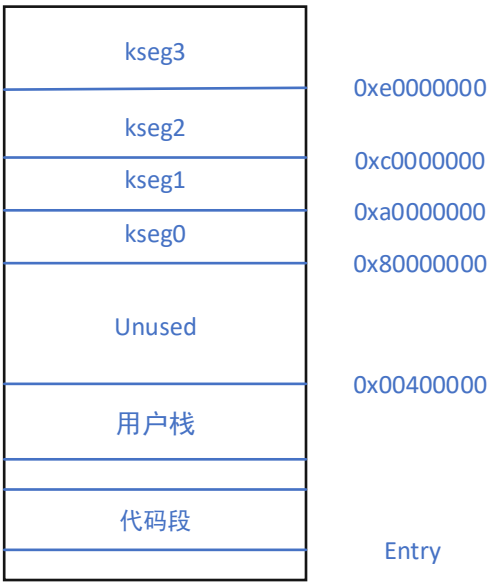
Project 5 Virtual Memory 设计文档

中国科学院大学
李云志
2017.12.24

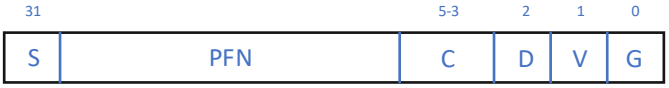
1. 用户态进程内存管理设计

(1) 用户态进程虚存布局
用户态进程虚存布局如下图：



注：基本与 MIPS 规范布局相同，但是实际任务中进程所需的页表仅为 1 页，

(2) 页表项结构是怎样的，包含哪些标记位？
页表项按照与 EntryLo 寄存器相近的布局设计，只不过 31 位设计为标记为，用于标记当前页所处位置：
1——交换区
0——物理页
其他 FLAG 位均与 MIPS 规定一致



(3) 任务 1 中用户态进程页表初始化做了哪些操作？使用了多少个页表项（PTE），以及使用了多少个物理页保存页表？
由于任务一需要实现直接分配物理页框，所以在进程页表初始化时就需要将相应的页表项填写完毕，包括 S\PFN\V 位，以及需要将相应的代码段拷贝到物理页框中。
如果没有记错的话，任务一时每个进程需要 16 个物理页框，同时需要 2 个物理页框用来保存页表。
相应的页表项也为 16 个。

(4) 任务 2 中用户态进程页表初始化做了哪些操作？使用了多少个页表项（PTE），以

及使用了多少个物理页保存页表？

由于任务二实现的为 on-demand paging，所以初始化时，并不需要为相应的进程分配物理页框，所以页表项初始化为 0 即可。

任务二时，实际上每个进程只需要一个页表项即可运行，同时物理页框只需要 4 个即可运行测试。

(5) 物理内存使用什么数据结构进行管理，描述物理内存的元数据信息有哪些，各有什么用途？此处的物理页分配策略是什么？

物理页框使用 page_map_entry_t 结构体进行管理，其包含数据信息如下：

```
typedef struct {
    uint32_t status;
    uint32_t page_ptr;
    uint8_t Read;
} page_map_entry_t;
```

