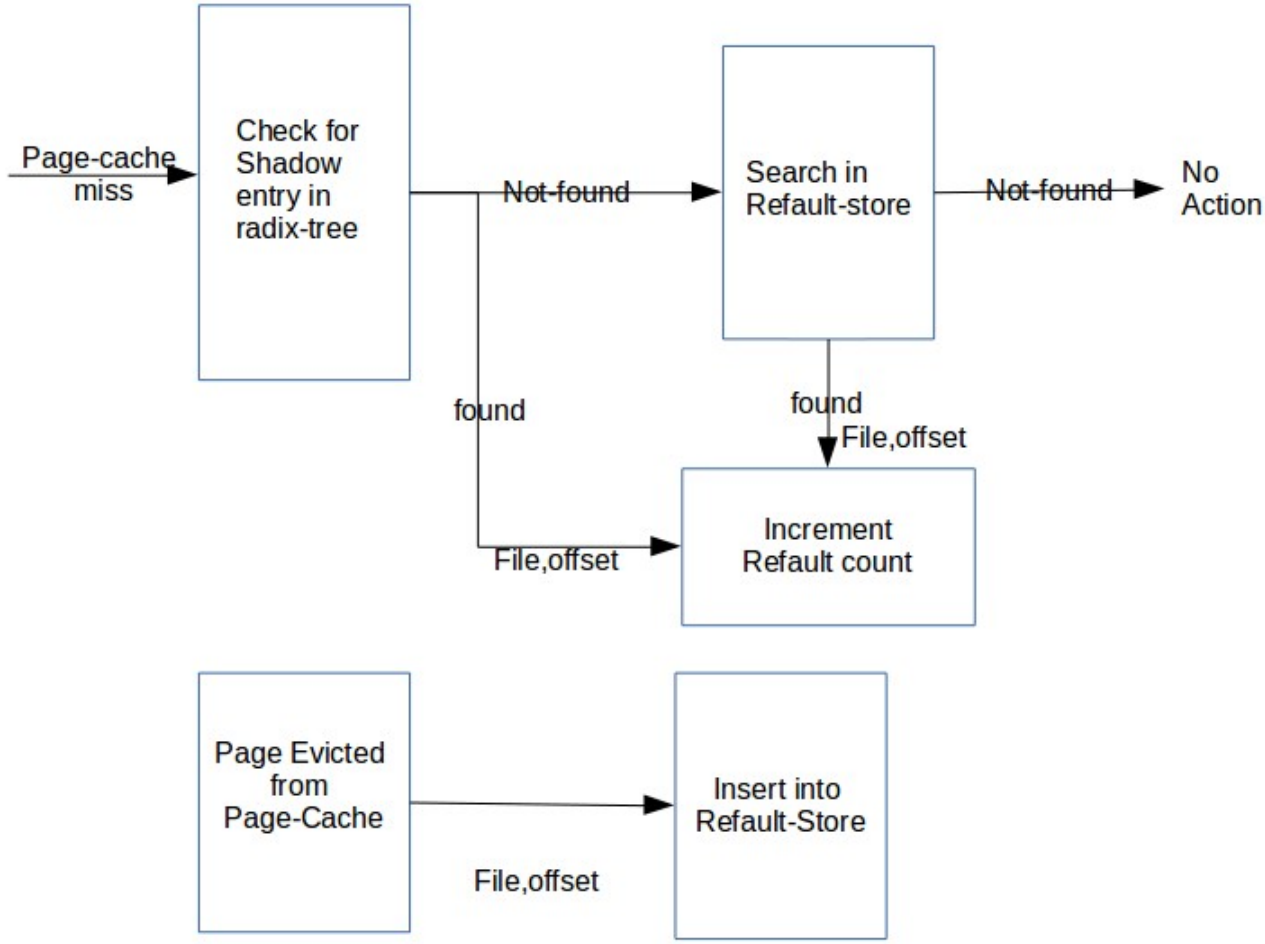
Author:- Mahendra kanani(133059006)

Project objective :-

For improving the performance of I/O, most OS caches the accessed pages from disk. After some time some of the evicted pages again accessed. If these pages would have been in memory then page-fault would have not occured. These kinds of faults called 'refaults'. The goal of the project is developing a tool which reports the reafult counts.

Methodology:-

The main idea is whenever the cached page got evicted we will store the identifiers of this page with refault count to be 0. In future, When there is any page fault, we will check whether it is for one of the pages which were previosly evicted. If it is then increment the counter. Below fig shows the high-level design semantics.



Implementation stages:-

Stage -1 :- Very basic implementation

Data-structure :- a linked list(list_head structure is used).

Hook put at the page fault location. Whenever there is any page fault, it will check for inode-offset pair within the linked list. If it is found then incrementes the counter. If not found then it will insert it into linked list with refault count is 0.

Stage-2 :-

Data-structure :- list of linked list for each file(inode).

Changes in Linux-source-tree :- Yes, In file mm/filemap.c, 2 new hooks added.

One hook is put at the location where the page from cache is evicted. At this point, it will check whether this page's information already exist or not if not exist then insert.

