Project 4 Virtual Memory 设计文档

中国科学院大学
陈灿宇
2018.12.15

1　内存管理设计

(1) 你设计的页表项包含哪些内容？页表本身使用什么数据结构保存？

页表项用 32 位进行表示，结构示意图如下所示：

- 第 31 位到第 23 位为 PFN 域，表示物理页号;
- 第 5 位到第 3 位为 C 域，表示对应的 TLB 的一致性属性，在本实验中设置为 2;
- 第 2 位为 D 域，表示对应的 TLB 是否可写，在本实验中设置为 1;
- 第 1 位为 V 域，表示对应的 TLB 是否有效，如果对应的物理页换出到 swap 区，则将本位置 0，同时将 TLB 的此位置为 0;
- 第 0 位为 G 域，表示对应的 TLB 的全局位，如果忽略 ASID 的比较，则将此位置为 1, 否则置为 0;

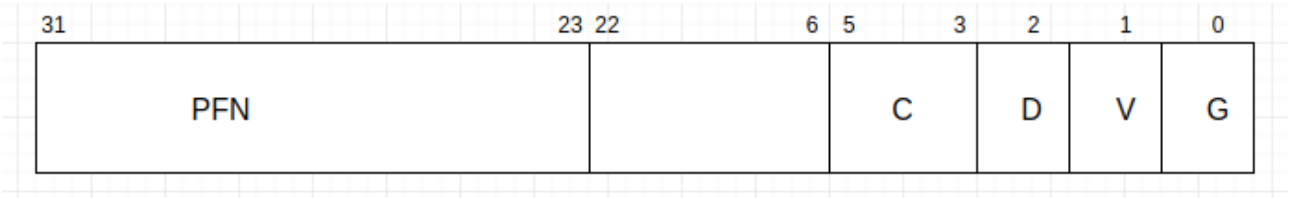

图 1: 页表项结构示意图

用 page_table_base_ptr 指向页表开始的地址，从而可以索引各个页表项。

```
1    uint32_t badvpn = ((badvaddr & 0xfffff000) >> 12);
2    uint32_t *page_table = (uint32_t *)page_table_base_ptr;
3    page_table[badvpn] = (PFN << 12) | PTE_C | PTE_D | PTE_V;
```

(2) 任务 1 和任务 2 中各自初始化了多少个页表项，以及使用了多少个物理页框保存页表？

任务 1 无 TLB 例外，所以只需要填满 32 个 TLB 项，对应 64 个页表项，用一个物理页框即可。

任务 2 需要用页表可以索引 2G 的虚拟地址空间，所以对应有 512K 页表项，一个物理页框可以容纳 1K 个页表项，所以需要 512 个物理页框来容纳页表，占用 2MB 物理空间。我的处理方式是在初始化内存的时候利用 set_up_page_table() 函数为页表分配 512 个物理页框来放置页表，并将其 pinned 在内存中，避免其换出。

(3) 如何管理物理页框？

```
typedef struct page_map_entry {
    uint32_t paddr;      //对应的物理地址
    uint32_t vaddr;      //对应的虚拟地址
    uint32_t VPN;        //对应的虚页号
    uint32_t PFN;        //物理页框号
    pid_t    pid;        //物理页框对应的进程号
    int      index;      //物理页框序列号
    bool_t   avail;      //标志物理页框是否可用
    bool_t   dirty;      //标志是否可写
    bool_t   pinned;     //标志是否pinned在内存中
    bool_t   swaped;     //标志是否换出
    int      swap_index; //索引换出到交换区的位置
    bool_t   R;          //用于页替换算法
} page_map_entry_t;

// Keep track of all page frames (in memory and in swap division): their vaddr,
    status, and other properties
static page_map_entry_t page_map[PAGEABLE_PAGES];
```

