lab4实验报告

思考题

Thinking 4.1

- 使用move k0,sp指令，将用户栈指针保存到k0
- 可以
- 在do_syscall函数中，我们拿到了函数，并且总内核栈中取出了参数
- 更改：
    - 将系统调用的返回值保存
    - 将栈指针保存在k0寄存器，返回的时候又重新写回

Thinking 4.2

因为使用函数的人可能传递非法的envid，如果没有这步判断可能会返回一个非法的进程地址。

Thinking 4.3

在envid2env()的实现中，如果envid是0，则会直接返回curenv，说明0这个数有特殊含义，所以mkenvid不会返回0

Thinking 4.4

C

Thinking 4.5

应该映射PTE_V位有效的页面

Thinking 4.6

1. `vpt` 和 `vpd` 的作用及使用方法

作用

- `vpt`（**Virtual Page Table**）
  指向 **当前进程的页表（Page Table）的虚拟地址**，用于直接访问页表项（PTE, Page Table Entry）。
- `vpd`（**Virtual Page Directory**）
  指向 **当前进程的页目录（Page Directory）的虚拟地址**，用于直接访问页目录项（PDE, Page Directory Entry）。

使用方法

- **访问页目录项（PDE）**：

  c

  复制

下载

```
pde_t *pde = &vpd[PDX(va)];  // PDX(va) 计算虚拟地址 va 的页目录索引
```

- **访问页表项（PTE）**：

c

复制

下载

```
pte_t *pte = &vpt[PTX(va)];  // PTX(va) 计算虚拟地址 va 的页表索引
```

其中：

  - `PDX(va)`：提取虚拟地址 `va` 的 **页目录索引**（高 10 位）。
  - `PTX(va)`：提取虚拟地址 `va` 的 **页表索引**（中间 10 位）。

---

2. 为什么进程能通过 `vpt` 和 `vpd` 访问自身的页表?

实现原理

- **页表自映射机制**（Self-Mapping）：
  - 操作系统的内存管理单元（MMU）在 **页目录（Page Directory）** 中设置了一个特殊的 **自映射条目**，使得 **页目录本身** 和 **页表** 可以通过 **固定的虚拟地址** 访问。
  - 在 x86 架构中，通常：
    - `vpd` 指向 `0x3FFFF000`（页目录的虚拟地址）。
    - `vpt` 指向 `0x3FF00000`（页表的虚拟地址）。
  - 这样，进程可以直接通过 `vpd` 和 `vpt` 访问自己的页目录和页表，而无需切换到内核模式。

优势

- **无需陷入内核**：用户态程序可以直接访问页表，提高效率。
- **简化页表管理**：操作系统可以通过固定的虚拟地址访问页表，便于实现 `fork()`、`exec()` 等需要修改页表的操作。

---

3. 如何体现自映射设计?

自映射的数学关系

在 x86 两级页表中：

- **虚拟地址** `0x3FFFF000` 映射到 **页目录自身**：
  - `PDX(0x3FFFF000) = 0x3FF`（页目录索引）
  - `PTX(0x3FFFF000) = 0x3FF`（页表索引）
  - 因此，`vpd = (pde_t *)0x3FFFF000` 直接指向页目录。
- **虚拟地址** `0x3FF00000` 映射到 **页表**：

- `PDX(0x3FF00000) = 0x3FF`（页目录索引）
- `PTX(0x3FF00000) = 0`（页表索引）
- 因此，`vpt = (pte_t *)0x3FF00000` 指向第一个页表。

自映射的页目录项

- 页目录的最后一项（`vpd[1023]`）指向 **页目录自身**，形成递归映射：

  c

  复制

  下载

  ```c
  vpd[1023] = (pde_t)vpd | PTE_P | PTE_W | PTE_U;
  ```

  这样，访问 `0x3FFFF000` 就能访问页目录。

---

4. 进程能通过 `vpt` 和 `vpd` 修改自己的页表项吗?

