Project 5 Virtual Memory 设计文档

中国科学院大学
付琳晴
2017/12/24

1. 用户态进程内存管理设计

1.1 用户态进程虚存布局

虚拟空间布局如图 1. 在任务一，进程的用户栈在内存空间，用户的代码段、数据段等在用户空间。在任务二，进程用户栈移到了用户空间。

Address range	Capacity (GB)	Mapping approach	Cacheable
0xC0000000 -- 0xFFFFFFFF	1	TLB lookup	Yes
	Kernel space		
0xA0000000 -- 0xBFFFFFFF	0.5	Base + offset	No
0x80000000 -- 0x9FFFFFFF	0.5	Base + offset	Yes
0x00000000 -- 0x7FFFFFFF	2	TLB lookup	Yes
	User space		

图 1. 虚址分布

1.2 页表项结构

页表项结构如图 2. 高 20 位为 PFN，后 12 位为 flag 位，而在此实验中我们只用到了 D（dirty）和 V（valid）位。D 位指示页的 dirty 属性，V 位指示页的有效属性。

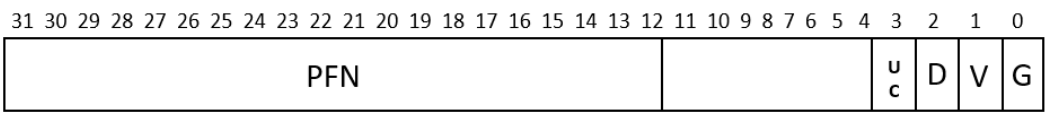

图 2. PTE 项结构

1.3 用户态进程页表操作

用户态进程页表初始化主要在 setup_page_table 中完成。主要流程是，首先分配一个物理页框用做进程的页表，然后分配一个物理页框用于存进程的代码段数据段，并在页表中插入一项页表项，其指向这个分配的物理页框。

在任务一中，在进程的页表中插入一项页表项，插入的页表项标志位需要设成有效，即需要立即给进程的代码段分配空间，并用 bcopy 将相关代码从磁盘拷贝到物理内存。关键代码段如图 3.

```
index = page_alloc(TRUE);
p = &pcb[pid];
pg = page_alloc(FALSE);
paddr = page_map[pg].paddr_start;

insert_page_table_entry(page_map[index].paddr_start, p->entry_point, paddr, 0x2, pid);
bcopy((char *)p->loc, (char *)paddr, PAGE_SIZE);
```

图 3. 任务一进程页框分配

在任务二中，需要在进程的页表中插入两项页表项，一项指向进程代码段空间，一项指向进程用户栈空间。初始化时只需要插入进程代码段对应的虚实地址转换项，插入的页表项标志位需要设成无效，即分配一个页框来放代码，但是并不调用 bcopy 将相关代码从磁盘拷贝到物理内存，当发生 page_fault 时，在 page_fault 处理函数里需要重新分配页框并再拷贝代码段。任务二中 setup_page_table 如图 4.

```
index = page_alloc(TRUE);
page_map[index].pid = pid;

p = &pcb[pid];

pg = page_alloc(FALSE);
page_map[pg].pid = pid;
insert_page_table_entry(page_map[index].paddr_start, p->entry_point, page_map[pg].paddr_start, 0x0, pid);
//bcopy((char *)p_loc, (char *)paddr, PAGE_SIZE);
```

图 4. 任务二进程页框分配

1.4 管理物理内存的数据结构

为了跟踪每一个页框的状态，设置一个数组 page_map，元素的个数为物理内存的页框数，每一个元素是一个数据结构，存有关于对应的页框的信息。该数据结构如图 5。其中，vaddr_start 和 paddr_start 分别为该物理页框对应的虚拟起始地址和物理起始地址。Dirty 域标记了该物理页框目前是否可写。pinned 域标记了该页是否可以被替换。Available 域标记该页信息是否有效。Pid 域标记该页框属于哪个进程。

```
typedef struct {
    // design here
  node_t node;
  uint32_t vaddr_start;
  uint32_t paddr_start;
  bool_t dirty;
  bool_t pinned;
  bool_t available;
  pid_t pid;
}page_map_entry_t;
```

