Project 5 Virtual Memory 设计文档

中国科学院大学
熊子威
2015K8009915050

2017 年 12 月 24 日

1 一点点吐槽

这个实验应该是这个学期到目前位置最消耗精力的实验了。学习理论课的时候还觉得这个部分比较简单，因为概念上非常简洁明了，要做的事情其实也不多。等真正开始写实验才发现一切的理论简单都是骗人的!!! 理论简洁只不过是忽略了实际操作上的一切琐碎细节，而这些全都是实际实现需要去考虑的。从 12 号开始写，弄到 24 号，期间精力各种各样的错误，尝试遍了能用的和打印无关的几乎所有 debug 方法，和好几个小伙伴一起合作找问题才让代码顺利运转。最犯人的并不是实现上有多难，而是出错了完全没有办法去检查到底是哪里的问题，任何显示的检查的方式都有可能引入新的错误，只能一遍一遍静态分析自己的代码，打印简略到不行的辅助信息来分析错误，还要考虑自己的代码是如何和老师的代码联系在一起。感觉整个完成实验的过程并不是在设计方案-实现方案-找出问题-解决问题，而是在设计方案-实现方案-猜测问题-试错-解决错误。整个 debug 的过程并没有让我对虚拟内存的理解加深，因为我在处理的总是一系列奇奇怪怪，因为自己的代码和老师的框架没有很好整合在一起的问题，有时候总会想还不如自己写来得痛快，好歹自己知道什么样的问题是在哪里触发的。

2 用户态进程内存管理设计

2.1 测试的用户进程的虚拟内存布局

本次任务中并没有将进程代码段和用户段区分开，因此我实现时也就没有分开（好像如果直接使用 createimage 来做 image 且区分这两个段的话，还不如不区分而一并放入内存要更简单）。而两个进程在内存的布局和 makefile 中给的数据略有出入。Process 1 占据了虚拟地址中 0x00100000 开始到 0x0010ffff 的位置，但是实际上进程 1 大小仅为 3472KB（代码段和数据段，不包含会动态增长的数据，事实上也没有会动态增长的数据），甚至不足以装满一页。而进程 2 占据虚拟地址中 0x00200000 开始

的位置，实际大小仅 3344KB。虚存中这两块地址是这两个测试程序运行时应当所在的位置，两个进程实际存储的位置在 makefile 并未进行页对齐，我为了后面方便拷贝代码段和数据段，将它们分别对齐到了 0xa0805000 和 0xa0806000 的位置。这个位置影响到了 kernel 内 page_map 的大小，如果使用 256 项的页框将会破坏进程 1 的代码，因此减小到 64 项，后来只保留 16 项，到测试页替换是仅保留 5 项（一级页表）和 7 项（二级页表）

2.2 页表项结构

为了减少在汇编内处理页表的工作量，我希望页表项的结构与 EntryLo0 和 EntryLo1 保持一致，因此页表项结构从高到低分别是 26 位 PFN 和 6 位 flag 域，其中 flag 由老师提供的宏来取或生成。

2.3 任务一：

任务一由于不是按需的页表，因此在初始化的时候一次性分配好所有使用的页表。这里按照 process 的 pcb 中 size 域来确定使用的页表项数目，由于进程非常小，其实一页已经足够。因此每个进程都只有一个页表项在页表中，这一项在初始化的时候完成填充，当然，初始化也需要使用 page_alloc 分配一个页给页表，各自使用一个物理页框来储存各自的页表。而且设计上将页表放置在虚拟地址的内核区，然后将这个内核的直接映射得到的物理页框作为储存页表的页框。因为如果放置在用户区的话，为了找到页表和所在物理页框的映射需要查询页表，但是此时映射还没有建立，所以无法查询页表，但是不查询页表无法找到映射，陷入死循环。所以其实就是用给定的 page_map 的项的物理地址加上 0xa0000000 作为一个进程可以使用的储存页表的页框的基址。

2.4 任务二：

任务二是严格按需分配的，凡是还没有真正被使用就不会分配任何页框，因此初始化的时候甚至不需要分配页框来储存页表，但是这样可能导致在 handle_tlb_c 内触发新的 TLB miss 或者 page fault，因此为了方便，默认一开始在内核中提供一页的大小供进程储存自己的页表，如果以后页表项增多则动态增长。初始化时分配完毕之后不建立任何页表项，但是要把页表所在物理页框清理为 0，这个可以利用页框出于非映射段来直接完成清零，然后运行时需要查询到某一项时再生成页表项，判断是否缺页，如果缺页则拷贝对应需要拷贝代码（用户栈不能拷贝，因此这里有一步区分的操作）然后填写页表。填写总是保证写入的页表项已经是 valid 的状态而且对应的物理页框页的确是 valid 状态，这样汇编内不用再进行十分复杂的处理。

