Lab8

一、 实验目的

1. 系统掌握分页式内存管理机制的核心原理与实现方式
2. 掌握页表访问机制及虚拟地址到物理地址的转换过程
3. 在操作系统内核层面实现完整的内存管理功能

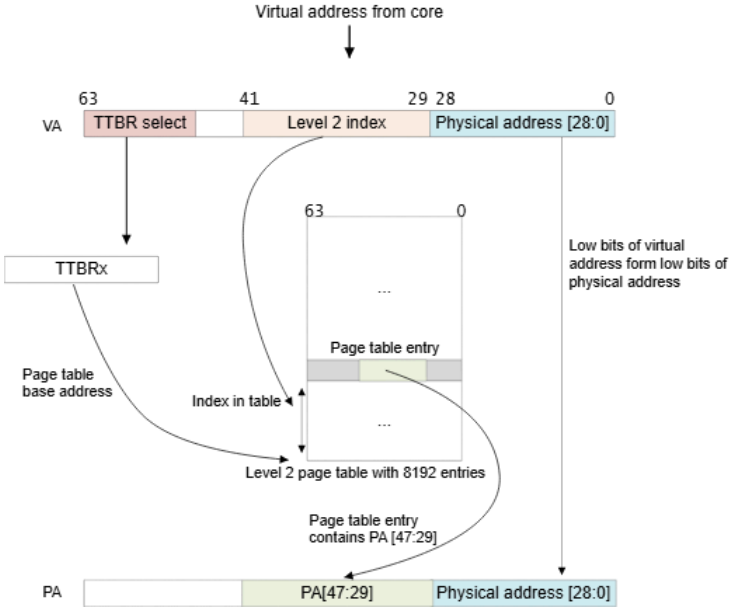
二、 实验过程

1.首先需要理解Armv8架构的地址转换：

Armv8架构采用双页表基址寄存器设计：TTBR0_ELn和TTBR1_ELn。TTBR0负责管理虚拟地址空间的下半部分，通常分配给用户应用程序使用；TTBR1则管理虚拟地址空间的上半部分，主要供内核使用。值得注意的是，TTBR0在EL1、EL2和EL3三个异常级别都存在，而TTBR1仅存在于EL1级别。

虚拟地址的最高位决定了使用哪个基址寄存器进行转换：当最高位为1时使用TTBR1_ELx，为0时则使用TTBR0_ELx。这种设计理念表明，不是寄存器选择决定了地址空间的使用，而是地址空间的选择决定了使用哪个寄存器进行转换。

下图展示了一个典型的48位虚拟地址转换示例，采用4级页表结构，每页大小为4KB：



页表从左到右依次为0级页表到3级页表。

举例说明转换过程：

输入虚拟地址：0xFFFF FFFF F000

转换步骤如下：

1. 使用bit[47:39]作为索引，从页表基址获取下一级页表基址(0x1000)
2. 使用bit[38:30]作为索引，从0x1000获取下一级页表基址(0x2000)
3. 使用bit[29:21]作为索引，从0x2000获取下一级页表基址(0x3000)
4. 使用bit[20:12]作为索引，从0x3000获取物理页地址(0x80000000)
5. 最终物理地址为页地址加上偏移量：0x80000000 + 0x0000 = 0x80000000

输出物理地址：0x80000000

这个例子展示了4级页表模式下，将虚拟地址0xFFFF FFFF F000映射到物理地址0X80000000的完整过程。每级页表最多包含512个表项，整个映射过程仅占用16KB内存空间，相比单级页表显著减少了内存占用。

2.接着分版块对本实验的核心代码mmu.c进行分析：

```
14  static mmu_mmap_region_s g_mem_map_info[] = {
15      {
16          .virt      = 0x0,
17          .phys      = 0x0,
18          .size      = 0x40000000, // 1G size
19          .max_level = 0x2,  // 不应大于3
20          .attrs     = MMU_ATTR_DEVICE_NGNRNE | MMU_ACCESS_RWX, // 设备
21      }, {
22          .virt      = 0x40000000,
23          .phys      = 0x40000000,
24          .size      = 0x40000000, // 1G size
25          .max_level = 0x2, // // 不应大于3
26          .attrs     = MMU_ATTR_CACHE_SHARE | MMU_ACCESS_RWX, // 内存
27      }
28  };
```

这个数组g_mem_map_info定义了虚拟地址到物理地址的映射关系，每个数组元素表示一个内存映射区域，包含以下信息：虚拟起始地址、物理起始地址、映射区域大小、最大页表级别以及内存属性。

```c
static U64 mmu_get_tcr(U32 *pips, U32 *pva_bits)
{
    U64 max_addr = 0;
    U64 ips, va_bits;
    U64 tcr;
    U32 i;
    U32 mmu_table_num = sizeof(g_mem_map_info) / sizeof(mmu_mmap_region_s);

    // 根据g_mem_map_info表计算所使用的虚拟地址的最大值
    for (i = 0; i < mmu_table_num; ++i) {
        max_addr = MAX(max_addr, g_mem_map_info[i].virt + g_mem_map_info
        [i].size);
    }

    // 依据虚拟地址最大值计算虚拟地址所需的位数，
    // 实际上应该分别计算物理地址的ips和虚拟地址的va_bits，而不是如下同时进行。
    if (max_addr > (1ULL << MMU_BITS_44)) {
        ips = MMU_PHY_ADDR_LEVEL_5;
```

