Project 4 Virtual Memory 设计文档

中国科学院大学
贾志杰
2020 年 12 月 8 日

1. 内存管理设计

（1）你设计的页表是几级页表，最大能索引到多大的物理空间？页表项的数据结构是什么？页表本身的数据结构是什么？ 你设计的页表需要几个物理页框保存？

页表为一级页表，每个进程支持 0x0 - 0x7fffffff 的虚地址空间，所有用户进程共同索引 0x20000000 - 0x3fffffff 的物理地址空间。

页表项的数据结构为：

```
typedef struct PTE{
    int VPN2;
    ENTRYLO_t entrylo[2];
    int disk_addr;
    char ASID;
    char setup;
    char inmemory;
} PTE_t;
```

其中 ENTRYLO_t 结构体为：

```
typedef struct ENTRYLO{
    int G : 1;
    int V : 1;
    int D : 1;
    int C : 3;
    int PFN : 24;
    int PFNX : 2;
} ENTRYLO_t;
```

页表本身的数据结构是一个页表项类型的数组。但是实际使用中我们不需要分配一个大数组，只需要规定好各页表的首地址即可。

一个页表大小为 5MB，需要 640 个物理页框保存。

（2）在 A-Core 中，你处理 TLB miss 的流程是怎样的？

TLB miss 即找不到 TLB 项，触发 TLB refill 例外。TLB 重填根据 context 寄存器中的触发例外的虚页号，找到对应的页表项，查看其 setup 和 inmemory 标志位。如果都为 1，说明该虚地址已经建立虚实映射，且对应的物理页框在内存中，那么将页表项填入 TLB。如果不全为 1，有多种可能情况，但是 TLB 重填不需要处理和区分它们，此时只将页表项的 entryhi 部分填入 TLB，entrylo 部分填入 0，这样之后再次执行该指令时触发 TLB invalid 的例外，届时再交给缺页处理程序来区别和处理各种情况。

（3）设计或实现过程中遇到的问题和得到的经验

一开始设计页表项的数据结构时，entrylo 的各项都使用了一个单独的 int 型，这样页表项的大小很大，页表占据较多的内存。后来学习并使用了结构体定制比特位的方法，使用一个 int 类型来表示一个 entrylo 项，并且把 setup 之类的标志位缩小到 char 类型大小。这样节省了 2/3 的页表项空间，减小了页表的内存开销。

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？

在 TLB 中查找到虚地址对于的 TLB 项，但是其 V 位为 0，触发 TLB invalid 例外，这时需要缺页处理。缺页有 2 种可能，一是该虚地址未建立虚实映射，二是已经建立虚实映射，但对于的物理页框不在内存而在磁盘中。缺页处理程序首先根据虚页号找到页表项，查看其 setup 和 inmemory 标志位确定是哪种缺页情况。第一种为其分配物理页框，完善页表项，查找 TLB 中对应的 TLB 项，用新的页表项信息更新该 TLB 项。第二种也为其分配一个物理页框，然后将磁盘中的页框移到内存中，更新页表项，查找 TLB 中对应的 TLB 项，用新的页表项信息更新该 TLB 项。

（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？

物理页框的数据结构：

```
typedef struct PF{
    int setup;
    int swap_times;
    int ins;
    int VPN2;
    int ASID;
    int PFN[2];
} PF_t;
```

用户进程使用 0x20000000 - 0x3fffffff 的物理地址空间，在部分空间上管理物理页框。物理地址低 0.5GB 则由内核所在的 unmapped 段直接映射，不需要管理。

缺页分配时从上次分配的 index 处开始，遍历物理页框找到第一个空闲的物理页框，将其分配给该进程，更新 index。当进程结束时回收其使用的物理页框，将其标记为空闲。

（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？

没有完全钉住的物理页框，因为内核使用的物理页是 unmapped，硬件直接映射的。但是对于用户进程来说，在它需要换页时，会优先选择自己的页框。

3. C-Core 设计

（1）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？

为了能够达到换页的条件，实际通过页表映射可访问的物理页框只有 14 个。Swap 由一个专门的 IO 进程完成 SD 卡的读写。

（2）你设计的页替换策略是怎样的，有什么优势和不足么？

采用 FIFO 的替换策略，同时规定了换页时进程优先换自己的页框，而不是其他进程的页框。优势在于避免不同进程交叉换页造成干扰、甚至把 IO 进程自己的页框换出而导致死锁，并且实现该算法的开销较小。不足在于最先添加的页但频繁使用的页，比如栈和代码段，也会被换出，造成性能下降。

