OS-344 Lab - 4

Instructions

- This assignment must be completed in a group.
- Group members should ensure that one member submits the completed assignment before the deadline.
- Each group should submit a report, with relevant screenshots/ necessary data and findings related to the assignments.
- We expect a sincere and fair effort from your side. All submissions will be checked for plagiarism through software, and plagiarised submissions will be penalised heavily, irrespective of the source and copy.
- There will be a viva associated with the assignment. Attendance of all group members is mandatory.
- Assignment code, report and viva all will be considered for grading.
- An early start is recommended; any extension to complete the assignment will not be given.
- If any required information is missing, reasonable assumptions may be made. All such assumptions must be mentioned in the report to ensure transparency and consistency.
- Before you start coding, read Chapters 3 and 4 of the xv6 book, paying close attention to page tables and trap handling.

——————————————————————————

Task 4.1: Implement Copy-on-Write (COW) Fork
In this task, you will modify the xv6 kernel to implement a more efficient version of the fork() system call using a technique called **Copy-on-Write (COW)**.

Objective:
—————

The default fork() implementation in xv6 is simple but inefficient. It allocates new physical memory for the child process and explicitly copies the entire memory of the parent. This can be very slow for large processes. Your goal is to change fork() so that the parent and child initially share all physical memory pages. A new page is only allocated for the child (and the content copied) when one of the processes tries to write to that shared memory.

1. Implement Reference Counting for Physical Pages
To know when it's safe to free a physical page, you must track how many processes' page tables refer to it.
    - Create a reference counting system: You need a way to store an integer reference count for each physical page. A simple approach is a global integer array protected by a spinlock.
    - Modify kalloc() and kfree():
        i. When a page is allocated with kalloc(), its reference count should be initialized to 1.
        ii. When kfree() is called on a page, it should decrement the page's reference count. The page should only be added to the free list if its reference count becomes 0.
    - Add helper functions: Create functions to increment and decrement a page's reference count. These will be used in fork(), the page fault handler, and uvmunmap().

2. Modify fork() for COW

   Change the fork() memory-copying logic to share pages instead of copying them.

   - Update uvmcopy(): This function is called by fork() to duplicate the parent's address space. Modify it to:
     i. Map the parent's physical pages into the child's page table instead of allocating new ones.
     ii. For each shared page, clear the PTE_W (write) flag in both the parent's and child's page table entries (PTEs). This makes the shared pages read-only.
     iii. Increment the reference count for each physical page that is being shared.

3. Implement a Page Fault Handler for COW

   This is the core of the "copy-on-write" mechanism. When a process tries to write to a read-only page, you will handle the resulting page fault.

   - Modify usertrap(): In the trap handler, add a case to handle store/write page faults (scause register value 15).
   - Inside the handler:
     i. Get the faulting virtual address from the stval register.
     ii. Find the physical address of the page and check its reference count.
     iii. If the reference count > 1 (the page is shared):
        - Allocate a new physical page using kalloc().
        - Copy the content of the original (shared) page to the new page.
        - Update the faulting process's page table to map the virtual address to the new physical page with write permissions (PTE_W set).
        - Decrement the reference count of the old physical page.
     iv. If the reference count == 1 (the page is not shared):
        - There is no need to copy. Simply update the process's PTE to add the PTE_W flag, making it writable.
     v. Resume the user process. The instruction that caused the fault will now succeed.

Deliverables:

—————

1. Modified Kernel Files: Submit all kernel files you modified. This will likely include:
   - kernel/kalloc.c (for reference counting and kfree).
   - kernel/vm.c (for uvmcopy and other VM changes).
   - kernel/trap.c (for the page fault handler).
   - Any other files you needed to change (e.g., kernel/proc.h, kernel/riscv.h).

2. Test Program: Create a new user program cowtest.c that clearly demonstrates your COW implementation is working correctly. The program should:
   - Allocate a page of memory.
   - Write an initial value to it.
   - Call fork().
   - The child process should attempt to write a new value to the page.
   - The parent process should wait() for the child and then read the page. The parent must see the original value, proving that the child's write was correctly isolated to a new copy.

Task 4.2: Implement Demand Paging with MRU Page Replacement
In this task, you will extend the xv6 operating system to support **demand paging** using a **Most Recently Used (MRU)** page replacement policy. You are required to modify the kernel to support swapping of user pages, add new **system calls** to monitor paging activity, and implement a test program to demonstrate the correctness of your implementation.

Objective:
—————

1. MRU Page Replacement
   - Maintain a global MRU list of all resident user pages.
   - On every memory access (detected via page faults or through helper functions like copyin/copyout), update the MRU list so the accessed page becomes the most recently used.
   - When physical memory is exhausted, evict the most recently used page from the MRU list.

2. Swap-Out and Swap-In
   - On eviction, swap the victim page out to disk (e.g., a per-process swap file). Update the process metadata and clear the page table entry.
   - On a page fault, if the faulting page was previously swapped out, swap it back in and update the MRU list.
   - If the page fault corresponds to an invalid address, terminate the process.

3. System Calls

   You must add new system calls in xv6 by modifying the appropriate kernel source files and headers. Specifically, implement:
   - int getpagestat(int pid, struct pagestat *st)
     i. Returns the number of page faults, swap-ins, and swap-outs for the given process.
     ii. You must define a new structure struct pagestat in a kernel header file.

   - int dumpmru(void): Prints the current MRU list to the console (showing process IDs and virtual page numbers in MRU order).

   Make the necessary changes in syscall.c, syscall.h, sysproc.c, user.h, and any other relevant kernel files to integrate these new system calls.

4. Test Program (mrumem.c)
   - Implement a user program named mrumem.c that:
     i. Allocates more memory than available physical RAM (using sbrk).
     ii. Accesses the allocated pages randomly to trigger page faults and swapping.
     iii. Periodically calls getpagestat() to display the number of page faults, swap-ins, and swap-outs.
     iv. Calls dumpmru() to print the MRU list so you can verify that eviction is following the MRU policy.

   - The program output must demonstrate that:
     i. The **most recently used page** is always the one evicted.
     ii. Swapped-in pages contain correct data.
     iii. Kernel and page table pages are never swapped.

Deliverables:
—————

- Modified xv6 source code, including all changes to kernel files and headers for the new system calls.
- The test program mrumem.c.
- In your report include:
    i. A clear description of your MRU page replacement mechanism.
    ii. The kernel modifications you made for system calls (list the files changed).
    iii. Example outputs from running mrumem.c, showing paging statistics and MRU ordering.
    iv. Screenshots/logs demonstrating correctness of your MRU implementation.