2.5 物理内存的管理

物理页框由老师提供的 page_map 提供抽象，这个结构体描述了该项物理页框的基址（分配页框后建立页表项时需要使用），映射到这个页框的虚拟地址的基址（后来并没有用上），当前物理页框是否空闲（分配的时候需要使用），当前物理页框容纳的页是否被 pin 住（分配时确定，用户栈和页表使用的页都是 pin 住的），当前页框被哪一个进程使用（进行页替换时需要将被替换的页在对应进程的页表中的项设置为无效）。

因此包含的重要的信息是 paddr, unused, pinned, pid。一开始的设计还使用了 valid 域，dirty 域，VPN 域，PFN 域，前两者其实并不需要，而后面两个只是在使用页框需要用到页号时可以不用进行移位而直接获得，只是一个偷懒用的数据域，并不是必需的。而这个时候分配策略也很简单，顺序扫描 page_map 数组，寻找到的第一个未被使用的页框即可用来分配。但是当处理到页替换的时候，则需要扫描两次，一次寻找没有用的页框，一次在没有可用页框的情况下寻找未被 pin 住的页框，然后进行分配。

2.6 TLB miss

TLB miss 会在进程运行到一个未经映射的地址时发生（应该是 TLB 中没有映射信息时发生，但是我的代码中如果 TLB 内无对应项，那么一定这个地址对应的页表项也是无效，也即未经映射的）。

当 TLB miss 发生的时候，程序流从正常的执行跳转到 general exception handler，在这里检测是 TLB 的异常之后转入 handle tlb，这个汇编函数将直接跳转到 handle tlb c 内。c 函数中，首先通过 badvaddr 值查询这个进程的页表，这个时候由于页表项一定是无效的，因此可以直接分配页框，然后检查当前地址是否满足拷贝代码的条件，如果满足则进行拷贝，拷贝完毕之后将对应的页表项设置为有效，然后回到汇编函数内，汇编函数将对应的两个页表项送到 EntryLo0 和 EntryLo1 寄存器并准备 EntryHi，然后写 TLB。

2.7 遇到的问题

1. LPPPPP：第一次实现的时候,线程完成第一轮打印之后整个程序都会停止下来疯狂打印 LPPPPP，我在这个问题上耗费了至少三天的时候，但是由于打印信息无法使用（打印函数会导致更多的错误），只能靠静态分析代码，最后另一个同学比对所有的 enter critical 和 leave critical 的时候发现在 c simple handler 内，老师不处理进入 simple handler 的异常，因此直接 return 了，但是这个时候 return 回到汇编后汇编内有一个 leave critical，这个当然就是 LPPP 的来源，为 c simple handler 还没有来得及 enter critical 就返回了。

2. 无休止的 TLB miss：这个问题当时检查时发现这个 tlb miss 是在汇编代码访问页表的时候发生的，由于访问页表出发了一次 TLB miss，然后程序又进入到了 tlb handler，而这个时候又去访问

页表，于是又出发了一轮 TLB miss，如此循环不停。后来检查却没有发现任何问题。直到后来重新检查 scheduler 里面对 pcb 的定义时才发现我设置的 page table 在 pcb 内的偏移量是错的。因为使用了老师的 nested count 宏的偏移来作为起始地址，然后往下数到 page table 所在的位置，但是没想到中间隔了一个 64 位的 deadline 域，于是我访问的 page table 实际上并不 page table，这个问题修复之后任务一就正常运转了。

3 缺页处理和 swap

3.1 缺页处理

缺页处理思路还算比较清晰，在进程访问了一个 TLB 里不能映射的地址，而且这个地址在页表内没有有效的记录，那么就说明不是一个 TLB miss 的错误，而是一次 page fault，因为只有 page fault 才能在查询 TLB 无效时发现页表内也无效。

