

实验二：内存管理

练习题1:

练习题 1: 完成 `kernel/mm/buddy.c` 中的 `split_chunk`、`merge_chunk`、`buddy_get_pages` 和 `buddy_free_pages` 函数中的 LAB 2 TODO 1 部分, 其中 `buddy_get_pages` 用于分配指定阶大小的连续物理页, `buddy_free_pages` 用于释放已分配的连续物理页。

提示:

- 可以使用 `kernel/include/common/list.h` 中提供的链表相关函数和宏如 `init_list_head`、`list_add`、`list_del`、`list_entry` 来对伙伴系统中的空闲链表进行操作
- 可使用 `get_buddy_chunk` 函数获得某个物理内存块的伙伴块
- 更多提示见代码注释
- `split_chunk` 代码如下:

```
1 static struct page *split_chunk(struct phys_mem_pool *pool, int order,
2                                 struct page *chunk)
3 {
4     /* LAB 2 TODO 1 BEGIN */
5     /*
6      * Hint: Recursively put the buddy of current chunk into
7      * a suitable free list.
8      */
9     /* BLANK BEGIN */
10    if (chunk->order == order) {
11        return chunk;
12    }
13
14    // split the chunk to two parts
15    chunk->order--;
16    struct page *buddy_chunk = get_buddy_chunk(pool, chunk);
17    buddy_chunk->order = chunk->order;
18
19    list_add(&buddy_chunk->node,
20            &(pool->free_lists[chunk->order].free_list));
21    pool->free_lists[chunk->order].nr_free += 1;
22
23    buddy_chunk->pool = chunk->pool;
24    buddy_chunk->allocated = 0;
25
26    return split_chunk(pool, order, chunk);
27
28    /* BLANK END */
29    /* LAB 2 TODO 1 END */
30 }
```

如果当前chunk的order就是我们需要的order直接返回, 如果不是则需要split。

先将order-1, 调用 get_buddy_chunk 获取 buddy_chunk 同时为其属性复制, 再递归调用 split_chunk。

- merge_chunk 代码如下:

```
1 static struct page *merge_chunk(struct phys_mem_pool *pool, struct page
  *chunk)
2 {
3     /* LAB 2 TODO 1 BEGIN */
4     /*
5      * Hint: Recursively merge current chunk with its buddy
6      * if possible.
7      */
8     /* BLANK BEGIN */
9     if (chunk->order == (BUDDY_MAX_ORDER - 1)) {
10         return chunk;
11     }
12
13     struct page *buddy_chunk = get_buddy_chunk(pool, chunk);
14     if (buddy_chunk == NULL || buddy_chunk->order != chunk->order) {
15         return chunk;
16     }
17     if (buddy_chunk->allocated == 1) {
18         return chunk;
19     }
20
21     list_del(&buddy_chunk->node);
22     pool->free_lists[chunk->order].nr_free--;
23
24     buddy_chunk->order++;
25     chunk->order++;
26
27     if (chunk > buddy_chunk) {
28         return merge_chunk(pool, buddy_chunk);
29     } else {
30         return merge_chunk(pool, chunk);
31     }
32     /* BLANK END */
33     /* LAB 2 TODO 1 END */
34 }
```

当 chunk 达到最大, 或者 buddy_chunk 不存在, 已被分配, 不是整块, 就不能 merge 直接返回 如果能够 merge, 则把 buddy_chunk 从对应的 free_list 删去, 设置对应 order, 并选取与 chunk 对应更前的地址, 作为新的 chunk 地址。

- buddy_get_page 补全代码如下:

```
1     cur_order = order;
2     for (; cur_order < BUDDY_MAX_ORDER; cur_order++) {
3         if (pool->free_lists[cur_order].nr_free > 0) {
4             break;
5         }
6     }
7     // if there is no free chunk, return NULL
8     if (cur_order == BUDDY_MAX_ORDER) {
```

```

9         page = NULL;
10     } else {
11         free_list = pool->free_lists[cur_order].free_list.next;
12         page = list_entry(free_list, struct page, node);
13         pool->free_lists[page->order].nr_free--;
14         page = split_chunk(pool, order, page);
15         page->allocated = 1;
16         list_del(&page->node);
17     }

