



North South University

Department of Electrical and Computer Engineering

Project Report XV6 Memory Extension

Course Code: CSE323

Course Title: Operating Systems

Section: 10

Date of Submission: **17th December 2025**

Student Name	ID
Shafin Rahman	2233146642
Asmaul Islam Tisha	2021934642

Course Instructor: **Md Salman Shamil**

[GitHub Repository](#)

Contents

.....	1
Project Goals:.....	3
Overview of Implementation.....	3
How Fork Works in Default XV6	3
Problems We Addressed	3
What our solution does.....	3
Modifications.....	4
Key Implementation Challenges.....	4
Detailed Kernel Modifications.....	4
Evaluation.....	6
Test Results and Analysis.....	6
Conclusion:.....	7
What we couldn't implement:.....	7
Potential Improvements:.....	7
Team Contribution:.....	7

Project Goals:

Overview of Implementation

This project has added a memory tracking system to XV6 based on x86, that monitors how child processes use memory after forking from their parent. The system includes a new “memstats()” system call that reports three key metrics: **shared pages (memory still identical to parent), private pages (memory unique to this process), and modified pages (memory that has been written to)**. We also track which process is the parent and when the fork took place.

How Fork Works in Default XV6

In the original XV6 kernel, when a process calls fork(), the operating system creates a child process by copying everything. The “copyuvvm()” function allocates fresh physical memory for each page and uses “memmove()” to copy the contents. This approach is simple and safe but has significant costs. Because child gets full allocated memory even if it’s not in used meaning it takes unnecessary space making the system slow.

Problems We Addressed

So, based on our understanding the eager copying approach causes two main problems. First, it's slow. Programs that fork frequently spend a lot of time in the copy loop. Second, it wastes memory. Many programs fork and then immediately call “exec()” to run a different program, throwing away all those copied pages. Our tracking system makes this inefficiency visible. By counting shared versus private versus modified pages, we can see exactly how much unnecessary copying takes place. In Task 1, shared pages will always be zero because every page gets copied. This baseline measurement proves the problem exists and gives us numeric understanding to compare when we implement copy-on-write in Task 2.

What our solution does

CMDT gives developers insight into memory behavior that was previously invisible and COW reduces memory usage by sharing pages until modification is necessary. These features bring XV6 closer to production operating systems like Linux and BSD, which have used COW fork for decades.

Modifications

Key Implementation Challenges

Challenge 0: Understanding at least necessary files their functions and how it's working was the most challenging part, even if understood to a certain degree working was tough because we were forgetting "where" & "what" is taking place and how we need to modify to change so that we could get our desired output.

Challenge 1: Understanding Page Tables. The biggest challenge was figuring out how to walk through page tables correctly. XV6 uses a two-level page table structure on x86. To find a specific virtual address, we needed to use the top 10 bits as an index into the page directory, the next 10 bits as an index into the page table, and the bottom 12 bits as an offset within the 4KB page.

Challenge 2: Iterating Over Virtual Memory. We needed to count pages across the entire user address space without crashing. In fact while implementing I couldn't figure out why the whole xv6 got stuck after "make qemu-nox". We iterated from address 0 to KERNBASE (0x80000000) in 4KB steps, calling "walkpgdir" for each page. The function returns 0 if no page table exists for that address.

Challenge 3: System Call Mechanics. Adding a system call requires coordinating changes across many files. The assembly stub in "usys.S" must match the number in "syscall.h". But it became easy since we had to implement a "demo()" system call giving us an idea how to perform a system call. So, We added one new system call: int "memstats(void)". When a user program calls "memstats()", the assembly stub executes the int instruction with syscall number 23. The CPU switches to kernel mode and the handler calls "myproc()" to get the current process, then calls "getmemstats()" to analyze the page tables, prints formatted output using "cprintf", and returns 0 to user space. The system call must run in kernel mode because user programs cannot access page tables or process structures.

Detailed Kernel Modifications

Process Structure (proc.h): We added five fields to struct proc: parent_pid, fork_time, shared_pages, private_pages, and modified_pages. These fields go at the end of the structure. The parent_pid stores curproc->pid

from when fork() was called, which is different from the parent pointer that changes if the parent exits.

Page Counting Functions (vm.c): We added “countpages(pde_t *pgdir, int check_cow, int check_writable)” which loops through every possible page from 0 to KERNBASE. For each 4KB-aligned address, it calls “walkpgdir” to get the page table entry. We also added “getmemstats(struct proc *p, int *shared, int *private_pg, int *modified)” as a wrapper that calls “countpages()” with the right parameters. In Task 1, it sets shared to 0 and counts all pages as private. This answers the questions which was asked during our first demon on “why the shared page(s) are always 0 not any other number?”

System Call Handler (sysproc.c): The “sys_memstats” function gets the current process, calls “getmemstats()” to fill in the statistics, and prints them using “cprintf” with a formatted output that includes process name, PID, parent PID, and all page counts.

Fork Tracking (proc.c): Inside “fork()”, we added two lines after “pid = np->pid”; to capture the parent relationship and timestamp, “np->parent_pid = curproc->pid”; “np->fork_time = ticks”;

System Call Registration: Four files needed changes: syscall.h defines SYS_memstats as 23, syscall.c adds the extern declaration and array entry, usys.S adds SYSCALL(memstats), and user.h adds the declaration for user programs.

added “cowhandler()” function: Implements the page fault handler that performs actual copying when a COW page is written to. It validates the fault is for a COW page and Allocates a new physical page then copies content from shared page to new page and updates PTE to point to new page with write permission.

Trap Handling (trap.c): we extended the default case in trap() to intercept page faults

Evaluation

Testing Methodology

We developed three user programs to validate the implementation:

memtest.c: Tests Task 1 (CMDT) functionality by performing fork operations and memory allocations while tracking page states at each step.

cowtest.c: Tests Task 2 (COW) functionality by verifying that pages are shared after fork and become private only when written to.

testall.c: Executes both tests sequentially and generates a comprehensive report comparing behavior before and after modifications.

Test Results and Analysis

Stage	Metric	Task 1 (No COW)	Task 2 (COW)
After fork (child)	Shared pages	1 to 0 quickly	Remains Shared
After fork (child)	Private pages	Increases immediately	Minimal Increase
After write	Modified pages	All pages	Only written pages
Parent after child exit	Memory change	Unchanged	Unchanged

So, Child process quickly transitions to fully private memory, in task 1, whereas pages remain shared immediately after “fork()”, in task 2.

Conclusion:

What we couldn't implement:

Due to semester time constraints, we did not implement reference counting for shared pages.

Potential Improvements:

Like modern systems some updates could be

1. Full reference counting, implement proper recounting to safely share pages between multiple processes and free them only when all references are dropped.
2. Multi process sharing, Currently, COW only optimizes parent-child sharing immediately after fork

Team Contribution:

GitHub will provide a perfect understanding of the scenario based on commits done by each member of the group.

Shafin Rahman: Commit performed under the name Shafin & shafinawake

1. Task 2
2. Proposal task 2
3. Final Report, Modifications and Evaluation
4. Editing
5. Evaluation

Asmaul Islam Tisha: Commit performed under the name AsmaullIslamTisha

1. Task 1
2. Proposal task 1
3. Final Report, Project Goal and Conclusion
4. Editing

5. Evaluation