但是一开始我并没有这么考虑，开始我的考虑是一个进程可能只触发 TLB miss 而不触发 page fault，于是设计了两个函数一个处理 TLB miss，一个处理 page fault。TLB miss 的处理函数一定在调用了 page fault 之前进行调用。不过后来发现其实根本没有必要进行区分，因为引入了页替换之后，一旦发生替换，我将清楚当前页的原所有者关于这个页的页表信息，宣布这个页已经被别人占用，因此只要替换之后一定只能触发 page fault，而第一次运转时由于是按需分配，因次也一定触发 page fault，因此两个函数可以合并一起。不过虽然合并了，我还是在函数内部区分到底只是 TLB miss 还是触发了 page fault。

之前在 review 的时候和老师谈到缺页的话我们不得不考虑当前进程运行到了代码段的哪一个部分，因为如果进程超过一个页，我们需要记录当前进程运行的位置，然后将对应地址所在的页从磁盘中拷贝到物理页框中，因此可能需要添加一些新的信息来记录这个运行位置。但是后来实际开始写的时候我觉得并不需要添加额外的信息。所谓运行到的位置，无非就是当前地址当进程入口地址的偏移，而这个偏移也就是这个进程运行到代码段中哪个位置的偏移，也就是在磁盘内到这个进程储存位置起点的偏移，而这个偏移利用 badvaddr 寄存器和 pcb.entry point 可以直接计算出，那么直接利用这个偏移我们就能确定需要拷贝的页是哪一个页而不需要添加额外的信息了。缺页时处理流程如下：

1. 确认已经触发了 page fault（即页表内对应表项的 valid 位为 0）
2. 计算进程当前发生错误的地址到 entry point 的偏移，然后抹去低 12 位得到页偏移
3. 分配一个物理页框给这个进程
4. 从磁盘内（image 的进程代码段）拷贝对应页偏移处的页到这个物理页框中
5. 回到汇编函数，写 TLB

其中拷贝操作不需要填写 TLB 之后再做，因为我们使用的物理页框只是非映射段的一部分，因此可以利用虚实地址相差 0xa0000000 的特点来直接完成代码拷贝，就不用回到汇编函数写 TLB 然后再回到 c 函数拷贝代码。

不过这个处理流程在使用了用户栈之后有一点点变化，因为分配的页框是否 pin 住，是否进行拷贝都需要进行判断，看看到底是访问的代码还是用户栈，如果是栈，pin 住页且不拷贝，否则不 pin 住，但是需要拷贝。

这样的处理在测试时会呈现 TLB miss 和 page fault 次数相等，而且是调度次数的两倍。说明运行正确，因为我们没有单独触发 TLB miss 的情况，而每调度一次都会发生一次替换（进程太小，每次替换整个进程都被换出去了），替换一次会导致一次新的 page fault，因此 page fault 的次数是进程调度次数的两倍。

3.2 swap

对于页替换，由于我们的进程太小只使用一个页，所以无论什么替换算法其实都是一样的，而效率最高的替换算法应该是固定替换，我们总共使用 3 个页框，两个装固定的页表，一个装代码（使用用户栈的话 5 个页框，还多了了个栈，做二级页表时 7 个页框，因为还需要装两个页目录），这样的话直接每次替换第三个页框即可，不需要查询，时间复杂度 O(1)。

当然我并没有这么做，一开始的做法是每次从头扫描，寻找没用过的页，找不到的话则寻找未 pin 住的页，但是这个方法不是真的 FIFO，因为每一次其实替换的都是一个非 pin 页，后来改为时钟算法，用一个 static 变量记录当前替换发生的位置，下一轮替换是从这个位置后一个页（如果是最后一个，则从第一个页开始）进行寻找。严格来说这并不是 clock 的方式，但是我们并没有 R 位，也没有 M 位，Dirty 位对于栈才有效，但是栈是 pin 住的，所以就只是以 clock 的方式进行轮询，但是不涉及其他的 clock 方式中的操作。

3.3 遇到的问题

这个实验中遇到的问题是最多的，各种各样奇奇怪怪的问题，这里只是记下最印象深刻的几个

1. 触发 4 号例外。按照老师的定义，4 号例外是 ADEL 例外，也就是用对齐访问的指令访问了一个非对齐的地址。我第一次碰到这个问题是 EPC 居然记录了一个 00002831。这个错误的解决其实也很有趣。我前面提到缺页处理是需要计算一个偏移，然后通过这个偏移来寻找磁盘中正确的拷贝起始位置，然后抹去低 12 位，整页拷贝到一个物理页框中。但是我一直没有意识到的是，老师的内存布局中，没有把进程的 loc 域做页对齐，于是我的拷贝就出了错误，从一个错误的地方拷贝了不知道什么东西到了物理页框，然后运行的时候读出了什么指令也不知道。于是触发了 4 号例外，于是我把 loc 改成了一个页对齐的位置，之后 4 号例外就消失了。