Status 域用于标识此物理页框的状态（未分配、已分配、pin）

Page_ptr 域用于存储映射到此页框的页表项地址

Read 位用于 Bonus 的表针算法。

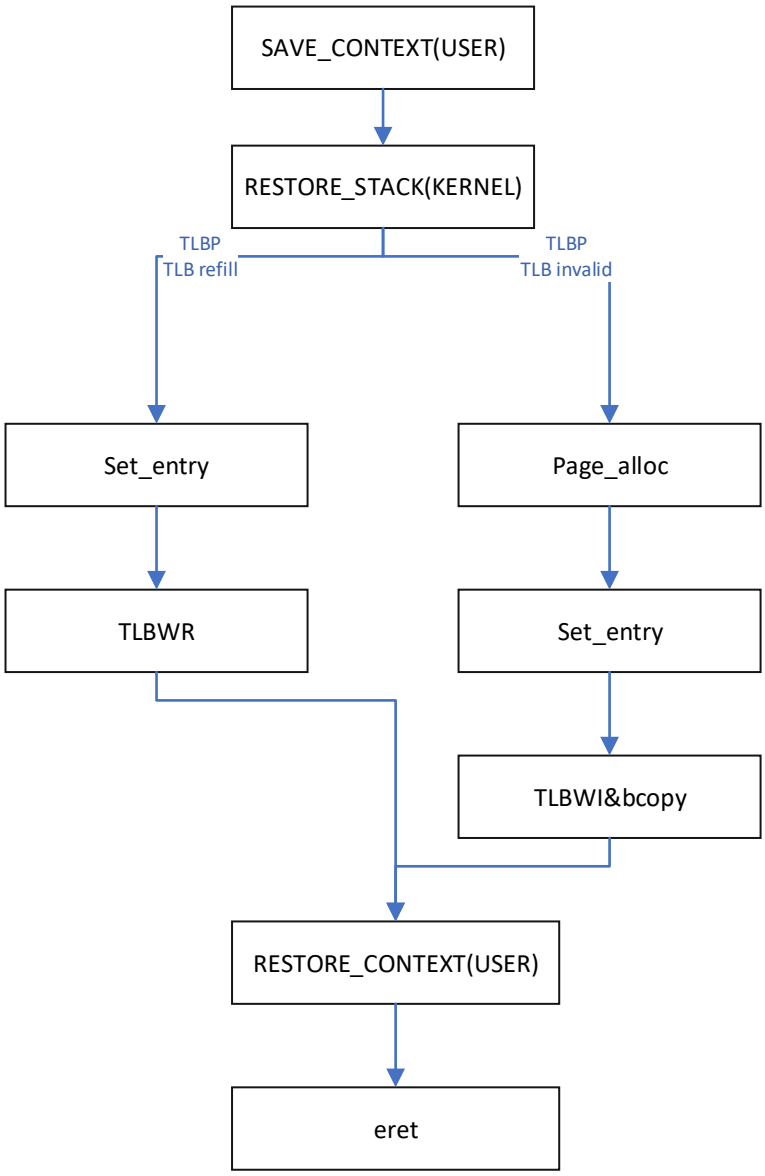
(6) TLB miss 何时发生？你处理 TLB miss 的流程是怎样的？

TLB miss 包括两种情况：

a) TLB refill

b) TLB invalid

由于 2 种例外入口点相同，所以需要手动判断，判断流程图如下：



2. 缺页中断与 swap 处理设计

(1) 任务 1 和任务 2 中是否有缺页中断？若有，何时发生缺页中断？你设计的缺页中断处理流程是怎样的？

任务 1 中不会发生缺页中断，因为初始化时需要用到的所有页已经完成分配

任务 2 中会发生缺页中断，缺页中断的处理流程对应于 TLB invalid 的处理流程，具体细节参考上图即可。

(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？

页表所处的物理页框是被 pin 住的，而其余的用户进程代码页以及用户栈均可以被交换。

Normal 任务使用的为 FIFO 策略，Bonus1 使用的表针算法，具体细节参考 Bonus 设计。

