Project 5 Virtual Memory 设计文档

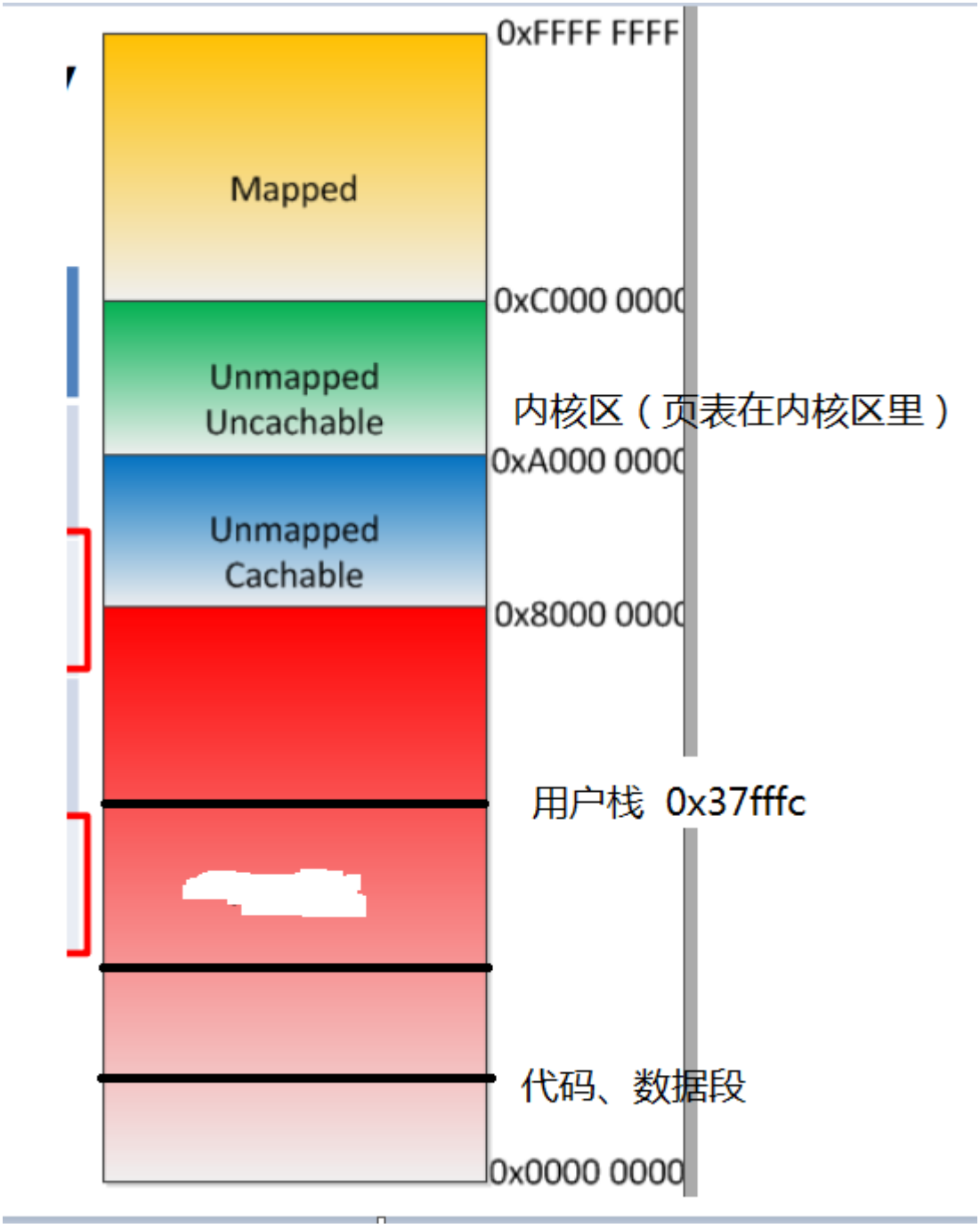
中国科学院大学
[王苑铮]
[2017.12.24]

