Assignment 2 - uaccess

2025 年 4 月 1 日

1 背景

内核如果想访问用户的地址空间，一般我们不让它直接对用户地址进行访存操作，如指针解引用和 memcpy；而是需要借助一些专门用于访问用户空间的函数进行访存操作，如 `copy_from_user` 和 `copy_to_user`，我们称这种函数为 uaccess primitive (用户空间访问原语)。

在进行系统调用时，用户程序会按需传递用户地址，并令内核对其进行读取或写入。如 `read(2)` 和 `write(2)` 系统调用中，用户会传入一个缓冲区 `buf` 和其长度 `count`，表示内核允许读取、写入最多 `count` 字节的数据。

```
ssize_t read(int fd, void buf[.count], size_t count);
ssize_t write(int fd, const void buf[.count], size_t count);
```

在 *console.c* 中，我们对 `sys_write` 系统调用的处理如下（简化版本）：我们首先在内核空间申请一个长度为 `len` 的缓冲区，并使用 `copy_from_user` 从用户态拷贝所要打印的数据，最后依次打印到串口上。

```
int64 user_console_write(uint64 __user buf, int64 len) {
    struct proc *p = curr_proc();
    struct mm *mm = p->mm;

    char *kbuf = kmalloc(len);
    copy_from_user(mm, kbuf, buf, len);

    for (int64 i = 0; i < len; i++) {
        consputc(kbuf[i]);
    }

    return len;
}
```

为什么我们需要 uaccess 原语来操作用户内存，而不能直接对用户指针进行解引用操作：

- 用户传入的指针是不可信的，使用原语可以强制每次访问时检查指针是否合法。
- 不对用户指针进行检查而直接进行解引用是非常危险的，用户可能故意传入一个合法的内核地址而不是用户地址。这种情况下，如果直接进行解引用，会导致对内核内存的访问。使用原语通过强制检查规避掉这样的漏洞。

Build commit hash: 49adc62

- 如果在内核模式可以直接进行用户内存访问，编译器可能会对内核代码进行访存优化，导致 Double Fetch 等安全问题。如果你感兴趣，推荐观看：【Windows 内核大揭秘】编译器优化如何引发潜在危机？

2 问题描述

在当前的 xv6 内核中，内核页表与用户页表是互相隔离的，内核的页表 (`kernel_pagetable`) 不包含某个用户进程的页表项 (`p->mm->pgt`)；反之，用户进程的页表也不包含任何内核的页表项。在 `copy_from_user` 里面，我们需要先手动将用户的虚拟地址转化为物理地址（即 `walkaddr` 方法）。由于用户态下的内存均是由 `kalloc` 模块中的 `kallocpage` 方法分配的，这些物理地址位于 Kernel Direct Mapping 的物理内存区域内，所以我们可以将其转换内核虚拟地址 (KVA) 进行访问：

```
// 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)
{
    uint64 n, va0, pa0;

    while (len > 0) {
        va0 = PGROUNDDOWN(srcva);
        pa0 = walkaddr(mm, va0);
        if (pa0 == 0)
            return -1;
        n = PGSIZE - (srcva - va0);
        if (n > len)
            n = len;
        memmove(dst, (void *)(PA_TO_KVA(pa0) + (srcva - va0)), n);

        len -= n;
        dst += n;
        srcva = va0 + PGSIZE;
    }
    return 0;
}
```

这显然是不高效的，地址翻译应该由 CPU 的 MMU 执行，并且翻译结果可以被 TLB 缓存；而目前的 `copy_from_user` 实现在每次调用时都至少要进行一次 walk 页表。

在本次作业中，请你在给定的代码基础上修改，改进 uaccess 原语，使其能够使用 CPU 的页表机制进行地址翻译，同时保证 uaccess 原语的安全性，对用户传入的地址进行检查并阻止错误的访存。

本次作业一共 4 分，有 4 个 Checkpoint 和一份报告。每个 Checkpoint 各 1 分，报告不占分，但是强制要求写。你的报告应该符合以下要求：

- PDF 文件格式的报告。
- 每个 checkpoint 运行成功的截图。
- 对于指引中提出的思考问题，你不需要在报告中回答。
- 记录完成这个作业一共花了多少时间。
- （可选）描述你在完成这个作业时遇到的困难，或者描述你认为本次作业还需要哪些指引。
- （可选）你可以在报告中分享你完成这个作业的思路。
- 越简洁越好。

要求 你应该使用 `make run` 启动单核心的 xv6。本次作业我们提供了 `usertest`，在启动内核后，你应该能在 `applist:` 提示符后看到一项 `test_uaccess`。你需要通过 `test_uaccess 1-4`，能稳定通过即视为通过 Checkpoint。