图 5. 管理物理内存的数据结构 page_map_entry_t

物理页框分配由 page_alloc 函数完成，任务一中不需要考虑页替换，因为页框数充足。此时的页框分配策略就是，遍历所有页框，找到一页 available 域为 1，即分配。

1.5 TLB_miss 处理流程

TLB miss 发生在进程入口时，TLB 中必定不会有进程的入口地址到物理地址的转换项，此时将发生 TLB miss. 发生 TLB miss 后需要在页表中找到该虚实转换项，并进行 TLB refill。在我的设计中，任务一中 TLB refill 的操作均在汇编代码中完成。

首先从 current_running 中取出 pid、page_table、badvaddr 域，根据 badvaddr 中的虚地址在 page_table 中索引，找到对应的 PTE 项，提取该 PTE 项中的物理地址信息，和 flag 位一起放入 cp0_entryLo0 和 cp0_entryLo1，将虚拟地址和 Pid 一起存入 cp0_entryhi，最后使用 tlbwr，在 TLB 中添加一项映射。

1.6 遇到的问题

（1）createimage 扇区数数错

解决过程：任务一写了一天，其他各种细微地小错几乎都忘了，一个最大的错误现象是进程 2 一直出不来，进程 1 可以跑出小飞机，而线程的打印信息显示一直在进入进程 2。查了一天的错，经历了各种 lppppp 等根本没有指向根本错误原因的错误。最后，终于，在和刘国栋同学的闲聊中，发现是载入磁盘的进程的扇区数算错。0x7400 减 0x6600 我算成了十进制，导致进程 2 加载的不完全，导致进程 2 的打印信息根本没有。

（2）insert_page_table_entry 函数插入的 PTE 项错误

解决过程：计算出的物理地址有问题，说明插入的 PTE 项有问题，仔细看了文档发现 EntryLo 寄存器里 PFN 都是 26 位，而我的设计中 PFN 只有 20 位，因此在 insert_page_table_entry 时我将物理地址右移 6 位，从而与 EntryLo 中设计相同。

2. 缺页中断与 swap 处理设计

2.1 缺页中断

任务一中不会发生缺页中断，因为任务一中每次插入 PTE 都直接分配一个页框并填入代码段，valid 域是 1. 但任务二中会发生缺页中断，在任务二中 setup_page_table 插入 PTE 时 valid 域是 0，表示缺页。理论上，发生 page fault 情况有两种：（1）TLB 中找到了该项但 valid 域为 0. （2）TLB 中未找到该项，在页表中找到了该项但 valid 域为 0.

在任务二中，我设计的缺页中断处理流程为：（1）先在 TLB 中查找是否有该项，若有，一定是 valid 域为 0，跳至 page_fault 处理程序处。（2）在页表中查该项，若 valid 域为 0，也跳至 page_fault 处理程序处。（3）最终都进行 TLB refill 操作。

Page_fault 处理程序我用 c 代码实现的，因为需要进行重新分配页等复杂操作。在 inserrupt.c 文件中实现该函数。需要分两种情况考虑，在任务二时要使用用户栈，意味着进程用户栈也需要分配物理页框。用户栈与用户代码产生的 page_fault 最大的区别在于，用户栈分配的物理页框需要 pinned，并且用户栈分配物理页框后不需要拷贝代码。需要 pinned 的页框为：每个进程的页表和每个进程的用户栈。

如何区分是用户栈还是用户代码产生的 page_fault 呢，是根据 current_running 中 cp0_badvaddr 域存的虚地址大小来做区分。我给用户栈分配的虚拟地址是 0x37fff0，而进程代码段的虚拟地址一定小于 0x300000。Handle_page_fault 函数主要代码段如图 6.

```
if((current_running->user_tf).cp0_badvaddr > 0x300000){
  index = page_alloc(1);
} else {
  index = page_alloc(0);
}
//printf(29, 40, "p%d:find a unpinned:%d",pid, index);
page_map[index].pid = pid;
paddr = page_map[index].paddr_start;

if((current_running->user_tf).cp0_badvaddr > 0x300000){
  insert_page_table_entry(page_table, 0x37fff0, paddr, 0x6, pid);
} else {
  insert_page_table_entry(page_table, p->entry_point, paddr, 0x6, pid);
  bcopy((char*)p->loc, (char*)paddr, PAGE_SIZE);
}
```

