


```

9  /* Step 2: map PHYSMEM_START ~ PERIPHERAL_BASE with 2MB granularity */
10 vaddr = PHYSMEM_START;
11 for (; vaddr < PERIPHERAL_BASE; vaddr += SIZE_2M) {
12     boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
13         (vaddr)
14         | UXN /* Unprivileged execute never */
15         | ACCESSED /* Set access flag */
16         | NG /* Mark as not global */
17         | INNER_SHARABLE /* Sharebility */
18         | NORMAL_MEMORY /* Normal memory */
19         | IS_VALID;
20 }
21
22 /* Step 2: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity */
23 for (vaddr = PERIPHERAL_BASE; vaddr < PHYSMEM_END; vaddr += SIZE_2M) {
24     boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
25         (vaddr)
26         | UXN /* Unprivileged execute never */
27         | ACCESSED /* Set access flag */
28         | NG /* Mark as not global */
29         | DEVICE_MEMORY /* Device memory */
30         | IS_VALID;
31 }
32 /* LAB 2 TODO 1 END */

```

思考题 3：请思考在 `init_boot_pt` 函数中为什么还要为低地址配置页表，并尝试验证自己的解释。

在 `start_kernel` 函数之前，使用的是物理地址，需要无缝地从物理地址切换到虚拟地址。

那么需要保证在之前使用物理地址必须和虚拟地址是一一对应的，这样切换到虚拟地址时是无缝的。否则启动 MMU 之后，使用虚拟地址的低地址，而这一段没有进行映射，会产生地址地址翻译错误。

练习题 4：完成 `kernel/mm/buddy.c` 中的 `split_page`、`buddy_get_pages`、`merge_page` 和 `buddy_free_pages` 函数中的 LAB 2 TODO 2 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。

```

1  static struct page *split_page(struct phys_mem_pool *pool, u64 order,
2                                struct page *page)
3  {
4      /* LAB 2 TODO 2 BEGIN */
5      /*

```

```

6      * Hint: Recursively put the buddy of current chunk into
7      * a suitable free list.
8      */
9
10     BUG_ON(page->order < order);
11     if(page->order==order) {
12         page->allocated = 1;
13         return page;
14     }
15     u64 order_tmp = page->order - 1;
16     page->order = order_tmp;
17     page->allocated = 0;
18     struct page *buddy = get_buddy_chunk(pool, page);
19     BUG_ON(buddy==NULL);
20     buddy->order = order_tmp;
21     buddy->allocated = 0;
22     list_add(&(buddy->node), &(pool->free_lists[order_tmp].free_list));
23     pool->free_lists[order_tmp].nr_free++;
24     return split_page(pool, order, page);
25
26     /* LAB 2 TODO 2 END */
27 }

```

对空闲chunk进行split操作

1. 首先要确保要分割的chunk的order比所需要的chunk的order大，否则出现问题
2. 如果order相同，直接返回
3. 否则把被分割的chunk的order减一，然后把这个chunk分成两个小的chunk
4. 选择其中一个chunk继续进行split，而另一个加入到free list
5. 递归调用直至找到对应order的chunk

```

1  struct page *buddy_get_pages(struct phys_mem_pool *pool, u64 order)
2  {
3      /* LAB 2 TODO 2 BEGIN */
4      /*
5       * Hint: Find a chunk that satisfies the order requirement
6       * in the free lists, then split it if necessary.
7       */
8
9      u64 i = order;
10     struct list_head *ptr = NULL;
11     while(i < BUDDY_MAX_ORDER) {
12         if(pool->free_lists[i].nr_free > 0) {
13             ptr = pool->free_lists[i].free_list.next;
14             break;
15         }
16         ++i;
17     }
18     if(ptr==NULL) {
19         return NULL;

```

```

20     }
21     list_del(ptr);
22     pool->free_lists[i].nr_free--;
23     struct page *page = list_entry(ptr, struct page, node);
24     return split_page(pool, order, page);
25
26     /* LAB 2 TODO 2 END */
27 }

