Lab03: Page Tables(关于页表)

Print a Page Table (Easy)

1) 实验目的

本实验的目的是在 xv6 操作系统中实现一个函数 `vmprint`，用于打印页表的内容。这将有助于理解 RISC-V 页表的结构，并为未来的调试提供帮助。

2) 实验步骤

1. **定义 `vmprint` 函数**:
   - 在 `kernel/vm.c` 文件中实现 `vmprint` 函数。
2. **声明 `vmprint` 函数原型**:
   - 在 `kernel/defs.h` 中添加 `vmprint` 函数的原型：

```
void vmprint(pagetable_t pagetable);
```

3. **实现 `vmprint` 函数**:
   - 在 `kernel/vm.c` 文件中添加以下代码：

```
void
vmprint(pagetable_t pagetable)
{
    printf("page table %p\n", pagetable);
    vmprint_rec(pagetable, 0);
}

void
vmprint_rec(pagetable_t pagetable, int level)
{
    for (int i = 0; i < 512; i++) {
        pte_t pte = pagetable[i];
        if (pte & PTE_V) {
            uint64 pa = PTE2PA(pte);
            for (int j = 0; j < level; j++) {
                printf("..");
            }
            printf("%d: pte %p pa %p\n", i, pte, pa);
            if ((pte & (PTE_R | PTE_W | PTE_X)) == 0) {
                pagetable_t next_level = (pagetable_t)pa;
                vmprint_rec(next_level, level + 1);
            }
        }
```

```
    }
}
```

4. **修改 `exec` 函数**：
    - 在 `kernel/exec.c` 中，确保在返回前打印出第一个进程的页表：

```
if (p->pid == 1) {
    vmprint(p->pagetable);
}
```

5. **编译和运行**：
    - 使用 `make qemu` 重新编译并运行 xv6，检查输出是否符合预期格式。

3) 实验中遇到的困难和解决办法

1. **递归打印页表**：
    - **困难描述**：递归打印页表时，容易陷入无限递归或打印无效 PTE。
    - **解决办法**：确保仅在 PTE 有效时递归，并且正确处理页表层级关系。
2. **正确解析 PTE**：
    - **困难描述**：解析 PTE 时，可能会错误地提取物理地址或 PTE 属性。
    - **解决办法**：使用 `kernel/riscv.h` 中的宏来解析 PTE，并仔细检查每个字段的含义。

4) 实验心得

通过本次实验，我深入理解了 RISC-V 页表的结构和工作原理。实现 `vmprint` 函数不仅帮助我熟悉了页表的遍历和递归处理，还让我学会了如何利用打印输出调试复杂的数据结构。实验过程中遇到的问题让我更加熟悉了 PTE 的解析和页表的层级关系。在调试过程中，我学会了如何通过逐步检查输出来排查问题，这对后续的实验和项目开发具有重要意义。整体上，这次实验不仅巩固了我对操作系统课程理论知识的理解，也提升了我在实际操作系统开发中的实践能力。

A Kernel Page Table Per Process (Hard)

1) 实验目的

本实验的目的是为每个进程在内核中维护自己的内核页表，并在进程切换时切换内核页表。这将使得内核能够直接解引用用户指针，而无需先将其转换为物理地址。

2) 实验步骤

1. **修改 `struct proc`**：
    - 在 `kernel/proc.h` 中添加一个字段，用于存储每个进程的内核页表：

```
struct proc {
  // existing fields...
  pagetable_t kpagetable;  // Kernel page table for this process
};
```

