(Challenge) Copy-on-Write

Posted on December 28, 2023
16 minute read

Introduction

The `fork()` system call in xv6, and as we have seen it so far in class, copies all of the parent's address space into the child's address space. This is okay if the process's address space is small, but if the parent is large, then this becomes very slow. Also, most often, a lot of the pages that are shared between the parent and the child are never written to. So keeping two copies of the same page that is only being read is very wasteful. Furthermore, copies of the full address space might be wasteful since calls to `fork()` are often followed by calls to `exec()` in the child, which completely replaces the address space of the child. In that case, the copy that we did on `fork()` was very wasteful.

In this lab, we will implement an optimization to `fork()` referred to as **copy-on-write** (COW).

Learning Objectives

At the end of this lab, you should be able to:

- Implement cow forking in the xv6 operating system.
- Implement reference counting as a way to track shared pages before freeing them.
- Manipulate process page tables in the xv6 operating system.

Overview

Unlike our regular `fork()`, a COW `fork()` will defer the allocation and copying of physical memory frame for the child until the copies are really needed, if ever.

COW `fork()` creates just a page table for the child, which PTEs point to the parent's physical frames that are now shared between the parent and the child. However, COW `fork()` marks all PTEs, in both the parent and the child, as **read-only**. In other words, neither the parent nor the child can write to their pages after a `fork()` system call.

When either the parent or the child attempt to write to a page, a page fault occurs and the kernel is called upon to handle the exception. The kernel will detect a COW protected page and will then allocate a frame of physical memory for the faulting process, copy the content of the original page to the new one, map the new frame in the faulty process's page table, and unmaps the old page appropriately. However, when mapped in the page table, the kernel marks the corresponding PTE as writable now, and thus the faulty process can continue execution normally after that.

The COW `fork()` makes freeing physical frame trickier. A frame of memory may be mapped in multiple processes's page tables, and thus cannot be really free'd back to the kernel until the last mapping is remove. We will explore this tricky step in this lab.

Getting the Source Code

To obtain the source code for this lab, follow the instructions below.

1. Commit and push your changes to your current branch

First, make sure all your changes to your current branch are pushed to your repo. Recall that you can use `git branch` to check which branch you are currently on.

Follow the standard `git add`, `git commit`, and `git push` workflow to push your changes to your own private repo.

If at any point, you get permission issues, this most likely means that you are trying to push to the class repo, which you do not have access for. To push to your own `main` branch, you can use:

```
$ git push origin main
```

2. Fetch the changes from our repo

From your Linux terminal, fetch our changes using:

```
$ git fetch upstream
```

Then, make sure you can see this lab's branch, namely `upstream/cow`.

```
$ git branch -a
* clab_solution
main
remotes/origin/clab_solution
remotes/origin/main
remotes/upstream/clab
remotes/upstream/heapmm
remotes/upstream/main
remotes/upstream/buddy
remotes/upstream/cow
```

You might see more branches locally (under `origin`) depending on what you have done, but you should be good if `remotes/upstream/cow` shows up.

3. Get the code in a new branch

Next, let's checkout the `cow` branch and create a new local branch for its solution.

```
$ git checkout -b cow_solution upstream/cow
branch 'cow_solution' set up to track 'upstream/cow'.
Switched to a new branch 'cow_solution'
```

4. Push the changes to your repo

Finally, push your changes to your own repo to make sure the code is there and you can start editing.

```
$ git push --set-upstream origin cow_solution
Enumerating objects: 119, done.
Counting objects: 100% (119/119), done.
Delta compression using up to 56 threads
Compressing objects: 100% (64/64), done.
Writing objects: 100% (111/111), 42.50 KiB | 42.50 MiB/s, done.
Total 111 (delta 42), reused 111 (delta 42), pack-reused 0
remote: Resolving deltas: 100% (42/42), completed with 8 local objects.
remote:
remote: Create a pull request for 'cow_solution' on GitHub by visiting:
remote:      https://github.com/user/csse332-labs-noureddi/pull/new/cow_solution
remote:
To github.com:user/csse332-labs-noureddi.git
 * [new branch]      cow_solution -> cow_solution
branch 'cow_solution' set up to track 'origin/cow_solution'.
```

Step 0: Making Sure Tests Fail

In this first step, we just want to make sure that your code fails the first simple test case of our tests. We have provided you with a set of test cases that tests various `fork()` behavior. The first of those tests, called `simple`, allocates more than half of the available physical memory for one process, and then calls `fork()`. This fork will fail because there is not enough space in the system to create a copy of the parent process, with all of its address space.

