Lab-Report-2

519021910685 王浩天

Grizzy@sjtu.edu.cn

思考题 1：请思考多级页表相比单级页表带来的优势和劣势（如果有的话），并计算在 AArch64 页表中分别以 4KB 粒度和 2MB 粒度映射 0～4GB 地址范围所需的物理内存大小（或页表页数量）。

优势：多级页表相对于单级页表，可以显著减少页表的大小，否则一张页表映射4kb全部64位地址空间就需要33554432GB的空间，而多级页表允许页表结构存在空洞，可以部分创建，减少页表占据的空间。

劣势：需要多次访存，复杂度高

若以4KB映射4GB的地址范围，则一共需要映射2^{20}页

需要2^{20}个三级页表条目，即2^{11}个三级页表

需要2^{11}个二级页表条目，即2^{2}个二级页表

共计2048 + 4 + 1 + 1 = 2054个页表项

若以2MB粒度映射4GB的地址范围，一共需要映射2^{11}页

需要2^{11}个二级页表条目，即2^{2}个二级页表

共计4 + 1 + 1 = 6个页表

练习题 2：请在 init_boot_pt 函数的 LAB 2 TODO 1 处配置内核高地址页表（boot_ttbr1_l0、 boot_ttbr1_l1 和 boot_ttbr1_l2），以 2MB 粒度映射。

```
/* LAB 2 TODO 1 BEGIN */
/* Step 1: set L0 and L1 page table entry */
vaddr = KERNEL_VADDR;
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;

```

```
/* Step 2: map PHYSMEM_START ~ PERIPHERAL_BASE with 2MB granularity */
vaddr = PHYSMEM_START;
for (; vaddr < PERIPHERAL_BASE; vaddr += SIZE_2M) {
        boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                (vaddr)
                | UXN /* Unprivileged execute never */
                | ACCESSED /* Set access flag */
                | NG /* Mark as not global */
                | INNER_SHARABLE /* Sharebility */
                | NORMAL_MEMORY /* Normal memory */
                | IS_VALID;
}

/* Step 2: map PERIPHERAL_BASE ~ PHYSMEM_END with 2MB granularity */
for (vaddr = PERIPHERAL_BASE; vaddr < PHYSMEM_END; vaddr += SIZE_2M) {
        boot_ttbr1_l2[GET_L2_INDEX(vaddr)] =
                (vaddr)
                | UXN /* Unprivileged execute never */
                | ACCESSED /* Set access flag */
                | NG /* Mark as not global */
                | DEVICE_MEMORY /* Device memory */
                | IS_VALID;
}
/* 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。

```
static struct page *split_page(struct phys_mem_pool *pool, u64 order,
                               struct page *page)
{
        /* LAB 2 TODO 2 BEGIN */
        /*
```

```
         * Hint: Recursively put the buddy of current chunk into
         * a suitable free list.
         */

        BUG_ON(page->order < order);
        if(page->order==order) {
                page->allocated = 1;
                return page;
        }
        u64 order_tmp = page->order - 1;
        page->order = order_tmp;
        page->allocated = 0;
        struct page *buddy = get_buddy_chunk(pool, page);
        BUG_ON(buddy==NULL);
        buddy->order = order_tmp;
        buddy->allocated = 0;
        list_add(&(buddy->node), &(pool->free_lists[order_tmp].free_list));
        pool->free_lists[order_tmp].nr_free++;
        return split_page(pool, order, page);

        /* LAB 2 TODO 2 END */
}
```

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

```
struct page *buddy_get_pages(struct phys_mem_pool *pool, u64 order)
{
        /* LAB 2 TODO 2 BEGIN */
        /*
         * Hint: Find a chunk that satisfies the order requirement
         * in the free lists, then split it if necessary.
         */

        u64 i = order;
        struct list_head *ptr = NULL;
        while(i < BUDDY_MAX_ORDER) {
                if(pool->free_lists[i].nr_free > 0) {
                        ptr = pool->free_lists[i].free_list.next;
                        break;
                }
                ++i;
        }
        if(ptr==NULL) {
                return NULL;
```

```
	}
	list_del(ptr);
	pool->free_lists[i].nr_free--;
	struct page *page = list_entry(ptr, struct page, node);
	return split_page(pool, order, page);

	/* LAB 2 TODO 2 END */
}
```

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

