COL331 - A3

Nikhil Zawar - 2022CS11106
Yash Bansal - 2022CS51133

April 30, 2025

Contents

1 Memory Printer

1.1 Introduction

The task involves implementing a handler in the xv6 operating system that detects the Ctrl+I key combination and prints the number of pages residing in RAM for each process categorized as SLEEPING, RUNNING, and RUNNABLE, with PID greater than or equal to 1. This report documents the implementation details and procedures followed to achieve this functionality.

1.2 Detecting Ctrl+I

In `console.c`, a global variable `ctrli` is defined within the `consoleintr` function. This variable is set upon detecting the Ctrl+I key combination, which continuously checks for the signal within an infinite loop. Once `ctrli` is set, the `print_mem_pages()` function, defined in `defs.h`, is invoked.

1.3 Handling Ctrl+I Signal

The `print_mem_pages()` function initializes and prints basic header lines, then iterates over the process table to identify processes with valid PIDs (greater than or equal to 1). For each valid process, it calls the `count_user_pages()` function to calculate and print the number of user-space pages currently residing in RAM.

The logic of `count_user_pages()` is given below:

```
int count_user_pages(pde_t *pgdir){
    int num_pages = 0;
    for (int i = 0; i < NPDENTRIES; i++) {
        if (pgdir[i] & PTE_P) {
            pte_t *pte = (pte_t *)P2V(PTE_ADDR(pgdir[i]));
            for (int j = 0; j < NPTENTRIES; j++) {
                uint va = (i << 22) | (j << 12);
                if ((pte[j] & PTE_P) && (va < KERNBASE)) {
                    num_pages++;
                }
            }
        }
    }
    return num_pages;
}
```

Listing 1: count_user_pages function definition

This function iterates through all page directory entries (PDEs) and their corresponding **page table entries (PTEs)**. It checks if the page is present in memory and within **the user space** (i.e., below `KERNBASE`), incrementing the page count accordingly. The checking should be down within the user-space, which is a important thing to notice here. This is achieved by the use of the KERNBASE macro.

2 Swapping Implementation in xv6

2.1 Disk Layout Modification for Swapping Support

To implement paging and swapping in `xv6`, we have modified the traditional disk layout to accommodate swap space. Enhanced disk layout is:

```
[ boot block | superblock | swap blocks | log |
        inode blocks | bitmap | data blocks ]
```

Each swap slot is designed to hold one memory page, which is 4096 bytes in size. Since each disk block is 512 bytes, a swap slot consists of 8 contiguous disk blocks. We reserve space for a total of 800 such swap slots, leading to an allocation of total swap blocks = 6400.

3 Swap Management Implementation

3.1 Helper Function Declarations in `defs.h`

To facilitate swap management, the following functions have been declared in `defs.h`:

- `void init_swap_slots(void);` – Initializes the swap slot tracking table.
- `int count_free_pages(void);` – Returns the number of free pages in physical memory.
- `void adaptive_page_swap(void);` – Core logic to perform page swapping when memory is low.
- `void update_rss(void);` – Updates the resident set size (RSS) of each process.
- `void free_swap_spaces(struct proc *);` – Frees all swap space used by a given process.
- `int free_swap_slot(int);` – Frees a specific swap slot by index.

3.2 Superblock Modification in `fs.h`

The `superblock` structure in `fs.h` has been extended to include metadata for managing the swap block region:

```
1 struct superblock {
2 ...
3   uint n_swap;                      // Number of swap blocks
4   uint swap_start;                  // Starting block number of swap region
5   uint swap_size;                   // Total size of swap space
6   uint n_swap_blocks_per_page;      // Blocks per swap page
7 };
```

3.3 Swap Slot Structure

A new data structure has been added to represent each swap slot:

```
struct swap_slot_struct {
    int page_perm;
    int is_free;
};
```

Each entry tracks the permission of the swapped-out page and whether the slot is currently free.