注：Checkpoint 2-4 需要在已经通过 Checkpoint 1 的情况下才记分。

3 指引步骤

3.1 Checkpoint. 1

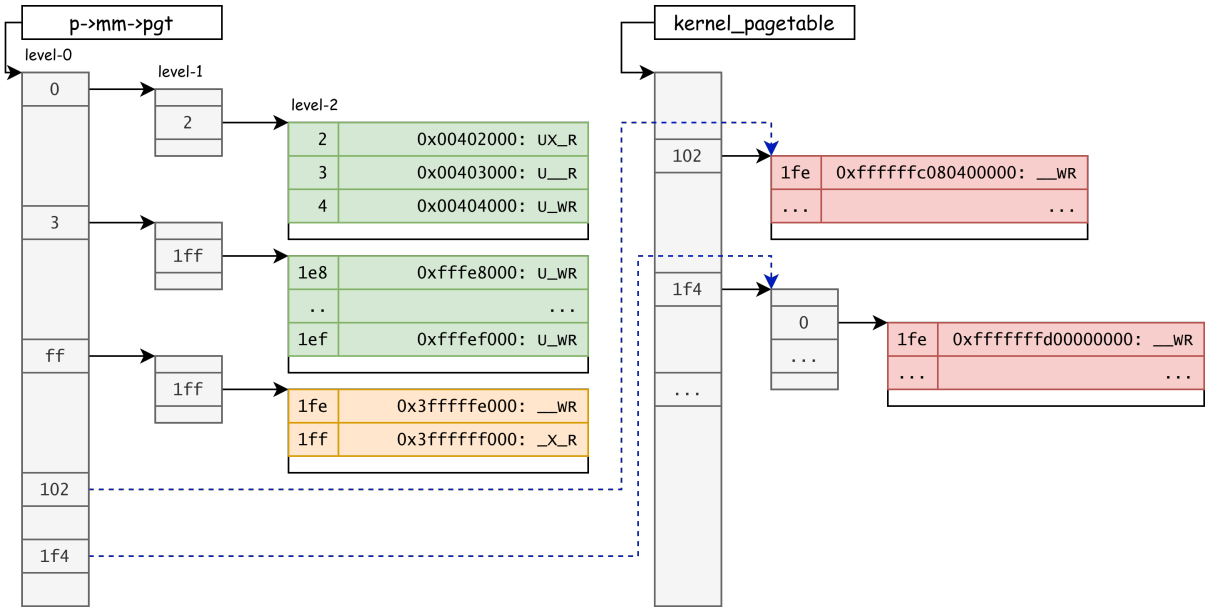
修改 xv6 的用户与内核的页表模型，使得：

- 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。

Hint. 内核页表在内核启动完成后是不变的。你可以直接复制内核页表中第一级页表中的 PTE 到用户页表中，使其直接指向内核页表中的第二级页表。



Hint. 在修改 `satp` 后，使用 `sfence.vma zero, zero` 清除 TLB 中的所有缓存。

Hint. scheduler 永远不会是第一个被 `swtch` 的对象。

Checkpoint Passed. 你能使用 `make run` 和 `make runsmp` 正常启动内核。在 sh 中执行 `test_uaccess 1`，你应该会看到输出：

```
sh >> test_uaccess 1
-> checkpoint 1 passed
```

3.2 Checkpoint. 2

将 *uaccess.c* 中的 `copy_to_user` 和 `copy_from_user` 函数中的 `while(...)` 循环全部删除，换为 `memmove` 直接进行内存拷贝。（你不需要修改 `copystr_from_user` 函数，我们的 testcase 不会覆盖它）如下所示：

```
int copy_to_user(struct mm* mm, uint64 __user dstva, char *src, uint64 len) {
    memmove((void *)dstva, src, len);
    return 0;
}
int copy_from_user(struct mm* mm, char *dst, uint64 __user srcva, uint64 len)
{
    memmove(dst, (void *)srcva, len);
    return 0;
}
```

使用 `make run` 启动内核。你应该会遇到内核异常，原因为 15 - Store/AMO page fault。该异常的 backtrace 如下所示（你可以参照后续的 Debug 章节了解如何对 Trapframe 进行 backtrace）：

```
(qemu) gef   bt
#0  0xffffffff802065a0 in memmove (dst=dst@entry=0xfffffffe00001ee0, src=0x404028, n=n@entry=0x12) at os/
string.c:36
#1  0xffffffff80207f70 in copy_from_user (mm=mm@entry=0xffffffffd000fffc8, dst=dst@entry=0xfffffffe00001ee0
"\310\377\017", srcva=<optimized out>,
    len=len@entry=0x12) at os/uaccess.c:39
#2  0xffffffff8020082c in user_console_write (buf=<optimized out>, len=0x12) at os/console.c:178
#3  0xffffffff80207134 in sys_write (fd=<optimized out>, va=<optimized out>, len=<optimized out>) at os/
syscall.c:142
#4  syscall () at os/syscall.c:186
#5  0xffffffff80207d44 in usertrap () at os/trap.c:142
#6  0x00000000004021d8 in ?? ()
```

