COL331
ASSIGNMENT 3 REPORT

Bhyri Siddhartha Roy
2022CS11105

Mudavath Sai Nikhil
2022CS11633

Swapping Implementation in xv6

2.1 Disk Layout Changes

the previous disk layout was
[boot block | sb block | log | inode blocks | free bit map | data blocks]

Now we modified it to include the swap blocks partition
[boot block | sb block | swap blocks | log | inode blocks | free bit map | data blocks]

- Now to accommodate swap blocks we need to modify the super block and size of the disk.
- Added the following parameters in super block

```
struct superblock {
  uint swapstart;
  uint nswapblocks;
};
```

- Modified the disk size and defined the number of swap slots and blocks on disk for a swap slot.

```
#define NSWAP        800
#define BLOCKSPERSLOT 8
#define SWAPBLOCKS    (NSWAP*BLOCKSPERSLOT)
#define FSSIZE       (1000+SWAPBLOCKS)
```

Modifications in mkfs.c

- Adjusted the nmeta to include the swap blocks and modified the start of the log, inode, bitmap blocks.
- Augmented balloc function to reserve blocks 0…nmeta−1 (boot, super, swap, log, inode, bitmap) by writing setting the bit of these blocks to 1 in bitmap.

2.2 Data Structure for Page Replacement

- Defined swap slot as swappage_t and array of swap slots swap_table[NSWAP];

```
struct swappage_t
{
    int page_perm;
    int is_free;
    int pid;
};
extern struct swappage_t swap_table[NSWAP];
```

- Implemented init_swap_table(), alloc_swap_slot(), free_swap_slot().

2.3 Swapping out a page: Victim selection

(`choose_victim_proc()`):

- Pick runnable/sleeping process with highest rss, tie-break by lower PID.
- Now we fetch the pgdir of the process and get the page table entries using proxytowalkpgdir function. Since walkpgdir is static and cant be included (even declared via extern) in pageswap.c . we write another function proxytowalkpgdir which is not static, which internally calls walkpgdir, so we can use it by declaring it as extern to walk the page directory.
- The loop variable va ranges over all page-aligned virtual addresses of the process.
- For each va, pte is the Page Table Entry describing the status/location/permissions of that page.
- For each PTE: if (PTE_P && !PTE_A) this is the victim page.
- Now we need to swap this page to disk. So we allocate swap slot, write this page to the disk.
- We now save the page_perm = *pte & (PTE_U|PTE_W), pack slot into bits 12–31, clear PTE_P|PTE_A, flush TLB, kfree(pa), decrement p→rss.
- On pages with PTE_A, now we are unsetting the PTE_A bit for 10% of pages so that page eviction doesn't stop because of no access pages(e.g. via (va/PGSIZE)%10==0)

2.4 Adaptive Page Replacement Strategy

```
extern int get_free_pages(void);
void handle_low_mem(void)
{
    int freepg=get_free_pages();
    if(freepg<=th)
    {
        cprintf("Current Threshold = %d, Swapping %d pages\n",th,npg);
        for(int i=0;i<npg;i++)
        {
            int x=handle_page_swap();
            if(x==-1) break;
        }
        th-=(th*beta)/100;
        npg+=(npg*alpha)/100;
        if(npg>=SWAP_LIMIT) npg=SWAP_LIMIT;
    }
    // cprintf("free pages after is %d\n",get_free_pages());
}
```

- **Workload bursts**: Evicting a single page per fault can lead to frequent faults under heavy allocation.
- **Adaptive tuning**: Adjust batch size (Npg) and threshold (T) dynamically to match demand.
- **Threshold (T)** and **batch size (Npg)** are initialized to configurable values (e.g. T=100, Npg=4).
- the system checks the current number of free pages. If free pages drop below the threshold, it evicts a batch of N pages and logs a diagnostic message.
- After eviction, the threshold is decreased by a factor $(1-\beta/100)$, and the batch size is increased by $(1+\alpha/100)$, bounded by a maximum limit, making the policy more aggressive under sustained memory pressure.
- This approach balances prompt reclamation under heavy load with minimal overhead during normal operation.