（3）你设计的测试用例是怎样的？

测试程序循环访问 10 个不同的页，由于 shell 和 IO 进程，还有该测试进程的栈和代码段需要页框，因此实际需要的页框大于实际可用的 14 个物理页框，这时就会不断地循环换页。在换页时记录每个物理页框的换页次数，测试程序每循环访问一次就打印一次所以页框的替换次数，观察替换次数来查看换页的效果。

（4）设计或实现过程中遇到的问题和得到的经验

SD 卡读写函数的内存地址参数都是虚地址，在读写之前需要先建立好虚实映射，填充 TLB，之后才能正常运行 SD 读写函数。

4. 关键函数功能

（1）TLB refill 函数

```
void do_TLB_Refill(uint64_t entryhi0, uint64_t context)
{
    int asid = (entryhi0 & 0xff);
    int badVPN2 = (context & 0x7fffff) >> 4;
    int i = badVPN2 - (USER_VADDR_BASE>>13);
    uint64_t entryhi, entrylo0, entrylo1;
    int ptid = proc_ptn[asid];

    if(page_table[ptid][i].setup == 1){
        if(asid == page_table[ptid][i].ASID){
            if(page_table[ptid][i].inmemory == 1){
                entryhi = ((uint64_t)(page_table[ptid][i].VPN2) << 13) | ((uint64_t
)(page_table[ptid][i].ASID) & 0xff);
                entrylo0 = *(int*)&(page_table[ptid][i].entrylo[0]);
                entrylo1 = *(int*)&(page_table[ptid][i].entrylo[1]);
                if(swap_reg.swap_index == count && swap_reg.swap_valid == 1)
                    count = (count + 1) % NUM_TLB;
                set_a_tlb_entry(entryhi, entrylo0, entrylo1, count);
            }
            else{
                page_table[ptid][i].entrylo[0].V = 0;
                page_table[ptid][i].entrylo[1].V = 0;
                entryhi = ((uint64_t)(page_table[ptid][i].VPN2) << 13) | ((uint64_t
)(page_table[ptid][i].ASID) & 0xff);
                if(swap_reg.swap_index == count && swap_reg.swap_valid == 1)
                    count = (count + 1) % NUM_TLB;
```

```
                set_a_tlb_entry(entryhi, 0, 0, count);
            }
        }
        else{
            protection_fault(asid, page_table[ptid][i].ASID, ptid);
        }
    }
    else{
        page_table[ptid][i].VPN2 = badVPN2;
        page_table[ptid][i].ASID = asid;
        page_table[ptid][i].entrylo[0].V = 0;
        page_table[ptid][i].entrylo[1].V = 0;
        page_table[ptid][i].entrylo[0].G = 0;
        page_table[ptid][i].entrylo[1].G = 0;
        entryhi = ((uint64_t)(page_table[ptid][i].VPN2) << 13) | ((uint64_t)(page_t
able[ptid][i].ASID) & 0xff);
        if(swap_reg.swap_index == count && swap_reg.swap_valid == 1)
            count = (count + 1) % NUM_TLB;
        set_a_tlb_entry(entryhi, 0, 0, count);
    }
    count = (count + 1) % NUM_TLB;
}
```