这个函数的主要功能是根据g_mem_map_info提供的内存映射信息，计算并构建ARM处理器MMU所需的TCR（转换控制寄存器）值。该寄存器负责配置内存管理单元的地址转换机制。

根据Armv8架构手册，TCR寄存器各字段功能如下：

比特位	长度	功能说明
IPS	b001 << 32	36位地址空间，支持64GB寻址
TG1	b10 << 30	TTBR1_EL1使用4KB页大小
SH1	b11 << 28	页表内存区域设置为内部可共享
ORGN1	b01 << 26	页表内存使用外部可写回缓存策略
IRGN1	b01 << 24	页表内存使用内部可写回缓存策略
EPD1	b0 << 23	启用TTBR1_EL1进行页表遍历
A1	b1 << 22	TTBR1_EL1.ASID定义ASID
T1SZ	b011100 << 16	虚拟地址空间为36位
TG0	b00 << 14	使用4KB页大小
SH0	b11 << 12	页表内存区域设置为内部可共享
ORGN0	b01 << 10	页表内存使用外部可写回缓存策略
IRGN0	b01 << 8	页表内存使用内部可写回缓存策略
EPD0	b0 << 7	启用TTBR0_EL1进行页表遍历
0	b0 << 6	保留位
T0SZ	b011100 << 0	虚拟地址空间为36位

```
90  static U32 mmu_get_pte_type(U64 const *pte)
91  {
92      return (U32)(*pte & PTE_TYPE_MASK);
93  }
94
95  // 根据页表项级别计算当个页表项表示的范围（位数）
96  static U32 mmu_level2shift(U32 level)
97  {
98      if (g_mmu_ctrl.granule == MMU_GRANULE_4K) {
99          return (U32)(MMU_BITS_12 + MMU_BITS_9 * (MMU_LEVEL_3 - level));
100     } else {
101         return (U32)(MMU_BITS_16 + MMU_BITS_13 * (MMU_LEVEL_3 - level));
102     }
103 }
```

第一个函数用于获取页表项类型，第二个函数则根据页表级别返回对应的位移量，用于计算每个页表项所表示的地址范围。

```
106 static U64 *mmu_find_pte(U64 addr, U32 level)
107 {
108     U64 *pte = NULL;
109     U64 idx;
110     U32 i;
111
112     if (level < g_mmu_ctrl.start_level) {
113         return NULL;
114     }
115
116     pte = (U64 *)g_mmu_ctrl.tlb_addr;
117
118     // 从顶级页表开始，直到找到所需level级别的页表项或返回NULL
119     for (i = g_mmu_ctrl.start_level; i < MMU_LEVEL_MAX; ++i) {
120         // 依据级别i计算页表项在页表中的索引idx
121         if (g_mmu_ctrl.granule == MMU_GRANULE_4K) {
122             idx = (addr >> mmu_level2shift(i)) & 0x1FF;
123         } else {
124             idx = (addr >> mmu_level2shift(i)) & 0x1FFF;
125         }
126
```

这个函数的功能是在页表中定位给定地址和级别的页表项地址。

```
153  static U64 *mmu_create_table(void)
154  {
155      U32 pt_len;
156      U64 *new_table = (U64 *)g_mmu_ctrl.tlb_fillptr;
157
158      if (g_mmu_ctrl.granule == MMU_GRANULE_4K) {
159          pt_len = MAX_PTE_ENTRIES_4K * sizeof(U64);
160      } else {
161          pt_len = MAX_PTE_ENTRIES_64K * sizeof(U64);
162      }
163
164      // 根据页表粒度在页表区域新建一个页表（4K或64K）
165      g_mmu_ctrl.tlb_fillptr += pt_len;
166
167      if (g_mmu_ctrl.tlb_fillptr - g_mmu_ctrl.tlb_addr > g_mmu_ctrl.
         tlb_size) {
168          return NULL;
169      }
```

mmu_creat_table函数根据指定的页表粒度（4KB或64KB）在页表区域创建新页表。新创建的页表初始状态为空，所有表项均设置为PTE_TYPE_FAULT类型，最后返回该页表的起始地址。

```
216  static S32 mmu_add_map(mmu_mmap_region_s const *map)
217  {
218      U64 virt = map->virt;
219      U64 phys = map->phys;
220      U64 max_level = map->max_level;
221      U64 start_level = g_mmu_ctrl.start_level;
222      U64 block_size = 0;
223      U64 map_size = 0;
224      U32 level;
225      U64 *pte = NULL;
226      S32 ret;
227
228      if (map->max_level <= start_level) {
229          return -2;
230      }
231
```