理论上是可行的，但通常不允许

- **用户态可以读取页表**（如查询 PTE 的 `PTE_P`、`PTE_W` 等标志位）。
- **但直接修改页表项会破坏内存隔离**，因此：
  - **操作系统会限制用户态对页表的写权限**（如清除 `PTE_U` 或 `PTE_W` 标志）。
  - **修改页表必须通过系统调用**（如 `mmap`、`mprotect`），由内核完成。

Thinking 4.7

1. 什么时候会出现“异常重入”?

“异常重入”指的是 **在处理一个异常的过程中，又触发了新的相同或不同类型的异常。**在 `do_tlb_mod`（处理 TLB 修改异常）中，可能出现以下情况：

可能的场景

1. **TLB 缺失（TLB Miss）嵌套**：
   - 当内核处理一个 **TLB 修改异常**（如写一个只读页）时，可能需要访问用户态页表，而该页表项本身可能 **不在 TLB 中**，导致 **嵌套的 TLB 缺失异常。**
2. **页错误（Page Fault）嵌套**：
   - 如果内核在 `do_tlb_mod` 中访问某个用户态地址，但该地址 **尚未映射或权限不足**，会触发 **页错误异常**，形成嵌套。
3. **系统调用嵌套**：
   - 如果 `do_tlb_mod` 调用了某些可能触发系统调用的函数（如 `copyout`），而该系统调用又需要修改页表，可能再次触发 TLB 异常。

为什么需要支持“异常重入”?

- **避免死锁**：如果不支持重入，内核可能在处理异常时再次触发异常，导致无限递归或死锁。
- **保持一致性**：嵌套异常可能导致现场（`Trapframe`）被覆盖，需保存现场以确保正确恢复。

---

2. 为什么需要将 `Trapframe` 复制到用户空间?

在 `do_tlb_mod` 中，内核可能会 **将异常现场（`Trapframe`）复制到用户空间的异常处理栈**，主要原因包括：

(1) 支持用户态异常处理

- 某些操作系统（如 **Xv6/JOS**）允许 **用户态自定义异常处理程序**（类似 Linux 的信号处理机制）。
- 当异常（如 `TLB_MOD`）发生时，内核需要 **将控制权交给用户态处理程序**，而用户态处理程序需要访问原始的 `Trapframe`（如寄存器状态、错误地址等）。

(2) 恢复执行现场

- 用户态异常处理程序在完成处理后，可能需要 **修改** `Trapframe`（如修复错误地址、调整寄存器值），然后通过 `sigreturn` **或类似机制** 恢复执行。
- 将 `Trapframe` 复制到用户栈，使得用户程序可以 **安全地读取和修改它**，而不会破坏内核数据。

(3) 避免内核栈泄露

- 如果直接让用户态访问内核栈上的 `Trapframe`，可能引发 **安全漏洞**（如用户程序篡改内核数据）。
- 复制到用户栈可以 **隔离内核和用户数据**，增强安全性。

(4) 支持“异常重入”

- 如果异常处理程序本身又触发异常（如嵌套 TLB 异常），内核需要 **保存多个** `Trapframe`。
- 将 `Trapframe` 存储在用户栈上，可以 **支持多层异常嵌套**（每层异常都有自己的 `Trapframe` 副本）。

Thinking 4.8

1. 减少内核态-用户态切换开销

- **内核态处理**：每次页写入异常都需要 **陷入内核**（`trap`），由内核修改页表权限（如设置 `PTE_W`），再返回用户态，导致 **上下文切换开销**。
- **用户态处理**：直接在用户态修复页表权限（如通过 `mprotect` 或自映射页表），**避免频繁陷入内核**，提高性能。

---

2. 灵活性更高

- **内核态处理**：通常采用固定策略（如 `COW` 或直接报错），难以适应不同应用的需求。
- **用户态处理**：应用程序可以 **自定义异常处理逻辑**，例如：
    - 实现 **写时复制（COW）** 的优化版本。
    - 处理 **内存映射文件（mmap）** 的延迟写入。
    - 支持 **调试工具**（如监视特定内存写入）。