Second hook is put at the location when there is shadow entry found instead of page in radix-tree for currently faulted page.

While the hook at the page-fault location will only check whether it is refault or not. If it is refault then increment corresponding count by one. Here difference with the first approach is that it will not insert the data on page-fault but when page will be removed from cache it will be inserted.

Effects :-

Searching time will improve since first we are checking the inode. e.g. Suppose 1GB file accessed once and then there are many small files accessed. In this case, this approach will give eliminates the searching on 1GB files data.

If we carefully observe the effects of addition of 2nd hook then it just delays the insertion time for certain page and always active page cached page will never be inserted.

Other data-structure options:-

radix-tree data-structures :- In case of huge files, searhing will be efficient compare to linked list but if we want to restrict the number of entries and enforce lru on that then we will not be able to do that.

Stage-3 :-

Only changes in module , no kernel-source-tree modifications.

For files of size 1GB, if its all pages will be accessed then our structure will consume ~4MB space. Each page entry consumes 16 bytes in our structure. Suppose these kinds of files only accessed once or rarely accessed, So it is not fruitful to reserved space behind these kinds of files. It is better to timely remove the older entries.

On every timeout, it will set flag to inactive value for each inodes/files and offset. On next timeout, the entries having inactive flag value will be removed. So the refaults count will be available which are frequently occurring within this time-interval.

I have set time-interval to be 30sec.

Possible extensions:-

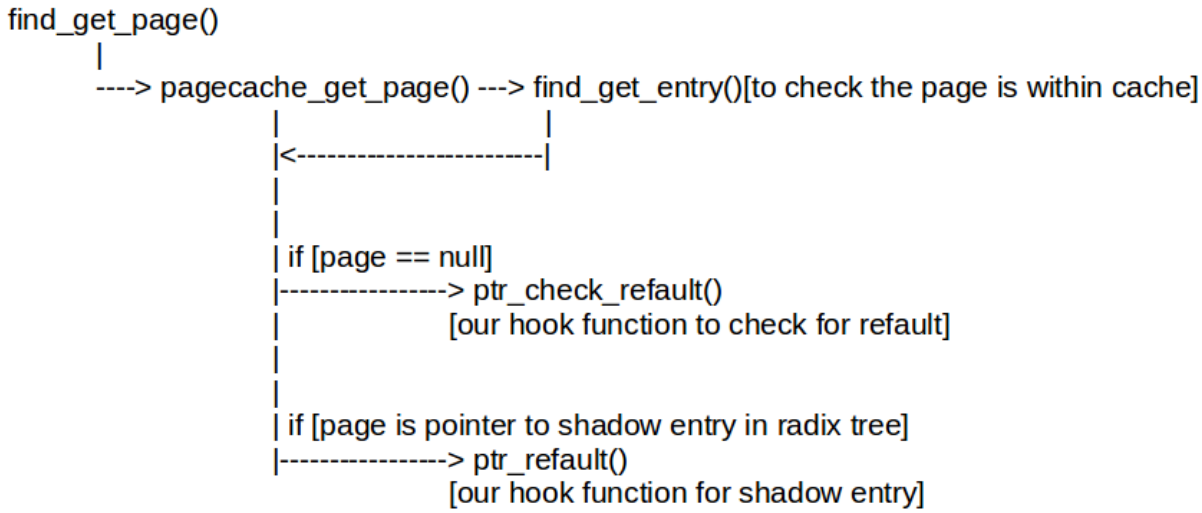
In hook function we can also pass the size of file. If it is huge file(can be decided based on threshold) then we can extend previous data-structure which will restrict the number of entries to some max number. Once max number is reached, lru can be implemented on this individual list.

