Project 4 Virtual Memory 设计文档

中国科学院大学
张磊
2019 年 12 月 2 日星期一

1. 内存管理设计

（1） 你设计的页表是几级页表，页表项的数据结构是什么？页表本身的数据结构是什么？

我设计的页表是一级页表，页表本身是一个数组，页表项的数据结构见下图：

```
//PTE Flags // R = Reserved
// 31                              12 11 9  8   7   6  5            1    0
// +--------------20-----------------+----+---+---+---+----+-----+---+---+
// |       Page Table Index          |  C | D | V | G |    |     |IMM|PTEV
// +---------------------------------+----+---+---+---+----+-------------+

typedef struct pte {
    uint32_t ptentry;
} pte_t;
```

PTEV 表示该页表项是否有效，IMM 位表示该页表项指示的物理内存是在内存当中还是在硬盘里，当 PTEV == 1 && IMM == 1 时，高 20bits 用于表示物理页框号，当 PTEV == 1 && IMM == 0 时，高 20bits 表示在硬盘中的位置，剩下的 C,V,D,G 位是为方便填充 ENTRYLO0 和 ENTRYLO1 特意设置，其实没有必要存在页表项中，完全看个人取舍；

（2） 任务 2 开发时初始化了多少个页表项，能索引到多大的物理空间，以及使用了多少个物理页框保存页表？

任务 2 开发时，为实现从 0x0000_0000 – 0x7fff_ffff 的空间共 2GB 的映射，初始化了 2GB/4KB = 524288 项页表项；

除了留给内核，例外处理程序，POMON 的内存，共初始化了 25MB 的物理内存，所以用了 6400 个物理页框；

每个页表项大小为 4B，所以存储 524288 项页表项共需 512 个物理页框；

（3） 任务 3 中，进程的用户态栈的起始地址是多少，栈空间是多大？

任务 3 中，进程用户栈起始地址为 8 * 1024 - 4，栈空间为 8KB；

（4） TLB miss 何时发生？在任务 2 中，你处理 TLB miss 的流程是怎样的？

TLB miss 发生在用户访问 mapping 段的地址但是 TLB 中没有对应虚拟地址到物理地址的映射时；

在发生 TLB 例外后，充填例外会跳转到 0x8000_0000 的位置，在这里将 TLB 例外处理程序引导至 handle_TLB，在 hadnle_TLB 中会最终跳转到 do_TLB_refill 例外处理函数，但是我发现实际并不需要这个 0x8000_0000 的例外地址入口，发生 TLB 例外后，也会从我们之前设计的例外处理出口进入 handle_TLB；

```
NESTED(TLBexception_handler_entry, 0, sp)
TLBexception_handler_begin:
    //TODO: TLB exception entry
    //jmp exception_handler[i] which decided by CP0_CAUSE
    CLI
    SAVE_CONTEXT(USER)
    j       handle_TLB
    nop
TLBexception_handler_end:
END(TLBexception_handler_entry)
```

在 do_TLB_refill 函数中，首先设置好 entryhi 寄存器的值，然后使用 tlbp 指令查询 TLB 中是否存在对应的 TLB 项，根据查询结果分为 TLB 重填和 TLB 无效 2 种情况，分别使用 tlbwr 和 tlbwi 指令填充 TLB；

由于在任务 2 中，我们已经填充好了页表项，所以不会存在 page_fault；

2. 缺页处理设计

（1） 何时会发生缺页处理？你设计的缺页处理流程是怎样的？

当发生 TLB 例外后，do_TLB_refill 查询对应虚拟位置的页表项，如果没有建立对应虚拟地址和物理地址的对应关系则会出发 page_fault；

当发生 page_fault 后，首先判断是否有剩余可用的物理内存，如果有则填充对应的页表项，然后根据页表项中的对应关系完成 TLB 项的填充；

如果没有剩余可用的物理空间则进行换页，采用 FIFO 的原则，将最先使用的物理页替换到 SD 卡中，并将腾出的物理内存分配给当前进程；

（2） 你使用什么数据结构管理物理页框，管理多少物理页框（例如管理哪些地址范围内的物理页框）？在缺页分配时，按照什么策略或原则进行物理页框分配？

我采用链表管理物理页框，共 6400 个，管理了 0x0001_2000 – 0x0080_0000 以及 0x1200_0000 – 0x 1fff_ffff 的物理空间，发生 page_fault 时采用 FIFO 的原则进行物理页框分配；

```
typedef struct pageframe
{
    uint32_t paddr;         // physical address
    uint32_t vaddr;         // virtual address
    struct pgframe *prev;
    struct pgframe *next;
}pageframe_t;
```