2. **初始化进程的内核页表**：

   - 在 `allocproc` 中为新进程创建内核页表：

```
pagetable_t
proc_kvminit(void)
{
    pagetable_t pagetable;

    // Allocate root page-table
    pagetable = (pagetable_t) kalloc();
    if(pagetable == 0)
        return 0;
    memset(pagetable, 0, PGSIZE);

    // Copy kernel mappings
    if(kvmmap(pagetable, UART0, UART0, PGSIZE, PTE_R | PTE_W) < 0)
        goto bad;
    if(kvmmap(pagetable, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W) < 0)
        goto bad;
    if(kvmmap(pagetable, CLINT, CLINT, 0x10000, PTE_R | PTE_W) < 0)
        goto bad;
    if(kvmmap(pagetable, PLIC, PLIC, 0x400000, PTE_R | PTE_W) < 0)
        goto bad;
    if(kvmmap(pagetable, KERNBASE, KERNBASE, (uint64)etext - KERNBASE,
PTE_R | PTE_X) < 0)
        goto bad;
    if(kvmmap(pagetable, (uint64)etext, (uint64)etext, PHYSTOP -
(uint64)etext, PTE_R | PTE_W) < 0)
        goto bad;

    return pagetable;

  bad:
    proc_kvmunmap(pagetable, 0, PGSIZE, 1);
    return 0;
}
```

3. **分配和释放内核页表**：

   - 在 `allocproc` 中调用 `proc_kvminit` 为新进程分配内核页表：

```
p->kpagetable = proc_kvminit();
if (p->kpagetable == 0) {
```

```
    freeproc(p);
    release(&p->lock);
    return 0;
}
```

- 在 `freeproc` 中释放进程的内核页表：

```
if (p->kpagetable) {
    proc_kvmunmap(p->kpagetable, 0, PGSIZE, 1);
    kfree((void*)p->kpagetable);
    p->kpagetable = 0;
}
```

4. **修改调度器**：

- 在 `scheduler` 函数中加载进程的内核页表到 SATP 寄存器中：

```
void
scheduler(void)
{
    struct proc *p;
    for(;;){
        // Avoid deadlock by ensuring that devices can interrupt.
        intr_on();

        for(p = proc; p < &proc[NPROC]; p++){
            acquire(&p->lock);
            if(p->state == RUNNABLE){
                // Switch to chosen process. It is the process's job
                // to release its lock and then reacquire it
                // before jumping back to us.
                p->state = RUNNING;

                w_satp(MAKE_SATP(p->kpagetable));
                sfence_vma();

                swtch(&c->scheduler, &p->context);

                w_satp(MAKE_SATP(kernel_pagetable));
                sfence_vma();

                // Process is done running for now.
                // It should have changed its p->state before coming
back.
            }
            release(&p->lock);
        }
    }
}
```

3) 实验中遇到的困难和解决办法

1. **页表映射问题**：
   - **困难描述**：初始化页表时可能会遇到映射错误或缺失映射，导致页表不完整。
   - **解决办法**：仔细检查页表映射代码，确保所有必须的内核地址都正确映射到新的页表中。
2. **SATP 切换问题**：
   - **困难描述**：在进程切换时，未正确加载新的页表到 SATP 寄存器，导致页表缺失或无效。
   - **解决办法**：在 `scheduler` 函数中正确调用 `w_satp` 和 `sfence_vma`，确保页表切换后立即生效。
3. **调试和验证**：
   - **困难描述**：难以确定页表是否正确工作，特别是在复杂的内存访问场景下。
   - **解决办法**：使用 `vmprint` 函数打印页表内容，检查输出是否符合预期；并通过运行 `usertests` 验证页表功能。

4) 实验心得

通过本次实验，我深入理解了操作系统中页表的管理和切换机制。为每个进程维护独立的内核页表，不仅提高了系统的安全性和隔离性，也为后续实验中内核直接解引用用户指针打下了基础。实验过程中遇到的映射问题和页表切换问题让我更加熟悉了页表的初始化和管理。在调试过程中，我学会了如何通过打印页表内容来排查问题，这对后续的实验和项目开发具有重要意义。整体上，这次实验不仅巩固了我对操作系统课程理论知识的理解，也提升了我在实际操作系统开发中的实践能力。