```

1. 从最小的order依次向较大的order的free list寻找
2. 若最终没有找到，则说明没有足够空间，返回 `NULL`
3. 否则，将找到的order的chunk暂时从free list移除，然后对其进行split

```

1  static struct page *merge_page(struct phys_mem_pool *pool, struct page *page)
2  {
3      /* LAB 2 TODO 2 BEGIN */
4      /*
5       * Hint: Recursively merge current chunk with its buddy
6       * if possible.
7       */
8
9      int order = page->order;
10     if(order==BUDDY_MAX_ORDER-1) {
11         list_add(&(amp;page->node), &(pool->free_lists[order].free_list));
12         pool->free_lists[order].nr_free++;
13         return page;
14     }
15     struct page *buddy = get_buddy_chunk(pool, page);
16     if(buddy == NULL || buddy->allocated || page->order!=buddy->order) {
17         list_add(&(amp;page->node), &(pool->free_lists[order].free_list));
18         pool->free_lists[order].nr_free++;
19         return page;
20     }
21     else if(page->order==buddy->order) {
22         list_del(&(amp;buddy->node));
23         pool->free_lists[order].nr_free--;
24         u64 addr = (u64)page_to_virt(page) & (u64)page_to_virt(buddy);
25         struct page* upper_page = virt_to_page((void*)addr);
26         upper_page->order = order + 1;
27         return merge_page(pool, upper_page);
28     }
29     else {
30         BUG_ON(1);
31     }
32
33     /* LAB 2 TODO 2 END */
34 }

```

1. 如果要合并的chunk的order已经是最大的，就不能继续合并了，直接返回
2. 否则找到这个chunk的buddy，判断它的buddy是不是合法，并且没有被分配或者分割，如果都符合，可以将

这个chunk和它的buddy进行合并，然后将合并得到的大的chunk进行递归合并

3. 如果buddy不合法，或者已经被分配，或者已经被分割，那么就不能合并，直接返回

```
1 void buddy_free_pages(struct phys_mem_pool *pool, struct page *page)
2 {
3     /* LAB 2 TODO 2 BEGIN */
4     /*
5      * Hint: Merge the chunk with its buddy and put it into
6      * a suitable free list.
7      */
8
9     page->allocated = 0;
10    merge_page(pool, page);
11
12    /* LAB 2 TODO 2 END */
13 }
```

1. free一个chunk，将其allocated字段置为false
2. 尝试合并

练习题 5：完成 kernel/arch/aarch64/mm/page_table.c 中的 query_in_pgtbl、map_range_in_pgtbl、unmap_range_in_pgtbl 函数中的 LAB 2 TODO 3 部分，分别实现页表查询、映射、取消映射操作。

```
1 int query_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t *pa, pte_t **entry)
2 {
3     /* LAB 2 TODO 3 BEGIN */
4     /*
5      * Hint: Walk through each level of page table using `get_next_ptp`,
6      * return the pa and pte until a L0/L1 block or page, return
7      * `-ENOMAPPING` if the va is not mapped.
8      */
9
10    ptp_t *cur_ptp = (ptp_t *)pgtbl;
11    u32 level = 0;
12    ptp_t *next_ptp;
13    pte_t *pte;
14    int ret = get_next_ptp(cur_ptp, level, va, &next_ptp, &pte, false);
15    while(ret==NORMAL_PTP && level < 3) {
16        cur_ptp = next_ptp;
17        level++;
18        ret = get_next_ptp(cur_ptp, level, va, &next_ptp, &pte, false);
19    }
20    if(ret==BLOCK_PTP || ret==NORMAL_PTP) {
21        paddr_t offset, pfn;
```

```

22         switch (level) {
23             case 1:
24                 BUG_ON(ret!=BLOCK_PTP);
25                 offset = GET_VA_OFFSET_L1(va);
26                 pfn = pte->l1_block.pfn;
27                 *pa = (pfn << L1_INDEX_SHIFT) | offset;
28                 break;
29             case 2:
30                 BUG_ON(ret!=BLOCK_PTP);
31                 offset = GET_VA_OFFSET_L2(va);
32                 pfn = pte->l2_block.pfn;
33                 *pa = (pfn << L2_INDEX_SHIFT) | offset;
34                 break;
35             case 3:
36                 BUG_ON(ret!=NORMAL_PTP);
37                 offset = GET_VA_OFFSET_L3(va);
38                 pfn = pte->l3_page.pfn;
39                 *pa = (pfn << L3_INDEX_SHIFT) | offset;
40                 break;
41             default:
42                 BUG_ON(1);
43         }
44         *entry = pte;
45         return 0;
46     }
47     else {
48         return -ENOMAPPING;
49     }
50     /* LAB 2 TODO 3 END */
51 }

```

1. 进行一个循环，直到找到对应页（4K或者大页均可）的pte或未分配标志
2. 如果是页的pte，那么根据level判断是4K、2M还是1G，然后根据offset和pfn算出物理地址
3. 如果是未分配，根据传入参数决定是否要分配一个页

```

1  int map_range_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t pa, size_t len,
2                          vmr_prop_t flags)
3  {
4      /* LAB 2 TODO 3 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     */
11     BUG_ON(va % 0x1000ul != 0);
12     BUG_ON(len % 0x1000ul != 0);
13     ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
14     pte_t *l0_pte, *l1_pte, *l2_pte;
15     pte_t *entry;