2. 触发 10 号例外。10 号例外是保留指令，我对这个问题印象十分深刻。当时好几个小伙伴都触发了保留指令例外。但是大家都没有点子为什么会触发这个例外。我困惑了很久之后，突然灵光闪现将 page_map 修改为只有 128 项，再运行时 10 号例外就消失，然后我才意识到问题在哪里。问题在于我将进程一对齐到了 0xa0805000 的位置，而 kernel 已经生长到了 0xa0804320 左右的位置，而这个位置只是代码的大小，并不包含各种数据，page_map 内每一个项有 8 个字，256 项共计 256 * 8 = 2048，早就吃掉了进程 1 的代码。减小之后就没有问题了。然后后来我一直使用的 16 个页框，需要页替换时则使用更少，也就没有保留指令的问题了。

3. 用户栈触发 LPPPP。这个问题也是挺有趣的，我为了使用用户栈，依据老师的提示添加了 restore stack(KERNEL) 的操作，但是很快就会在线程遭遇时钟中断时发生 LPPPPPP，这个让我很困惑。直到后来细想才意识到，线程如果使用了 restore stack(KERNEL)，它取出来的 stack 未必是正确的，也就是老师提示的那样，线程拿到了一个错误的信息，这个信息具体是什么就不好说了，而且内核线程用户栈和内核栈是混用的，这就无法得知这个时候恢复出来的东西是什么。但是进程不一样，进程的内核栈就是随便使用的，供进程在内核为各个函数提供栈空间，一旦离开内核态，这个栈里面的内容就作废了。所以我在 general exception handler 里添加了区分内核线程和进程的代码，只对进程恢复核心栈，然后 LPPPP 就消失了。

4. Process2 加法算到 68 就触发 EPPPPP。这个 bug 我至今还没能修复出来，因为我根本不知道这是哪里的问题。我以为是递归深度太深爆栈了，但是用核心栈测试的时候发现一页栈足够时候，而 EPPPP 的来源也无法确定，不知道从何下手。

4 Bonus 的设计

4.1 页替换

最开始的页替换是 FIFO，使用一个队列来管理 pin 住的页，当一个页框被分配为 pin 住的，那么就将这个页框放到这个队列里面，然后在需要替换的时候从这个队列中出队一个页。后来想更换成 LRU 算法，于是在页框信息中添加了一个时间域，但是后来实现的时候意识到我们根本无法判断一个页到底有没有被访问，所以我没有采用这个方法。后来想了想，我们的 page_map 是以数组的形式储存的，天然有制作成为循环链表的优势，于是决定使用 Clock 算法。

我们的页面没有关于访问的信息，标准 clock 算法每次都要清理一个页访问/读写的标记，这里我们一方面是无法获取一个页被访问的信息，如果我们能获取，那么说明已经进入例外了，这个时候访问信息已经没有意义了。所以这里的 Clock 只是记录下当前发生替换时选中的页框，下一次需要替换的时候就从这里开始选择。

4.2 二级页表

二级页表实现上逻辑比较清晰，就在于添加了页目录之后需要通过两次寻址来找到最终的页表。

二级页表本质上是一个二维数组，这个数组一地址的高 10 位中间低 10 位为横纵坐标，数组中储存的数据是一个个 PFN。这样理解就让代码好处理了。首先把页目录放置到内核，页表页则可以放到用户空间（仿照 linux 的做法），然后用 badvaddr 的高 10 位索引找到一个页表页的入口，对这个入口的高 20 为寻址则找到了这个页表页映射的物理页号，如果这个页表页本身并不可用，分配空间。所以二级页表管理的本质就是手工管理了一个二维数组，这个数组的脚标来自 badvaddr，然后行指针已经在内核分配号，我们需要时不时分配列指针。