The swap system is initialized early during boot by invoking in main.c:

```
init_swap_slots();    // Initialise the swap slot tracking table
```

3.4 Swap Slot Initialization Logic in `pageswap.c`

The `init_swap_slots()` function sets up the global swap table and marks all slots as free:

```
void init_swap_slots() {
    initlock(&s_table_lock, "swap lock");

    cprintf("swap slots initialized\n");
    for (int i = 0; i < SWAP_PAGES; i++) {
        swap_table[i].is_free = 1;
        swap_table[i].page_perm = 0;
    }
}
```

3.5 Swapping-in pages - Control Flow

To enable adaptive swapping during memory allocation, a call to the swap manager is added in `allocuvm()`:

```
adaptive_page_swap();
```

This ensures that whenever a new page is to be allocated, the system checks if memory pressure requires swapping pages to disk to free up space. The function `adaptive_page_swap()` acts as the core logic for initiating a page swap operation whenever physical memory is under pressure. It dynamically adjusts two parameters:

- `Th`: the threshold for the number of free physical pages below which swapping should occur.
- `Npg`: the number of pages to be swapped out in one iteration.

If the number of free pages is greater than `Th`, no action is taken. Otherwise, `Npg` pages are selected and swapped out. The values of `Th` and `Npg` are adjusted adaptively using decay and growth factors (`BETA` and `ALPHA`, respectively), capped by an upper limit (`LIMIT`).

3.6 Page Victim Selection and Swapping Out

To evict pages, the system uses a helper function named `get_victim_proc_page()`, which identifies a candidate process and one of its virtual pages for swapping. Once a victim page is identified, the function `swap_out_page()` is called.

The `swap_out_page()` function is responsible for:

- Finding a free slot in the swap space using `find_free_swap_slot()`.
- Writing the contents of the victim page to disk (using multiple blocks, one per disk sector).
- Storing the original page's permission bits.
- Marking the swap slot as occupied and updating the process's page table entry to reflect the page's new location on disk.
- Releasing the memory frame occupied by the evicted page.

The `get_victim_proc_page()` function returns a process pointer and the virtual address (VA) of a suitable page for eviction.
The function first traverses the process table to identify a suitable process from which a page can be evicted. It avoids system processes (with `pid < 3`) and only considers user-space processes that are in `SLEEPING`, `RUNNING`, or `RUNNABLE` state. Among them, the one with the highest resident set size (RSS) is selected.

```
1  struct proc *victim_p = 0;
2  int p_allocated = 0;
3  int curr_max_rss = 0;
4  acquire(&ptable.lock);
5  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
6      if (p->state == UNUSED || p->pid < 3)
7          continue;
8      if (p->state == SLEEPING || p->state == RUNNING || p->state == RUNNABLE)
           {
9          int rss = count_user_pages(p->pgdir);
10         p->rss = rss;
11         if (!p_allocated || p->rss > curr_max_rss) {
12             victim_p = p;
13             curr_max_rss = p->rss;
14             p_allocated = 1;
15         }
16     }
17 }
18 release(&ptable.lock);
```

Listing 2: Finding the Victim Process Based on Maximum RSS

Once a victim process is selected, its page table is scanned to identify a candidate page for eviction. Priority is given to user pages that are present and have their access bit unset. If no such page is found, the function resets the access bits of all present pages and selects the first present page as the victim.

These steps ensure a basic approximation of a second-chance page replacement strategy, by first trying to evict pages that haven't been accessed recently (access bit unset), and giving a "second chance" to others by resetting their access bits.

```
for (va = 0; va < KERNBASE; va += PGSIZE) {
    if ((pte = walkpgdir_proc(p->pgdir, (char *)va)) == 0)
        continue;
    if ((*pte & PTE_P) && !(*pte & PTE_A)) {
        *victim_pte = pte;
        *victim_va = va;
        return p;
    }
}
for (va = 0; va < KERNBASE; va += PGSIZE) {
    if ((pte = walkpgdir_proc(p->pgdir, (char *)va)) == 0)
        continue;
    if (*pte & PTE_P) {
        *pte &= ~PTE_A;
        if (!found) {
            *victim_pte = pte;
            *victim_va = va;
            found = 1;
        }
    }
}
```

Listing 3: Finding a Victim Page within the Selected Process

3.7 Swap Slot Management

The system maintains a fixed-size swap table that tracks the state of each swap slot. Key functions managing this table include:

- `find_free_swap_slot()`: Searches for an available swap slot and marks it as used.
- `count_free_slots()`: Returns the number of currently available swap slots.

3.8 Reclaiming Swap Slots

To efficiently manage limited swap space, the system incorporates a mechanism to release swap slots that are no longer needed.