To run this, first compile xv6 using `make qemu`, and then run the `cowtest` executable using:

```
$ cowtest
simple: fork() failed
```

Your output should show exactly the output shown above. Once that is done, you are ready to move on to Step 1.

Step 0.5: Identifying Segmentation Faults

The default behavior of xv6 when it encounters a segmentation fault is that it will kill the faulty process. What really happens behind the scenes is that a user trap is encountered when accessing an unmapped page in the page table, and the code in `usertrap` in `kernel/trap.c` is triggered.

Specifically, when a page fault happens, the hardware will trigger an exception. In RISC-V, the hardware will record the cause of the exception in the `scause` register, save the faulty PC in `sepc`, and the value of the pointer that caused the page fault in the `stval` register. Then, the hardware will trap into the kernel to handle the page fault, which will then call on `usertrap` to decipher what happens, since this is an exception that happened from user space.

Take a look at the code in `kernel/trap.c`, specifically at `usertrap()` for a sample of what happens. You can see for example that if the trap is caused by a system call issued from the user, the value of the `scause` register will be 8 as shown in the following snippet of code:

```
if(r_scause() == 8) {
  // system call
  ...
}
```

As it currently stands, xv6 does not have a handler for segmentation faults. Instead, the kernel will kill the process if it is caused by an unhandled trap. In order to figure out what happens when a page fault occurs, let's create one. In the user directory, we have provided you with a file called `user/fault.c` that forces a page fault as follows:

```
int
main(int argc, char **argv)
{
  char *p = (char *)0xdeadbeef;

  /* Try to access an invalid pointer and check what happens. */
  *p = 0xff;
  printf("The ptr %p points to %c", p, *p);

  exit(0);
}
```

To run this program, compile the xv6 kernel using `make qemu` and run the `fault` executable as follows:

```
$ fault
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x0000000000000010 stval=0x00000000deadbeef
```

You can see that for a page fault, the `scause` register contains the value 15 (0xf in hex), and the `stval` register contains the address that caused the fault (`0xdeadbeef` in our case). We can therefore modify the source code for `usertrap()` to print a segmentation fault as follows:

```
else if((which_dev == devintr()) != 0) {
  // ok
} else if(r_scause() == 15) {
  printf("Segmentation fault from process %d at address %p\n", p->pid, r_stval());
  p->killed = 1
} else {
  ...
}
```

Now compiling this code using `make qemu` and running the `fault` executable results in:

```
$ fault
Segmentation fault from process 3 at address 0x00000000deadbeef
$
```

We will return to this handler later on in Step 2.

Step 1: Modifying `fork` Behavior

Understanding the Copying `fork` Behavior

First, take a look at the implementation of the `fork()` system call. You can find the source code in `kernel/proc.c`. The most important snippet of code we care about is the following:

```
// Copy user memory from parent to child.
if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
  freeproc(np);
  release(&np->lock);
  return -1;
}
np->sz = p->sz;
```

This code will copy the parent process (called `p`)'s address space and create a copy of each page in the child process's address space. `uvmcopy` copies all of the mapped pages in the parent's address space and creates corresponding pages in the child's address space, and maps them in the child's page table. You can find the source code for `uvmcopy` in

`kernel/vm.c`.