因为周日才做完，马上提交就截止了，所以这个文档写的稍微草了一点

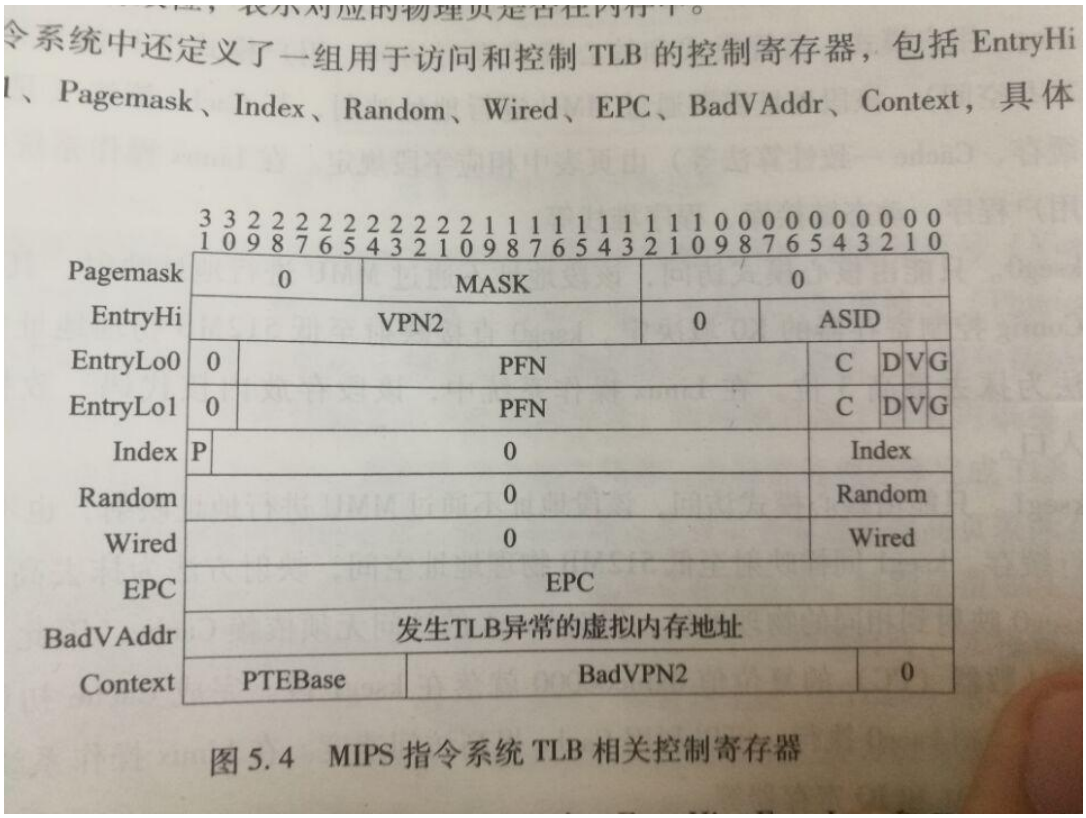
1. 用户态进程内存管理设计

请至少包含以下内容

（1） 测试的用户态进程虚存布局是怎样的？请说明



（2）你设计的页表项结构是怎样的，包含哪些标记位？



页表项：与图中的 EntryLo0/1 是一样的，用一个 unit32 来存储，所以没专门定义这个结构体。PFN：物理页框号。标记位：C:使用的 cache 算法（体系结构的课本上讲的，本次实验不用管），D：这一个物理页是否被写过（dirty），V：这页是否在内存中（valid），G：是否全局匹配。若 G 为 1 则匹配 tlb 项时不看进程号（asid）

（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？

跳过 task1 直接做的 task2

（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？

```
uint32_t setup_page_table( int pid ) {
    uint32_t index = page_alloc(TRUE);

    uint32_t page_dir = pa2va(page_map[index].paddr);
    page_map[index].pid = pid;
    page_map[index].vaddr = page_dir;
    bzero((char*)page_dir, 0x1000);

    // flush entry point in TLB
    return page_dir;
}
```

操作：找到一个可用的物理页框（如果所有物理页框都被占用了，就进行页替换）用来放页表。将这个页框标上这个进程的 pid，标上这个进程页表的虚拟地址。现在这个物理页框里存的就是这个进程页表了（如果进程页表需要多个页才能放下，则这是页表第一个页）。把页表项全部刷为 0，返回这个页表的虚地址赋给 pcb 里对应的域

物理页：一开始只有一个物理页用来保存页表

页表项：一页大小 4KB，一个页表项是 unit32,即 4B，故有 1K 个页表项

（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？

```
typedef struct {
    // design here
    uint32_t VPN;
    uint32_t PFN;
    uint32_t paddr;
    uint32_t vaddr;
    pid_t pid;
    bool_t unused;
    bool_t dirty;
    bool_t pinned;
} page_map_entry_t;
```

VPN:这个物理页框对应的虚拟页号

PFN：物理页框号

paddr：物理页起始物理地址

vaddr：这个物理页框对应的虚拟页的起始虚拟地址

pid：使用这个物理页框的进程的 pid

unused：没有 task 正在使用这个物理页框

dirty：这个物理页框里的虚拟页被写过，换出时需要写回 disk

pinned：这个物理页框里面的虚拟页被 pin 住，不可以替换出去

分配策略：需要物理页框时，先遍历物理页框，如果找到一个 unused && !pinned 物理页框，就分配这个物理页框，否则找到一个!pinned 物理页框，如果 dirty 就先写回内存。把这个物理页框分配出去。

