Lab 6 Copy-on-Write Fork in xv6
Report

-- 140050002, 140050007

How to run xv6 :

Change permission of submission6 folder.
Open folder in terminal and run ‘make qemu’

Modification in Data Structure :

We have added three member (**num_free_pages, max** and **reference_count[ ]**) of type int to struct **kmem** in kalloc.c.

> **num_free_pages** stands for the number of free pages at a given time available for allocation. This is just the count of the elements of the free pool linked list already present in **kmem**.
>
> **Max** is a variable that stores the maximum value possible for **num_free_pages**.
> This is needed during **kinit2** itself but is used only for debugging purposes.
>
> **Reference_count[ ]** is the important data structure introduced for storing the number of processes pointing to a page. Here the index of this array of integers is actually the physical frame number (obtained by dividing the physical address by Page size) and the value stored at that index denotes the number of processes pointing to it.

System Call :

1. To create custom syscall first we add our new sys_getNumFreePages function in “**sysproc.c**”(line 93). This in turn calls the function **kfreepages** from kalloc.c(line 149)

2. Next we add their system call numbers in “**syscall.h**”(line 23) and create the mappings to the sys_functions in “**syscall.c**”(line 125) and further introducing the extern function from sysproc.c in “**syscall.c**”(line 101).

3. Then we expose the function to users in “**user.h**”(line 26) and finally add the corresponding new macros in “**usys.S**”(line 32).

4. The “**testcow.c**” code calls the syscalls viz. getNumFreePages() and we compile this user program by adding this file in EXTRA of “**Makefile**”(line 246) and set the user program executables in UPROGS of “**Makefile**”(line 176).

Implementation Details :

To maintain the **num_free_pages** we decrement it by 1 every time a successful **kalloc** happens and increment by 1 when a complete **kfree** finishes.
It can be seen in the respective functions in **kalloc.c** (line 137, 115).

Now we must see to it that kfree should not free a page unless no one is pointing to it.
For this we check the value of the reference count of this page and if it is more than 1 we simply **reduce it by one** but **do not push** the page to the **free pool.**
We do so only when there was just one process or none pointing to it.
It can be seen in **kalloc.c**(line 103).

For external modifications to the reference count to be made we created a function **refer_count(kalloc.c**[line 57]**)**. This takes the physical address of a page and increments the reference by the value provided to it.

We **acquire and release** *the lock* every time we access variables from **kmem**.

Modification in fork :

The **fork** user program now does not call copyuvm().

Instead a completely new function **changeflag(vm.c**[line 331]**)** is called.
This function modifies the flags in the page table entries which come within the pgdir and the size of the process.

The flags are set based on the value of writable. If **writable** is set to 0, it means to have the flags as read-only, we also increase the reference count of all the considered pages by one.

Changeflag refreshes the **cache** every time it is executed successfully.

Trap Handling :

There is another function **reference_counter**(**vm.c**[line 314]) which takes the pgdir and size and **increments** the *reference_count* of all the page table entries by the **value** provided. It also **returns** the *reference_count* of the of the first page in this range **after the modifications**.

Lastly we introduce the trap handler in **trap.c**(line 49). Here we first check the validity of the virtual address that we get from **rcr2()** and kill the process for invalid page faults.

Next we check if there are more than 1 processes pointing to the page which faulted. If there is such a fault we create a copy now (**using copyuvm**) and give the new pgdir to the process that trappe, **reduce the** *reference count* of the pages which trapped and convert this **new copy to writable**.(**trap.c**[line 57])

If the trapped process is the only process that is pointing to the pages then we simply **make these pages writable instead** of making a copy.(**trap.c**[line 66])

Now to finally make all these processes access these functions we add these new functions from **kalloc.c** and **vm.c** in **"defs.h".**

Note that every time we get into a trap for page fault we print a string to the console that shows the virtual address of the page that faulted and the number of processes still pointing to that page. This helps to understand the test case as explained below.

Test Case and Observations :

To run simply replace all the given files in the source directory of xv6 code and then type “**make qemu**” in terminal.
In the qemu terminal that opens type “**testcow**” to start the scheduler testing user program.

The following test case is on a 2 core system

```
$ testcow
In trap 16332 2
In trap 16320 1
Parent 56728
In trap 11228 2
In trap 11228 2
In trap 11228 1
Child 2 56588
In trap 12227 1
Child 56656
```

In this test case we invoke a process that creates a long string and calls two children. These children in turn modify some characters in the string.

Observation explained:

1. First we get a trap with 2 processes pointing to the same virtual address. These are probably the **‘init**’ and the **‘testcow’** processes. This happens when we try to write a string inside testcow parent.

2. Next trap is because of the init processes trying to write to the pages which it alone points to. Note that these **could be vice-versa too**.
3. Next the parent prints the total number of free pages.

4. Then we observe that a page fault happens at a page which is pointed by 2 processes. The next also shows 2. Well actually we clearly know that **if all the forks happened first** then this **would have been 3** with the processes being the **parent and its 2 childrens**.

5. Child 2’ s result is printed first with **number of free pages less than that of parent**. This is because **we copied the pages after the traps** occurred and assigned it to this child.
6. The first child then prints the free pages which also **occurs after the trap**. But now the freepages have increases because Child 2 has freed up some of its copied pages.