Hook points :-

1) When page-cache miss occurs :- to check for refaults

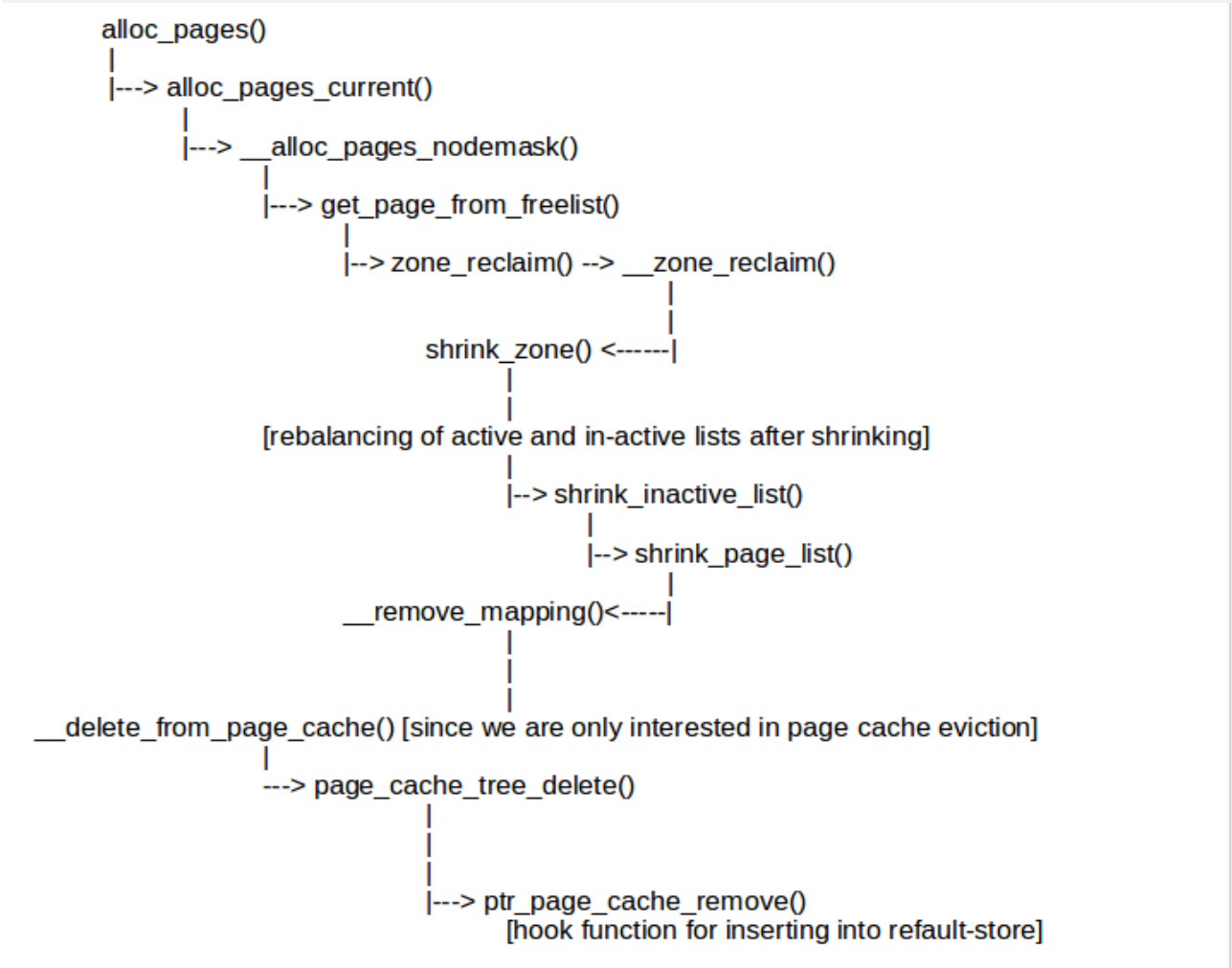
Whenever there is file-read request/write request is issued, it will first search whether page is already present in main memory or not. So it will call find_get_page() method.

Below fig shows the exact function traces when our hook function will be called and which will be called.



2) When cached page get evicted :- to insert inode-offset pair into refault-store.

Allocating a new page might cause some page to be removed if enough memory is not available. So allocating a page can lead to eviction of page cached pages. Below fig shows the hook point at the time of page-cache eviction.



In case of page replacement within lru-list rather than freeing it(i.e. replace_page_cache_page() funtion called), this replace_page_cache_page() is also calling the __delete_from_page_cache() method for removing old page mappings so our hook function ptr_page_cache_remove() will be called.

Experiments:-

1) **Deterministic experiment for proving the correctness of my solution**:-

steps:-

0) clear page cache
1) Module loaded
2) script started
2.0) iteration for 4 times
2.0.1) A process started which reads and writes the file of 10MB on every 2048th byte.
2.0.2) cache memory cleaned using command this will cause the insertion for refaults of above file.
3)script terminated
4) Module unloaded
5) dmesg will print the refaults for each page.

In log, the refault count for the file read by above program should be 3 for each offset. In first iteration, it will cache the pages and when the cache page dropped, all entries will be bookkept. On every successive iteration, there will be exactly one page fault.

2) **Finding refaults for a file which is read on “long”-interval basis:-**

- A programm which will be in infinite loop. Within this loop, it is reading ~10MB file and then go for sleep for 15 sec.
- Parallely reading and writing two ~160MB file one by one this will be performed only once. This was taking around 1min to execute.

What I was expecting is :- due to this parallel reading-writing, these files will cause refaults for this 10MB file. If all pages of this 10MB file is marked active and remained in cache then there will be no refaults for it. When this experiment performed, it found that there was none refaults for it. When there is shadow entry found during page-cache-miss, this page will be marked as active once page is brought into memory. So after some time the entry for this 10MB files into our refault-data structure will be marked as older since there were none refaults reported, and finally it will be deleted.

Apart from this, There is no notion of page-cache size so that we can restrict it to certain limit. We can not control page-cache size in direct manner. So While performing experiment there is no guarantee that these 10MB files pages will be replaced by these 160-MB files pages. I have tried to control the page cache size using changes certain parameters which affects the page cache like swappiness, dirty_expire_centisecs and vfs_cache_pressure. Though I was not able to succeed in this experiment.

Time division (40-50 hrs):-

stage-1 :- 20%

stage-2 :- 50% (finding the location at which page cache get evicted consumed so much time).

stage-3 :- 20%

reporting:- 10%