PROJECT DESIGN DOCUMENT

GROUP 13

Mandeep Bawa, K. Sai Anuroop, Ashish Kumar Bhoi

10th November 2019

Problem Statement

To provide page buffering for the PF layer using two page-replacement strategies to be chosen for a file at the time of opening it: LRU and MRU.

Our understanding of the problem statement

Here are the objectives of this problem statement:

1. Implement a buffer for the PF layer
2. Define the type of buffer to be used
3. Explore two replacement strategies for pages in the buffer; namely, LRU and MRU
4. Define indexing strategy for pages in the buffer
5. Define procedure for assigning 'dirty flags' to the modified pages
6. Define the policy to be implemented while writing back to the disk
7. Define parameters to gauge the performance of the buffer and analyse the statistics obtained

Design

- **Buffer Type**
  We plan to use an already implemented buffer in which the linked-lists are allocated to the entries of the PFhashtbl (buckets). The hash table is direct-mapped and the linked-list is fully associative.

- **Page Replacement Strategies**
  Least Recently Used (LRU)
  In this policy, the page which is least recently used gets evicted from the buffer, in case the buffer is running at its maximum capacity.
  Most Recently Used (MRU)
  In this policy, the page which is most recently used gets evicted from the buffer, in case the buffer is running at its maximum capacity.

- **Page Indexing in Buffer**
  We will use the indexing strategy implemented in PFhashFind() of hash.c wherein we find the bucket to which the page is allocated, and do a linear search through the linked-list of the pages to find the required page

- **Modifications to Pages in Buffer**

We will use ‘dirty flag’ of the page defined in buffer which helps us to determine the process to be done while evicting a page from the buffer. We plan to implement write-back policy for buffer writes.

- **Parameters**
  Parameters under consideration are:
  1. Buffer size PF_MAX_BUFS
  2. Replacement strategy
  3. Hash table size PF_HASH_TBL_SIZE

- **Statistics**
  1. Number of buffer read requests
  2. Number of buffer write requests
  3. Buffer hit rate
  4. Buffer miss rate
  5. Number of buffer evicts
  6. Number of logical I/Os
  7. Number of physical I/Os
  8. Average lifespan of an entry in buffer

Algorithm to be implemented

- When a page is called by the PF Layer, PFbufGet() is executed, which first checks for the bucket to which the page is allocated using PFhashFind(), and a linear search is done through the linked-list of the pages to find the required page.
- If the buffer contains that page, it will be directly fetched from the buffer without accessing the disk (free linked-list in this case). If it is not present in the buffer, it will be fetched from the disk to the PF Layer and will also be stored in buffer using PFbuInternalAlloc().
- When data is to be written to a page, the write is made in the buffer without writing to the disk (free linked-list in this case) and the dirty flag of that page in buffer is set to 1.
- When a new entry is to be placed in the buffer, if there is space in the buffer (PFnumbpage < PF_MAX_BUFS), it will be attached to the linked-list using malloc(), based on the replacement strategy used. If the buffer is at its maximum capacity, depending on the replacement strategy (LRU or MRU), the entry to be evicted is chosen and if its dirty flag is set to 1, it will be written to the disk and evicted. If its dirty flag is set to 0, then it is directly evicted. The new entry will be attached as the head of the linked-list.

Overview of using ToyDB code

We will implement our buffer for the PF layer as per the algorithm described above, by appropriately editing the buf.c file in the pflayer folder of ToyDB, as described below.

- As LRU replacement scheme is already implemented, we will first implement MRU scheme as follows:
  The part of the code for choosing the victim to be written out in PFbufInternalAlloc() of buf.c will be appropriately modified to remove the first page (head) in the buffer page linked-list.

- Next, we will vary the buffer and hash table size by changing the values of PF_MAX_BUFS and PF_HASH_TBL_SIZE respectively, and gauge the performance in both the LRU and MRU schemes.

- Statistics will be implemented as follows:
  1. Counter will be placed in PF_GetThisPage() function in pf.c to count buffer read requests
  2. Counter will be placed in PFwritefcn() to capture number of write requests
  3. Counter will be placed in if (error== PFE_PAGEFIXED) of pf.c to check the number of page fixes (hits) in the buffer.
  4. Counter will be placed in an else statement that will be written after line 530 to capture buffer misses and miss rate
  5. Counter for number of buffer evicts will be placed in PFbufInternalAlloc() of buf.c under else statement at line 147.
  6. Logical I/Os will equal the number of buffer read/write requests
  7. Physical I/Os will equal the number of accesses to the hash table PFhashtbl[] array
  8. Based on the counters defined above in buf.c, we will calculate average lifespan of a page in the buffer
- We will test the performance of the algorithm under the said parameters and replacement strategies using the test programs provided.

Output

Statistics file containing various statistics as described above will be printed out to the console at the end of execution of a test program.

END OF THE DOCUMENT