(4) 如何管理 TLB 项？

```
typedef struct tlb_entry {
    uint32_t index;      //TLB项序列号
    bool_t   empty;      //TLB项是否为空
    uint32_t VPN2;       //TLB项的VPN2域
    uint32_t PFN0;       //TLB项的PFN0域
    uint32_t PFN1;       //TLB项的PFN1域
    uint32_t pid;        //TLB项的ASID域
} tlb_entry_t;

tlb_entry_t tlb_table[TLB_ENTRIES_NUM];
```

(5) 任务 2 和任务 3 中，进程的用户态栈的起始地址各是多少，栈空间各是多大？

我的虚拟地址空间（0-2G）的设计中，1G512M 到 2G 为用户地址空间，栈空间有 512M。

(6) 任务 1 和任务 2 中你设计的操作系统实际通过页表可以访问到的物理内存有多大？

任务一有 64 个页表项，所以可以访问 $64 \times 4KB = 256KB$ 物理内存。
任务二可以访问全部 32MB 的物理内存。

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

当在 TLB 中找不到项与欲引用的地址映射匹配时发生 TLB refill 例外，当虚拟地址引用与 TLB 中某一项相匹配，但该项标记为无效时，发生 TLB invalid 例外。具体流程如下：

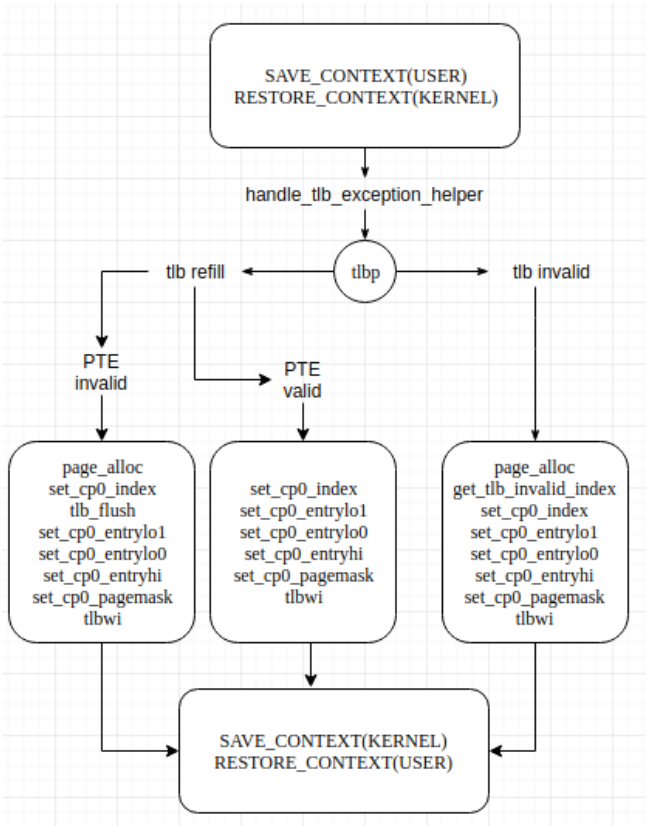

图 2: TLB miss 处理示意图

（8）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

- 有一点之前一直没有注意到,就是发生例外之后需要从用户态切换到内核态,所以除了需要 SAVE_CONTEXT(USER) 之外，还需要 RESTORE_CONTEXT(KERNEL)，将内核态与用户态作出清晰的划分。

```
1     NESTED(handle_tlb,0,sp)
2         SAVE_CONTEXT(USER)
3         RESTORE_CONTEXT(KERNEL)
4
5         jal     handle_tlb_exception_helper
6         nop
7
8         SAVE_CONTEXT(KERNEL)
9         RESTORE_CONTEXT(USER)
10
11        j       return_from_exception
12        nop
13    END(handle_tlb)
```