```

找到大于等于所需 order 的非空 free_list，从 pool 对该 free_list 的 chunk 进行 split，直到得到所需 order 的 page，然后从 free_list 中删去 page，并设置为已分配。

- buddy_free_pages 代码如下：

```

1 void buddy_free_pages(struct phys_mem_pool *pool, struct page *page)
2 {
3     int order;
4     struct list_head *free_list;
5
6     lock(&pool->buddy_lock);
7
8     /* LAB 2 TODO 1 BEGIN */
9     /*
10      * Hint: Merge the chunk with its buddy and put it into
11      * a suitable free list.
12      */
13     /* BLANK BEGIN */
14     page->allocated = 0;
15     page = merge_chunk(pool, page);
16
17     order = page->order;
18     free_list = &(pool->free_lists[order].free_list);
19     list_add(&page->node, free_list);
20     pool->free_lists[order].nr_free++;
21     /* BLANK END */
22     /* LAB 2 TODO 1 END */
23
24     unlock(&pool->buddy_lock);
25 }

```

将对应 page 设置为非分配，进行 merge 之后，放到对应的 free_list 中。

练习题2：

练习题 2：完成 kernel/mm/slab.c 中的 choose_new_current_slab、alloc_in_slab_impl 和 free_in_slab 函数中的 LAB 2 TODO 2 部分，其中 alloc_in_slab_impl 用于在 slab 分配器中分配指定阶大小的内存，而 free_in_slab 则用于释放上述已分配的内存。

提示：

- 你仍然可以使用上个练习中提到的链表相关函数和宏来对 SLAB 分配器中的链表进行操作
- 更多提示见代码注释
- choose_new_current_slab 代码如下：

```

1 static void choose_new_current_slab(struct slab_pointer *pool)
2 {
3     /* LAB 2 TODO 2 BEGIN */
4     /* Hint: Choose a partial slab to be a new current slab. */
5     /* BLANK BEGIN */
6     struct list_head *list;
7
8     list = &(pool->partial_slab_list);
9     if (list_empty(list)) {
10         pool->current_slab = NULL;
11     } else {
12         struct slab_header *slab;
13
14         slab = (struct slab_header *)list_entry(
15             list->next, struct slab_header, node);
16         pool->current_slab = slab;
17         list_del(list->next);
18     }
19     /* BLANK END */
20     /* LAB 2 TODO 2 END */
21 }

```

利用 `pool->partial_slab_list` 获得对应的 `partial slab` 链表，如果该链表为空，则返回 `NULL`，获得新的 `current slab` 失败，如果非空，则利用 `list_entry` 方法获得对应的 `slab` 的地址。然后赋值给 `pool` 的 `current_slab`，最后把该 `slab` 从 `partial slab` 链上去。

- `alloc_in_slab_impl` 代码如下：

```

1
2     /* LAB 2 TODO 2 BEGIN */
3     /*
4      * Hint: Find a free slot from the free list of current slab.
5      * If current slab is full, choose a new slab as the current
6      one.
7      */
8     /* BLANK BEGIN */
9     free_list = (struct slab_slot_list *)current_slab->free_list_head;
10    BUG_ON(free_list == NULL);
11
12    next_slot = free_list->next_free;
13    current_slab->free_list_head = next_slot;
14
15    current_slab->current_free_cnt--;
16
17    if (unlikely(current_slab->current_free_cnt == 0)) {
18        // try_insert_full_slab_to_partial(current_slab);
19        choose_new_current_slab(&slab_pool[order]);
20    }
21    /* BLANK END */
22    /* LAB 2 TODO 2 END */

```

根据提示，利用 `current_slab` 的 `free_list_head` 属性获取对应 `slot` 的 `free_list` 链表 然后通过操作指针从链表中获取 `slot`，把他从链表中删去。如果 `current_slab` 的中没有空余 `slot`，再选择新的 `current slab`。

- `free_in_slab` 代码如下：

```
1      /* LAB 2 TODO 2 BEGIN */
2      /*
3      * Hint: Free an allocated slot and put it back to the free
4      list.
5      */
6      /* BLANK BEGIN */
7      slot->next_free = slab->free_list_head;
8      slab->free_list_head = slot;
9      slab->current_free_cnt++;
10     /* BLANK END */
11     /* LAB 2 TODO 2 END */
```