```
static struct page *merge_page(struct phys_mem_pool *pool, struct page *page)
{
	/* LAB 2 TODO 2 BEGIN */
	/*
	 * Hint: Recursively merge current chunk with its buddy
	 * if possible.
	 */

	int order = page->order;
	if(order==BUDDY_MAX_ORDER-1) {
		list_add(&(page->node), &(pool->free_lists[order].free_list));
		pool->free_lists[order].nr_free++;
		return page;
	}
	struct page *buddy = get_buddy_chunk(pool, page);
	if(buddy == NULL || buddy->allocated || page->order!=buddy->order) {
		list_add(&(page->node), &(pool->free_lists[order].free_list));
		pool->free_lists[order].nr_free++;
		return page;
	}
	else if(page->order==buddy->order) {
		list_del(&(buddy->node));
		pool->free_lists[order].nr_free--;
		u64 addr = (u64)page_to_virt(page) & (u64)page_to_virt(buddy);
		struct page* upper_page = virt_to_page((void*)addr);
		upper_page->order = order + 1;
		return merge_page(pool, upper_page);
	}
	else {
		BUG_ON(1);
	}

	/* LAB 2 TODO 2 END */
}
```

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

```
void buddy_free_pages(struct phys_mem_pool *pool, struct page *page)
{
        /* LAB 2 TODO 2 BEGIN */
        /*
         * Hint: Merge the chunk with its buddy and put it into
         * a suitable free list.
         */

        page->allocated = 0;
        merge_page(pool, page);

        /* LAB 2 TODO 2 END */
}
```

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。

```
int query_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t *pa, pte_t **entry)
{
        /* LAB 2 TODO 3 BEGIN */
        /*
         * Hint: Walk through each level of page table using `get_next_ptp`,
         * return the pa and pte until a L0/L1 block or page, return
         * `-ENOMAPPING` if the va is not mapped.
         */

        ptp_t *cur_ptp = (ptp_t *)pgtbl;
        u32 level = 0;
        ptp_t *next_ptp;
        pte_t *pte;
        int ret = get_next_ptp(cur_ptp, level, va, &next_ptp, &pte, false);
        while(ret==NORMAL_PTP && level < 3) {
                cur_ptp = next_ptp;
                level++;
                ret = get_next_ptp(cur_ptp, level, va, &next_ptp, &pte, false);
        }
        if(ret==BLOCK_PTP || ret==NORMAL_PTP) {
                paddr_t offset, pfn;
```

```
                switch (level) {
                        case 1:
                                BUG_ON(ret!=BLOCK_PTP);
                                offset = GET_VA_OFFSET_L1(va);
                                pfn = pte->l1_block.pfn;
                                *pa = (pfn << L1_INDEX_SHIFT) | offset;
                                break;
                        case 2:
                                BUG_ON(ret!=BLOCK_PTP);
                                offset = GET_VA_OFFSET_L2(va);
                                pfn = pte->l2_block.pfn;
                                *pa = (pfn << L2_INDEX_SHIFT) | offset;
                                break;
                        case 3:
                                BUG_ON(ret!=NORMAL_PTP);
                                offset = GET_VA_OFFSET_L3(va);
                                pfn = pte->l3_page.pfn;
                                *pa = (pfn << L3_INDEX_SHIFT) | offset;
                                break;
                        default:
                                BUG_ON(1);
                }
                *entry = pte;
                return 0;
        }
        else {
                return -ENOMAPPING;
        }
        /* LAB 2 TODO 3 END */
}
```

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

```
int map_range_in_pgtbl(void *pgtbl, vaddr_t va, paddr_t pa, size_t len,
                       vmr_prop_t flags)
{
        /* LAB 2 TODO 3 BEGIN */
        /*
         * Hint: Walk through each level of page table using `get_next_ptp`,
         * create new page table page if necessary, fill in the final level
         * pte with the help of `set_pte_flags`. Iterate until all pages are
         * mapped.
         */
        BUG_ON(va % 0x1000ul != 0);
        BUG_ON(len % 0x1000ul != 0);
        ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
        pte_t *l0_pte, *l1_pte, *l2_pte;
        pte_t *entry;
```

```
        size_t mapped = 0;
        u32 index;
        int ptp_type;

        while(mapped < len) {
        l0_ptp = (ptp_t*)pgtbl;
        ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
        ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
        ptp_type = get_next_ptp(l2_ptp, 2, va + mapped, &l3_ptp, &l2_pte, true);
        index = GET_L3_INDEX(va + mapped);
        entry = &(l3_ptp->ent[index]);
        set_pte_flags(entry, flags, USER_PTE);
        entry->l3_page.is_page = 1;
        entry->l3_page.is_valid = 1;
        entry->l3_page.pfn = (pa + mapped) >> 12;
        mapped += (1u << 12);
        }
        return 0;

        /* LAB 2 TODO 3 END */
}
```

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