图 6. Page_fault 处理程序关键代码

2.2 page_alloc

采用的 page_alloc 分配策略是 FIFO，又因为在该实验中只有两个进程，而为了测试 page_fault 功能，只能留出一个 unpinned 的页框用作不停地被替换，因此页替换其实是体现不出来是什么策略的。

Page_alloc 相比于任务一增加的部分是：考虑页框分配已满，需要换出页框的情况。需要扫描一遍 page_map 数组，如果发现没有 available 的页框，则需要重新扫描数组，找到一个 pinned 域为 0 的页框，并替换掉原来的内容。这里需要注意的是，若要换出一个页框，需要将曾经拥有这个页框的进程的页表中，映射到该页框的那项 PTE 置为无效，这一点很关键。为了实现这个目的，调用一次 insert_page_table_entry 即可，这个函数里会调用 tlb_flush 来自动刷新 TLB。

2.3 遇到的问题

（1）LPPPPPP

解决过程：根据助教发的邮件提示，需要更改 entry.S 中 handle_int、handle_tlb、handle_syscall 三个函数，要在程序最开始 SAVE_CONTEXT(USER)之后添加 RESTORE_STACK(KERNEL)，主要目的是把 sp 寄存器换成指向内核栈，在内核栈进行这些处理程序。需要注意的是，在 handle_int 函数中要先判断是进程还是线程，如果是线程就不需要更改栈指针了，因为本来就在内核栈，判断代码如图 7.

```
  SAVE_CONTEXT(USER)
  la      k0, current_running
  lw      k0, 0(k0)
  lw      k0, NESTED_COUNT(k0)
  bne     $0, k0, not_restore
  nop
  nop
  RESTORE_STACK(KERNEL)
not_restore:
```

图 7. 判断是否需要恢复内核栈

（2）status 信息只打印了一遍就开始狂 LPPPPPPP

解决过程：在室友帮助下，发现在 page_fault 处理程序中 insert_page_table_entry 函数 flag 位置错，我传入的 flag 数是 0x2，即只置了 valid 位，dirty 位是 0，根据体系结构实验课的文档，就会发生 TLB modified 例外。

（3）process2 第二次进入时竟然状态显示 exited

解决过程：是 page_alloc 里考虑不周全，当进行页替换的时候，需要把先前拥有这个页的进程的页表中该项置为无效，否则，先前的进程还以为自己拥有这个物理页框，但事实上已经不属于它了。做法是调用 insert_page_table_entry，虚拟地址还是之前的虚拟地址，flag 位置成 0x0 即可。

（4）错误现象不记得了

解决过程：handle_tlb 汇编代码中，我进行 TLB refill 时，存入 EntryLo0 和 EntryLo1 的数据不太对，规定 EntryLo0 对应的物理页应该是偶页（虚拟地址第 12 位是 0），EntryLo1 对应的是奇页（虚拟地址第 12 位是 1），但我根本没有管虚拟地址第 12 位是什么，统统把欲找到那项 PTE 存入 EntryLo0，而地址加 4 处的 PTE 存入 EntryLo1。这样会导致 TLB 查询时查不到对应的项，即使确实有这一项。

3. Bonus 设计

未做 Bonus。

4. 关键函数功能

4.1 page_alloc

用来分配一个物理页框，返回值是该物理页框在 page_map 数组中的下标。任务二中完善了 page_alloc，使其能实现页替换功能，并且在页替换时将刷新之前拥有这个页框的进程的页表项。关键代码如图 8.

```
//IF don't have empty page
if (index == PAGEABLE_PAGES){
    for(index = 0; index < PAGEABLE_PAGES; index++){
        if(page_map[index].pinned == 0){
            /*clean the former*/
            pid = page_map[index].pid;
            page_table = pcb[pid].page_table;
            insert_page_table_entry(page_table, pcb[pid].entry_point, 0x0, 0x0, pid);
            break;
        }
    }
}
```