只要将 `alloc` 的过程逆向即可

练习题3:

练习题 3: 完成 `kernel/mm/kmalloc.c` 中的 `_kmalloc` 函数中的 `LAB 2 TODO 3` 部分，在适当位置调用对应的函数，实现 `kmalloc` 功能

提示:

- 你可以使用 `get_pages` 函数从伙伴系统中分配内存，使用 `alloc_in_slab` 从 SLAB 分配器中分配内存
 - 更多提示见代码注释
- `_kmalloc` 代码如下：

```
1 void *_kmalloc(size_t size, bool is_record, size_t *real_size)
2 {
3     void *addr;
4     int order;
5
6     if (unlikely(size == 0))
7         return ZERO_SIZE_PTR;
8
9     if (size <= SLAB_MAX_SIZE) {
10         /* LAB 2 TODO 3 BEGIN */
11         /* Step 1: Allocate in slab for small requests. */
12         /* BLANK BEGIN */
13         addr = alloc_in_slab(size, real_size);
14         /* BLANK END */
15 #if ENABLE_MEMORY_USAGE_COLLECTING == ON
16         if (is_record && collecting_switch) {
17             record_mem_usage(*real_size, addr);
18         }
19 #endif
20     } else {
21         /* Step 2: Allocate in buddy for large requests. */
22         /* BLANK BEGIN */
```

```

23         order = size_to_page_order(size);
24         addr = _get_pages(order, is_record);
25         /* BLANK END */
26         /* LAB 2 TODO 3 END */
27     }
28
29     BUG_ON(!addr);
30     return addr;
31 }

```

调用已有的 `alloc_in_slab` 分配小块的内存，调用 `get_pages` 分配大块内存即可。

练习题4:

练习题 4: 完成 `kernel/arch/aarch64/mm/page_table.c` 中的 `query_in_pgtbl`、`map_range_in_pgtbl_common`、`unmap_range_in_pgtbl` 和 `mprotect_in_pgtbl` 函数中的 LAB 2 TODO 4 部分，分别实现页表查询、映射、取消映射和修改页表权限的操作，以 4KB 页为粒度。

提示:

- 需要实现的函数内部无需刷新 TLB，TLB 刷新会在这些函数的外部进行
- 实现中可以使用 `get_next_ptp`、`set_pte_flags`、`virt_to_phys`、`GET_LX_INDEX` 等已经给定的函数和宏
- 更多提示见代码注释
- `query_in_pgtbl` 代码如下:

```

1  int query_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t *pa, pte_t **entry)
2  {
3      /* LAB 2 TODO 4 BEGIN */
4      /*
5       * Hint: walk through each level of page table using
6       * `get_next_ptp`,
7       * return the pa and pte until a L0/L1 block or page, return
8       * `-ENOMAPPING` if the va is not mapped.
9       */
10     /* BLANK BEGIN */
11     ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp, *ptp;
12     pte_t *pte;
13     int ret;
14
15     l0_ptp = (ptp_t *)pgtbl;
16     l1_ptp = NULL;
17     l2_ptp = NULL;
18     l3_ptp = NULL;
19
20     ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, false);
21     if (ret == -ENOMAPPING) {
22         return ret;
23     }
24
25     ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, false);
26     if (ret == -ENOMAPPING) {
27         return -ENOMAPPING;

```

```

27         } else if (ret == BLOCK_PTP) {
28             if (entry != NULL) {
29                 *entry = pte;
30             }
31             *pa = virt_to_phys((vaddr_t)l2_ptp) +
GET_VA_OFFSET_L1(va);
32             return 0;
33         }
34
35         ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, false);
36         if (ret == -ENOMAPPING) {
37             return ret;
38         } else if (ret == BLOCK_PTP) {
39             if (entry != NULL) {
40                 *entry = pte;
41             }
42             *pa = virt_to_phys((vaddr_t)l3_ptp) +
GET_VA_OFFSET_L2(va);
43             return 0;
44         }
45
46         ret = get_next_ptp(l3_ptp, L3, va, &ptp, &pte, false);
47
48         if (ret == -ENOMAPPING) {
49             return ret;
50         }
51
52         if (entry != NULL) {
53             *entry = pte;
54         }
55         *pa = virt_to_phys(ptp) + GET_VA_OFFSET_L3(va);
56         /* BLANK END */
57         /* LAB 2 TODO 4 END */
58         return 0;
59     }

