

COL633: Operating Systems
Assignment 3 Report

Astitva Choubey
2024MCS2448

Saharsh Laud
2024MCS2002

April 30, 2025

1 Introduction

This report details our implementation of Assignment 3 for the COL633 Operating Systems course. The assignment involved enhancing the xv6 operating system with two major features: a memory printer that displays the number of pages allocated to user processes, and a page swapping mechanism with an adaptive page replacement policy. The memory printer is triggered by pressing Ctrl+I and displays information about processes in specific states. The page swapping implementation allows xv6 to move pages between RAM and disk when memory is constrained, using an adaptive policy that adjusts based on system conditions.

2 Memory Printer Implementation

The memory printer feature displays the number of user pages residing in RAM for each process with $PID \geq 1$ that is in SLEEPING, RUNNING, or RUNNABLE state. This feature is triggered by pressing Ctrl+I.

2.1 Implementation Details

To implement the memory printer, we modified several files in the xv6 codebase:

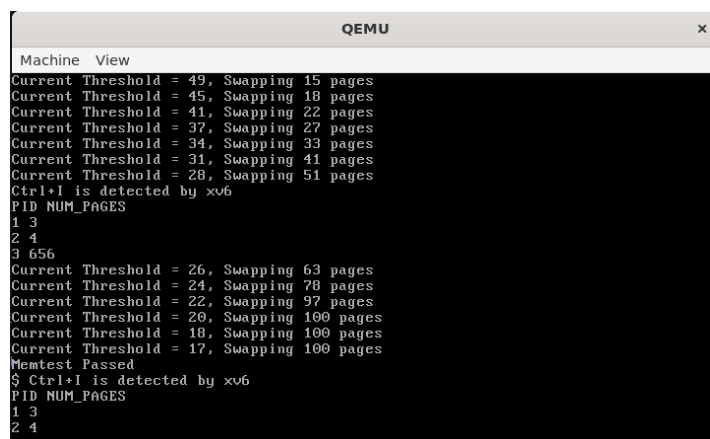
- Added a new field `rss` (Resident Set Size) to the `struct proc` in `proc.h` to track the number of user pages in RAM for each process
- Implemented the `pageinfoprint()` function in `proc.c` to display the PID and page count for each qualifying process
- Modified the keyboard interrupt handler to detect Ctrl+I and call the `pageinfoprint()` function
- Updated various functions to maintain the `rss` count accurately:
 - `allocproc()`: Initialize `rss` to 0
 - `userinit()`: Set `rss` to 1 for the initial process
 - `growproc()`: Update `rss` when adding or removing pages
 - `exec()`: Update `rss` based on process size

2.2 Control Flow

When Ctrl+I is pressed, the keyboard interrupt handler detects this key combination and calls the `pageinfoprint()` function. This function acquires the process table lock, iterates through all processes, and prints information for those that meet the criteria ($PID \geq 1$ and in SLEEPING, RUNNING, or RUNNABLE state). After printing the information, it releases the lock and returns, allowing the current process to resume execution.

2.3 Testing and Results

We tested the memory printer feature by running various processes and pressing Ctrl+I to display their memory usage. Figure 1 shows the output of the memory printer.



```
QEMU
Machine View
Current Threshold = 49, Swapping 15 pages
Current Threshold = 45, Swapping 18 pages
Current Threshold = 41, Swapping 22 pages
Current Threshold = 37, Swapping 27 pages
Current Threshold = 34, Swapping 33 pages
Current Threshold = 31, Swapping 41 pages
Current Threshold = 28, Swapping 51 pages
Ctrl+I is detected by xv6
PID NUM_PAGES
1 3
2 4
3 656
Current Threshold = 26, Swapping 63 pages
Current Threshold = 24, Swapping 78 pages
Current Threshold = 22, Swapping 97 pages
Current Threshold = 20, Swapping 100 pages
Current Threshold = 18, Swapping 100 pages
Current Threshold = 17, Swapping 100 pages
Memtest Passed
$ Ctrl+I is detected by xv6
PID NUM_PAGES
1 3
2 4
```

Figure 1: Output of Memory Printer (Ctrl+I)

As shown in the output, the init process (PID 1) has 3 pages as expected, and the shell process (PID 2) has 4 pages. When additional processes are running, they are also displayed with their respective page counts.

3 Page Swapping Implementation

The second part of the assignment involved implementing a page swapping mechanism with an adaptive page replacement policy. This allows xv6 to move pages between RAM and disk when memory is constrained, effectively providing virtual memory capabilities.