---

3. 减少内核复杂性

- **内核态处理**：需要内核维护复杂的页错误处理逻辑（如 `do_page_fault`），增加内核代码量和维护成本。
- **用户态处理**：将部分逻辑下放到用户态，**简化内核设计**，使其更专注于核心功能。

---

4. 支持更细粒度的内存管理

- **内核态处理**：通常以 **页（4KB）为单位** 修改权限。
- **用户态处理**：可以结合 **自定义内存分配器**，实现更细粒度的管理（如 `malloc` 中部分页只读，部分可写）。

---

5. 更好的隔离性与安全性

- **内核态处理**：若内核页错误处理有漏洞，可能导致 **特权升级**（如利用 `use-after-free` 攻击内核）。
- **用户态处理**：异常处理在用户态进行，**即使崩溃也不会影响内核**，攻击面更小。

Thinking 4.9

为了让子进程复制

如果放在写实复制保护机制完成之后，子进程就没有设置好entry，就无法实现写时复制保护，或者需要给子进程使用syscall_set_tlb_mod_entry

难点分析

进程间通信机制（IPC）

```
struct Env {
     // lab 4 IPC
     u_int env_ipc_value;//进程传递的具体数值
     u_int env_ipc_from;//发送方进程的ID
     u_int env_ipc_recving;//1：等待接受数据中，0：不可接受数据
     u_int env_ipc_dstva;接收到的页面需要与自身的哪个虚拟页面完成映射
     u_int env_ipc_perm;传递的页面的权限位设置
 };
```

sys_ipc_recv(u_int dstva)接受消息，放到dstva这一页里面，接受的值放到env_ipc_value这里面

- 将自身env_ipc_recving设置为1，表明盖进程准备接受发送方的消息
- 之后给env_ipc_dstva赋值，表明自己将接收到的页面与dstva完成映射
- 阻塞当前进程
- 放弃CPU

sys_ipc_try_send（u_int envid, u_int value, u_int srcva, u_int perm)发送消息给envid这个进程，发送srcva这一页，发送value这个值，

- 根据envid找到相应进程，如果指定进程为可接受状态，发送成功
- 否则，函数返回-E_IPC_NOT_RECV,表示目标进程未处于接受状态
- 清除接受进程的接收状态，将相应数据填入进程控制块，传递物理页面的映射关系
- 修改进程控制块中的进程状态

使用srcva为0的调用来表示值传value值

Fork

- user/lib/fork.c
    - int fork(void)
        - cow_entry处理写实复制的函数
        - env_user_tlb_mod_entry
        - syscall_set_tlb_mod_entry(0,cow_entry)
        - syscall_exofork()定义在kern/syscall_all.c里面
            - 分配env结构体
            - 设置env->tf为父进程上下文
            - 设置新进程返回值env_tf.regs[2]=0,表明是子进程
            - 设置env_status为ENV_NOT_RUNNABLE
        - ENVX(syscall+getenvid())
        - UXSTACKTOP
        - vpd[i]
        - duppage
        - syscall_set_tlb_mod_entry
        - syscall_set_env_status
    - duppage(u_int envid, u_int vpn)复制地址空间
        - syscall_mem_map
        - syscall_mem_unmap
    - static void __attribute__((noreturn)) cow_entry(struct Trapframe *tf)实现写时复制

实验体会

前面lab3说中断是一种异常，现在系统调用也是一种异常。

其实异常可以说是一种陷入内核的手段。

异常的处理方式，主要是要跳转到异常分发代码处（当然前面还需要设置EPC，EXL和Cause，分别是记录返回位置，陷入内核禁止中断，记录异常原因）

异常分发代码根据异常类型跳转到不同的处理程序

8号异常就是系统调用的处理程序