（2）TLB invalid/缺页处理函数

```
void do_page_fault(int badVPN2, int asid)
{
    int i = badVPN2 - (USER_VADDR_BASE>>13);
    uint64_t entryhi, entrylo0, entrylo1, index;
    int ptid = proc_ptn[asid];
    int pf_num;

    if(page_table[ptid][i].setup == 0){
        pf_num = get_free_pf();
        if(pf_num >= 0){
            page_table[ptid][i].setup = 1;
            page_table[ptid][i].inmemory = 1;
            physical_frame[pf_num].setup = 1;
            physical_frame[pf_num].VPN2 = badVPN2;
            physical_frame[pf_num].ASID = asid;
            page_table[ptid][i].entrylo[0].PFN = physical_frame[pf_num].PFN[0];
            page_table[ptid][i].entrylo[1].PFN = physical_frame[pf_num].PFN[1];
            page_table[ptid][i].entrylo[0].C = 2;
            page_table[ptid][i].entrylo[1].C = 2;
            page_table[ptid][i].entrylo[0].D = 1;
```

```
            page_table[ptid][i].entrylo[1].D = 1;
            page_table[ptid][i].entrylo[0].V = 1;
            page_table[ptid][i].entrylo[1].V = 1;
            page_table[ptid][i].entrylo[0].G = 0;
            page_table[ptid][i].entrylo[1].G = 0;

            entryhi = ((uint64_t)(page_table[ptid][i].VPN2) << 13) | ((uint64_t)(pa
ge_table[ptid][i].ASID) & 0xff);
            entrylo0 = *(int*)&(page_table[ptid][i].entrylo[0]);
            entrylo1 = *(int*)&(page_table[ptid][i].entrylo[1]);
            index = find_index(entryhi);
            set_a_tlb_entry(entryhi, entrylo0, entrylo1, index);
        }
        else{
            swap(asid);
        }
    }
    else if(page_table[ptid][i].inmemory == 0){
        pf_num = get_free_pf();
        if(pf_num >= 0){
            current_running->status = TASK_BLOCKED;
            current_running->inqueue = BLOCK;
            pcb[PROC_SWAP].block_task[++(pcb[PROC_SWAP].block_task_count)] = curren
t_running->pid;
            if(swap_reg.swap_inwork){
                do_scheduler2();
                return;
            }
            page_table[ptid][i].inmemory = 1;
            physical_frame[pf_num].setup = 1;
            physical_frame[pf_num].VPN2 = badVPN2;
            physical_frame[pf_num].ASID = asid;
            page_table[ptid][i].entrylo[0].PFN = physical_frame[pf_num].PFN[0];
            page_table[ptid][i].entrylo[1].PFN = physical_frame[pf_num].PFN[1];
            page_table[ptid][i].entrylo[0].V = 1;
            page_table[ptid][i].entrylo[1].V = 1;

            entryhi = ((uint64_t)(page_table[ptid][i].VPN2) << 13) | ((uint64_t)(pa
ge_table[ptid][i].ASID) & 0xff);
            index = find_index(entryhi);
            entryhi = ((uint64_t)(page_table[ptid][i].VPN2) << 13) | ((uint64_t)(PR
OC_SWAP) & 0xff);
            entrylo0 = *(int*)&(page_table[ptid][i].entrylo[0]);
            entrylo1 = *(int*)&(page_table[ptid][i].entrylo[1]);
```

```
            set_a_tlb_entry(entryhi, entrylo0, entrylo1, index);

            free_disk_pf(page_table[ptid][i].disk_addr);
            swap_reg.swap_addr = (uint64_t)(page_table[ptid][i].VPN2<<13);
            swap_reg.swap_disk_addr = page_table[ptid][i].disk_addr;
            swap_reg.swap_rw = 0;
            swap_reg.swap_i = i;
            swap_reg.swap_asid = asid;
            swap_reg.swap_index = index;
            swap_reg.swap_valid = 1;
            swap_reg.swap_inwork = 1;
            vt100_move_cursor(1, 1);
            printk("[swap read] ptid: %d, num: %d, vpn: 0x%x, asid: %d, pt_vpn: 0x%
x, pt_asid: %d, disk_addr: 0x%x", ptid, pf_num, (physical_frame[pf_num].VPN2<<13),
physical_frame[pf_num].ASID, (page_table[ptid][i].VPN2 << 13), page_table[ptid][i].
ASID, page_table[ptid][i].disk_addr);
            do_unblock_one(&pcb[PROC_SWAP]);
            do_scheduler2();
        }
        else{
            swap(asid);
        }
    }
}
```

