5b. Copy-on-write fork

The fork.c tests the behavior of the fork system call when multiple processes share the same memory page and one of them writes to it. The program creates N child processes using fork and attempts to modify a shared memory page in each child process, causing a page fault and triggering a copy-on-write operation by the operating system.

The purpose of this program is to test the correctness of the copy-on-write mechanism in the operating system's virtual memory subsystem. When a process forks, the operating system creates a new process that shares the same memory pages with the parent process, but marks them as read-only. If either process attempts to write to a shared memory page, a page fault occurs and the operating system creates a copy of the page for the process that wrote to it, and marks it as writable. This allows multiple processes to share the same memory pages without interfering with each other's data.

By testing the behavior of the fork system call in this way, the program ensures that the operating system correctly implements the copy-on-write mechanism and that multiple processes can safely share memory pages without causing data corruption or other errors.

To implement copy-on-write (COW) fork() in xv6, follow these steps:

1. Modify fork() in proc.c to create a new page table with non-writable entries.
2. Implement a reference count mechanism to track shared pages.
3. Update the page fault handler to handle COW cases.

To demonstrate that the copy-on-write fork works correctly and to compare its performance against the basic fork, the following user programs were added:

- Program with shared read-only data:

  Creating a program that has a large read-only data segment like a large constant array. Forking several child processes that will read and process the data without modifying it. In this scenario, the copy-on-write fork performs better than the basic fork, since it won't create unnecessary copies of the shared data.

- Program with shared and modified data:
  Creates a program that forks multiple child processes, each with its own data to modify. The copy-on-write fork performs similarly to the basic fork in this scenario since each child process will eventually need its own copy of the data.

By comparing the elapsed time for each program using the basic fork and the copy-on-write fork, you can see how the performance is affected by the different implementations. In the first program (shared read-only data), the copy-on-write fork should show better performance, while in the second program (shared and modified data), both implementations should have similar performance.