这个函数实现向MMU添加映射的核心功能：通过循环从起始级别开始，为指定地址范围建立映射关系。在每一级别，它首先尝试定位对应的页表项（调用mmu_find_pte函数），若查找失败则返回错误。对于找到的页表项，调用mmu_add_map_pte_process函数进行处理，包括创建下级页表（当前页表项无效且非末级）或直接设置页表项值（末级页表或达到指定级别）。成功添加映射后，更新虚拟地址、物理地址和已映射大小，继续处理剩余部分，直到完成整个映射区域的设置。

```
261  static inline void mmu_set_ttbr_tcr_mair(U64 table, U64 tcr, U64 attr)
262  {
263      OS_EMBED_ASM("dsb sy");
264
265      OS_EMBED_ASM("msr ttbr0_el1, %0" : : "r" (table) : "memory");
266      // OS_EMBED_ASM("msr ttbr1_el1, %0" : : "r" (table) : "memory");
267      OS_EMBED_ASM("msr tcr_el1, %0" : : "r" (tcr) : "memory");
268      OS_EMBED_ASM("msr mair_el1, %0" : : "r" (attr) : "memory");
269
270      OS_EMBED_ASM("isb");
271  }
272
```

这个函数通过嵌入式汇编指令直接操作处理器寄存器，设置TTBR、TCR和MAIR寄存器的值。这些寄存器共同配置处理器的内存管理单元，控制虚拟地址到物理地址的转换过程。

根据Armv8架构手册，MAIR寄存器支持定义8种预设内存属性，可分为memory类型和device类型。memory类型可配置缓存策略（如write back、write through等），device类型则不可缓存，主要用于外设寄存器访问。

MAIR寄存器主要包含三种属性：

- G（Gathering）：是否允许访问合并
- R（Re-ordering）：是否支持访问重排序
- E（Early Write Acknowledgement）：是否允许总线提前返回写响应

3.启用MMU

```
45      // 启用 MMU
46      BL      mmu_init
47      // 进入 main 函数
48      B       OsEnterMain
49
```

三、测试及分析

程序执行结果正常：

四、 Lab8作业

启用TTBR1，将地址映射到虚拟地址空间的高半部分，使用高地址访问串口，具体修改如下：

1. 在src/bsp目录下的print.c文件中添加宏定义：

```c
#define UART_0_REG_BASE (0xffffffff00000000 + 0x09000000)
```

2. 在src/bsp目录下的hwi_init.c文件中添加宏定义：

```c
#define GIC_DIST_BASE               (0xffffffff00000000 +
0x08000000)
#define GIC_CPU_BASE                (0xffffffff00000000 +
0x08010000)
```

3. **设置虚拟地址高半部分的页表基地址 (Translation Table Base Register 1 (EL1))**（修改 `mmu.c` 的 `mmu_set_ttbr_tcr_mair` 函数）

当前代码中只写入了 TTBR0_EL1，而 TTBR1_EL1 的写入被注释掉。应恢复写入 TTBR1_EL1，例如：

```c
OS_EMBED_ASM("msr ttbr1_el1, %0" : : "r"(table) : "memory");
```

4. **设置 TTBR1 区域大小（TCR_T1SZ）**（修改 `mmu.c` 的 TCR 配置代码）

函数 `mmu_get_tcr` 目前只设置了 `TCR_T0SZ`，对应映射低地址空间大小。要启用高地址区，需额外设置 `TCR_T1SZ`，例如：

```c
tcr |= ((64 - g_mmu_ctrl.va_bits) << 16);  // 相当于 TCR_T1SZ(va_bits)
```

- 这样设置了 TCR_EL1 寄存器中的 T1SZ 字段，即虚拟地址的高半部分（由 TTBR1_EL1 处理）使用多少位地址空间。
- `T1SZ` 表示：高地址空间使用多少位虚拟地址（由 TTBR1 管理）
- 它的值为 `(64 - 虚拟地址位数)`，这是 ARM 架构的规定

5. **修改 `g_mem_map_info` 映射表**（在 `mmu.c` 顶部）

在现有映射表 `g_mem_map_info` 中新增外设的高地址映射项。例如，在数组末尾添加：

```c
{
    .virt = 0xffffffff08000000,     // 高地址空间 GIC 映射起始
    .phys = 0x08000000,             // GIC 物理基地址
    .size = 0x00200000,            // 覆盖 0x08000000~0x08100000 范围（2MB 对齐）
    .max_level = 0x2,
    .attrs = MMU_ATTR_DEVICE_NGNRNE | MMU_ACCESS_RW,  // 设备类型、可读写
},
{
    .virt = 0xffffffff09000000,
    .phys = 0x09000000,             // UART 物理基址
    .size = 0x00100000,            // 1MB 对齐
    .max_level = 0x2,
    .attrs = MMU_ATTR_DEVICE_NGNRNE | MMU_ACCESS_RW,
}
```

这会在 TTBR1 映射空间内建立从 `0xffffffff08000000` 和 `0xffffffff09000000` 起的虚拟映射，分别对应物理 `0x08000000`（GIC）和 `0x09000000`（UART）。
`MMU_ATTR_DEVICE_NGNRNE` 保证设备类型映射无缓存，`MMU_ACCESS_RW` 允许读写。

6. 运行结果：