```
int unmap_range_in_pgtbl(void *pgtbl, vaddr_t va, size_t len)
{
        /* LAB 2 TODO 3 BEGIN */
        /*
         * Hint: Walk through each level of page table using `get_next_ptp`,
         * mark the final level pte as invalid. Iterate until all pages are
         * unmapped.
         */

        BUG_ON(va % 0x1000ul != 0);
        BUG_ON(len % 0x1000ul != 0);
        ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
        pte_t *l0_pte, *l1_pte, *l2_pte;
        pte_t *entry;
        size_t mapped = 0;
        u32 index;
        int ptp_type;

        while(mapped < len) {
        l0_ptp = (ptp_t*)pgtbl;
        ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
        ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
        ptp_type = get_next_ptp(l2_ptp, 2, va + mapped, &l3_ptp, &l2_pte, true);
        index = GET_L3_INDEX(va + mapped);
```

```
            entry = &(l3_ptp->ent[index]);
            entry->l3_page.is_valid = 0;
            mapped += (1u << 12);
            }
            return 0;

            /* LAB 2 TODO 3 END */
    }
```

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。

```
int map_range_in_pgtbl_huge(void *pgtbl, vaddr_t va, paddr_t pa, size_t len,
                            vmr_prop_t flags)
{
        /* LAB 2 TODO 4 BEGIN */
        BUG_ON(va % 0x1000ul != 0);
        BUG_ON(len % 0x1000ul != 0);
        ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
        pte_t *l0_pte, *l1_pte, *l2_pte;
        pte_t *entry;
        size_t mapped = 0;
        u32 index;
        int ptp_type;

        while(mapped < len) {

        if(len >= (1 << 30) + mapped)
                goto Map_1G;
        else if(len >= (1 << 21) + mapped)
                goto Map_2M;
        else
                goto Map_4K;

        Map_1G:
        {
        l0_ptp = (ptp_t*)pgtbl;
        ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
        index = GET_L1_INDEX(va + mapped);
        entry = &(l1_ptp->ent[index]);
        set_pte_flags(entry, flags, USER_PTE);
        entry->l1_block.is_table = 0;
        entry->l1_block.is_valid = 1;
        entry->l1_block.pfn = (pa + mapped) >> 30;
        mapped += (1u << 30);
```

```
        goto Loop;
        }

        Map_2M:
        {
        l0_ptp = (ptp_t*)pgtbl;
        ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
        ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
        index = GET_L2_INDEX(va + mapped);
        entry = &(l2_ptp->ent[index]);
        set_pte_flags(entry, flags, USER_PTE);
        entry->l2_block.is_table = 0;
        entry->l2_block.is_valid = 1;
        entry->l2_block.pfn = (pa + mapped) >> 21;
        mapped += (1u << 21);
        goto Loop;
        }

        Map_4K:
        {
        l0_ptp = (ptp_t*)pgtbl;
        ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
        ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
        ptp_type = get_next_ptp(l2_ptp, 2, va + mapped, &l3_ptp, &l2_pte, true);
        index = GET_L3_INDEX(va + mapped);
        entry = &(l3_ptp->ent[index]);
        set_pte_flags(entry, flags, USER_PTE);
        entry->l3_page.is_page = 1;
        entry->l3_page.is_valid = 1;
        entry->l3_page.pfn = (pa + mapped) >> 12;
        mapped += (1u << 12);
        goto Loop;
        }

        Loop:
        continue;
        }
        return 0;

        /* LAB 2 TODO 4 END */
}
```

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

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

```
        ptp_t *l0_ptp, *l1_ptp, *l2_ptp, *l3_ptp;
        pte_t *l0_pte, *l1_pte, *l2_pte;
        pte_t *entry;
        size_t mapped = 0;
        u32 index;
        int ptp_type;

        while(mapped < len) {

        if(len >= (1 << 30) + mapped)
                goto Map_1G;
        else if(len >= (1 << 21) + mapped)
                goto Map_2M;
        else
                goto Map_4K;

        Map_1G:
        {
        l0_ptp = (ptp_t*)pgtbl;
        ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
        index = GET_L1_INDEX(va + mapped);
        entry = &(l1_ptp->ent[index]);
        entry->l1_block.is_valid = 0;
        mapped += (1u << 30);
        goto Loop;
        }

        Map_2M:
        {
        l0_ptp = (ptp_t*)pgtbl;
        ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
        ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
        index = GET_L2_INDEX(va + mapped);
        entry = &(l2_ptp->ent[index]);
        entry->l2_block.is_valid = 0;
        mapped += (1u << 21);
        goto Loop;
        }