3. Bonus 设计

(1) Bonus 中你限制物理内存到多大？Bonus 任务 1 中的页替换策略是怎样的？和常规任务中的页替换策略比，Bonus 中的策略能减少页替换数量么？

物理页框限定为 5 个

使用 CLOCK 页替换策略，事实证明他是可以减少也替换数量的。

例如设计的测试程序：

访问页顺序如下：

0 => 1 => 2 => 3 => 4 => 1 => 2 => 5 => 1=>2 => 3 => 4 => 5

FIFO 替换策略运行过程如下：

0 => 1 => 2 => 3 => 4 => 1 => 2 => 5 => 1=>2 => 3 => 4 => 5

注：红色为缺页发生

CLOCK 替换策略运行过程如下：

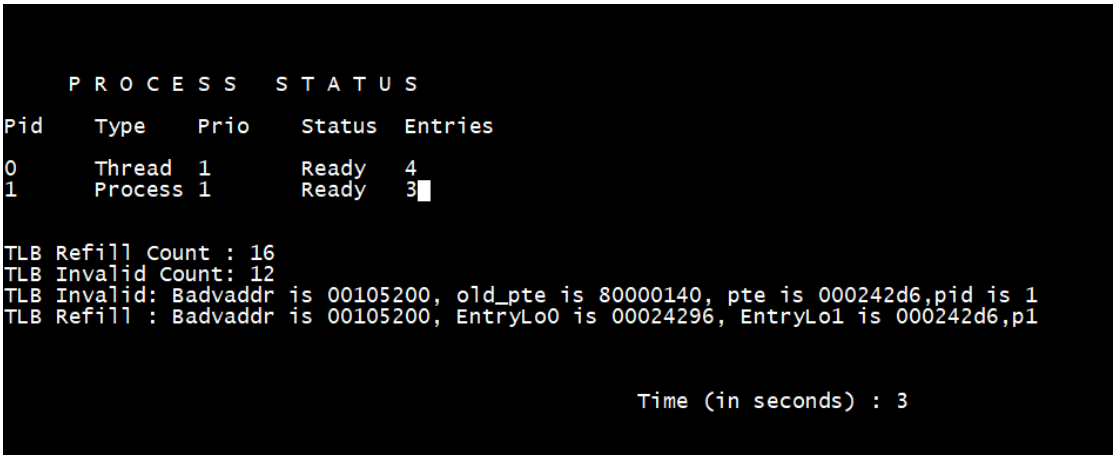
0 => 1 => 2 => 3 => 4 => 1 => 2 => 5 => 1=>2 => 3 => 4 => 5

注：红色为缺页发生

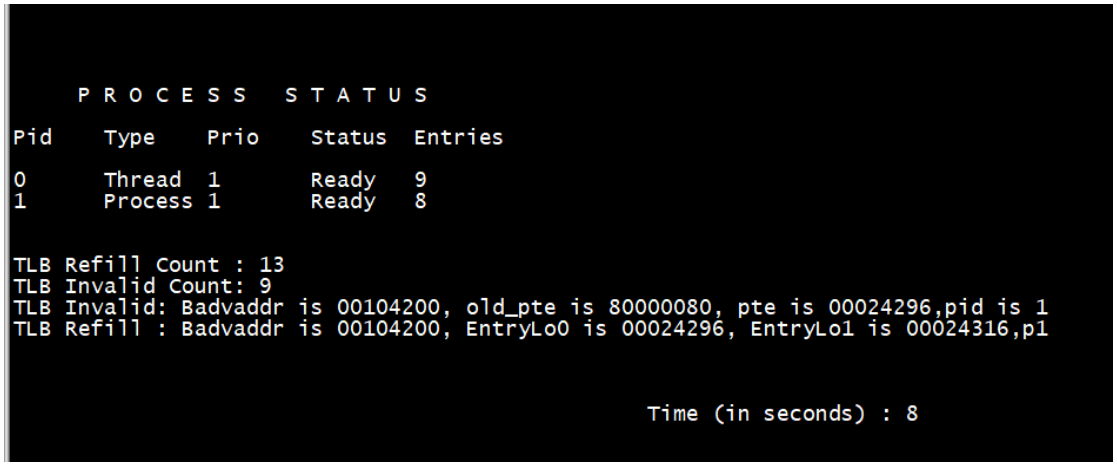
可以看到，CLOCK 替换策略比 FIFO 替换策略少发生 3 次缺页。

实际运行结果如下：

Non_clock



Clock:



4. 关键函数功能

```
uint32_t get_swap(int CLOCK) {
    ptr_page %= PAGEABLE_PAGES;
    if(CLOCK==1)
        while(page_map[ptr_page].status == PM_PINNED || page_map[ptr_page].Read) {
            if(page_map[ptr_page].Read) page_map[ptr_page].Read = 0;
            ptr_page = (ptr_page + 1) % PAGEABLE_PAGES;
        }
    else
        while(page_map[ptr_page].status == PM_PINNED) {
            if(page_map[ptr_page].Read) page_map[ptr_page].Read = 0;
            ptr_page = (ptr_page + 1) % PAGEABLE_PAGES;
        }
        return ptr_page++;
}//传入参数用于区分替换策略
```

```
int page_alloc( int pinned, uint32_t page_ptr ) {
    int free_index = PAGEABLE_PAGES, i;
    free_index = get_swap(1);
    ASSERT( free_index < PAGEABLE_PAGES );

    if(page_map[free_index].status != PM_FREE) {
        int swap_index = swap_alloc();
        *(uint32_t*)(page_map[free_index].page_ptr) = (1 << 31) | (swap_index << 6);
        tlb_flush2();
        bcopy(page_vaddr(free_index), swap_vaddr(swap_index), PAGE_SIZE);
    }
    if(pinned) {
        page_map[free_index].status = PM_PINNED;
        page_map[free_index].page_ptr = 0;
    } else {
        page_map[free_index].status = PM_ALLOCATED;
        page_map[free_index].page_ptr = page_ptr;
    }
    page_map[free_index].Read = 0;
    return free_index;
}
```

```
uint32_t setup_page_table( int pid ) {
    uint32_t page_table;
    // alloc page for page table
    int free_index = page_alloc(1, 0);
    page_table = page_vaddr(free_index);
    // initialize PTE and insert several entries into page tables using
insert_page_table_entry
    bzero(page_table, PAGE_SIZE);
    return page_table;
}
```

```
void handle_tlb_c(void){
    enter_critical();
    uint32_t EntryHi = get_cp0_EntryHi();
    uint32_t Badvpn2 = EntryHi >> 13;
    uint32_t Badvaddr = get_cp0_Badvaddr();
    uint32_t entrylo_offset = (Badvaddr & 0x1000) >> 12;
    ASSERT2((Badvpn2 << 3) < PAGE_SIZE, "Virtual Address Overflow !");
    uint32_t* p_pte = (uint32_t*)(current_running->page_table + (Badvpn2 << 3));
    asm volatile("tlbp");
    if((signed)get_cp0_index() >= 0) {
        tlb_invalid_count++;
        // ASSERT2(FALSE, "TLB Invalid !");
        int free_index;
        uint32_t v_begin, loc_begin, offset, old_pte, pte;
        free_index = page_alloc(0, (uint32_t)(p_pte + entrylo_offset));
        old_pte = *(p_pte + entrylo_offset);
        pte = ((page_paddr(free_index) >> 6) & 0xffffffc0) | PE_UC | PE_D | PE_V;
        *(p_pte + entrylo_offset) = pte;
        if(entrylo_offset) set_cp0_EntryLo1(*(p_pte + entrylo_offset));
        else set_cp0_EntryLo0(*(p_pte + entrylo_offset));
        asm volatile("tlbwi");
        if(old_pte & 0x80000000) {
            bcopy(swap_vaddr((old_pte & 0x7fffffff) >> 6), page_vaddr(free_index),
PAGE_SIZE);
        } else {
            v_begin = current_running->entry_point & 0xffffe000;
            loc_begin = current_running->loc - (current_running->entry_point - v_begin);
            offset = ((Badvpn2 << 13) | (entrylo_offset << 12)) - v_begin;
            if(offset < current_running->size + (current_running->entry_point -
v_begin))
                bcopy(loc_begin + offset, page_vaddr(free_index), PAGE_SIZE);
        }
```

```
        printf(19,1,"TLB Invalid: Badvaddr is %x, old_pte is %x, pte is %x,pid is %d",
Badvaddr, old_pte, pte,EntryHi & 0xff);
    } else {
        tlb_refill_count++;
        set_cp0_EntryLo0(*(p_pte));
        set_cp0_EntryLo1(*(p_pte + 1));
        asm volatile("tlbwr");
        printf(20,1,"TLB Refill : Badvaddr is %x, EntryLo0 is %x, EntryLo1 is %x,pid
is %d",  Badvaddr, *(p_pte), *(p_pte + 1),EntryHi & 0xff);
        if(*(p_pte + entrylo_offset) & 0x2)
        set_Read(*(p_pte + entrylo_offset));
    }
    printf(17,1,"TLB Refill Count : %d", tlb_refill_count);
    printf(18,1,"TLB Invalid Count: %d", tlb_invalid_count);
    return;}
```

感想：

这次实验真的是对调试能力以及忍耐能力的极大考验，最开始完成此实验的时候，由于发现时间根本不够了，所以打算先不实现用户栈了，所以直接进行了页替换部分的编写，幸好老师又给了我们几天喘息的机会，所以我打算先把页替换部分写完再去完成用户栈部分，但是没想到，这一下又花费了一天的时间，最后发现实验板的行为和预期根本不一致。明明完成了 TLB 表项的填写，但是仍然发生 TLB miss，甚至通过 TLBP 都可以查找到相应的页表项。经过了解还有很多同学也遇到了这个问题，最后不得不从头来过，完全转变思路，所以也附上前一版本未完成的代码（v1.0），来纪念我的时间。

参考文献

[1] [单击此处键入参考文献内容]