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，此处的物理页分配策略是什么？

有四种可能会发生缺页的情况：

- 软件访问的虚拟地址尚未建立虚拟地址映射，所以 TLB 中不包含对应的虚拟地址，页表项为空，所以会发生 TLB refill 例外，所以需要填页表项和 TLB 项
- 软件访问的虚拟地址尚未建立虚拟地址映射，页表项为空，但 TLB 中包含对应的虚拟地址（由于一个 TLB 项对应两个相邻的页表项），所以会发生 TLB invalid 例外，需要填对应的页表项和 TLB 项
- 软件访问的虚拟地址已经建立好虚拟映射，页表项不为空，物理页框不在物理内存中，TLB 中也不包含对应的虚拟地址，所以也会发生 TLB refill 例外，需要填对应的页表项，并且填任意一个 TLB 项
- 软件访问的虚拟地址已经建立好虚拟映射，页表项不为空，物理页框不在物理内存中，但 TLB 中包含对应的虚拟地址，所以会发生 TLB invalid 例外，需要填对应的页表项和 TLB 项

物理页分配策略是循环扫描物理页框数组，将扫描到的第一个 avail 位为 1 的物理页框作为分配出去的页框。如果物理内存中的所有的物理页框都不可用，就利用页替换算法将物理内存中的一个物理页框替换到 swap 区，将这个物理页框作为分配出去的物理页框。

（2）你设计中哪些页属于 pinning pages？你实现的页替换策略是怎样的？

我的实现中页表所占用的物理页框属于 pinning pages。

```
1    static void set_up_page_table()
2    {
3        bool_t pinned = TRUE;
4        int i = 0;
5        while(i < PAGE_TABLE_PAGES){
6            page_alloc(pinned,0,0);
7            i++;
8        }
9    }
```

我的页替换策略是 CLOCK 页替换算法，算法的核心思想是：把所有页框组织成环形链表，用一个表针指向最老的页，发生缺页时，按表针走动方向来检查页的 pinned 位和 R 位，跳过 pinned 位为 1 的物理页框，同时如果其 R 位为 1，将其置为 0，且表针向前移一格；如果其 R 位为 0，替换它。

3 Bonus 设计

（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？何时会触发 swap 操作？swap 操作是由专门的进程完成么？

bonus 部分通过页表访问的可用物理内存是 8MB，物理地址从 16MB 到 24M。

触发 swap 操作的情况是：发生缺页，并且软件访问的虚拟地址对应的物理页不在内存中或者内存中没有空闲页。

swap 操作没有由专门的进程完成。

（2）你设计的页替换策略是怎样的，有什么优势和不足么？

我的页替换策略是 CLOCK 页替换算法，优点是改进了 Second Chance 的替换开销，没有移动开销; 不足是如果内存很大，轮一遍需要很长时间。

具体的核心实现代码如下：

```
1     if(flag_find_free_page == 0){
2         while(page_map[clock_ptr].pinned == 1 || page_map[clock_ptr].R == 1){
3             if(page_map[clock_ptr].R == 1){
4                 page_map[clock_ptr].R = 0;
5             }
6             clock_ptr = (clock_ptr + 1) % FRAME_PAGES;
7         }
8
9         swap_out_index = clock_ptr;
10        swap_in_index = get_swap_in_index(swapin_valid) + FRAME_PAGES;
11    }
```

（3）你设计的测试用例是怎样的？

我针对可能发生 swap 操作的四种情况分别设计了测试用例，在测试用例中物理页框的大小为页表大小加上一个页面大小，swap 区的大小为 2 个页面大小。

- 软件访问的虚拟地址尚未建立虚拟地址映射，所以 TLB 中不包含对应的虚拟地址，页表项为空，所以会发生 TLB refill 例外，并且内存中没有空闲页
- 软件访问的虚拟地址尚未建立虚拟地址映射，页表项为空，但 TLB 中包含对应的虚拟地址（由于一个 TLB 项对应两个相邻的页表项），所以会发生 TLB invalid 例外，并且内存中没有空闲页
- 软件访问的虚拟地址已经建立好虚拟映射，页表项不为空，物理页框不在物理内存中，TLB 中也不包含对应的虚拟地址，所以也会发生 TLB refill 例外，并且软件访问的虚拟地址对应的物理页不在内存中
- 软件访问的虚拟地址已经建立好虚拟映射，页表项不为空，物理页框不在物理内存中，但 TLB 中包含对应的虚拟地址，所以会发生 TLB invalid 例外，并且软件访问的虚拟地址对应的物理页不在内存中