（3）TLB modify 处理函数

```
void do_tlb_modify(int badVPN2, int asid)
{
    int i = badVPN2 - (USER_VADDR_BASE>>13);
    uint64_t entryhi, entrylo0, entrylo1, index;
    int ptid = proc_ptn[asid];

    page_table[ptid][i].entrylo[0].D = 1;
    page_table[ptid][i].entrylo[1].D = 1;
    entryhi = ((uint64_t)(page_table[ptid][i].VPN2) << 13) | ((uint64_t)(page_table
[ptid][i].ASID) & 0xff);
    entrylo0 = *(int*)&(page_table[ptid][i].entrylo[0]);
    entrylo1 = *(int*)&(page_table[ptid][i].entrylo[1]);
    index = find_index(entryhi);
    set_a_tlb_entry(entryhi, entrylo0, entrylo1, index);
    printk("error: tlb_modify badVPN2: %d ptid: %d", badVPN2, ptid);
}
```

（4）swap 换页处理函数

```
void swap(int asid)
{
    int i, ptid;
    uint64_t entryhi, entrylo0, entrylo1, index = -1;
    int disk_base;
    uint64_t mem_addr;

    if(asid == PROC_SWAP){
        vt100_move_cursor(1, 2);
        printk("[swap error] proc swap lost page ");
        uint64_t badvddr= get_cp0_badvaddr();
        int bd_hi = (int)(badvddr>>32);
        int bd_lo = (int)(badvddr & 0xffffffff);
        printk("badvaddr: 0x%x %x", bd_hi, bd_lo);
    }
    current_running->status = TASK_BLOCKED;
    current_running->inqueue = BLOCK;
    pcb[PROC_SWAP].block_task[++(pcb[PROC_SWAP].block_task_count)] = asid;
    if(swap_reg.swap_inwork){
        do_scheduler2();
        return;
    }
    find_page_out(asid);
    i = physical_frame[page_out].VPN2 - (USER_VADDR_BASE>>13);
    ptid = proc_ptn[physical_frame[page_out].ASID];
    entryhi = ((uint64_t)(page_table[ptid][i].VPN2) << 13) | ((uint64_t)(page_table
[ptid][i].ASID) & 0xff);
    index = find_index(entryhi);
    set_a_tlb_entry(entryhi, 0, 0, index);

    entryhi = ((uint64_t)(page_table[ptid][i].VPN2) << 13) | ((uint64_t)(PROC_SWAP)
 & 0xff);
    entrylo0 = *(int*)&(page_table[ptid][i].entrylo[0]);
    entrylo1 = *(int*)&(page_table[ptid][i].entrylo[1]);
    set_a_tlb_entry(entryhi, entrylo0, entrylo1, ++index);
    page_table[ptid][i].inmemory = 0;
    page_table[ptid][i].entrylo[0].V = 0;
    page_table[ptid][i].entrylo[1].V = 0;
    swap_reg.swap_addr = (uint64_t)(page_table[ptid][i].VPN2<<13);
    swap_reg.swap_disk_addr = new_disk_pf_addr();
    page_table[ptid][i].disk_addr = swap_reg.swap_disk_addr;
    swap_reg.swap_rw = 1;
    swap_reg.swap_asid = asid;
    swap_reg.swap_index = index;
```

```
    swap_reg.swap_valid = 1;
    swap_reg.swap_inwork = 1;
    vt100_move_cursor(1, 1);
    printk("[swap write] ptid: %d, num: %d, vpn: 0x%x, asid: %d, pt_vpn: 0x%x, pt_a
sid: %d, disk_addr: 0x%x", ptid, page_out, (i<<13), physical_frame[page_out].ASID,
(page_table[ptid][i].VPN2 << 13), page_table[ptid][i].ASID, page_table[ptid][i].dis
k_addr);
    do_unblock_one(&pcb[PROC_SWAP]);
    do_scheduler2();
}
```

（5）换页 IO 进程

```
void __attribute__((section(".entry_function"))) _start(void)
{
    uint64_t mem_buff;
    uint64_t disk_base;
    int mode, i;

    while(1){
        sys_get_swapreg(&swap_reg);
        if(!swap_reg.swap_valid){
            sys_block();
            sys_get_swapreg(&swap_reg);
        }
        mem_buff = swap_reg.swap_addr;
        disk_base = swap_reg.swap_disk_addr;
        mode = swap_reg.swap_rw;
        if(mode == 1){
            sys_sdwrite(mem_buff, disk_base, 0x2000);
            sys_update_pf();
        }
        else{
            sys_sdread(mem_buff, disk_base, 0x2000);
            sys_update_pf();
        }
    }
    sys_exit();
}
```