Specifically, we care about the following loop in `uvmcopy`:

```
for(i = 0; i < sz; i += PGSIZE){
  if((pte = walk(old, i, 0)) == 0)
    panic("uvmcopy: pte should exist");
  if((*pte & PTE_V) == 0)
    panic("uvmcopy: page not present");
  pa = PTE2PA(*pte);
  flags = PTE_FLAGS(*pte);
  if((mem = kalloc()) == 0)
    goto err;
  memmove(mem, (char*)pa, PGSIZE);
  if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
    kfree(mem);
    goto err;
  }
}
```

This code will first walk the parent's page table for each virtual page number in the parent's address space. It does some error checking and then allocates a page in the child's address space using:

```
if((mem = kalloc()) == 0)
  goto err;
```

and then copies the content of the old page into the new page using:

```
memmove(mem, (char*)pa, PGSIZE);
```

Finally, we create a mapping for the new page in the child's page table using:

```
if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
  kfree(mem);
  goto err;
}
```

It is absolutely crucial that you understand how this function works for you to be able to continue on with this lab, so please ask questions about it before you move on.

Implementation: Mapping the Parent's Pages Without Copy

Your first task in this lab is to modify the `uvmcopy` function to create non-writable pages that are shared between the parent and the child. In other words, you will **NOT** be creating new pages for the child, instead all the mappings in the parent's page table will be the same in the child's page table, and all the pages in the shared address space will be marked as non-writable.

The goal of making those pages non-writable is to cause segmentation faults to occur when either the parent or the child try to write to these pages. Then, it is only at that point that we create new pages, copy the content of the old pages, and remap those pages in the corresponding process's page table.

However, this creates a small conundrum. What if a page was originally marked as non-writable? At that point, we do not want the kernel to copy and remap that page in the page table, we would like to keep that page non-writable. Therefore, to achieve a distinction between pages that are non-writable because of the copy-on-write mechanism, and those that are non-writable by default, we will make use of an additional bit in each page table entry (PTE). This bit is called the RSW

bit (where RSW stands for Reserved for SoftWare). Therefore, we will set that bit to 1 whenever a page is marked non-writable because of the cow mechanism. We have already defined that bit for you in the `kernel/riscv.h` definitions as follows:

```
#define PTE_RSW (1L << 5) // reserved for software
```

Task: Change `uvmcopy`

Your task here is to change the behavior of `uvmcopy` to share the pages between the parent process and the child process. In other words, we do not want to create new pages (or frames) for the child that are copies of each other, we want the parent and the child to **share** the same pages, and thus the same mappings.

However, for all those pages (in both parent and child), we would need to unset the Write flag (i.e., disable writing to hose pages from **both** processes). To unset a flag from a page table entry (say `pte`), we would perform a bitwise operation as follows:

```
*pte &= ~PTE_W;
```

Additionally, for those pages that were not originally write-protected, we would like to set the `PTE_RSW` bit. We can do so using `*pte |= PTE_RSW;`. Note that per our discussion above, not all pages should receive this bit.

At the end of this task, the parent's page table and the child's page table should be exact copies of each other, with no page marked as writable, and those pages that were not write-protected now having the *reserved for software* bit.

Testing

At this point, your xv6 kernel will break. Recall that `fork()` is an essential system call in everything the operating system does. And as of now, when we fork a process, all of the pages for both the parent and child are marked as non-writable, which will cause our operating system to break.

When I tried to run my kernel using `make qemu`, I get something like the following:

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
usertrap(): unexpected scause 0x000000000000000c pid=1
            sepc=0x0000000000000394 stval=0x0000000000000394
panic: init exiting
```

Your output might look slightly different, but basically the `init` process should fail since we cannot `fork()` properly. We will take care of that in the next step.

Note that it is okay if your kernel doesn't fail out of the box, you might sometimes need to press `<Enter>` before the kernel breaks; that is perfectly acceptable for this step.

Step 2: Handling Segmentation Faults

It is now time to go back to our page fault handler to make sure that we create new pages and copy then when we encounter a page fault caused by a cow mapping. What we would like to do in this step is to modify the page fault handler that we created in Step 0.5 to handle cow mapping.

Task: Add your custom handler

Your task now is to handle page faults that are due to the copy-on-write mechanism. Before you start writing code, make sure you answer the following question:

> *What is a page considered to be a cow page?*

You need to have a clear answer to the question above before being able to implement this task.

Please note that if you detect a page fault on a non cow page, that you don't need to do anything else, simply kill the process as any other unhandled page.

For those cow pages, now is the time to do the copying. You will need to copy the page, unmap it from the process's page table, and then map it again with the copy with appropriate permission flags.

Some hints

- You will most likely need to walk the process's page table. You do not have to rewrite this function, you can use the `walk` function call from `kernel/vm.c`. To do so, add the following line at the top of your file, right below the `extern char trampoline[] ....` line:

```
extern pte_t *walk(pagetable_t pagetable, uint64 va, int alloc);
```

- Look at the functions in `kernel/vm.c` to checkout ways to manipulate the page table (create new mappings, remove existing one). Make sure you understand what a function does exactly before you use it.

Testing

Implementing this step does not completely solve all of our problems, but let's try to test a few things out so far. The main problem that we have not handled yet is what happens when a process dies, so by definition our tests will fail. We will take care of that in the next step.

However, we have provided you with a partial test that would tell you if your implementation so far is correct. Checkout the code in `user/simplefork.c`. This is very similar to the `simple` test in `cowtest.c`, however it does not call the `wait` system call since we haven't handled yet what happens when a process dies.

To test things so far, compile the xv6 kernel using `make qemu` and run the `simplefork` executable using:

```
$ simplefork
simple: ok
usertrap(): unexpected scause 0x000000000000000c pid=1
            sepc=0x0000000000000fa4 stval=0x0000000000001000
panic: init exiting
```

Note that the test passes but then the kernel panics. This is due to the fact that we do not clean up after ourselves when doing COW. Let's take care of that in the next step.

Step 3: Adding Reference Counting

A big problem arises when we share pages between processes, and that is that of cleaning up allocated memory. Under normal circumstances, when a process dies, it will unmapp and free all of its allocated pages. However, under COW, that page that we are freeing might still be shared with a child process, and that is problematic if the child process still needs access to it.

This is a very similar problem to one that might arise when dealing with garbage collection in many high level languages. For each physical frame in our system, we must keep track of a **reference count** that indicates how many processes are still actively trying to access that frame. As long as the reference count is greater than 0, we must not free that frame. It is only when the reference count reaches 0 (i.e., no further processes need that frame) that we can completely free the frame.

Your job in this step is to implement reference counting for each physical frame in the xv6 system. You will need to modify the code in `kernel/kalloc.c` to keep a reference count for each allocate frame in the system. The design of xv6 makes things easy for us since we already know before hand the maximum number of physical frames that can be allocated in our system. Check out the code in `kernel/memlayout.h`, specifically,

```
// the kernel expects there to be RAM
// for use by the kernel and user pages
// from physical address 0x80000000 to PHYSTOP.
#define KERNBASE 0x80000000L
#define PHYSTOP (KERNBASE + 128*1024*1024)
```

The macro `PHYSTOP` indicates that largest physical address that we can reach. The macro `KERNBASE` is the address where the kernel's memory is allocated. Therefore the total number of frames that can be allocated for the users is `(PHYSTOP - KERNBASE) / PGSIZE`.

Task: Implement reference counting

Your next task is to add reference counting to your cow mechanism implementation. Since the number of frames in xv6 is fixed and will not change, we suggest that you use a simple array to keep track of the reference count for every frame in physical memory.

Whenever a frame is allocated (using `kalloc`), its reference count should be increased. Similary, anytime you share a frame between two or more processes, the refence count for that frame should be increased.

Contrarily, when a frame is free'd (using `kfree`), its reference count must be decremented. **It is only after the reference count for that frame reached 0** that it would be actually free'd and released back to the kernel.

Some hints

1. You will need to modify code `kernel/kalloc.c`.

2. You will need to modify your `uvmcopy` function to make use of reference counting.

   Hint: To expose a function from `kernel/kalloc.c` to your other files, add that function in `kernel/kalloc.c` and then use extern to expose it.

   For example, if I create a function called `void my_function(uint64 pa)` in `kernel/kalloc.c`, then if I need it anywhere else in the kernel, you can use:

   ```
   extern void my_function(uint64);
   ```

3. To find the frame number for a given *physical* address, you can use the following:

   ```
   #define FRINDEX(pa) ((uint64)pa - KERNBASE) / PGSIZE
   ```

4. Since xv6 runs on two processors by default, some synchronization must be done. Since we have not talked about that yet, you can turn that off using `make CPUS=1 qemu`.

   Alternatively, it is a really simple solution, anytime you need to access the reference count array, you must have a lock on your hands. For me, I did the following:

```
acquire(&kmem.lock);
// do stuff with the reference count array.
release(&kmem.lock);
```

Testing

Now we can test things out a bit more. To do so, uncomment the lines in `user/simplefork.c` that are marked with `UNCOMMENT THIS FOR STEP 3` (there is one in the `simpletest()` function and one in `main`). Then, compile the kernel using `make qemu` and run the `simplefork` executable as follows:

```
$ simplefork
simple: ok
simple: ok
$
```

Both simple tests should pass at this point. We can also make more extensive tests using `cowtest`, however not all tests will pass. Specifically, `simple` and `three` will pass, but `file` will fail, something that looks like the following:

```
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: eerrorerrorr:r orrea: readd :  rfaeafdi laedifalielde
d

$
```

Note that your error might look a bit different, but it will fail regardless. We will fix the `filetest` in the next and final step.

Step 4: Modifying `copyout`

The final case we would like to take care of is the case of `copyout` in `kernel/vm.c`. `copyout` is a function that copies bytes of data from the kernel address space to the user's address space. When a process calls `copyout` on a shared cow page, it must first allocate a new page and copy it before it can write to it. However, it will not go through the same page fault mechanism that we implemented in the previous steps since `copyout` is usually called in kernel space, and thus has a different trap handler. To simplify things, we will directly modify `copyout`'s source code to handle cow pages.

To complete this lab, modify the source code for `copyout` to handle cow pages before writing to them. Your code will look very much similar to the code you used in step 2. In a nutshell, the source code of `copyout` is copying data from the kernel space to the user space page by page. For each page to copy, the code will first find `pa0`, which is the address of the physical frame to write to, and then calls `memmove` to copy the data from the kernel frames to the user's frame. Your code must reside after the line `va0 = PGROUNDDOWN(dstva);` and it should set the appropriate value for `pa0`.

> *Hint: In the case where the virtual page to copy to is not a cow page, you simply need to set `pa0 = walkaddr(pagetable, va0);`.*

> *Hint: You will need to fail `copyout` if the user runs out of space. In other words, if `va0` is greater than `MAXVA`, then you need to return -1.*

Testing

Now, all `cowtest` tests should pass:

```
$ cowtest
simple: ok
simple: ok
three: ok
three: ok
three: ok
file: ok
ALL COW TESTS PASSED
```

Running the full grading script

Once you are done implementing all the above programs, run the grading script using

```
$ make grade
```

from your Linux terminal (not your xv6 terminal window).

Submitting Your Code

From the Linux terminal, issue the command (make sure you are in the `xv6-riscv` directory in your repository):

```
./create_submission.sh <username>
```

and replace `<username>` with your RHIT username (without the `<` and `>`). For example, for me, that would be:

```
./create_submission.sh noureddi
```

If you get a message saying that you don't have permission to run `./create_submission.sh`, then issue the following command first

```
chmod +x ./create_submission.sh
```

Here's the output as it shows up on my end:

```
Cleaning up xv6 directory...
Process started: writing temporaries to /tmp/dab9bfedf8d508c2a1c3f1c95e6ba1fc.txt
Found the following modified files:
./user/rhmalloc.c
Creating the submission zip file.
  adding: user/rhmalloc.c (deflated 54%)
Done...
################################################################
        submission_noureddi.zip has been created.
   Please submit THIS FILE AND THIS FILE ONLY to Gradescope.
################################################################
```

This will generate a single file called `submission-username.zip` (for me, it would be `submission-noureddi.zip`). That is all you need to upload to Gradescope.

Submission Checklist

- [ ] My code compiles and generates the right executables.
- [ ] I ran `make grade` to double check the test cases for all of my code.
- [ ] I ran the submission script to generate my `zip` file.
- [ ] I submitted the `zip` file to Gradescope.

If you notice any typos or inaccuracies, please open a GitHub issue on this repository.