        Map_4K:
        {
        l0_ptp = (ptp_t*)pgtbl;
        ptp_type = get_next_ptp(l0_ptp, 0, va + mapped, &l1_ptp, &l0_pte, true);
        ptp_type = get_next_ptp(l1_ptp, 1, va + mapped, &l2_ptp, &l1_pte, true);
        ptp_type = get_next_ptp(l2_ptp, 2, va + mapped, &l3_ptp, &l2_pte, true);
        index = GET_L3_INDEX(va + mapped);
        entry = &(l3_ptp->ent[index]);
        entry->l3_page.is_valid = 0;
        mapped += (1u << 12);
        goto Loop;
        }

```

```
        Loop:
        continue;
        }
        return 0;

        /* LAB 2 TODO 4 END */
}
```

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，这是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

程序有诸如data、bss、text等多个段，这些段的权限是不同的，不能用一个粗粒度的映射一概而论

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。

```
// ttbr1_el1 virtual addr
u64 ttbr1_el1 = get_pages(0);
volatile u64 a = 1;
vmr_prop_t flag1, flag2, flag3;
flag1 = 0;
flag2 = VMR_DEVICE;
flag3 = VMR_DEVICE;
map_range_in_pgtbl(ttbr1_el1, 0xffffff0000000000, 0x0, 0x3f000000, flag1);
map_range_in_pgtbl(ttbr1_el1, 0xffffff003f000000, 0x3f000000ul, 0x1000000ul, flag2);
map_range_in_pgtbl(ttbr1_el1, 0xffffff0040000000, 0x40000000ul, 0x40000000ul, flag3);
```

```
u64 phy_addr = virt_to_phys(ttbr1_el1);
asm volatile("msr ttbr1_el1, %[value]" : :[value] "r" (phy_addr));
flush_tlb_all();
kinfo("[remap] 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。

注意，在 `set_pte_flags` 中非常容易出错，需要将 `PXN` 设置为 `false`，以下是修改过后的函数

```
static int set_pte_flags(pte_t *entry, vmr_prop_t flags, int kind)
{
        if(kind == KERNEL_PTE) {
                // attention : set PXN false
                // kernel may execute in previleged permission
                entry->l3_page.PXN = AARCH64_MMU_ATTR_PAGE_PX;
                entry->l3_page.UXN = AARCH64_MMU_ATTR_PAGE_UXN;
                entry->l3_page.AF = AARCH64_MMU_ATTR_PAGE_AF_ACCESSED;
                entry->l3_page.nG = 1;
                entry->l3_page.SH = INNER_SHAREABLE;
                if (flags & VMR_DEVICE) {
                        entry->l3_page.attr_index = DEVICE_MEMORY;
                        entry->l3_page.SH = 0;
                } else if (flags & VMR_NOCACHE) {
                        entry->l3_page.attr_index = NORMAL_MEMORY_NOCACHE;
                } else {
                        entry->l3_page.attr_index = NORMAL_MEMORY;
                }
                return 0;
        }

        /*
         * Current access permission (AP) setting:
         * Mapped pages are always readable (No considering XOM).
         * EL1 can directly access EL0 (No restriction like SMAP
         * as ChCore is a microkernel).
         */
        if (flags & VMR_WRITE)
                entry->l3_page.AP = AARCH64_MMU_ATTR_PAGE_AP_HIGH_RW_EL0_RW;
        else
                entry->l3_page.AP = AARCH64_MMU_ATTR_PAGE_AP_HIGH_RO_EL0_RO;

        if (flags & VMR_EXEC)
                entry->l3_page.UXN = AARCH64_MMU_ATTR_PAGE_UX;
        else
                entry->l3_page.UXN = AARCH64_MMU_ATTR_PAGE_UXN;

        // EL1 cannot directly execute EL0 accessiable region.
        entry->l3_page.PXN = AARCH64_MMU_ATTR_PAGE_PXN;
        // Set AF (access flag) in advance.
        entry->l3_page.AF = AARCH64_MMU_ATTR_PAGE_AF_ACCESSED;
        // Mark the mapping as not global
        entry->l3_page.nG = 1;
```

```
        // Mark the mappint as inner sharable
        entry->l3_page.SH = INNER_SHAREABLE;
        // Set the memory type
        if (flags & VMR_DEVICE) {
                entry->l3_page.attr_index = DEVICE_MEMORY;
                entry->l3_page.SH = 0;
        } else if (flags & VMR_NOCACHE) {
                entry->l3_page.attr_index = NORMAL_MEMORY_NOCACHE;
        } else {
                entry->l3_page.attr_index = NORMAL_MEMORY;
        }

        return 0;
}
```