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。

Checkpoint Passed. 通过 `test_uaccess 2`。

3.3 Checkpoint. 3

在 sh 中执行 `test_uaccess 3`，其源代码位于 *user/src/test_uaccess.c*，`ktest` 部分内核代码位于 os/ktest/ktest_syscall.c。`test_uaccess 3` 会在调用 `write` 时故意传入一个内核地址，而不是用户地址。

修改 uaccess.c 中的代码，使用 `access_ok` 检查带有 `__user` 标签的地址是否真的是属于用户空间地址的区域。注意 Trapframe 和 Trampoline 不是用户可以访问的地址。**不要**通过页表（`p->mm->pgt`）检查来判断地址是否合法。你不需要判断地址是否是一个被映射的虚拟地址。

Hint. 在我们的操作系统设计中,用户地址处于总 512 GiB 地址空间的低 256 GiB 内 (`0x0000_0000_0000_0000` 到 `0x0000_003f_ffff_ffff`)，而内核地址处于高地址内。你可以通过比较地址的大小来判断地址是否为一个合法的用户空间地址。

Checkpoint Passed. 通过 `test_uaccess 3`。

3.4 Checkpoint. 4

在 sh 中执行 `test_uaccess 4`，你将遇到 Kernel Panic，并且 `stval` 的值是一个未被映射的虚拟地址 0x8000_0000。

修改内核代码，使得内核如果在 uaccess 过程中的发生**位于用户地址**的 Page Fault，使用 `exit(-9)` 退出进程而不是造成 Kernel Panic。

Hint. 用户态通过 syscall 进入内核态时，`stvec` 会被改为 `kernel_trap`。而在原本的设计中，`kernel_trap` 从不应该处理内核异常，而是直接 Kernel Panic 终止内核。

```
    if (cause & SCAUSE_INTERRUPT) {
        // handle interrupt
    } else {
        // kernel exception, unexpected.
        goto kernel_panic;
    }
// ...
kernel_panic:
    errorf("============ Kernel Panic ============");
    print_sysregs(true);
    print_ktrapframe(ktf);
    panic("kernel panic");
```

Hint. 在调用 uaccess 系列 copy_from_user 等函数时，我们会持有 p->mm->lock，并在函数返回后释放锁。而 Page Fault 异常会打断正常的函数调用流程。流程如下：

```
// enter syscall (write), then user_console_write
acquire(&p->mm->lock);
copy_from_user(p->mm, dst, src, len);
{
    // copy_from_user function body:
    memmove(mm, dst, src, len);
    {
        // memmove function body:
        while (n-- > 0)
            *d++ = *s++;    // <-- Page Fault Here.
    }
}
release(&p->mm->lock);
// return syscall
```

Checkpoint Passed. 通过 test_uaccess 4。你可能会无法通过 proctest，但是本次作业我们不需要关心该测试集。

4 提交

将你的报告放到与该 PDF 文件的同目录下，运行 make handin，这会生成一个 handin.zip 压缩包。上传该文件到 Blackboard 即可.

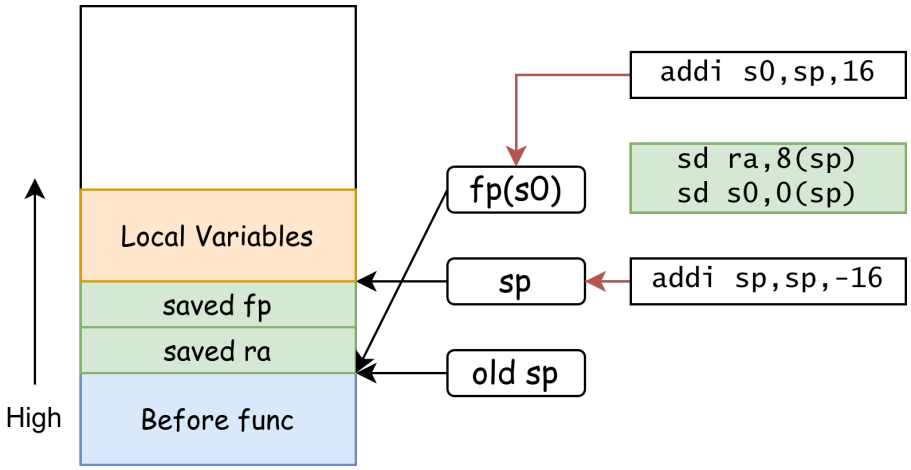
5 Debug

5.1 Stack Layout

我们在诸多计算机课程上都学习过栈 (stack) 的结构。函数在调用时会在栈上构造栈帧 (Stack Frame)。函数入口 (prologue) 会将函数的返回地址 `ra` 和当前的 frame pointer 值压入栈，并将当前 stack pointer 赋值给 frame pointer 寄存器。函数出口 (epilogue) 会从栈上恢复 `ra` 和进入函数前的 `fp`。