So whenever kalloc fails, which means there is no free page available on memory, we invoke adaptive page replacement policy, so that pages are swapped out to disk which causes free pages in memory.

2.5 Swap-In (Page Fault Handler)

In trap.c:

```
case T_PGFLT:
    handle_page_fault();
    return;
```

When a page fault occurs (since the page is not in the memory but in the swap space) we need to bring that page back to memory. So in trap.c when a trap occurs because of a page fault then we invoke a function call which moves the page back to memory from disk.

handle_page_fault():

- Read faulting VA from rcr2().
- Since walkpgdir is static and cant be included (even declared via extern) in pageswap.c . we write another function proxytowalkpgdir which is not static, which internally calls walkpgdir, so we can use it by declaring it as extern to walk the page directory.
- Now we iterate over over the page directory and get the page table entry of the faulting virtual address. We stored the swap slot number of the disk in this entry. So we fetch it.
- pte = walkpgdir(pgdir, va, 0).
- extract the swap space slot via slot = (*pte>>12).
- Now to accommodate the page in the memory, we allocate a page using kalloc().
- Now if kalloc fails, which means memory is full, then we invoke adaptive page replacement policy, so that there are free/available pages in memory.
- Read 8 sectors, copy into mem, free_swap_slot(slot), restore PTE to V2P(mem)|perms|PTE_P, flush TLB, increment p→rss.

2.6 Effect of parameters on overall efficiency of the system

The two parameters α (“alpha”) and β (“beta”) are the determining factors in adaptive swapping policy. They govern **when** pages are swapped out (via the threshold T) and **how many** are swapped (via the batch size N). Tuning them well can make the difference between a responsive system and one that thrashes or under-utilizes RAM.

β (Threshold Decay Rate): Controlling when eviction occurs

- **Role of β**: After each swap-out event, the free-page threshold is reduced as

```
threshold-=(threshold*beta)/100;
```

- **Small β** : T only decays slowly, so swaps are triggered even after freeing pages, leading to more frequent but smaller swap batches. This keeps free-page headroom low, conserving RAM for user pages but at the cost of higher page-fault overhead.

- **Large β** : T rapidly decreases after the first set of swaps, so page replacement doesn't take place again until free pages drop very low. That gives stability under temporary bursts, but risks running completely out of pages if subsequent allocations arrive before your next swap check.

α (Batch-Size Growth Rate): Controlling How Many Evictions occur

- **Role of α**: After each swap event we expand the number of pages to be swapped next as

```
npg+=(npg*alpha)/100;
```

- **Small α**
  Each eviction grows only slightly larger than the last. Under sustained pressure, the system will swap in many small rounds, giving recently accessed pages ample chance to be reclassified as recently used.

- **Large α**
  The batch size doubles each time, rapidly ramping up to the maximum. This clears large amounts of memory quickly but can overshoot the working set, causing a subsequent burst of page faults

Thus α governs the "learning rate" of batch sizing: low α means cautious, incremental increases; high α means rapid swappings.

Combined Impact on Efficiency

- **Low α, Low β**
  In this case many small, frequent evictions occur .This approach works well when a program uses a small and consistent set of memory pages, but it can lead to more time being spent handling each page fault.

- **High α, High β**
  Leads to infrequent but very large removals of pages from memory. This can help free up a lot of memory quickly when there's a strong need, but if the system isn't actually using that many pages, it can end up removing useful ones, leading to frequent slowdowns from having to bring them back.

- **Moderate α (20–50 %), moderate β (10–20 %)**
  Maintains a steady balance: the condition for swapping becomes slightly stricter over time, and the number of pages removed increases at a controlled rate. This avoids constantly triggering the mechanism too often, while still freeing up enough memory to prevent repeated delays.