Each swap slot holds metadata including usage status and page permissions. When a page is brought back into memory or is no longer needed, the corresponding slot must be cleared. The function `free_swap_slot(int slot)` checks for a valid index and sets the slot's `is_free` flag. It also resets stored permission bits to zero. A lock is acquired during the update to ensure atomicity.

```
int free_swap_slot(int slot) {
    if (slot > 0 && slot < SWAP_PAGES) {
        acquire(&s_table_lock);
        swap_table[slot].page_perm = 0;
        swap_table[slot].is_free = 1;
        release(&s_table_lock);
        return 0;
    }
    return -1;
}
```

Listing 4: Freeing a Swap Slot

3.8.1 Process-Wide Swap Cleanup

When a process exits or no longer needs its swapped pages, the kernel must iterate through its virtual address space to release all associated swap slots. This is handled by the `free_swap_spaces(struct proc *p)` function.

It traverses the address space below `KERNBASE`, identifies page table entries (PTEs) that represent swapped-out pages (i.e., not present in memory but holding a swap index), and calls `free_swap_slot()` to free them. The swap slot index is encoded in the upper bits of the PTE.

```
void free_swap_spaces(struct proc *p) {
    for (uint va = 0; va < KERNBASE; va += PGSIZE) {
        pte_t *pte = walkpgdir_pageswap(p->pgdir, (void *)va);
        if (!pte) continue;
        if ((!(*pte & PTE_P)) && (*pte >> 12) != 0) {
            int slot_num = *pte >> 12;
            if (free_swap_slot(slot_num) < 0) {
                cprintf("cannot free swap slot\n");
            }
        }
    }
}
```

3.9 Adaptive Page Replacement Strategy

The Adaptive Page Replacement Strategy dynamically reacts to the system's memory pressure by monitoring free pages and adjusting swapping behavior accordingly.

When the number of free physical pages in the system drops below a dynamically maintained threshold Th, the kernel proactively evicts a certain number of pages (Npg) from memory to disk (swap space). This helps avoid sudden memory exhaustion and allows the system to operate smoothly under varying workloads.

At each allocation point or during a periodic memory usage check, the kernel evaluates the current number of free pages using `count_free_pages()`. If this number falls below the current threshold Th, the page replacement mechanism is invoked.

3.9.1 Adaptive Swapping Procedure

When the trigger condition is met, the following steps occur:

1. Print the current threshold and number of pages being swapped: `Current Threshold = Th, Swapping Npg pages`
2. Swap out Npg pages using a victim selection strategy implemented in `get_victim_proc_page()` and `swap_out_page()`.
3. Update the threshold Th and the number of pages Npg adaptively:

$$Th = \left\lfloor Th \times \left(1 - \frac{\beta}{100}\right) \right\rfloor \tag{1}$$

$$N_{pg} = \min\left(LIMIT, \left\lfloor Npg \times \left(1 + \frac{\alpha}{100}\right) \right\rfloor\right) \tag{2}$$

Initial values:

- $Th = 100$
- $Npg = 4$
- $\alpha = 25$, $\beta = 10$ (set in Makefile)
- $LIMIT = 100$

3.10 Efficiency Tuning using α and β

Role of α (Growth Factor)

α determines the rate at which the system increases the number of pages to be swapped in future iterations. A higher α causes exponential growth in Npg, allowing the system to quickly reclaim large memory chunks. This is suitable for bursty memory usage patterns but may lead to over-swapping.

Example: If $\alpha = 25$, then after each round:

$$Npg = \lfloor Npg \times 1.25 \rfloor$$

Role of β (Decay Factor)

β governs the decay of the threshold Th. A higher β lowers the threshold more aggressively, allowing the system to delay future swap actions longer. While this reduces swapping frequency, it increases the risk of running critically low on memory.

Example: If $\beta = 10$, then:

$$Th = \lfloor Th \times 0.9 \rfloor$$

3.11 Trade-offs and Observations

- **High** α: Speeds up memory reclamation but may cause performance degradation due to frequent disk I/O.
- **Low** α: Safer under low memory pressure but slower to respond to heavy loads.
- **High** β: Makes the system lenient about when to trigger swapping, risking late reaction.
- **Low** β: Ensures earlier and more frequent swapping but could reduce performance unnecessarily.

3.12 Conclusion

The above mechanism implements a basic but functional paging and swapping system in xv6. It introduces a dedicated swap space in the disk layout, uses metadata to manage each slot, and dynamically responds to memory pressure using adaptive thresholding and victim selection. The system ensures correctness by synchronizing access to the swap table and handling edge cases such as invalid slot access or failed memory allocations.

The adaptive page replacement mechanism enhances memory management by learning from runtime conditions. By tuning α and β, one can balance between responsiveness and system stability.