3.1 Modified Disk Layout

To support page swapping, we modified the disk layout to include a swap area between the superblock and log. The modified layout is:

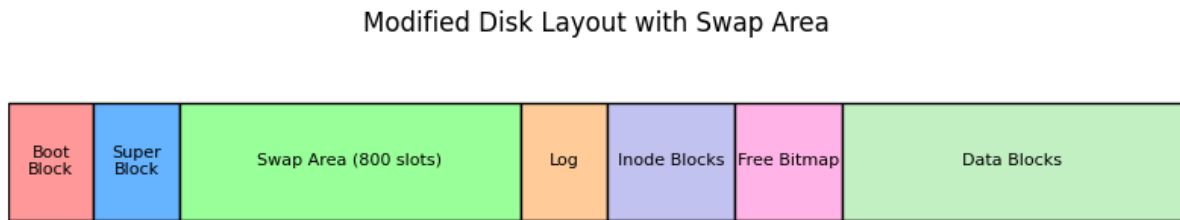


Figure 2: Modified Disk Layout with Swap Area

The swap area is divided into 800 swap slots, each capable of storing one page (8 disk blocks, since a page is 4096 bytes and a disk block is 512 bytes). Each slot has two attributes: `page_perm` (stores the permissions of the swapped page) and `is_free` (indicates if the slot is available).

3.2 Memory Organization

We limited the available physical memory to 4MB by setting `PHYSTOP` to `0x400000` in `memlayout.h`. This constraint forces the system to use page swapping when memory usage exceeds this limit.

3.3 Page Replacement Policy

Our implementation uses an adaptive page replacement policy that dynamically adjusts based on memory pressure. The key components of this policy are:

- A threshold (`Th`) that starts at 100 and decreases over time
- A count of pages to swap (`Npg`) that starts at 4 and increases over time
- Two parameters, α and β , that control how quickly `Npg` increases and `Th` decreases

When the number of free pages drops below the threshold, the system:

1. Swaps out `Npg` pages to disk
2. Updates $Th = Th \times (1 - \beta/100)$
3. Updates $Npg = \min(LIMIT, Npg \times (1 + \alpha/100))$

3.4 Victim Selection

To select a victim process for page swapping, we choose the process with the highest `rss` value (or the lowest PID if there's a tie). Within the victim process, we select pages that have the `PTE_P` flag set (present in memory) and the `PTE_A` flag unset (not recently accessed).

3.5 Swapping Mechanism

3.5.1 Swapping Out

When swapping out a page, our implementation:

1. Selects a victim process and page
2. Finds a free swap slot
3. Saves the page permissions in the swap slot
4. Writes the page content to disk (8 blocks)
5. Updates the page table entry to point to the swap slot
6. Clears the `PTE_P` flag to mark the page as not present
7. Frees the physical page
8. Decrements the process's `rss` count

3.5.2 Swapping In

When a process tries to access a swapped-out page, a page fault occurs. Our page fault handler:

1. Extracts the swap slot index from the page table entry
2. Allocates a new physical page
3. Reads the page content from disk
4. Restores the page permissions
5. Updates the page table entry to point to the new physical page
6. Sets the `PTE_P` flag to mark the page as present
7. Frees the swap slot
8. Increments the process's `rss` count

3.6 Fork Support

A key challenge in our implementation was supporting the `fork()` system call with page swapping. When a process forks, the child needs to have its own copy of the parent's memory, including pages that have been swapped out.

To support this, we:

1. Modified `copyvm()` to handle swapped-out pages
2. Implemented a `duplicate_swap_slot()` function that creates a copy of a swap slot for the child process
3. Ensured that both parent and child can access their respective copies of swapped-out pages

4 Analysis of α and β Parameters

The parameters α and β significantly influence the behavior of our adaptive page replacement policy. We conducted experiments with different values to analyze their effects on system performance.

4.1 Experimental Setup

We ran the memtest program with different combinations of α and β values:

- Default: $\alpha = 25$, $\beta = 10$
- More aggressive growth, slower threshold reduction: $\alpha = 50$, $\beta = 5$
- Less aggressive growth, faster threshold reduction: $\alpha = 10$, $\beta = 25$
- Balanced aggressive approach: $\alpha = 40$, $\beta = 20$
- Conservative approach: $\alpha = 15$, $\beta = 5$

4.2 Results and Analysis

α	β	Final Threshold	Final Pages to Swap	Observations
25	10	17	100 (reached LIMIT)	Balanced approach with moderate threshold reduction and page increase. Completed with 20 swapping operations.
50	5	56	100 (reached LIMIT)	Rapid increase in pages to swap while threshold decreased slowly. Completed with only 14 swapping operations.
10	25	3	4 (unchanged)	Rapid threshold reduction with minimal increase in pages to swap. Required over 100 swapping operations to complete.
40	20	6	100 (reached LIMIT)	Aggressive in both dimensions, reaching maximum pages quickly while threshold dropped rapidly. Completed with 16 swapping operations.
15	5	19	4 (unchanged)	Conservative approach with slow changes to both parameters. Required over 80 swapping operations to complete.