```

```

16     size_t mapped = 0;
17     u32 index;
18     int ptp_type;
19
20     while(mapped < len) {
21         l0_ptp = (ptp_t*)pgtbl;
22         ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
23         ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
24         ptp_type = get_next_ptp(l2_ptp, 2, va + mapped, &l3_ptp, &l2_pte, true);
25         index = GET_L3_INDEX(va + mapped);
26         entry = &(l3_ptp->ent[index]);
27         set_pte_flags(entry, flags, USER_PTE);
28         entry->l3_page.is_page = 1;
29         entry->l3_page.is_valid = 1;
30         entry->l3_page.pfn = (pa + mapped) >> 12;
31         mapped += (1u << 12);
32     }
33     return 0;
34
35     /* LAB 2 TODO 3 END */
36 }

```

1. 进行一个while循环，如果还没有映射完全，就继续进行映射
2. 由于映射4K大小的page，因此需要三级页表，在第三级页表中存储的是4K page的pte
3. 将pfn等字段一一设置

```

1  int unmap_range_in_pgtbl(void *pgtbl, vaddr_t va, size_t len)
2  {
3      /* LAB 2 TODO 3 BEGIN */
4      /*
5       * Hint: Walk through each level of page table using `get_next_ptp`,
6       * mark the final level pte as invalid. Iterate until all pages are
7       * unmapped.
8       */
9
10     BUG_ON(va % 0x1000ul != 0);
11     BUG_ON(len % 0x1000ul != 0);
12     ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
13     pte_t *l0_pte, *l1_pte, *l2_pte;
14     pte_t *entry;
15     size_t mapped = 0;
16     u32 index;
17     int ptp_type;
18
19     while(mapped < len) {
20         l0_ptp = (ptp_t*)pgtbl;
21         ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
22         ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
23         ptp_type = get_next_ptp(l2_ptp, 2, va + mapped, &l3_ptp, &l2_pte, true);
24         index = GET_L3_INDEX(va + mapped);

```

```

25     entry = &(l3_ptp->ent[index]);
26     entry->l3_page.is_valid = 0;
27     mapped += (1u << 12);
28     }
29     return 0;
30
31     /* LAB 2 TODO 3 END */
32 }

```

unmap操作就是把map操作反过来执行一次，不再赘述

练习题 6：在上一个练习的函数中支持大页（2M、1G 页）映射，可假设取消映射的地址范围一定是某次映射的完整地址范围，即不会先映射一大块，再取消映射其中一小块。

```

1  int map_range_in_pgtbl_huge(void *pgtbl, vaddr_t va, paddr_t pa, size_t len,
2                                vmr_prop_t flags)
3  {
4      /* LAB 2 TODO 4 BEGIN */
5      BUG_ON(va % 0x1000ul != 0);
6      BUG_ON(len % 0x1000ul != 0);
7      ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
8      pte_t *l0_pte, *l1_pte, *l2_pte;
9      pte_t *entry;
10     size_t mapped = 0;
11     u32 index;
12     int ptp_type;
13
14     while(mapped < len) {
15
16         if(len >= (1 << 30) + mapped)
17             goto Map_1G;
18         else if(len >= (1 << 21) + mapped)
19             goto Map_2M;
20         else
21             goto Map_4K;
22
23     Map_1G:
24     {
25         l0_ptp = (ptp_t*)pgtbl;
26         ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
27         index = GET_L1_INDEX(va + mapped);
28         entry = &(l1_ptp->ent[index]);
29         set_pte_flags(entry, flags, USER_PTE);
30         entry->l1_block.is_table = 0;
31         entry->l1_block.is_valid = 1;
32         entry->l1_block.pfn = (pa + mapped) >> 30;
33         mapped += (1u << 30);

```



```

34         goto Loop;
35     }
36
37     Map_2M:
38     {
39         l0_ptp = (ptp_t*)pgtbl;
40         ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
41         ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
42         index = GET_L2_INDEX(va + mapped);
43         entry = &(l2_ptp->ent[index]);
44         set_pte_flags(entry, flags, USER_PTE);
45         entry->l2_block.is_table = 0;
46         entry->l2_block.is_valid = 1;
47         entry->l2_block.pfn = (pa + mapped) >> 21;
48         mapped += (1u << 21);
49         goto Loop;
50     }
51
52     Map_4K:
53     {
54         l0_ptp = (ptp_t*)pgtbl;
55         ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
56         ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
57         ptp_type = get_next_ptp(l2_ptp, 2, va + mapped, &l3_ptp, &l2_pte, true);
58         index = GET_L3_INDEX(va + mapped);
59         entry = &(l3_ptp->ent[index]);
60         set_pte_flags(entry, flags, USER_PTE);
61         entry->l3_page.is_page = 1;
62         entry->l3_page.is_valid = 1;
63         entry->l3_page.pfn = (pa + mapped) >> 12;
64         mapped += (1u << 12);
65         goto Loop;
66     }
67
68     Loop:
69     continue;
70 }
71 return 0;
72
73 /* LAB 2 TODO 4 END */
74 }