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

tlb miss：要访问的虚拟地址在非映射区，且 tlb 里没有这个进程的这个虚拟页对应的 tlb 表项，这时 tlb 就会通过硬件发出 tlb miss 异常

```
NESTED(handle_tlb,0,sp)
  // find the bad virtual address
  jal   handle_tlb_c
  nop
  
  la  t0, current_running
  lw  t0, 0(t0)
  lw  t0, PTBASE(t0)        // t0 = page table
  la  t1, current_running
  lw  t1, 0(t1)
  lw  t7, PID(t1) // t7 = pid
  mfc0  t1, CP0_BADVADDR    // t1 = badvaddr
  srl t1, 13
  sll t1, 13  // t1 = VPN2
  or  t3, t1, t7  // EntryHi
  mtc0  t3, CP0_ENTRYHI
  srl t1, 10  // t1 to be the offset in page table
  add t0, t0, t1
  lw  t6, (t0)  // t6 = even page entry
  lw  t5, 4(t0)     // t5 = odd page entry
  mtc0  t6, CP0_ENTRYLO0
  mtc0  t5, CP0_ENTRYLO1
  mtc0  zero, CP0_PAGEMASK
  tlbwr
  j return_from_exception
  nop
END(handle_tlb)
```

```c
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* PTEntry = (uint32_t*)(*((uint32_t*)(PTDItem & 0xfffff000)));
    uint32_t PTEntryItem = PTEntry[(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_point;
    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);
            PTEntry[(badvaddr>>12) & 0x3ff] = ((page_paddr(index)+0xa0000000)) | 0xc00;

            if(!(PTBase[VPN] & PE_V)){
                printf(line++, 1, "Memory Allocation %x", current_running->user_tf.cp0_badvaddr);

                index = page_alloc(badvaddr >= 0x300000);
                PTBase[VPN] = (page_paddr(index) & 0xfffff000) >> 6 | (PE_V | PE_D);
                uint32_t entry_hi = ((VPN << 12) & 0xfffe000) | (current_running->pid & 0xff);
                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);
            }
        }
    }
}
```

处理 tlb miss：在 handle_tlb_c 中，在进程页表里配置好要填进 tlb 的这个页表项（包括分配物理页、把物理页框号和 flag 填进页表项），然后在汇编代码中，找到这个页表项填入 entryhi，entrylo 寄存器（奇偶两个连续的页表项）。再由触发异常的虚拟地址 badvaddr 配置好 enrtyhi 寄存器，然后把配置好的 pagemask，entryhi，entrylo0，entrylo1 用 tlbwr 随机写进 tlb 里的某个表项，然后进行例外返回

无论 tlb miss 还是 page fault 都是走这两个函数进行处理的

（7）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）

2. 缺页中断与 swap 处理设计

请至少包含以下内容

（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？

直接做的 task2。如果 tlb 项里有虚拟地址对应的 tlb 项，但是 invalid 位为 0，就会触发 page fault。从另一个角度看，就是 tlb 中有这个虚拟页的项，但是物理页框中没有，就会触发 page fault

处理流程见上面的 tlb miss，都是走的上面那两个函数

（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？

储存页表的页是 pin 的。页替换策略一开始是找到第一个 unpinned 页就把它替换掉。bonus 里用了 clock 的替换方法

（3）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）

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？

用了 clock 的替换方法，把一个页替换后，下次从这个页后面的页开始找用于替换的页。可以减少页替换的次数，因为被用过的页可能有更大概率会被再次使用，这种方法可以让这种页在物理页框里面停留更久，下次再用到这个页时就不会发生 miss 或 page fault，减少了页替换次数

```
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;
}
```

课本上这个算法是用循环链表实现的。但是由于之前的 task 里我们用一个数组来管理物理页框，所以这里为了简单起见，我们继续用之前的数组，并且用 clock 角标来代替双向循环链表中的表针指针。

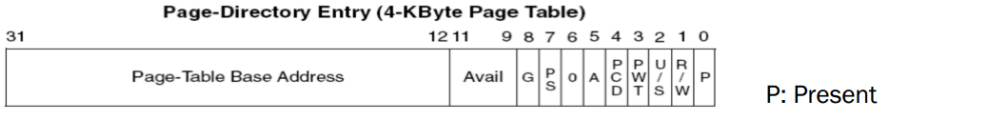
（2） Bonus 任务 2 的二级页表中，你设计了多少个页表项（PTE），使用了多少物理页保存二级页表，和常规任务中的一级线性页表相比，使用物理页数有减少么？二级页表的页目录项结构是怎样的，包含哪些信息？

依然是每个页有 1K 个页表项，页目录里也是一页有 1K 个项

两个页放页目录，两个页放页表

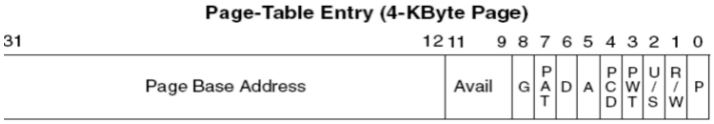
因为时间不够，没有设计新的实验用例。所以在运行原有的用例的时，由于又引入了页目录，所以需要的物理页数反而变多了。

- Two-level page table
  - Page directory entry



P: Present
R/W: Read/Write
U/S: User/Supervisor
D: Dirty

  - Page table entry



二级页表页表项按这个设计的。

（3）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）

4. 关键函数功能

关键代码已经贴在前面了

参考文献

[1] [单击此处键入参考文献内容]