4.3 测试

测试只在老师提供的两个进程上测试，没有设计自己的测试用例。因为读任务书不认真，当我发现需要自己设计用例的时候已经没有时间设计并调试顺利然后提交了。。。。

5 关键代码

1. 处理用户栈时避免栈指针出错的代码。这里让用户进程使用核心栈而对内核线程不做处理，避免时钟中断导致线程触发 LPPPP

```
    NESTED(general_exception_handler,0,sp)
exception_handler_start:
        .set    noat
        .set    mips32
        SAVE_CONTEXT(USER)
        la      k0, current_running
        lw      k0, (k0)
        lw      k0, NESTED_COUNT(k0)
        bnez    k0, 1f
        nop
        RESTORE_STACK(KERNEL)
1:
        CLI
        mfc0    k0, CP0_CAUSE
        andi    k0, k0, CAUSE_EXCCODE   /* k0 = {exc_code,00} */
        la      k1, interrupt_handlers
```

```
        add     k0, k0, k1
        lw      k0, 0(k0)
        jr      k0      /* interrupt_handlers[exc_code](); */
exception_handler_end:
```

2. 时钟替换算法

```
int page_alloc( int pinned ) {
// code here

int free_index;
int find = 0;
static int clock = 0;
for(free_index = 0; free_index < PAGEABLE_PAGES; free_index++){
    if(page_map[free_index].unused){
        find = 1;
        break;
    }
}

if(!find){
    for(free_index = clock; free_index < PAGEABLE_PAGES; free_index++){
        printf(4, 1, "searching");
        if(!page_map[free_index].pinned){
            find = 1;
            clock = (free_index + 1) % PAGEABLE_PAGES;
            break;
        }
    }
    int pid = page_map[free_index].pid;
    int VPN2 = page_map[free_index].vaddr & 0xffffe000;
    uint32_t* page_table = (uint32_t*)pcb[pid-1].page_table;
    page_table[page_map[free_index].vaddr>>12] = 0;
    tlb_flush(VPN2 | pid);
}
```

```
    bzero((char*)(page_map[free_index].paddr + 0xa0000000), PAGE_SIZE);
    page_map[free_index].pid = current_running->pid;
    page_map[free_index].VPN = current_running->user_tf.cp0_badvaddr >> 12;
    page_map[free_index].vaddr = current_running->user_tf.cp0_badvaddr;
    page_map[free_index].unused = FALSE;
    page_map[free_index].pinned = pinned;
    page_map[free_index].dirty = FALSE;
    return free_index;
}
```

其中 dirty 域并没有使用到。

3. 处理页替换和页表查询

```
    void handle_tlb_c(void)
{
    static int tlb_miss = 0;
    static int page_fault = 0;
    int line = 28;

    uint32_t VPN = current_running->user_tf.cp0_badvaddr >> 12;
    uint32_t badvaddr = current_running->user_tf.cp0_badvaddr;
    uint32_t* PTDir = (uint32_t*)current_running->page_dir;
    uint32_t PTDItem = PTDir[badvaddr >> 22];
    uint32_t PTEntryItem = ((uint32_t*)PTDItem)[(badvaddr >> 12) & 0x3ff];
    uint32_t* PTBase = (uint32_t*)(*((uint32_t*)(PTEntryItem & 0xfffff000)));
    uint32_t index;
    int offset = current_running->user_tf.cp0_badvaddr - current_running->entry_po
    current_running->page_table = (uint32_t)PTBase;


    printf(line++, 1, "TLB MISS: %d at %x", ++tlb_miss, badvaddr);
    if(!(PTDItem & 0xc00)){
        index = page_alloc(badvaddr >= 0x300000);
        PTDir[(badvaddr>>22)] = ((page_paddr(index) + 0xa0000000)) | 0xc00;
        if(!(PTEntryItem) & 0xc00){
```

```
                index = page_alloc(badvaddr >= 0x300000);
                ((uint32_t*)PTEntryItem)[(badvaddr>>12) & 0x3ff] = ((page_paddr(index)
                if(!(*PTBase & PE_V)){
                    printf(line++, 1, "Memory Allocation %x", current_running->user_tf
                    index = page_alloc(badvaddr >= 0x300000);
                    *PTBase = (page_paddr(index) & 0xfffff000) >> 6 | (PE_V | PE_D);
                    uint32_t entry_hi = ((VPN << 12) & 0xffffe000) | (current_running-
                    tlb_flush(entry_hi);
                    if(current_running->task_type == PROCESS){
                        uint32_t source = (current_running->loc + offset) & 0xfffff000
                        if((badvaddr) <= 0x300000){
                            bcopy((char*)source,
                                  (char*)((uint32_t)(page_paddr(index) + 0xa0000000)),
                                  PAGE_SIZE);
                        }
                    }
                    printf(line++, 1, "PAGE FAULT: %d", ++page_fault);
                }
            }
        }
    }
```