```

1. while循环，和映射4K page是一样的
2. 根据剩余的len选择是大页还是4K页
3. 如果是大页，就是一级页表（1G大页）或者二级页表（2M大页）
4. 对pte设置pfn等字段

```

1 int unmap_range_in_pgtbl_huge(void *pgtbl, vaddr_t va, size_t len)
2 {
3     /* LAB 2 TODO 4 BEGIN */

```

```

4      ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
5      pte_t *l0_pte, *l1_pte, *l2_pte;
6      pte_t *entry;
7      size_t mapped = 0;
8      u32 index;
9      int ptp_type;
10
11     while(mapped < len) {
12
13         if(len >= (1 << 30) + mapped)
14             goto Map_1G;
15         else if(len >= (1 << 21) + mapped)
16             goto Map_2M;
17         else
18             goto Map_4K;
19
20     Map_1G:
21     {
22         l0_ptp = (ptp_t*)pgtbl;
23         ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
24         index = GET_L1_INDEX(va + mapped);
25         entry = &(l1_ptp->ent[index]);
26         entry->l1_block.is_valid = 0;
27         mapped += (1u << 30);
28         goto Loop;
29     }
30
31     Map_2M:
32     {
33         l0_ptp = (ptp_t*)pgtbl;
34         ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
35         ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
36         index = GET_L2_INDEX(va + mapped);
37         entry = &(l2_ptp->ent[index]);
38         entry->l2_block.is_valid = 0;
39         mapped += (1u << 21);
40         goto Loop;
41     }
42
43     Map_4K:
44     {
45         l0_ptp = (ptp_t*)pgtbl;
46         ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
47         ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
48         ptp_type = get_next_ptp(l2_ptp, 2, va + mapped, &l3_ptp, &l2_pte, true);
49         index = GET_L3_INDEX(va + mapped);
50         entry = &(l3_ptp->ent[index]);
51         entry->l3_page.is_valid = 0;
52         mapped += (1u << 12);
53         goto Loop;
54     }
55

```

```

56         Loop:
57             continue;
58         }
59         return 0;
60
61         /* LAB 2 TODO 4 END */
62     }

```

和unmap普通的4K页相似，只是需要判断一下映射的是大页还是4K页，不再赘述

思考题 7：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，并在发生缺页异常（实际上是permission fault）时如何处理。

Copy-on-Write是通过access control进行配置的，可以设置AP字段为Read-only

这样，当一个程序在写内存时因为权限不足处罚缺页异常，操作系统检测缺页异常发现是由于尝试修改只读内存，这时就会将对应的内存重新分配一个物理页，同时将权限位配置为可读可写，重新映射给该程序。

思考题 8：为了简单起见，在 ChCore 实验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。

可能导致大页内存分配了但是未使用完，造成内存的浪费

可能会产生很多内存碎片，减少内存的利用率

挑战题 9：使用前面实现的 page_table.c 中的函数，在内核启动后重新配置内核页表，进行细粒度的映射。

```

1  // ttbr0_l0 virtual addr
2  u64 ttbr1_l0 = get_pages(0);
3  vmr_prop_t flag1, flag2, flag3;
4  flag1 = 0;
5  flag2 = VMR_DEVICE;
6  flag3 = VMR_DEVICE;
7  map_range_in_pgtbl(ttbr1_l0, 0xffffffff00000000, 0x0, 0x3f000000ul, flag1);
8  map_range_in_pgtbl(ttbr1_l0, 0xffffffff00000000 + 0x3f000000ul, 0x3f000000ul, 0x40000000ul
- 0x3f000000ul, flag2);
9  map_range_in_pgtbl(ttbr1_l0, 0xffffffff00000000 + 0x40000000ul, 0x40000000ul,
0x40000000ul, flag3);

```

```
10  u64 phy_addr = virt_to_phys(ttbr1_l0);
11  // TODO : inline assemble code
12  asm volatile("msr ttbr0_el1, %[value]" : :[value] "r" (phy_addr) :);
13  flush_tlb_all();
14  kinfo("remap finished\n");
```

在 `mm_init` 函数的最后加入以上代码，实现对内核页表的细粒度重映射。