```

利用 `get_next_ptp` 依次遍历对应 `va` 的每级页表，利用 `ret` 判断为 `block` 则返回对应页表项和物理页。

如果未分配，则返回 `-ENOMAPPING`；如果遍历到物理页且分配，则返回对应页表项和物理页。

- `map_range_in_pgtbl_common` 代码如下：

```

1  static int map_range_in_pgtbl_common(void *pgtbl, vaddr_t va, paddr_t
pa,
2                                     size_t len, vmr_prop_t flags, int
kind)
3  {
4      /* LAB 2 TODO 4 BEGIN */
5      /*
6          * Hint: walk through each level of page table using
`get_next_ptp`,
7          * create new page table page if necessary, fill in the final
level
8          * pte with the help of `set_pte_flags`. Iterate until all pages
are

```

```

9      * mapped.
10     * Since we are adding new mappings, there is no need to flush
    TLBS.
11     * Return 0 on success.
12     */
13     /* BLANK BEGIN */
14     u64 total_page_cnt;
15     ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
16     pte_t *pte;
17     int ret;
18     int pte_index;
19     int i;
20
21     BUG_ON(pgtbl == NULL);
22     BUG_ON(va % PAGE_SIZE);
23
24     total_page_cnt = len / PAGE_SIZE + (((len % PAGE_SIZE) > 0) ? 1
: 0);
25     l0_ptp = (ptp_t *)pgtbl;
26
27     l1_ptp = NULL;
28     l2_ptp = NULL;
29     l3_ptp = NULL;
30
31     while (total_page_cnt > 0) {
32         ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte, true);
33         BUG_ON(ret != 0);
34
35         ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte, true);
36         BUG_ON(ret != 0);
37
38         ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte, true);
39         BUG_ON(ret != 0);
40
41         pte_index = GET_L3_INDEX(va);
42         for (i = pte_index; i < PTP_ENTRIES; ++i) {
43             pte_t new_pte_val;
44
45             new_pte_val.pte = 0;
46             new_pte_val.l3_page.is_valid = 1;
47             new_pte_val.l3_page.is_page = 1;
48             new_pte_val.l3_page.pfn = pa >> PAGE_SHIFT;
49             set_pte_flags(&new_pte_val, flags, kind);
50             l3_ptp->ent[i].pte = new_pte_val.pte;
51
52             va += PAGE_SIZE;
53             pa += PAGE_SIZE;
54
55             total_page_cnt -= 1;
56             if (total_page_cnt == 0)
57                 break;
58         }
59     }
60     /* BLANK END */
61     /* LAB 2 TODO 4 END */
62     return 0;

```


利用 `total_page_cnt = len / PAGE_SIZE + (((len % PAGE_SIZE) > 0) ? 1 : 0)`; 计算总的需要映射的物理页数，然后利用 `get_next_ptp` 找到对应的 `va` 的第三级页表，并分配可能没有分配的页表页，最后遍历最后一级页表，依次分配物理页，直到达到对应的数目。

- `unmap_range_in_pgtbl` 代码如下：

```

1  int unmap_range_in_pgtbl(void *pgtbl, vaddr_t va, size_t len)
2  {
3      /* LAB 2 TODO 4 BEGIN */
4      /*
5       * Hint: walk through each level of page table using
6       * `get_next_ptp`,
7       * mark the final level pte as invalid. Iterate until all pages
8       * are
9       * unmapped.
10      * You don't need to flush tlb here since tlb is now flushed
11      * after
12      * this function is called.
13      * Return 0 on success.
14      */
15      /* BLANK BEGIN */
16      u64 total_page_cnt;
17      ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
18      pte_t *pte;
19      int ret;
20      int pte_index;
21      int i;
22
23      BUG_ON(pgtbl == NULL);
24      BUG_ON(va % PAGE_SIZE);
25
26      total_page_cnt = len / PAGE_SIZE + (((len % PAGE_SIZE) > 0) ? 1
27      : 0);
28
29      l1_ptp = NULL;
30      l2_ptp = NULL;
31      l3_ptp = NULL;
32
33      while (total_page_cnt > 0) {
34          l0_ptp = (ptp_t *)pgtbl;
35          ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte,
36          false);
37
38          if (ret == -ENOMAPPING) {
39              total_page_cnt -= L0_PER_ENTRY_PAGES;
40              va += L0_PER_ENTRY_PAGES * PAGE_SIZE;
41              continue;
42          }
43
44          ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte,
45          false);
46
47          if (ret == -ENOMAPPING) {
48              total_page_cnt -= L1_PER_ENTRY_PAGES;
49              va += L1_PER_ENTRY_PAGES * PAGE_SIZE;
50          }
51      }
52  }