（6）换页测试程序

```
void __attribute__((section(".entry_function"))) _start(void)
{
    uint64_t swap_task_base = 0x100000;
    int i, j, k;
```

```
    uint32_t print_location = 1;
    for(i=0; i<500; i++){
        for(k=0; k<10; k++){
            *(int *)(swap_task_base + k*0x2000) = 1;
        }
        for(j=0; j<NUM_PF; j++){
            sys_get_pfnum(pf_swap_num);
            sys_move_cursor(1, print_location + j);
            printf("physical frame[%d]: swap times: %d ", j, pf_swap_num[j]);
        }
    }
    sys_exit();
}
```

（7）页表和物理页框回收

```
void release_page_table(pid_t pid)
{
    int i;
    int ptid = proc_ptn[pid];
    for(i=0; i<NUM_PAGE_TABLE_ENTRY; i++){
        page_table[ptid][i].setup = 0;
        page_table[ptid][i].inmemory = 0;
    }
    page_table_free[ptid] = 1;
    proc_ptn[pid] = -1;
    for(i=0; i<NUM_PF; i++){
        if(physical_frame[i].ASID == pid){
            physical_frame[i].setup = 0;
            physical_frame[i].ins = 0;
        }
    }
}
```

（8）共享内存相关函数

```
uint64_t do_shmat(uint64_t shmid)
{
    if(shmid >= NUM_SHARE_PF)
        return -1;
    int i = share_pf[shmid].VPN2;
    uint64_t entryhi, entrylo0, entrylo1;
    int ptid = proc_ptn[current_running->pid];
    if(page_table[ptid][i].setup == 0){
        page_table[ptid][i].setup = 1;
        page_table[ptid][i].inmemory = 1;
```

```
        page_table[ptid][i].VPN2 = i;
        page_table[ptid][i].ASID = current_running->pid;
        page_table[ptid][i].entrylo[0].PFN = share_pf[shmid].PFN[0];
        page_table[ptid][i].entrylo[1].PFN = share_pf[shmid].PFN[1];
        page_table[ptid][i].entrylo[0].C = 2;
        page_table[ptid][i].entrylo[1].C = 2;
        page_table[ptid][i].entrylo[0].D = 1;
        page_table[ptid][i].entrylo[1].D = 1;
        page_table[ptid][i].entrylo[0].V = 1;
        page_table[ptid][i].entrylo[1].V = 1;
        page_table[ptid][i].entrylo[0].G = 0;
        page_table[ptid][i].entrylo[1].G = 0;
        entryhi = ((uint64_t)(page_table[ptid][i].VPN2) << 13) | ((uint64_t)(page_t
able[ptid][i].ASID) & 0xff);
        entrylo0 = *(int*)&(page_table[ptid][i].entrylo[0]);
        entrylo1 = *(int*)&(page_table[ptid][i].entrylo[1]);
        if(swap_reg.swap_index == count && swap_reg.swap_valid == 1)
            count = (count + 1) % NUM_TLB;
        set_a_tlb_entry(entryhi, entrylo0, entrylo1, count);
        count = (count + 1) % NUM_TLB;
        return (((uint64_t)share_pf[shmid].VPN2)<<13);
    }
    else{
        return -1;
    }
}
uint64_t do_shmdt(uint64_t addr)
{
    int i;
    int vpn2 = (addr >> 13);
    uint64_t entryhi, index;
    int ptid = proc_ptn[current_running->pid];
    for(i=0; i<NUM_SHARE_PF; i++){
        if(share_pf[i].VPN2 == vpn2){
            share_pf[i].setup--;
            page_table[ptid][vpn2].setup = 0;
            page_table[ptid][vpn2].inmemory = 0;
            entryhi = ((uint64_t)(page_table[ptid][vpn2].VPN2) << 13) | ((uint64_t)
(page_table[ptid][vpn2].ASID) & 0xff);
            index = find_index(entryhi);
            set_a_tlb_entry(0, 0, 0, index);
            return 0;
        }
    }
```

```
    return -1;
}
```

参考文献

[1] MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set Reference Manual, Revision 6.05, 2016

[2] 龙芯 GS264 处理器核用户手册, v1.0, 2018: 49-54, 78-91