Memory Reclamation

SWE3015
Sung-hun Kim

This slide is based on Jaeho Hwang’s lecture slide
Memory Overcommit

• Most OSes allow memory overcommit
  – Allocate more virtual memory than physical memory
• How does this work?
  – Physical pages allocated on demand only
  – Allocated pages are not reclaimed until it is actually needed
    • Even if it will never accessed.
  – If free space is low...
    • OS frees some pages non-critical pages (e.g., cache)
    • Worst case, page some stuff out to disk
Memory Overcommit

• To swap a page out...
  – Save contents of page to disk
  – What to do with page table entries pointing to it?
    • Clear the PTE_P bit

• If we get a page fault for a swapped page...
  – Allocate a new physical page
  – Read contents of page from disk
  – Re-map the new page (with old contents)
Reclaiming Pages

• When system runs out of memory
  – Kernel needs to reclaim physical pages for other uses
  – Doesn’t necessarily mean we have zero free memory
    • Maybe just below a “uncomfortable” level

• Where to get free pages?
  – Goal: Minimal performance disruption
    • Should work on phone, supercomputer, and everything in between
Types of Pages

- Unreclaimable:
  - Free pages (obviously)
  - Pinned/wired pages
  - Locked pages

- Swappable: anonymous pages

- Dirty Cache: data waiting to be written to disk

- Clean Cache: contents of disk reads
General Principles

• Free harmless pages first
  – Consider dropping clean disk cache (can read it again)
  – Steal pages from user programs
    • Especially those that haven’t been used recently
    • Must save them to disk in case they are needed again
  – Consider dropping dirty disk cache
    • But have to write it out to disk first
    • Doable, but not preferable

• When reclaiming page, remove all references at once
  – Removing one reference is a waste of time
  – Consider removing entire object (needs extra linked list)
Finding Candidates to Reclaim

• Try reclaiming pages not used in a while
  – All pages are on one of 2 LRU lists: active or inactive
  – Access causes page to move to the active list
  – If page not accessed for a while, moves to the inactive list

• How to know when an inactive page is accessed?
  – Remove PTE_PRESENT bit
    • Page fault is cheap compared to paging out bad candidate

• How to know when page isn’t accessed for a while?
  – Would page fault too often on false candidates
  – Use PTE_ACCESSSED bit (e.g., clock algorithm)
• [http://www.reddit.com/r/linux/comments/1hk5ow/free_buffer_swap_dirty_procmeminfo_explained/](http://www.reddit.com/r/linux/comments/1hk5ow/free_buffer_swap_dirty_procmeminfo_explained/)
When Reclamation Starts?
When Reclamation Starts?

• Low on memory reclaiming
  – Failed to allocate a new buffer page
  – Failed to allocate the temporary buffer heads
  – alloc_pages() failed to allocate contiguous pages

• Hibernation reclaiming
  – Must free memory for entering in the suspend-to-disk state
    • No more discussion for this class

• Periodic reclaiming
  – The kswapd kernel threads check pages_high watermark
The LRU Lists

- Active/inactive list
  - All user mode pages or page caches are belonged one of them.
    - Current implementation divides each list for anon. and file.
  - Unreferenced pages in active list are moved to inactive list.
  - Referenced pages in inactive list are move to active list.
  - Unreferenced pages in inactive list are opt to be reclaimed.
The LRU Lists

- **Active/inactive list**
  - Refer to mm/swap.c/__activate_page
    - `del_page_from_lru_list(zone, page, lru)`
    - `add_page_to_lru_list(zone, page, lru)`
    - Argument “lru” indicates INACTIVE or ACTIVE
Low On Memory Reclaiming

- **free_more_memory()**
  - `wakeup_pdflush()`: write back some dirty pages
  - Calls `try_to_free_pages()`

- **try_to_free_pages()**
  - Does loop from priority 12 to 0
    - Priority means portion of scanning area in zone
    - Call `shrink_zones()`
    - Call `shrink_slab()`
    - If got the goal (reclaiming 32 pages) exit the loop
Low On Memory Reclaiming

