1 Overview

In this assignment, you will learn how to implement copy-on-write for xv6. To get started, download a new copy of the xv6 source code from here. Do NOT use the source code of the previous project. Words of wisdom: first, please start early! This project is more challenging than the previous ones. Second, please make minimal changes to xv6; you do not want to make it hard for us to grade!

2 Copy-on-write for Xv6 (100%)

In UNIX, the fork syscall creates an exact copy of the parent process. Xv6 implements it by simply making a copy of the parent’s memory and other resources (fork in proc.c). Specifically, it first allocates a process control block for the child process (allocproc), clones the parent’s memory with copyuvm, and duplicates the files with filedup etc. In this project, you will replace copyuvm with copyuvm_cow. Copyuvm takes two parameters: pgdir is the parent’s page directory, sz is the size of the user-mode memory (proc->sz). Copyuvm first maps the kernel to the child process (setupkvm). The return value of setupkvm is the page directory for the child process. For each user page, copyuvm first reads the parent’s page table entry (walkpgdir), which consists of the address of the physical frame and the protection bits. It then allocates a new page (kalloc) and copies the parent’s memory to this page (memmove). Finally, it maps the new page to the child process (mappages). Note that page table uses physical address while the kernel can only access memory using virtual address. You can use p2v and v2p to convert them.

In this project, you will replace copyuvm with copyuvm_cow. Similar to copyuvm, copyuvm_cow first maps the kernel to the child. However, it does not clone the user-mode memory. Instead, it maps the parent’s physical frame read-only in both the parent and the child. In other words, it clears the PTE_W flag from the corresponding page table entries of the parent and the child. Moreover, the function needs to flush the TLB because the active page table is changed. Insert a call to flush_tlb_all before return. Hint, use code similar to the following to clear a flag:

```
flag &= ~PTE_W;
```

Replace the call to copyuvm in fork (proc.c) with copyuvm_cow. However, this is not over yet, read on.

Copyuvm_cow maps the parent’s memory read-only in both the parent and the child. When either of them tries to write to it, the CPU will raise a page fault exception (the faulting address can be retrieved from the control register cr2.) In the page fault handler, you should clone the old page into a new page, and map this new page writable in the current process (proc). Before getting into the details, copy the code for the following two functions (read_cr2 and flush_tlb_all) into trapasm.S:

```
.globl read_cr2
read_cr2:
    movl %cr2, %eax
```

1Xv6 uses the two-level page table of x86. The first level is called page directory, and the second level is called page table. X86 also supports the three-level page table.
Copy the following function declaration to defs.h:

```c
pde_t* copyuvm_cow(pde_t*, uint);
void handle_pgflt (void);
uint read_cr2 (void);
void flush_tlb_all (void);
```

In xv6, exceptions are handled in the trap function (trap.c). You need to call the page fault handler when there is a page fault (T_PGFLT). Add the following code to trap after the case for syscall (T_SYSCALL).

```c
if (tf->trapno == T_PGFLT) {
    proc->tf = tf;
    handle_pgflt ();
    return;
}
```

Your task is to write the handler for the page faults handle_pgflt (in addition to copyuvm_cow, add both functions to vm.c). Handle_pgflt first reads the faulting address from cr2 (read_cr2). It then gets the physical address of the currently mapped physical frame from the page table (walkpgdir) and clones the frame into a new page. Finally, it maps this new page into the current process writable and flushes the TLB (flush_tlb_all). Roughly speaking, the responsibilities of copyuvm are now split into copyuvm_cow and handle_pgflt.

However, there is one more complication. Assume a physical page $p$ is shared by process $A$ and $B$. When process $A$ exits before process $B$ does, xv6 will reclaim the shared memory to the kernel and may use it for other purposes. This will lead to problems in process $B$ if the memory is subsequently modified. A solution is to keep track of how many processes are using the page and only release it to the kernel if no other process is using it. However, a quick and dirty solution is not to free the page if it is ready-only. Change the deallocuvm function for this purpose. Note that this solution leaks memory.

If everything works, your COW-ified xv6 should behave exactly like the original system externally. Particularly, it should be able to execute commands and pass the forkttest test.

# 3 Deliverables

Submit in the blackboard your modified source code of xv6 as a gzip compressed tarball. Include in the tarball all files necessary for a successful build! The name of your attachment should be

```
cop4610-project3-yournames.tar.gz
```

with yournames replaced by your CS account name. Your submission will be graded by compiling and running it and reviewing the source code.

- Please make sure your source code can compile. Absolutely no credit if it does not compile.

- Please don’t include the binary files. Do a `make clean` before submission. You’ll make grading harder for us if you do.
• Please don’t leave out any files! You’ll make grading harder for us if you do.

• Please don’t modify any files you don’t need to! You’ll make grading harder for us if you do.

• Please don’t send us the meta-information from your revision control system! You’ll make grading harder for us.

And if we have a hard time grading, you’ll have a hard time getting points!

4 Useful References

- Xv6 book/commentary, Chapter 2
- Xv6 Homepage