一、实验名称

分页内存管理

二、实验目的

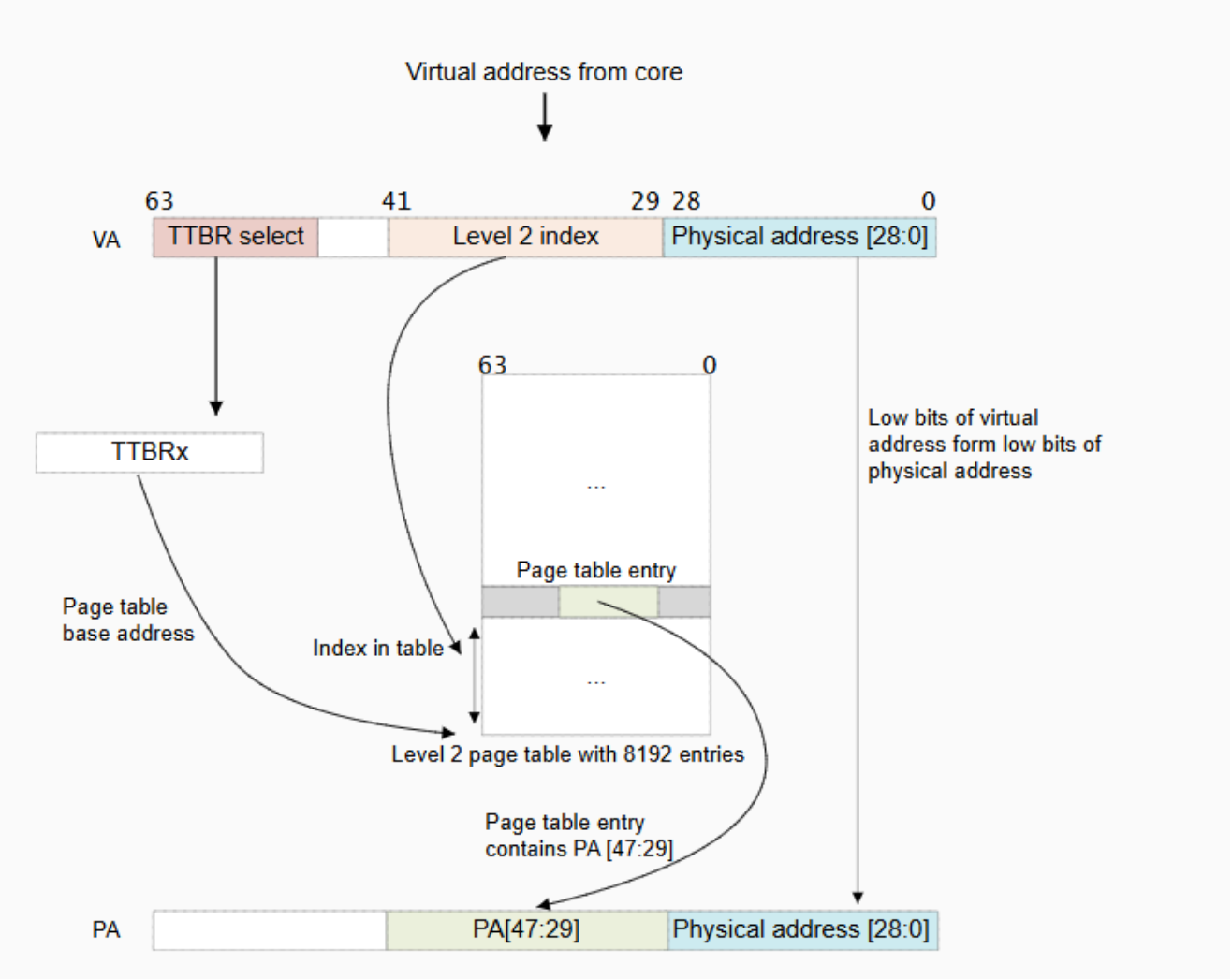
1. 全面深入地理解分页式内存管理的基本方法;
2. 理解页表的访问、完成地址转换等的方法;
3. 在操作系统内核中实现一个内存管理系统。

三、实验任务

Armv8的地址转换

ARM Cortex-A Series Programmer's Guide for ARMv8-A 中提到：For EL0 and EL1, there are two translation tables. TTBR0_EL1 provides translations for the bottom of Virtual Address space, which is typically application space and TTBR1_EL1 covers the top of Virtual Address space, typically kernel space. This split means that the OS mappings do not have to be replicated in the translation tables of each task. 即TTBR0指向整个虚拟空间下半部分通常用于应用程序的空间，TTBR1指向虚拟空间的上半部分通常用于内核的空间。其中TTBR0除了在EL1中存在外，也在EL2 and EL3中存在，但TTBR1只在EL1中存在。