上图为物理页框的内容，下图中的两个链表分别用来保存空闲的和正在使用的物理页框；

```
queue_t free_mem_list;      // save free physical memory frames
queue_t busy_mem_list;      // save busy physical memory frames
```

（3） 你的设计中是否有 pinned 的物理页框？若有，具体是保存什么内容的物理页框？

我的设计中没有 pinned 的物理页框； 如果有的话，我觉得经常使用的物理页框，

比如用户栈对应的物理页框需要保存下来；

3. Bonus 设计（还有 bug，后续改进后上传）

（1） Bonus 中你设计的操作系统通过页表访问的可用物理内存是多少？ swap 操作是由专门的进程完成么？

Bonus 中，我设计的操作系统通过页表访问的物理内存可以达到 8GB，但是 swap 操作没有用其他进程完成；

（2） 你设计的页替换策略是怎样的，有什么优势和不足么？

我采用了最简单的 FIFO 的替换策略，优势就是很简单，便于实现，不足就是由于没有预见性，极有可能将经常使用的访存的物理页给 swap 到 SD 卡中，加大了访存的延迟；

（3） 你设计的测试用例是怎样的？

我的测试用例：在测试时，将物理页框号缩小为 8，分配给 shell，process1，process2 以及主进程，当 process2 执行访存指令时，由于页表中没有建立对应的虚拟地址到物理内存的对应关系，会触发 page_fault，在 page_fault 中，由于没有可用的物理内存，所以会出发换页操作；

下图为，采用 FIFO 原则选择替换对应物理页；

```
// 1. Get item = Physical Memory Block form busy_mem_list.head
pageframe_t *item0, *item1;
item0 = (pageframe_t *)pmem_queue_dequeue(&busy_mem_list);
item1 = (pageframe_t *)pmem_queue_dequeue(&busy_mem_list);

current_running->page_table[((item0->vaddr)>>12)].ptentry = 0;
current_running->page_table[((item1->vaddr)>>12)].ptentry = 0;

SD_addr0 = SD_Base + SD_Page_Index * PAGE_SIZE;
SD_Page_Index++;
SD_addr1 = SD_Base + SD_Page_Index * PAGE_SIZE;
SD_Page_Index++;

current_running->page_table[((item0->vaddr)>>12)].ptentry = 0x1 | SD_addr0;
current_running->page_table[((item1->vaddr)>>12)].ptentry = 0x1 | SD_addr1;
```

然后，将原物理页中的内容 swap 到 SD 中；

```
// 2. Write the context of item into SD card
// Write_SD_card();
sdwrite((uint8_t *)(item0->vaddr & 0xfffff000),SD_addr0,PAGE_SIZE);
sdwrite((uint8_t *)(item1->vaddr & 0xfffff000),SD_addr1,PAGE_SIZE);
```

随后，建立新的从虚拟地址到物理内存的映射；

```
// 3. Paging the vaddr to item block
uint32_t PPN0,PPN1;

PPN0 = current_running->page_table[(badvaddr >> 12) & 0xfffffffe].ptentry = (item0->paddr & 0xfffff000)
PPN1 = current_running->page_table[(badvaddr >> 12) | 0x1].ptentry = (item1->paddr & 0xfffff000) | (CD

pmem_queue_push(&busy_mem_list,item0);
pmem_queue_push(&busy_mem_list,item1);
```

最后，填充 TLB 项，完成 swap 操作；

```
// 4. Replace TLB entry
entryhi = (((badvaddr >> 12) >> 1) << 13) | (current_running->pid & 0xff);
entrylo0 = PPN0 >> 6;
entrylo1 = PPN1 >> 6;

SET_CP0_ENTRYHI(entryhi);
SET_CP0_ENTRYLO0(entrylo0);
SET_CP0_ENTRYLO1(entrylo1);
asm volatile("tlbwi");
```

