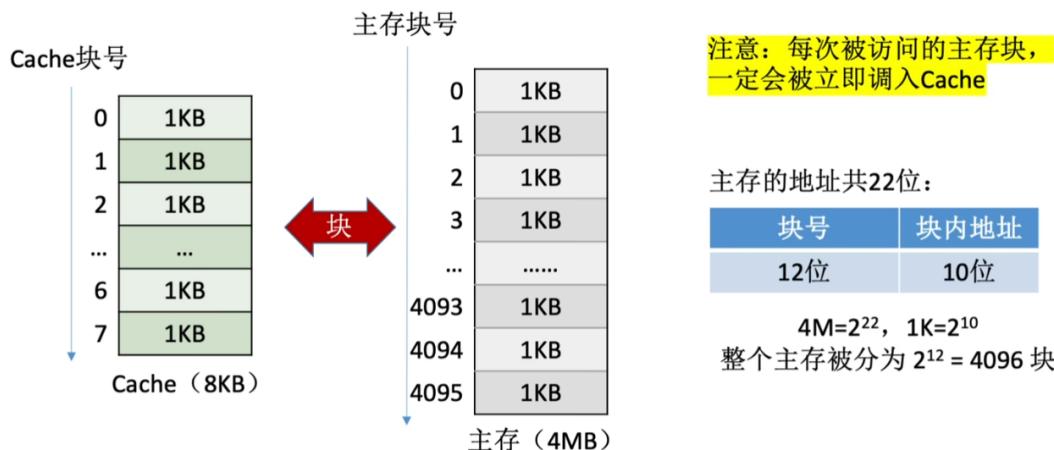


Cache和主存块的替换算法

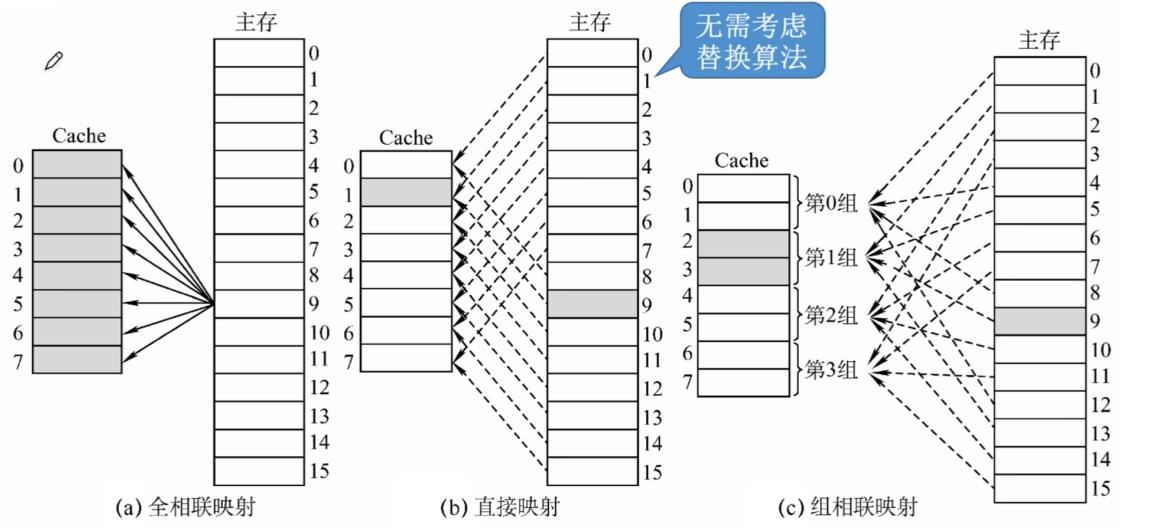


有待解决的问题



- 如何区分 Cache 与 主存 的数据块对应关系? —— Cache 和主存的映射方式
- Cache 很小, 主存很大。如果 Cache 满了怎么办? —— 替换算法
- CPU 修改了 Cache 中的数据副本, 如何确保主存中数据母本的一致性? —— Cache 写策略

替换算法解决的问题



Cache完全满了才需要替换
需要在全局选择替换哪一块

如果对应位置非空，则
毫无选择地直接替换

分组内满了才需要替换
需要在分组内选择替换哪一块

随机算法 (RAND)

随机算法 (RAND, Random) ——若Cache已满，则随机选择一块替换。



设总共有 4 个 Cache 块，初始整个 Cache 为空。采用全相联映射，依次访问主存块 {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}

歪，你有freestyle吗？

访问主存块	1	2	3	4	1	2	5	1	2	3	4	5
Cache #0	1	1	1	1	1	1	1	1	1	1	4	4
Cache #1		2	2	2	2	2	2	2	2	2	2	2
Cache #2			3	3	3	3	5	5	5	5	5	5
Cache #3				4	4	4	4	4	4	3	3	3
Cache命中?	否	否	否	否	是	是	否	是	是	否	否	是
Cache替换?	否	否	否	否	否	否	是	否	否	是	是	否