TTBR0_ELn 和 TTBR1_ELn 是页表基地址寄存器，地址转换的过程如下所示。



In a simple address translation involving only one level of look-up. It assumes we are using a 64KB granule with a 42-bit Virtual Address. The MMU translates a Virtual Address as follows:

1. If VA[63:42] = 1 then TTBR1 is used for the base address for the first page table. When VA[63:42] = 0, TTBR0 is used for the base address for the first page table.
2. The page table contains 8192 64-bit page table entries, and is indexed using VA[41:29]. The MMU reads the pertinent level 2 page table entry from the table.
3. The MMU checks the page table entry for validity and whether or not the requested memory access is allowed. Assuming it is valid, the memory access is allowed.
4. In the above Figure, the page table entry refers to a 512MB page (it is a block descriptor).
5. Bits [47:29] are taken from this page table entry and form bits [47:29] of the Physical Address.
6. Because we have a 512MB page, bits [28:0] of the VA are taken to form PA[28:0]. See Effect of granule sizes on translation tables
7. The full PA[47:0] is returned, along with additional information from the page table entry.

In practice, such a simple translation process severely limits how finely you can divide up your address space. Instead of using only this first-level translation table, a first-level table

entry can also point to a second-level page table.在仅涉及一级查找的简单地址翻译中，假设我们使用64KB粒度与42位虚拟地址。MMU按如下方式翻译虚拟地址：
若虚拟地址位[63:42] = 1，则使用TTBR1作为一级页表的基地址；当虚拟地址位[63:42] = 0时，使用TTBR0作为一级页表的基地址。
页表包含8192个64位页表条目，通过虚拟地址位[41:29]索引。MMU从表中读取对应的一级页表条目。
MMU检查页表条目的有效性和内存访问权限。若验证通过，则允许内存访问。
在上述机制中，页表条目指向一个512MB的页（即块描述符）。
页表条目的位[47:29]被提取为物理地址的位[47:29]。
由于采用512MB页，虚拟地址的位[28:0]直接映射为物理地址的位[28:0]。最终返回完整的物理地址PA[47:0]及页表条目中的附加信息。
需注意，这种简单的翻译机制会严重限制地址空间的划分粒度。实际中，一级页表条目通常可指向二级页表以实现更精细的地址空间管理。