4 关键函数功能

请列出上述各项功能设计里，你觉得关键的函数或代码块，及其作用

最为关键的一个函数是 handle_tlb_exception_helper()，由于这个函数设计得较为复杂，下面只放出针对“软件访问的虚拟地址已经建立好虚拟映射，页表项不为空，物理页框在物理内存中，TLB 中也不包含对应的虚拟地址，所以会发生 TLB refill 例外，不需要填页表项，只需要填 TLB 项”这种 TLB 例外情况的处理代码：

```
1  void handle_tlb_exception_helper()
2  {
3      uint32_t cp0_pagemask = 0;
4      uint32_t EntryHi = get_cp0_entryhi();
5      uint32_t badvpn2 = ((EntryHi & 0xffffe000) >> 13);
6      uint32_t badvaddr = get_cp0_badvaddr();
7      uint32_t EntryLo_is_odd = ((badvaddr & 0x1000) >> 12);
8      uint32_t badvpn = ((badvaddr & 0xfffff000) >> 12);
9      uint32_t *page_table = (uint32_t *)page_table_base_ptr;
10     uint32_t pid = current_running->pid;
11     asm volatile("tlbp");
12     uint32_t index = get_cp0_index();
13     //TLB refill handler
14     if(((index & 0x80000000) >> 31) == 1){
15         tlb_refill_count++;
16         //check whether have the right to have access to the virtual address
17         if(check_tlb_wrong_access() == 1){
18             vt100_move_cursor(1, 45);
19             printk("ERROR! Current running pid is %d, but virtual address is in pid %d
                   address space", pid, wrong_pid);
20             return;
21         }
22         //PTE is not empty and PFN is in memory
23         if(page_table[badvpn] != 0 && ((page_table[badvpn] & PTE_V) == PTE_V)){
24             tlb_entry_index_ptr = (tlb_entry_index_ptr + 1) % TLB_ENTRIES_NUM;
25             set_cp0_index(tlb_entry_index_ptr);
26             tlb_flush_no_check();
27             uint32_t PFN = ((page_table[badvpn] & 0xfffff000) >> 12);
28             if(EntryLo_is_odd == 1){
29                 uint32_t EntryLo1 = (PFN << 6) | PTE_C | PTE_D | PTE_V;
30                 set_cp0_entrylo1(EntryLo1);
31             }
32             else{
33                 uint32_t EntryLo0 = (PFN << 6) | PTE_C | PTE_D | PTE_V;
34                 set_cp0_entrylo0(EntryLo0);
35             }
36             set_cp0_pagemask(cp0_pagemask);
37             EntryHi = ((EntryHi & 0xffffe000) | pid);
38             set_cp0_entryhi(EntryHi);
39             asm volatile("tlbwi");
```

```
40              tlb_table[tlb_entry_index_ptr].pid = current_running->pid;
41              tlb_table[tlb_entry_index_ptr].empty = 0;
42              tlb_table[tlb_entry_index_ptr].index = tlb_entry_index_ptr;
43              if(EntryLo_is_odd == 1){
44                  tlb_table[tlb_entry_index_ptr].PFN1 = ((get_cp0_entrylo1() & 0xffffffc0)
                        >> 6);
45              }
46              else{
47                  tlb_table[tlb_entry_index_ptr].PFN0 = ((get_cp0_entrylo0() & 0xffffffc0)
                        >> 6);
48              }
49              tlb_table[tlb_entry_index_ptr].VPN2 = ((get_cp0_entryhi() & 0xffffe000) >>
                    13);
50
51              return;
52          }
53      }
54  }
```