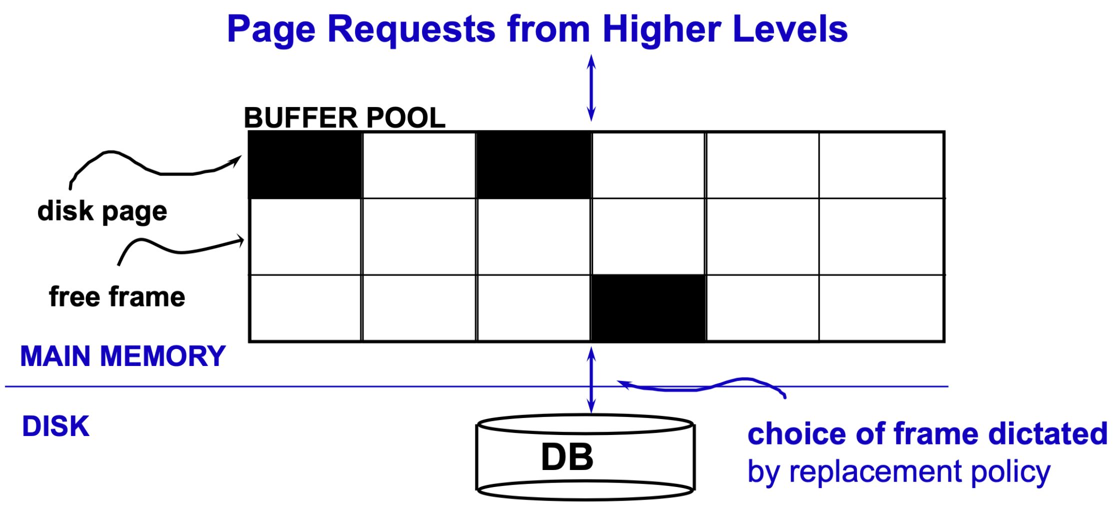
CAS CS 561: Data Systems Architectures
Data-intensive Systems and Computing Lab
Department of Computer Science
College of Arts and Sciences, Boston University
http://bu-disc.github.io/CS561/

CS561 Spring 2024 - Systems Project

Title: *Implementing a Bufferpool Manager of a DBMS*

Background: The bufferpool of a DBMS always maintains *in memory* a set of pages, and allows incoming page requests (writes or reads) to be served without accessing the disk. The purpose of the bufferpool is to improve database system performance. Since data can be accessed much faster from memory than from disk, the fewer times the database manager needs to access disk, the better the performance.
When an application accesses a page, the database manager checks the bufferpool first. If the requested page is in the bufferpool (*page hit*), the database manager does not need to go out to the disk to fetch the requested data. However, if the requested page is not found in the bufferpool, then a *page miss* occurs and the database manager needs to access the disk to retrieve the corresponding page. If we have a page miss and there is no space in the bufferpool, the *page replacement policy* selects a page to be evicted from the bufferpool. If the page has not been updated (it is *clean*), it is just removed from the bufferpool. Otherwise, if the page has been updated (*dirty*), it is written back to the disk before being removed. In essence, the page replacement policy dictates the order in which page access requests reach disk.


Page Requests from Higher Levels
BUFFER POOL
disk page
free frame
MAIN MEMORY
DISK
DB
choice of frame dictated by replacement policy

Popular Page Replacement Policies:
The goal of a page replacement algorithm is to select a victim page for eviction. An ideal page replacement policy is supposed to minimize the number of disk access (minimize *page miss*). However, it is impossible to select one best page replacement policy because the workload dominates which page replacement policy will be best. Here we list some popular page replacement policies used in Database and Operating System communities.

- **LRU** is the most commonly used policy for page replacement because of its simplicity and competitive performance in traditional hard disks. LRU considers temporal locality, which means it assumes that a page accessed (referenced) more recently is likely to be accessed (referenced) again in the near future. LRU maintains the page list in the order of last access time and selects the least recently accessed page as a victim.
- **LFU** keeps track of how many times each page has been accessed since the page was brought in the bufferpool. The page that has been referenced the least is selected for eviction
- **FIFO** as the name suggests maintains a FIFO (First In First Out) list and evicts in that manner.
- **CFLRU** is optimized for flash-based storage. CFLRU maintains the LRU order of the pages and divides the LRU list into two regions: working region and clean-first region. The working region contains the recently accessed pages. In order to minimize the number of writes, CFLRU evicts clean pages from the clean-first region and when there are no clean pages in that region, it evicts dirty pages following classical LRU. The size of the clean-first region is decided by a parameter called the *window size*. Although, the optimal window size depends on the workload characteristics, the rule-of-thumb is that if N is the size of the bufferpool, then N/3 is a good window size.
- **LRU-WSR** or LRU with Write Sequence Reordering (LRU-WSR) is another policy optimized for flash storage. It delays evicting *cold* dirty pages to reduce the number of writes. Each page in LRU-WSR maintains a *cold* flag which is cleared every time the page is referenced. If the candidate page for eviction is dirty, it will be evicted only if its cold flag is set; otherwise the page is moved to MRU position while setting the cold flag and another candidate page is selected based on the LRU order. If the candidate page is clean, it is evicted irrespectively of the status of its cold flag.
- **SIEVE** is an efficient cache eviction algorithm designed for web caches. SIEVE offers better performance than traditional LRU-based methods while maintaining ease of implementation. Unlike complex eviction policies that rely on multiple queues or SIEVE requires only one FIFO queue and one pointer called “hand”. The queue maintains the insertion order between objects. Each object in the queue uses one bit to track the visited/non-visited status. The hand points to the next eviction candidate in the cache and moves from the tail to the head.

Objective: The objective of the project is to implement a Bufferpool Manager with at least three page replacement policies: LRU, CFLRU, and LRU-WSR [1, 2, 3].

(a) Review and understand how bufferpool is used in DBMS to optimize performance.
(b) Understand the impact of different page replacement policies and how they play out. Learn about various page replacement policies: LRU, LFU, CFLRU, LRU-WSR.

You should go over the CFLRU and LRU-WSR papers to understand how they work.

(c) Develop a bufferpool and implement the three page replacement policies (LRU, CFLRU, and LRU-WSR) by cloning the API available to you at: https://github.com/BU-DiSC/cs561_templatebufferpool. By default, without simulation on disk, you can just count the page miss/hit without performing real reads and writes. However, you are also required to implement the simulation on disk. With adding "--simulation_on_disk" flag, each read should go to the specific page with a specific offset, and read the whole entry; each write should update one entry within a specific page and offset using the input new entry. You must perform a comparative analysis on the three page-replacement policies based on different workloads with/without simulation on disk. Some useful metrics: #page misses, #page hits, #total writes, execution latency, etc.

[1] E. J. O'Neil, P. E. O'Neil, G. Weikum, "The LRU-k Page Replacement Algorithm for Database Disk Buffering", Proc. ACM SIGMOD Conf., May, 1993.

[2] Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jinsoo Kim, and Joonwon Lee. 2006. CFLRU: a replacement algorithm for flash memory. In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES).234–241. https://doi.org/10.1145/1176760.1176789

[3] Hoyoung Jung, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. 2008. LRU-WSR: integration of LRU and writes sequence reordering for flash memory. IEEE Trans. Consumer Electron. 54, 3(2008), 1215–1223. https://doi.org/10.1109/TCE.2008.4637609