4. 关键函数功能

（1） Do_TLB_refill 函数：

```
void do_TLB_Refill()
{
//    vt100_move_cursor(0,0);
//    printk("-> TLB Exception %d                ",TLB_miss_time);
//    TLB_miss_time++;
//    current_running->user_context.cp0_epc = current_running->user_context.cp0_epc + 4;

    uint32_t CDVG_Flags;
    CDVG_Flags = ((PTE_C << 3) | (PTE_D << 2) | (PTE_V << 1)) & 0x3f;

    uint32_t entryhi,entrylo0,entrylo1,index,context,badvaddr;

    entryhi  = GET_CP0_ENTRYHI();
    context  = GET_CP0_CONTEXT();
    badvaddr = GET_CP0_BADVADDR();

    uint32_t PPN0 = current_running->page_table[(badvaddr >> 12) & 0xfffffffe].ptentry;
    uint32_t PPN1 = current_running->page_table[(badvaddr >> 12) | 0x1].ptentry;

//    context = context << 9;
    SET_CP0_ENTRYHI((entryhi & 0xfffff000) | (0xff & current_running->pid));
    asm volatile("tlbp");
    index = GET_CP0_INDEX();

    if(index & 0x80000000)
    {
        if(PPN0 & 0x1) // PTE Valid
        {
            if(PPN0 & 0x2) // Page In Memory
            {
                entrylo0 = PPN0 >> 6;
                entrylo1 = PPN1 >> 6;
                SET_CP0_ENTRYLO0(entrylo0);
                SET_CP0_ENTRYLO1(entrylo1);
                asm volatile("tlbwr");
            }
            else        // Page In SD_card
            {
                swap_page_readback(badvaddr); // re
            }
        }
        else
        {
            do_page_fault(badvaddr);
        }
    }
```

```
    else
    {
        //TLB invalid
        if(PPN0 & 0x1) // PTE Valid &
        {
            if(PPN0 & 0x2) // Page In Memory
            {
                entrylo0 = PPN0 >> 6;
                entrylo1 = PPN1 >> 6;
                SET_CP0_ENTRYLO0(entrylo0);
                SET_CP0_ENTRYLO1(entrylo1);
                asm volatile("tlbwr");
            }
            else        // Page In SD_card
            {
                swap_page_readback(badvaddr); //
            }
        }
        else
        {
            do_page_fault(badvaddr);
        }
    }
}
```

首先设置 entryhi 寄存器，通过 tlbp 指令查找 TLB 中是否有对应的项，然后分为 TLB 充填和 TLB 无效两种例外，进行 TLB 例外的处理；

查找对应的页表项，若对应页表项有效且对应物理内存在内存中，则将 TLB 填充，否则，如果在 SD 卡中，则需要将其从 SD 卡中读取回来，为此我们需要先腾出一块物理内存；

若对应页表项无效，则触发 page_fault 例外，进项 page_fault 的处理；

（2） Do_Page_fault 函数：

```
void do_page_fault(uint32_t badvaddr)
{
//    vt100_move_cursor(0,0);
//    printk("-> Page Fault Exception %d                ",Page_Fault_time);
//    Page_Fault_time++;

    uint32_t CDVG_Flags;
    CDVG_Flags = ((PTE_C << 3) | (PTE_D << 2) | (PTE_V << 1)) & 0x3f;

    uint32_t entryhi,entrylo0,entrylo1;

    if(free_mem_list.head != NULL)
    {
        pageframe_t *frame;
        frame = palloc();
        current_running->page_table[(badvaddr >> 12) & 0xfffffffe].ptentry = (frame->paddr & 0xfffff000) |
        frame->vaddr = badvaddr & 0xffffe000;
        frame = palloc();
        current_running->page_table[(badvaddr >> 12) | 0x1].ptentry = (frame->paddr & 0xfffff000) | (CDVG_F
        frame->vaddr = (badvaddr & 0xfffff000) | 0x1000;

        uint32_t PPN0 = current_running->page_table[(badvaddr >> 12) & 0xfffffffe].ptentry;
        uint32_t PPN1 = current_running->page_table[(badvaddr >> 12) | 0x1].ptentry;

        entryhi = (((badvaddr >> 12) >> 1) << 13) | (current_running->pid & 0xff);
        entrylo0 = PPN0 >> 6;
        entrylo1 = PPN1 >> 6;
```

```
        entryhi = (((badvaddr >> 12) >> 1) << 13) | (current_running->pid & 0xff);
        entrylo0 = PPN0 >> 6;
        entrylo1 = PPN1 >> 6;

        SET_CP0_ENTRYHI(entryhi);
        SET_CP0_ENTRYLO0(entrylo0);
        SET_CP0_ENTRYLO1(entrylo1);
        asm volatile("tlbwi");
    }
    else
    {
        swap_page(badvaddr);
    }
}
```

这里主要分为两种情况，如果有空闲的物理页框则将其分配给对印度个页表项，建立心新的虚拟地址和物理内存间的映射关系，完成 TLB 的填写；

如果没有空闲的物理页框，则进行换页操作，swap_page；

参考文献

[1] 龙芯 2F 处理器手册

[2] XV6 源码