- **shrink_zones()**
  - Calls shrink_zone for each zone

- **shrink_zone()**
  - Extracts LRU list and calls shrink_list()
  - Determines how aggressively the lists should be scanned
    - nr_to_scan
  - Fore each LRU list, calls shrink_list
    - Repeat until obtaining the scan count
• **shrink_list()**
  – If the list is an active list, calls shrink_active_list
    • Only if the inactive list has insufficient pages
    • Return 0 because nothing reclaimed
  – If the list is an inactive list, calls shrink_inactive_list

• **shrink_active_list()**
  – Scans inactive and evictable pages in the active list
    • And moves them to the inactive list

• **shrink_inactive_list()**
  – Isolates maximum 32 evictable pages to local list
  – Calls shrink_page_list_list and pass the local list as an argument
  – Reassigns failed-to-reclaim pages to the (in)active lists
  – Return the number of actually reclaimed pages
Low On Memory Reclaiming

• shrink_page_list()
  – Determines whether it is currently to be activated, kept, or reclaimable
  – Do actual page reclaiming process
    • Reclaim only reclaimable pages
**Periodic Reclaiming**

- **kswapd kernel thread**
  - Wakes up periodically and checks free memory for each zone
    - To guarantee enough free pages.
    - To leverage cpu power during idle
  - Starts reclamation if less free pages are in some zones
    - Same mechanism to previous explanation
  - `__alloc_pages()` could wake it up
    - When the free page rate is over its threshold (WMARK_LOW)
• If reclamation is failed?
  – __alloc_pages calls out_of_memory() (OOM).
  – determines a victim process which
    • Owns a large number of page frames
    • Might lose a small amount of work
    • Is a low static priority process
    • Is not a root process
    • Does not directly access hardware devices
    • Is not neither swapper nor init
  – Sends SIGKILL to the victim
• Memory pages are reclaimed if
  – No sufficient pages to allocate
  – The number of free pages is under the threshold
• Simple (in)active LRU list is maintained
  – A page has been referenced since the last check is active.
  – Pages to be evicted are collected in inactive lists.
  – If a page to be evicted becomes active or is being written back, it is reassigned to its list.
  – During eviction, some pages can be swappable, and written to the swap device.
• Almost all flow is done in mm/vmscan.c
• introduced to offer a backup on disk for unmapped pages
  
  – Three kinds of pages that must be handled by the swapping
    • But we know/talk only one: anonymous pages
Swap Area

• Stores the swapped-out pages
  – Either as a disk partition or as a file
  – Multiple swap area can be defined (MAX_SWAPFILES)

• Each swap area consists of a sequence of page slots
  – 4KB blocks: PAGE_SIZE
  – Slot 0 has info structure including magic number
    • “SWAPSPACE2”
Creating a swap area

```
$ dd bs=4096 count=10 if=/dev/zero of=swap
10+0 records in
10+0 records out
40960 bytes (41 kB) copied, 0.000397511 s, 103 MB/s

$ sudo mkswap swap -L os_project
Setting up swapspace version 1, size = 36 KiB
LABEL=os_project, UUID=ab2b6bef-11cc-4af5-a1cc-c11d827c5

$ sudo swapon swap

$ cat /proc/swaps
Filename   Type     Size   Used   Priority
/dev/dm-1  partition  520188  56      -1
/home/hahaman5/swap  file    36      0       -2

$ sudo swapoff swap

$ cat /proc/swaps
Filename   Type     Size   Used   Priority
/dev/dm-1  partition  520188  56      -1
```

24/37
Swapping Out Pages

- Inserting the page frame in the swap cache
  - In `shrink_page_list()`, if a victim page is anonymous and not swap cache, `add_to_swap()` is called to allocate a new slot in a swap area

- Writing the page into the swap area
  - Invoked in `pageout()` called in `shrink_page_list()`
  - `pageout()` invokes `writepage` method of the page’s `address_space` structure.
    - The implementation is `swap_writepage()`.
      - See the code if you are interested.
Swapping Out Pages

- Removing the page frame from the swap cache
  - shrink_list() verifies that no process has tried to access the page frame.
  - Then invokes delete_from_swap_cache() to reclaim the physical page frame.
Swapping in Pages

- The page fault handler triggers a swap-in operation if
  - The page for the fault is a valid one.
  - The page is not present in memory.
  - The pte for the page is not null but the Dirty bit is clear.
  - Then handle_ptd_fault() invokes do_swap_page()
    - Also refer the code if you are interested.