随机算法——实现简单，但完全没考虑局部性原理，命中率低，实际效果很不稳定

先进先出算法 (FIFO)

先进先出算法 (FIFO, First In First Out) ——若Cache已满，则替换最先被调入Cache的块

设总共有4个Cache块，初始整个Cache为空。采用全相联映射，依次访问主存块 {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}

访问主存块	1	2	3	4	1	2	5	1	2	3	4	5
Cache #0	1	1	1	1	1	1	5	5	5	5	4	4
Cache #1		2	2	2	2	2	1	1	1	1	1	5
Cache #2			3	3	3	3	3	3	2	2	2	2
Cache #3				4	4	4	4	4	4	3	3	3
Cache命中?	否	否	否	否	是	是	否	否	否	否	否	否
Cache替换?	否	否	否	否	否	否	是	是	是	是	是	是

先进先出算法——实现简单，最开始按#0#1#2#3放入Cache，之后轮流替换#0#1#2#3
FIFO依然没考虑局部性原理，最先被调入Cache的块也有可能是被频繁访问的

先进先出算法 (FIFO, First In First Out) ——若Cache已满，则替换最先被调入Cache的块

设总共有4个Cache块，初始整个Cache为空。采用全相联映射，依次访问主存块 {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}

访问主存块	1	2	3	4	1	2	5	1	2	3	4	5
Cache #0	1	1	1	1	1	1	5	5	5	5	4	4
Cache #1		2	2	2	2	2	1	1	1	1	1	5
Cache #2			3	3	3	3	3	3	2	2	2	2
Cache #3				4	4	4	4	4	4	3	3	3
Cache命中?	否	否	否	否	是	是	否	否	否	否	否	否
Cache替换?	否	否	否	否	否	否	是	是	是	是	是	是

抖动现象：
频繁的换
入换出现
象（刚被
替换的块
很快又被
调入）

先进先出算法——实现简单，最开始按#0#1#2#3放入Cache，之后轮流替换#0#1#2#3
FIFO依然没考虑局部性原理，最先被调入Cache的块也有可能是被频繁访问的

近期最少使用算法 (LRU)

近期最少使用算法 (LRU, Least Recently Used) ——为每一个Cache块设置一个“计数器”，用于记录每个Cache块已经有多久没被访问了。当Cache满后替换“计数器”最大的

设总共有4个Cache块，初始整个Cache为空。采用全相联映射，依次访问主存块 {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}

计数器	访问主存块	1	2	3	4	1	2	5	1	2	3	4	5
0 ← 3	Cache #0	1	1	1	1	1	1	1	1	1	1	1	5
3 2	Cache #1		2	2	2	2	2	2	2	2	2	2	2
1 0	Cache #2			3	3	3	3	5	5	5	5	4	4
2 1	Cache #3				4	4	4	4	4	4	3	3	3
Cache命中?	否	否	否	否	否	是	是	否	是	是	否	否	否
Cache替换?	否	否	否	否	否	否	否	是	否	否	是	是	是

Cache块的总数
 $=2^n$ ，则计数器
只需n位。且
Cache装满后所
有计数器的值
一定不重复

- ①命中时，所命中的行的计数器清零，比其低的计数器加1，其余不变；
- ②未命中且还有空闲行时，新装入的行的计数器置0，其余非空闲行全加1；
- ③未命中且无空闲行时，计数值最大的行的信息块被淘汰，新装入的块的计数器置0，其余全加1。

LRU算法——基于局部性原理，近期被访问过的主存块，在不久的将来也很有可能被再次访问，因此淘汰最久没被访问过的块是合理的。LRU算法的实际运行效果优秀，Cache命中率高。

若被频繁访问的主存块数量>Cache行的数量，则有可能发生“抖动”，如：{1,2,3,4,5,1,2,3,4,5,1,2...}

最不经常使用算法 (LFU)

最不经常使用算法 (LFU, Least Frequently Used) —— 为每一个Cache块设置一个“计数器”，用于记录每个Cache块被访问过几次。当Cache满后替换“计数器”最小的

设总共有 4 个Cache块，初始整个Cache为空。采用全相联映射，依次访问主存块 {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}

计数器	访问主存块	1	2	3	4	1	2	5	1	2	3	4	5
2	Cache #0	1	1	1	1	1	1	1	1	1	1	1	1
2	Cache #1		2	2	2	2	2	2	2	2	2	2	2
0	Cache #2			3	3	3	3	5	5	5	3	3	5
1	Cache #3				4	4	4	4	4	4	4	4	4
	Cache命中？	否	否	否	否	是	是	否	是	是	否	是	否
	Cache替换？	否	否	否	否	否	否	是	否	否	是	否	是

LFU算法——曾经被经常访问的主存块在未来不一定会用到（如：微信视频聊天相关的块），并没有很好地遵循局部性原理，因此实际运行效果不如 LRU