No. of CPU = 1

output of testcow.c: (After implementing copy-on-write)

Parent start: no. of free page = 56776
Child1: no. of free page just after fork = 56674
Child1: no. of free page after some write = 56673
Child2: no. of free page just after fork = 56708
Child2: no. of free page after some write = 56707
Parent end: no. of free page = 56776

output of testcow.c: (Without implementing copy-on-write)

Parent start: no. of free page = 56776
Child1: no. of free page just after fork = 56636
Child1: no. of free page after some write = 56636
Child2: no. of free page just after fork = 56636
Child2: no. of free page after some write = 56636
Parent end: no. of free page = 56776

No. of CPU = 2

output of testcow.c:(After implementing copy-on-write)

Parent start: no. of free page = 56775
Child1: no. of free page just after fork = 56639
Child1: no. of free page after some write = 56638
Child2: no. of free page just after fork = 56638
Child2: no. of free page after some write = 56637
Parent end: no. of free page = 56775

output of testcow.c:(without implementing copy-on-write)

Parent start: no. of free page = 56775
Child1: no. of free page just after fork = 56635
Child1: no. of free page after some write = 56635
Child2: no. of free page just after fork = 56635
Child2: no. of free page after some write = 56635
Parent end: no. of free page = 56775

Explanation:

Without implementing copy-on-write, all the pages needed by new process are allocated at time of fork itself, thus there is no change in number of freepage just after the fork statement, and after writing in memory, in case of both child1 and child2.

However, after implementing, copy-on-write, at time of fork, new pages are allocated, only at time of write, at the time of fork no user memory is written, thus there is a difference in number of free page, just after fork and after something is written into user part of memory image of the process, both in the case of child1 and child2.

After both child exits, the number of freepage at the statrt of parent process = no. Of freepage at the end of process just before exiting.

Changes Done:

1. kalloc.c
    a.) defined new variable numfreepages to keep track of number of freepage
    b.) defined an array pg_ref_array to keep track of reference count corresponding to each physical location
    c.) defined a new function pgcountinit to be called from main.c, to initialize above two variables
    d.) wrote new kfree and kalloc function, which modifies numfreepages and reference count accordingly
    e.) defined a struct pg_ref for using lock
    f.) defined function increase_ref_count(uint location), to increase reference count of that location
    g.) defined function decrease_ref_count(uint location), to decrease reference count of that location
    h.) defined function set_ref_count_to_one(uint location), to set reference count = 1 of that location

2. main.c
    a.) call function pginitcount to intialize at start

3. user.h
    a.) defined function getNumFreePages(), which can be called externally

4. vm.c
    a.) wrote new copyuvm to implement copy-on-write and handle_pgflt function to handle pagefault

5. testcow.c
    a.) test case file

6. trap.c
    a.) trap handler to handle T_PGFLT and call handle_pgflt function

7. sysproc.c
    a.) defined system call sys_getNumFreePages

8. syscall.h
    a.) system call number corresponding to SYS_getNumFreePages

9. syscall.c
    a.) defined SYS_getNumFreePages = sys_getNumFreePages

10. defs.h
    a.) defined all variable and functions so that they can be accessed from other files as well

11. usys.S
    a.) defined SYSCALL(getNumFreePages)

12. Makefile
    a.) added _testcow, to compile testcow.c
    b.) CPUS changed accordingly to test one 1 or 2 cpus

Copy-on-Write Implementation

fork() system call calls copyuvm() to make memeory image of child process, which in turn call setupvm() function to set up kernel parts of memory images, and then originally kalloc was called for getting new page for each user part of memeory image and new page is given to the child at the time of fork itself.

However, in the modified copyuvm(), only setupkvm() is called. For user part of memory image, new page table entry is made for child process, and the page table entry corresponds to same physical address pointed by the parent, rather than allocating new page using kalloc(). The permission of parent page table entry is set to NOT WRITABLE, and same permission(also NOT WRITABLE), is set for childs parents table, and the reference count of corresponding physical address is incresed by 1, and tlb is flushed and pagetable is reinstalled.

Now, whenever a process wants to write and the permission of corresponding page table entry is NOT WRITABLE, a trap is generated, which in turn call handle_pgflt(). handle_pgflt() allocates new page using kalloc(), and set its permission to WRITABLE, and decrease the refernce count of the old physical address by 1, and the page table is reinstalled.

Since kalloc() always allocates a new page, in kalloc function(), it sets the refernce count of that physical address to be 1, and decreases the number of free pages by 1.

kfree() is also called on the physical address which cause the page fault. The modified kfree, first decreses its reference count by 1, and if the reference count goes to 0, it frees up that page as done by kfree() earlier, and increases the number of free pages by 1.

number of free page is initialized with 0, and number of reference count for all physical location by 1, since main function will call kinit1() and kinit2() for free pages, which will eventually call kfree(), and the reference count will be set to 0 there.