```

```

42         continue;
43     } else if (ret == BLOCK_PTP) {
44         pte->pte = PTE_DESCRIPTOR_INVALID;
45         va += PAGE_SIZE * PTP_ENTRIES * PTP_ENTRIES;
46
47         total_page_cnt -= PTP_ENTRIES * PTP_ENTRIES;
48         continue;
49     }
50
51     ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte,
false);
52
53     if (ret == -ENOMAPPING) {
54         total_page_cnt -= L2_PER_ENTRY_PAGES;
55         va += L2_PER_ENTRY_PAGES * PAGE_SIZE;
56         continue;
57     } else if (ret == BLOCK_PTP) {
58         pte->pte = PTE_DESCRIPTOR_INVALID;
59         va += PAGE_SIZE * PTP_ENTRIES;
60
61         total_page_cnt -= PTP_ENTRIES;
62         continue;
63     }
64
65     pte_index = GET_L3_INDEX(va);
66     for (i = pte_index; i < PTP_ENTRIES; ++i) {
67         l3_ptp->ent[i].pte = PTE_DESCRIPTOR_INVALID;
68
69         va += PAGE_SIZE;
70
71         total_page_cnt -= 1;
72         if (total_page_cnt == 0)
73             break;
74     }
75
76     /* BLANK END */
77     /* LAB 2 TODO 4 END */
78
79     dsb(ishst);
80     isb();
81
82     return 0;
83 }

```

在调用 `get_next_ptp` 时，将 `alloc` 设为 `false`，并以此判断对应的页表项是块描述符还是页描述符，还是指向下一级页表的基地址。如果是块描述符，直接 `unmap` (`unmap` 操作通过对 `pte` 页表项赋值为 0 实现)后，需要对 `va` 加上对应块的大小，并对计数用的 `total_page_cnt` 减掉对应的物理页数；如果是页描述符，`unmap` 后，需要对 `va` 加上对应页的大小，并对计数用的 `total_page_cnt` 减1；如果未映射，则省去 `unmap` 操作，直接对 `va`、`total_page_cnt` 做对应操作。

- `mprotect_in_pgtbl` 代码如下：

```

1  int mprotect_in_pgtbl(void *pgtbl, vaddr_t va, size_t len, vmr_prop_t
flags)
2  {

```

```

3      /* LAB 2 TODO 4 BEGIN */
4      /*
5          * Hint: walk through each level of page table using
`get_next_ptp`,
6          * modify the permission in the final level pte using
`set_pte_flags`.
7          * The `kind` argument of `set_pte_flags` should always be
`USER_PTE`.
8          * Return 0 on success.
9          */
10     /* BLANK BEGIN */
11     s64 total_page_cnt;
12     ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
13     pte_t *pte;
14     int ret;
15     int pte_index;
16     int i;
17
18     BUG_ON(pgtbl == NULL);
19     BUG_ON(va % PAGE_SIZE);
20
21     total_page_cnt = len / PAGE_SIZE + (((len % PAGE_SIZE) > 0) ? 1
: 0);
22     l0_ptp = (ptp_t *)pgtbl;
23
24     l1_ptp = NULL;
25     l2_ptp = NULL;
26     l3_ptp = NULL;
27
28     while (total_page_cnt > 0) {
29         ret = get_next_ptp(l0_ptp, L0, va, &l1_ptp, &pte,
false);
30         if (ret == -ENOMAPPING) {
31             total_page_cnt -= L0_PER_ENTRY_PAGES;
32             va += L0_PER_ENTRY_PAGES * PAGE_SIZE;
33             continue;
34         }
35
36         ret = get_next_ptp(l1_ptp, L1, va, &l2_ptp, &pte,
false);
37         if (ret == -ENOMAPPING) {
38             total_page_cnt -= L1_PER_ENTRY_PAGES;
39             va += L1_PER_ENTRY_PAGES * PAGE_SIZE;
40             continue;
41         } else if (ret == BLOCK_PTP) {
42             set_pte_flags(pte, flags, USER_PTE);
43             va += PAGE_SIZE * PTP_ENTRIES * PTP_ENTRIES;
44
45             total_page_cnt -= PTP_ENTRIES * PTP_ENTRIES;
46             continue;
47         }
48
49         ret = get_next_ptp(l2_ptp, L2, va, &l3_ptp, &pte,
false);
50         if (ret == -ENOMAPPING) {
51             total_page_cnt -= L2_PER_ENTRY_PAGES;