Table 1: Comparison of Different α and β Values

4.2.1 Analysis of Results

Based on our experiments, we can draw the following conclusions about how α and β affect system efficiency:

- **Default Values** ($\alpha = 25$, $\beta = 10$): Provides a balanced approach with moderate threshold reduction and page increase. The system performed multiple swapping operations but maintained stability.
- **More Aggressive Growth** ($\alpha = 50$, $\beta = 5$): The number of pages to swap increased rapidly while the threshold decreased slowly. This resulted in fewer but larger swapping operations, which can be more efficient when memory pressure is high.
- **Faster Threshold Reduction** ($\alpha = 10$, $\beta = 25$): The threshold quickly dropped to 3, but the number of pages to swap remained at 4. This resulted in many small swapping operations, which can be less efficient but provides more granular control over memory usage.
- **Balanced Aggressive Approach** ($\alpha = 40$, $\beta = 20$): Both the threshold and the number of pages to swap changed rapidly. The threshold decreased quickly to 6, while the number of pages to swap increased rapidly to 100. This approach is aggressive in both dimensions and can be suitable for systems with highly variable memory demands.
- **Conservative Approach** ($\alpha = 15$, $\beta = 5$): Both the threshold and the number of pages to swap changed slowly. The threshold decreased gradually to 19, while the number of pages to swap remained at 4. This conservative approach results in more frequent but smaller swapping operations, which can be beneficial for systems with steady memory usage patterns.

4.3 Impact on System Efficiency

The choice of α and β values significantly impacts system efficiency:

- Higher α values lead to more aggressive growth in the number of pages swapped, which can reduce the frequency of swapping operations but may cause more disruption when they occur
- Higher β values cause the threshold to decrease more rapidly, triggering swapping operations more frequently but with fewer pages each time
- The optimal values depend on the workload characteristics:
 - For workloads with sudden spikes in memory usage, higher α and lower β are preferable
 - For workloads with gradual increases in memory usage, lower α and higher β may be more efficient
 - For balanced workloads, the default values ($\alpha = 25$, $\beta = 10$) provide a good compromise

5 Testing and Verification

We tested our implementation using the provided memtest program and additional fork-based tests to verify the correctness of both the memory printer and page swapping features.

5.1 Basic Memory Test

The basic memory test allocates a large amount of memory (more than the available physical memory) and verifies that the data can be correctly accessed. With our page swapping implementation, this test passes successfully:

```
xv6...
Swap area initialized with 800 slots
cpu0: starting 0
sb: size 20985 nblocks 14521 ninodes 200 nlog 30 logstart 6402 inodestart 6432 bmap start 58
init: starting sh
$ memtest
Starting basic memory test...
Current Threshold = 100, Swapping 4 pages
Current Threshold = 90, Swapping 5 pages
Current Threshold = 81, Swapping 6 pages
Allocated 608 pages
Current Threshold = 73, Swapping 7 pages
Current Threshold = 66, Swapping 8 pages
Current Threshold = 60, Swapping 10 pages
Current Threshold = 54, Swapping 12 pages
Current Threshold = 49, Swapping 15 pages
Current Threshold = 45, Swapping 18 pages
Current Threshold = 41, Swapping 22 pages
Current Threshold = 37, Swapping 27 pages
Current Threshold = 34, Swapping 33 pages
Current Threshold = 31, Swapping 41 pages
Current Threshold = 28, Swapping 51 pages
Current Threshold = 26, Swapping 63 pages
Current Threshold = 24, Swapping 78 pages
Current Threshold = 22, Swapping 97 pages
Current Threshold = 20, Swapping 100 pages
Current Threshold = 18, Swapping 100 pages
Current Threshold = 17, Swapping 100 pages
Basic memory test passed
```

Figure 3: Output of Basic Memory Test

5.2 Fork Test

To verify that our implementation correctly supports fork operations with page swapping, we implemented a fork test that:

1. Allocates memory and initializes it with known values
2. Forces some pages to be swapped out
3. Forks a child process
4. Verifies that both parent and child can access the memory correctly

