OS Lab2 内存管理

520030910393 马逸川

思考题1:

优势： 允许页表内存在空洞，在多数情况下降低了内存占用的空间；

劣势： 多级页表访问时间比单级页表长；代码实现稍微复杂一些；在系统使用的虚拟内存和总的虚拟地址空间相近时，多级页表使用的内存实际上反而大于单级页表。

计算:

4KB :

$$num = \frac{4GB}{4KB} = 1,000,000$$

2MB:

$$num = \frac{4GB}{2MB} = 2000$$

练习题2:

实现思路: 首先将PHYSMEM_START地址加上高地址偏移量，跳转到高地址空间。

```
        vaddr = KERNEL_VADDR + PHYSMEM_START;
        boot_ttbr1_l0[GET_L0_INDEX(vaddr)] = ((u64)boot_ttbr1_l1) |
IS_TABLE
                                             | IS_VALID | NG;
        boot_ttbr1_l1[GET_L1_INDEX(vaddr)] = ((u64)boot_ttbr1_l2) |
IS_TABLE
                                             | IS_VALID | NG;
```

然后参考低地址空间的映射代码，将物理地址映射到高地址空间，如下：

```
        for (; vaddr < KERNEL_VADDR + PERIPHERAL_BASE ; vaddr += SIZE_2M) {
                boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                        (vaddr-KERNEL_VADDR) /* low mem, va = pa */
                        | UXN /* Unprivileged execute never */
                        | ACCESSED /* Set access flag */
                        | NG /* Mark as not global */
                        | INNER_SHARABLE /* Sharebility */
                        | NORMAL_MEMORY /* Normal memory */
                        | IS_VALID;
        }
```

运行后程序正常进入`main`函数。

思考题3:

参考课件和官方文档，TTBR0存放的是用户态的页表，TTBR1存放的是内核态的高地址页表。为低地址配置页表是为了用户态的应用程序能够使用物理内存。如果不为低地址配置页表，应用程序将无法正常使用物理内存。

练习题4:

这部分的实现参考了这篇博客的代码。

下面详细阐述每个函数部分的实现细节:

每个函数的返回值都是一个地址，其内容是各个`page`结构体的开头。

1. `split_page`

这一部分的实现参考了博客。

首先从pool中删除当前page，然后循环降低`page->order`，持续地分配伙伴页至`page->order` = `order`， 并将各个不同order值的`buddy-page`加入pool中。

2. `buddy_get_pages`

首先从所需的order出发找到order最小的free page。通过`list_entry`函数获取第一个page, 接着调用之前实现的`split_page(pool, order, page)`即可。

3. `merge_page`

删除现order的page，通过`get_buddy_chunk`获取伙伴页，将其记为已分配，page->order++至buddy_page已分配或者分配为其它大小。（第二个条件的设计参考了博客）

4. `buddy_free_pages`

将`page->allocated`记为0，调用`merge_page`。

练习题5:

这一部分的实现也参考了上面博客的部分内容。

1. `query_in_pgtbl`

（博客的实现好像有点问题）我的思路是:循环迭代level, 对L1, L2判断是否返回值为`-ENOMAPPING`， 若是，则返回，否则按照页表级数返回物理地址。

2. `map_range_in_pgtbl`

首先 len / page_size 计算page数目，通过`get_next_ptp`迭代至L3层，参考博客的实现分配新页，va, pa对应加上page_size, 当`page_num=0`时退出循环。

#####

3. `unmap_range_in_pgtbl`

同样，首先计算page数目，然后通过`get_next_ptp`依次取得各层级的页， 如果在l3之前已经返回

`-ENOMAPPING`，则按照大页的大小（如`L0_PER_ENTRY_PAGES`）增加va, 减去页数。如到达l3层，则逐页标识为`PTE_DESCRIPTOR_INVALID`，减去页数即可。

练习题6:

1. `map_range_in_pgtbl_huge`

这一部分的实现思路与`map_range_in_pgtbl`基本相同，区别在于page_size的修改。

首先，4GB_page的大小为`PAGE_SIZE * PTP_ENTRIES * PTP_ENTRIES`， 2MB_page的大小为`PAGE_SIZE * PTP_ENTRIES`。首先，按照4GB大小分配，此时层级为L1，在L1page上分配，具体的分配代码与`map_range_in_pgtbl`基本相同，len减去分配的页大小。然后，按照2MB大小在L2上分配。最后在L3层上分配。

2.`unmap_range_in_pgtbl_huge`

同样分不同level进行unmap操作，与`unmap_range_in_pgtbl`的实现思路基本相同，唯一的区别在于page_size的不同，具体的代码段如下例所示:

```
else if (err == BLOCK_PTP) {
        pte->pte = PTE_DESCRIPTOR_INVALID;
        va += PAGE_SIZE * PTP_ENTRIES * PTP_ENTRIES;

        page_num -= PTP_ENTRIES * PTP_ENTRIES;
        flag = false ;
```

上面展示的是L1_pages，当`get_next_ptp`返回`BLOCK_PTP`时，需将va加4GB以实现L1_page的unmap操作。

思考题7 :

参考文档的以下内容:

If the level 1 descriptor defines a page-mapped access then the level 2 descriptor specifies four access permission fields (ap3 to ap0), each corresponding to one quarter of the page:

- For small pages:
  - ap3 is selected by the top 1KB of the page
  - ap0 is selected by the bottom 1KB of the page.
- For large pages:
  - ap3 is selected by the top 16KB of the page
  - ap0 is selected by the bottom 16KB of the page.

The selected AP bits are then interpreted in exactly the same way as for a section (see Table 6.3. The only difference is that the fault generated is a subpage permission fault.

可知需要配置ap0-ap3字段。

思考题8:

可能的问题: 粗粒度映射导致内核对内存的利用效率降低。某些情况下可能出现内存碎片。