Assignment 2- uaccess

12212320 谭政

**Checkpoint. 1**

修改xv6 的用户与内核的页表模型，

使得：

• 用户页表包含整个内核页表(kernel\_pagetable)。（内核页表中的 PTE 均没有 PTE\_U 权限位）

• 内核线程使用用户页表（此时它应该包含内核页表），而不是内核页表。 同时，你需要修改Context Switch 和 Trampoline：

• 在使用swtch 进行Context Switch 时，切换到内核线程各自的页表 (p->mm->pgt)。

• 通过trampoline 在用户态与内核态之间切换时，保留用户页表，不需要切换到内核页表kernel\_satp。

• 对于没有用户空间的scheduler，直接使用 kernel\_pagetable 内核页表即可。

修改：

void mm\_copy\_kpgt(struct mm \*mm) {

    assert(holding(&mm->lock));

    for (uint64 i = 0x100; i < 512; i++) {

        pte\_t \*kpte = (pte\_t \*)&kernel\_pagetable[i];

        pte\_t \*upte = (pte\_t \*)&mm->pgt[i];

        // Assignment 2: Your code here

        if (\*kpte & PTE\_V) {

            \*upte = \*kpte & ~PTE\_U;

        }

}

static int access\_ok(struct mm\* mm, uint64 addr, uint64 len) {

    // Assignment 2: Your code here

    if (addr >= USER\_TOP)

        return 0;

    if (len > 0 && addr + len < addr)  // 溢出

        return 0;

    if (addr + len > USER\_TOP)

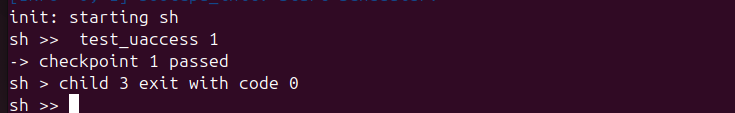
        return 0;

    return 1;

}

}

运行结果：



**Checkpoint. 2**

阅读特权级手册4.1.1.2 Memory Privilege in sstatus Register 中关于 SUM 标志位的解释，理解为什么这里 会遇到Page Fault，并且根据文档中的提示解决该问题。

因为内核在用户态下尝试直接访问用户空间内存，而没有正确设置权限位。通过设置 sstatus 寄存器的 SUM 位，允许内核直接访问用户空间内存。在拷贝操作完成后，清除 SUM 位以恢复默认行为

修改：

static void begin\_user\_access() {

    // Assignment 2: Your code here

    w\_sstatus(r\_sstatus() | SSTATUS\_SUM);

}

static void end\_user\_access() {

    // Assignment 2: Your code here

    w\_sstatus(r\_sstatus() & ~SSTATUS\_SUM);

}

// Copy from kernel to user.

// Copy len bytes from src to virtual address dstva in a given page table.

// Return 0 on success, -1 on error.

int copy\_to\_user(struct mm \*mm, uint64 \_\_user dstva, char \*src, uint64 len) {

    begin\_user\_access();

    memmove((void \*)dstva, src, len);

    end\_user\_access();

    return 0;

}

// Copy from user to kernel.

// Copy len bytes to dst from virtual address srcva in a given page table.

// Return 0 on success, -1 on error.

int copy\_from\_user(struct mm \*mm, char \*dst, uint64 \_\_user srcva, uint64 len) {

    if (!access\_ok(mm, srcva, len)) {

        return -1;

    }

    begin\_user\_access();

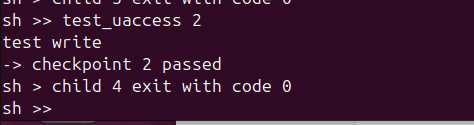
    memmove(dst, (void \*)srcva, len);

    end\_user\_access();

    return 0;

}

运行结果：



**Checkpoint. 3：**

在 checkpoint3 中，任务是确保内核不会访问非法的用户空间地址。access\_ok函数通过检查地址范围来确保用户地址是合法的。

修改：

static int access\_ok(struct mm\* mm, uint64 addr, uint64 len) {

    // Assignment 2: Your code here

    if (addr >= USER\_TOP)

        return 0;

    if (len > 0 && addr + len < addr)  // 溢出

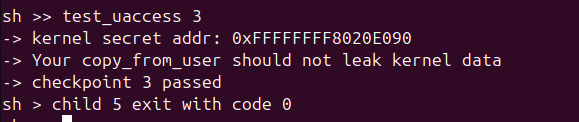
        return 0;

    if (addr + len > USER\_TOP)

        return 0;

    return 1;

}

运行截图：

**Checkpoint. 4**

修改：if (cause & SCAUSE\_INTERRUPT) {

        // correctness checking:

        if (mycpu()->inkernel\_trap > 1) {

            // should never have nested interrupt

            print\_sysregs(true);

            print\_ktrapframe(ktf);

            panic("nested kerneltrap");

        }

        if (panicked) {

            panic("other CPU has panicked");

        }

        // handle interrupt

        if (handle\_intr() == 0) {

            errorf("unhandled interrupt: %d", cause);

            goto kernel\_panic;

        }

    } else {

        if (exception\_code == 12 || exception\_code == 13 || exception\_code == 15) {

            struct proc \*p = curr\_proc();

            if (holding(&p->mm->lock)) {

                intr\_off();

                release(&p->mm->lock);

            }

            p->killed = 1;

            mycpu()->inkernel\_trap--;

            exit(-9);

            return;

        }

        // kernel exception, unexpected.

        goto kernel\_panic;

    }

**运行结果：**