在 RISC-V 平台上，栈指针是 `sp`，Frame Pointer 为 `fp`（也是 `s0`）

```
some_func:
    addi    sp,sp,-16
    sd      ra,8(sp)
    sd      s0,0(sp)
    addi    s0,sp,16
    # function body
    ld      ra,8(sp)
    ld      s0,0(sp)
    addi    sp,sp,16
    jr      ra
```



5.2 Backtrace 栈回溯

我们注意到，fp 所指向的区域中包含之前的 fp 的值，所以通过当前的 fp 我们即可找到所有过去的 Stack Frame！这就是 gdb 中 `backtrace` (`bt`) 命令的原理。

但是，Trap Handler 中构造的 ktrapframe 并没有完全按照这种设计。如果我们要在 Kernel Trap 的上下文中对发生 Trap 的原因进行 backtrace，我们则需要做出一点小小的改动。

运行 `make debug` 和 `gdb-multiarch`，进入 gef 中。我们在 `kernel_trap` 中的 `kernel_panic` 标签处打断点：`b kernel_trap:kernel_panic`。使用 `continue` 继续执行。

在断点命中时，使用 `bt` 指令进行栈回溯：

```
(qemu) gef   bt
#0  kernel_trap (ktf=0xffffffe00001de0) at os/trap.c:90
#1  0xffffffff802000b4 in kernel_trap_entry () at os/entry.S:55
Backtrace stopped: frame did not save the PC
```

将 fp 寄存器改为 `ktf` 结构体中保存的 fp：`set $s0 = ktf->s0`。

然后，我们即可在中断帧的上下文进行 backtrace 了。

```
(qemu) gef   bt
#0  kernel_trap (ktf=0xffffffe00001de0) at os/trap.c:90
#1  0xffffffff802000b4 in kernel_trap_entry () at os/entry.S:55
Backtrace stopped: frame did not save the PC
(qemu) gef   set $s0 = ktf->s0
(qemu) gef   bt
#0  kernel_trap (ktf=0xffffffe00001de0) at os/trap.c:90
#1  0xffffffff802084e8 in copy_from_user (mm=mm@entry=0xffffffd000fffc8, dst=dst@entry=0x0,
    srcva=srcva@entry=0x404028, len=len@entry=0x0) at os/uaccess.c:36
#2  0xffffffff8020084c in user_console_write (buf=0x404028, len=0x0) at os/console.c:187
#3  0xffffffff802076e8 in sys_write (fd=<optimized out>, va=<optimized out>, len=<optimized out>) at os/syscall
    .c:180
#4  syscall () at os/syscall.c:227
#5  0xffffffff8020831c in usertrap () at os/trap.c:183
#6  0x0000000000402184 in ?? ()
```

额外的，我们可以使用 `bt -full` 同时打印保存在栈上的局部变量。

```
(qemu) gef   bt -full
#0  kernel_trap (ktf=0xffffffe00001de0) at os/trap.c:90
    cause = 0xd
    exception_code = 0xd
    __func__ = "kernel_trap"
#1  0xffffffff802084e8 in copy_from_user (mm=mm@entry=0xffffffd000fffc8, dst=dst@entry=0x0,
srcva=srcva@entry=0x404028, len=len@entry=0x0) at os/uaccess.c:36
No locals.
#2  0xffffffff8020084c in user_console_write (buf=0x404028, len=0x0) at os/console.c:187
    ret = <optimized out>
    p = 0xffffffd0201ef10
    mm = 0xffffffd000fffc8
    kbuf = 0x0
#3  0xffffffff802076e8 in sys_write (fd=<optimized out>, va=<optimized out>, len=<optimized out>) at os/syscall
    .c:180
No locals.
#4  syscall () at os/syscall.c:227
    trapframe = 0xffffffc080d05000
    id = 0x17
    ret = <optimized out>
    args = {0x1, 0x404028, 0x12, 0xd, 0xa, 0xa}
    __func__ = "syscall"
#5  0xffffffff8020831c in usertrap () at os/trap.c:183
    which_dev = 0x0
    p = 0xffffffd0201ef10
    trapframe = 0xffffffc080d05000
    cause = <optimized out>
#6  0x0000000000402184 in ?? ()
```