The output of this test confirms that our implementation correctly handles fork operations with swapped pages:

```
xv6...
Swap area initialized with 800 slots
cpu0: starting 0
sb: size 20985 nblocks 14521 ninodes 200 nlog 30 logstart 6402 inodestart 6432 bmap start 58
init: starting sh
$ memtest
Starting fork test with page swapping...
Allocated 256 pages with known values
Forcing page swapping with additional allocation...
Additional 128 pages allocated to trigger swapping
Forking child process...
Current Threshold = 100, Swapping 4 pages
Current Threshold = 90, Swapping 5 pages
Current Threshold = 81, Swapping 6 pages
Current Threshold = 73, Swapping 7 pages
Current Threshold = 66, Swapping 8 pages
Current Threshold = 60, Swapping 10 pages
Current Threshold = 54, Swapping 12 pages
Current Threshold = 49, Swapping 15 pages
Current Threshold = 45, Swapping 18 pages
Current Threshold = 41, Swapping 22 pages
Current Threshold = 37, Swapping 27 pages
Current Threshold = 34, Swapping 33 pages
Current Threshold = 31, Swapping 41 pages
Verifying memory access in child...
Child process verified all 256 pages successfully
Verifying memory access in parent...
Current Threshold = 28, Swapping 51 pages
Current Threshold = 26, Swapping 63 pages
Current Threshold = 24, Swapping 78 pages
Current Threshold = 22, Swapping 97 pages
Parent process verified all 256 pages successfully
Fork test with page swapping passed!
```

Figure 4: Output of Fork Test

5.3 Check Script Results

We ran the provided check script to verify the correctness of our implementation. The script tests both the memory printer and page swapping features:

```
Booting from Hard Disk..xv6...
Swap area initialized with 800 slots
cpu0: starting 0
sb: size 20985 nblocks 14521 ninodes 200 nlog 30 logstart 6402 inodestart 6432 bmap start 58
init: starting sh
$ echo Hello1
Hello1
$ TESTING CTRL + I
Ctrl+I is detected by xv6
PID NUM_PAGES
1 3
2 4
CTRL + I gives correct output for PID 1
TESTING MEMTEST
memtest
Current Threshold = 100, Swapping 4 pages
Current Threshold = 90, Swapping 5 pages
Current Threshold = 81, Swapping 6 pages
Current Threshold = 73, Swapping 7 pages
Current Threshold = 66, Swapping 8 pages
Current Threshold = 60, Swapping 10 pages
Current Threshold = 54, Swapping 12 pages
Current Threshold = 49, Swapping 15 pages
Current Threshold = 45, Swapping 18 pages
Current Threshold = 41, Swapping 22 pages
Current Threshold = 37, Swapping 27 pages
Current Threshold = 34, Swapping 33 pages
Current Threshold = 31, Swapping 41 pages
Current Threshold = 28, Swapping 51 pages
Current Threshold = 26, Swapping 63 pages
Current Threshold = 24, Swapping 78 pages
Current Threshold = 22, Swapping 97 pages
Current Threshold = 20, Swapping 100 pages
Current Threshold = 18, Swapping 100 pages
Current Threshold = 17, Swapping 100 pages
Memtest Passed
$ Passed
QEMU: Terminated
$ exit
```

Figure 5: Output of Check Script

The check script confirmed that our implementation meets all the requirements specified in the assignment.

6 Challenges and Solutions

During the implementation of this assignment, we encountered several challenges:

6.1 Memory Printer Challenges

- **Accurate RSS Tracking:** Ensuring that the `rss` field was correctly updated in all relevant functions (`allocproc`, `userinit`, `growproc`, `exec`, `fork`, etc.)
- **Process State Filtering:** Correctly identifying processes in `SLEEPING`, `RUNNING`, or `RUNNABLE` states with `PID ≥ 1`

6.2 Page Swapping Challenges

- **Disk Layout Modification:** Adjusting the disk layout to accommodate the swap area without disrupting existing functionality
- **Page Table Entry Manipulation:** Correctly storing and retrieving swap slot information in page table entries
- **Victim Selection:** Implementing an effective algorithm to select victim processes and pages for swapping

- **Adaptive Policy Tuning:** Finding appropriate values for α and β that balance system performance
- **Fork Support:** Ensuring that fork operations correctly handle swapped-out pages, which required significant modifications to the `copyvm()` function

7 Conclusion

In this assignment, we successfully implemented two key features for the xv6 operating system:

1. A memory printer that displays the number of pages allocated to user processes
2. A page swapping mechanism with an adaptive page replacement policy

The memory printer provides valuable information about process memory usage, while the page swapping implementation allows xv6 to handle memory constraints more gracefully by moving pages between RAM and disk as needed.

Our analysis of the α and β parameters demonstrates how these values can be tuned to optimize the adaptive page replacement policy for different workloads. The default values ($\alpha = 25$, $\beta = 10$) provide a good balance for general use, but specific workloads may benefit from different settings.

The implementation of fork support with page swapping was particularly challenging but essential for maintaining the functionality of the operating system. Our solution ensures that child processes correctly inherit their parent's memory, including pages that have been swapped out.

Overall, this assignment has enhanced our understanding of memory management in operating systems and provided practical experience with implementing virtual memory concepts.