图 8. Page_alloc

4.2 setup_page_table

在 initialize_pcb 中被调用，用于给每个进程分配一个物理页框当页表。任务一和任务二该函数的主要区别在于，任务一中给代码段直接分配了一个页框，并且调用 bcopy 将代码段从磁盘拷贝进页框。而任务二中虽然分配了页框，但置 PTE 项为无效，并没有拷贝代码段。任务二的代码如图 9.

```
uint32_t setup_page_table( int pid ) {
    uint32_t page_table;
    int pg;
    uint32_t paddr, vaddr;
    int index;
    int j;
    pcb_t *p;
    index = page_alloc(TRUE);
    page_map[index].pid = pid;

    p = &pcb[pid];

    pg = page_alloc(FALSE);
    page_map[pg].pid = pid;
    insert_page_table_entry(page_map[index].paddr_start, p->entry_point, page_map[pg].paddr_start, 0x0, pid);
    //bcopy((char *)p_loc, (char *)paddr, PAGE_SIZE);


    page_table = page_map[index].paddr_start;
    return page_table;
}
```

图 9. Setup_page_table

4.3 insert_page_table_entry

在页表中插入一项 PTE，插入的位置根据虚拟地址高 20 位决定。PTE 项其实是个 32 位的数据。为了保持一致性，需要在最后刷新 TLB，无效掉该 PTE 对应的 TLB 项。代码段如图 10.

```
// TODO: insert page table entry to page table
void insert_page_table_entry( uint32_t *table, uint32_t vaddr, uint32_t paddr,
                              uint32_t flag, uint32_t pid ) {
    // insert entry
    uint32_t index = get_table_index(vaddr);
    table[index] = ((paddr & 0xfffff000)>>6)|(flag & 0xfff);
    tlb_flush((vaddr & 0xffffe000) | (pid & 0xfff));
    // tlb flush
}
```

图 10. Insert_page_table_entry

4.4 handle_page_fault

在 Interrupt.c 文件中，属于中断处理程序。需要对于用户栈和进程程序引起的 page_fault 分别进行处理。是任务二中新添加的函数。代码如图 11.

```
void handle_page_fault()
{
  //printf(29, 1, "jump in page_fault");
    // //printf(28,0,"%x", EntryLo0_temp);
  uint32_t index;
  uint32_t *page_table;
  uint32_t paddr;
  pid_t pid;
  pcb_t *p;

  page_fault++;
  printf(18, 1, "page_fault:%d", page_fault);
  pid = current_running->pid;
  page_table = current_running->page_table;
  p = &pcb[pid];
  if((current_running->user_tf).cp0_badvaddr > 0x300000){
    index = page_alloc(1);
  } else {
    index = page_alloc(0);
  }
  //printf(29, 40, "p%d:find a unpinned:%d",pid, index);
  page_map[index].pid = pid;
  paddr = page_map[index].paddr_start;

  if((current_running->user_tf).cp0_badvaddr > 0x300000){
    insert_page_table_entry(page_table, 0x37fff0, paddr, 0x6, pid);
  } else {
    insert_page_table_entry(page_table, p->entry_point, paddr, 0x6, pid);
    bcopy((char*)p->loc, (char*)paddr, PAGE_SIZE);
  }

}
```

图 11. Handle_page_fault

4.5 handle_tlb

完成了 TLB 产生中断之后的全部中断处理流程，用汇编代码。完成了先查 TLB 后查页表从而判断是哪种中断，然后跳到不同的地方分别进行处理。代码过长，不截图了。

4.6 handle_tlb_refill

在 handle_tlb 中进行 TLB refill 时调用的 c 程序，其作用只是打印出 tlb_refill 的次数。代码如图 12.

```
void handle_tlb_refill(){
  tlb_refill++;
  printf(17, 1, "tlb refill:%d", tlb_refill);
}
```

图 12. Handle_tlb_refill

4.7 set_pt

在 scheduler 中被调用，用于页表切换，其实就是更新 EntryHi 的 pid 域，但是很重要。代码如图 13.

```
LEAF(set_pt)
    la t6, current_running    # current_running
    lw t6, (t6)
    lw t0, 328(t6)            # pid (ASID)
    mfc0 t1, CP0_ENTRYHI
    li t3, 0xffffe000
    and t1, t1, t3
    or t1, t1, t0
    mtc0 t1, CP0_ENTRYHI
    j   ra
    nop
END(set_pt)
```

图 13. Set_pt