```

```

52         va += L2_PER_ENTRY_PAGES * PAGE_SIZE;
53         continue;
54     } else if (ret == BLOCK_PTP) {
55         set_pte_flags(pte, flags, USER_PTE);
56         va += PAGE_SIZE * PTP_ENTRIES;
57
58         total_page_cnt -= PTP_ENTRIES;
59         continue;
60     }
61
62     pte_index = GET_L3_INDEX(va);
63     for (i = pte_index; i < PTP_ENTRIES; ++i) {
64         set_pte_flags(&l3_ptp->ent[i], flags, USER_PTE);
65
66         va += PAGE_SIZE;
67
68         total_page_cnt -= 1;
69         if (total_page_cnt == 0)
70             break;
71     }
72 }
73 /* BLANK END */
74 /* LAB 2 TODO 4 END */
75 return 0;
76 }

```

基本思路和 `unmap_range_in_pgtbl` 一致，只是将 `unmap` 操作变成 `set_pte_flags(&l3_ptp->ent[i], flags, USER_PTE);`;

思考题5:

思考题 5: 阅读 Arm Architecture Reference Manual, 思考要在操作系统中支持写时拷贝 (Copy-on-Write, CoW) 需要配置页表描述符的哪个/哪些字段, 并在发生页错误时如何处理。 (在完成第三部分后, 你也可以阅读页错误处理的相关代码, 观察 ChCore 是如何支持 Cow 的)

- 为支持写时拷贝需要配置:
 - **Access Permissions (AP):** 页表描述符中的访问权限字段可以用于控制页面的读写权限, 对于共享的页面, 一般设置为只读, 防止不同进程修改页内容。
 - **Domain Access Control (DACR):** DACR 字段用于定义访问控制域, 用于控制哪些进程可以访问特定的内存区域。
- 发生页错误时的处理:
 - 引到对应的物理页, 将物理页的内容拷贝到另一块物理页中。
 - 然后将该物理页映射到对应的虚拟地址, 并在页表项中填写好对应的 AP 为可读可写。
 - 然后在触发异常的地址继续执行程序。

思考题6:

思考题 6: 为了简单起见, 在 ChCore 实验 Lab1 中没有为内核页表使用细粒度的映射, 而是直接沿用了启动时的粗粒度页表, 请思考这样做有什么问题。

- 被映射的物理页可能不能被充分利用, 存在较大的内部碎片。

练习题8:

练习题 8: 完成 `kernel/arch/aarch64/irq/pgfault.c` 中的 `do_page_fault` 函数中的 LAB 2 TODO 5 部分, 将缺页异常转发给 `handle_trans_fault` 函数。

- `do_page_fault` 代码如下:

```
1      /* LAB 2 TODO 5 BEGIN */
2      /* BLANK BEGIN */
3      ret = handle_trans_fault(current_thread->vmpace,
    fault_addr);
4      /* BLANK END */
5      /* LAB 2 TODO 5 END */
```

调用 `ret = handle_trans_fault(current_thread->vmpace, fault_addr);` 即可

练习题9:

练习题 9: 完成 `kernel/mm/vmpace.c` 中的 `find_vmr_for_va` 函数中的 LAB 2 TODO 6 部分, 找到一个虚拟地址找在其虚拟地址空间中的 VMR。

- `find_vmr_for_va` 代码如下:

```
1  struct vmregion *find_vmr_for_va(struct vmpace *vmpace, vaddr_t addr)
2  {
3      /* LAB 2 TODO 6 BEGIN */
4      /* Hint: Find the corresponding vmr for @addr in @vmpace */
5      /* BLANK BEGIN */
6      struct rb_node *node =
7          rb_search(&(vmpace->vmr_tree), addr, cmp_vmr_and_va);
8
9      if (node == NULL)
10         return node;
11
12     return rb_entry(node, struct vmregion, tree_node);
13     /* BLANK END */
14     /* LAB 2 TODO 6 END */
15 }
```

调用宏函数 `rb_search`, `rb_entry` 即可找到对应的 vmr。

练习题10:

练习题 10: 完成 `kernel/mm/pgfault_handler.c` 中的 `handle_trans_fault` 函数中的 LAB 2 TODO 7 部分 (函数内共有 3 处填空, 不要遗漏), 实现 `PMO_SHM` 和 `PMO_ANONYM` 的按需物理页分配。你可以阅读代码注释, 调用你之前见到过的相关函数来实现功能。

- `handle_trans_fault` 代码如下:

```
1      if (pa == 0) {
2          /*
3              * Not committed before. Then, allocate the
    physical
4              * page.
5              */
```

```

6      /* LAB 2 TODO 7 BEGIN */
7      /* BLANK BEGIN */
8      /* Hint: Allocate a physical page and clear it
to 0. */
9      pa = virt_to_phys(get_pages(0));
10     memset(phys_to_virt(pa), 0, PAGE_SIZE);
11     /* BLANK END */
12     /*
13      * Record the physical page in the radix tree:
14      * the offset is used as index in the radix tree
15      */
16     kdebug("commit: index: %ld, 0x%lx\n", index,
pa);
17     commit_page_to_pmo(pmo, index, pa);
18
19     /* Add mapping in the page table */
20     lock(&vmospace->pgtbl_lock);
21     /* BLANK BEGIN */
22     map_range_in_pgtbl(vmospace->pgtbl,
23                        fault_addr,
24                        pa,
25                        PAGE_SIZE,
26                        perm);
27     /* BLANK END */
28     unlock(&vmospace->pgtbl_lock);
29 } else {
30     /*
31     committed a
32
33     * physical page.
34     *
35     * For concurrent page faults:
36     *
37     threads
38
39     * When type is PMO_ANONYM, the later faulting
40
41     * of the process do not need to modify the page
42     * table because a previous faulting thread will
43     do
44
45     * that. (This is always true for the same
46     process)
47
48     * However, if one process map an anonymous pmo
49     for
50
51     * another process (e.g., main stack pmo), the
52     faulting
53
54     * thread (e.g., in the new process) needs to
55     update its
56
57     * page table.
58     * So, for simplicity, we just update the page
59     table.
60
61     * Note that adding the same mapping is
62     harmless.
63
64     *
65     threads
66
67     * When type is PMO_SHM, the later faulting
68
69     * needs to add the mapping in the page table.
70     * Repeated mapping operations are harmless.

```

```

50         */
51         if (pmo->type == PMO_SHM || pmo->type ==
PMO_ANONYM) {
52             /* Add mapping in the page table */
53             lock(&vmospace->pgtbl_lock);
54             /* BLANK BEGIN */
55             map_range_in_pgtbl(vmspace->pgtbl,
56                               fault_addr,
57                               pa,
58                               PAGE_SIZE,
59                               perm);
60             /* BLANK END */
61             /* LAB 2 TODO 7 END */
62             unlock(&vmospace->pgtbl_lock);
63         }
64     }

```

利用 `virt_to_phys(get_pages(0))`；分配物理页。

利用 `memset(phys_to_virt(pa), 0, PAGE_SIZE)`；清空物理页。

利用 `map_range_in_pgtbl(vmspace->pgtbl, fault_addr, pa, PAGE_SIZE, perm, &rss)`；对物理页进行映射。