mmu管理

新建 src/bsp/mmu.c 文件

```
#include "prt_typedef.h"        // 引入项目自定义类型定义
#include "prt_module.h"         // 引入模块相关函数声明
#include "prt_errno.h"          // 引入错误码定义
#include "mmu.h"                // MMU相关功能头文件
#include "prt_task.h"           // 任务管理相关头文件

// 外部变量声明
extern U64 g_mmu_page_begin;    // 内存映射区域的起始物理地址
extern U64 g_mmu_page_end;      // 内存映射区域的结束物理地址

// 外部函数声明
extern void os_asm_invalidate_dcache_all(void);  // 刷新所有数据缓存
extern void os_asm_invalidate_icache_all(void);  // 刷新所有指令缓存
extern void os_asm_invalidate_tlb_all(void);     // 刷新所有TLB条目

// 内存映射区域配置表
static mmu_mmap_region_s g_mem_map_info[] = {
    {
        .virt      = 0x0,                       // 虚拟地址起始
        .phys      = 0x0,                       // 物理地址起始
        .size      = 0x40000000,                // 区域大小（1GB）
        .max_level = 0x2,                       // 最大映射层级（不超3级）
        .attrs     = MMU_ATTR_DEVICE_NGNRNE | MMU_ACCESS_RWX,  // 设备属性（不可缓存）
    }, {
```

```
        .virt      = 0x40000000,               // 虚拟地址起始
        .phys      = 0x40000000,               // 物理地址起始
        .size      = 0x40000000,               // 区域大小（1GB）
        .max_level = 0x2,                      // 最大映射层级
        .attrs     = MMU_ATTR_CACHE_SHARE | MMU_ACCESS_RWX,    // 共享可读写
执行内存
    }
};

// MMU控制结构体
static mmu_ctrl_s g_mmu_ctrl = { 0 };        // 初始化所有成员为0

/**
 * 生成TCR寄存器值并计算虚拟地址位数
 * @param pips 返回物理地址位数（IPS）
 * @param pva_bits 返回虚拟地址位数（VA_BITS）
 * @return TCR寄存器配置值
 */
static U64 mmu_get_tcr(U32 *pips, U32 *pva_bits)
{
    U64 max_addr = 0;                          // 计算最大虚拟地址
    U64 ips, va_bits;                          // 物理地址位数/虚拟地址位数
    U64 tcr;                                   // TCR寄存器值
    U32 i;                                     // 循环计数器

    // 遍历所有内存区域计算最大虚拟地址
    for (i = 0; i < sizeof(g_mem_map_info)/sizeof(mmu_mmap_region_s); ++i) {
        max_addr = MAX(max_addr, g_mem_map_info[i].virt +
g_mem_map_info[i].size);
    }

    // 根据最大地址确定IPS和VA_BITS
    if (max_addr > (1ULL << MMU_BITS_44)) {
        ips = MMU_PHY_ADDR_LEVEL_5;            // 48位虚拟地址
        va_bits = MMU_BITS_48;
    } else if (max_addr > (1ULL << MMU_BITS_42)) {
        ips = MMU_PHY_ADDR_LEVEL_4;            // 44位虚拟地址
        va_bits = MMU_BITS_44;
    } else if (max_addr > (1ULL << MMU_BITS_40)) {
        ips = MMU_PHY_ADDR_LEVEL_3;            // 42位虚拟地址
        va_bits = MMU_BITS_42;
    } else if (max_addr > (1ULL << MMU_BITS_36)) {
        ips = MMU_PHY_ADDR_LEVEL_2;            // 40位虚拟地址
```

```
        va_bits = MMU_BITS_40;
    } else if (max_addr > (1ULL << MMU_BITS_32)) {
        ips = MMU_PHY_ADDR_LEVEL_1;          // 36位虚拟地址
        va_bits = MMU_BITS_36;
    } else {
        ips = MMU_PHY_ADDR_LEVEL_0;          // 32位虚拟地址
        va_bits = MMU_BITS_32;
    }

    // 构建TCR寄存器值
    tcr = TCR_EL1_RSVD | TCR_IPS(ips);       // 基础配置+IPS设置

    // 根据页表粒度设置缓存属性
    if (g_mmu_ctrl.granule == MMU_GRANULE_4K) {
        tcr |= TCR_TG0_4K | TCR_SHARED_INNER | TCR_ORGN_WBWA |
TCR_IRGN_WBWA;
    } else {
        tcr |= TCR_TG0_64K | TCR_SHARED_INNER | TCR_ORGN_WBWA |
TCR_IRGN_WBWA;
    }

    tcr |= TCR_T0SZ(va_bits);                // 设置虚拟地址位宽

    // 返回参数
    if (pips) *pips = ips;
    if (pva_bits) *pva_bits = va_bits;

    return tcr;
}

/**
 * 获取页表项类型
 * @param pte 页表项地址
 * @return 页表项类型（PTE_TYPE_BLOCK/PTE_TYPE_TABLE等）
 */
static U32 mmu_get_pte_type(U64 const *pte)
{
    return (U32)(*pte & PTE_TYPE_MASK);      // 提取类型标识位
}

/**
 * 计算不同级别页表的地址偏移量
 * @param level 页表级别
```

```
 * @return 地址索引位移量
 */
static U32 mmu_level2shift(U32 level)
{
    if (g_mmu_ctrl.granule == MMU_GRANULE_4K) {
        return (U32)(MMU_BITS_12 + MMU_BITS_9 * (MMU_LEVEL_3 - level));  //
4K粒度计算
    } else {
        return (U32)(MMU_BITS_16 + MMU_BITS_13 * (MMU_LEVEL_3 - level)); //
64K粒度计算
    }
}

/**
 * 查找指定地址的页表项
 * @param addr 虚拟地址
 * @param level 目标页表级别
 * @return 对应页表项指针（找不到返回NULL）
 */
static U64 *mmu_find_pte(U64 addr, U32 level)
{
    U64 *pte = NULL;
    U32 i;

    // 检查层级有效性
    if (level < g_mmu_ctrl.start_level) return NULL;

    pte = (U64 *)g_mmu_ctrl.tlb_addr;         // 从TLB基地址开始

    // 遍历页表层级
    for (i = g_mmu_ctrl.start_level; i < MMU_LEVEL_MAX; ++i) {
        // 计算当前层级的索引
        if (g_mmu_ctrl.granule == MMU_GRANULE_4K) {
            idx = (addr >> mmu_level2shift(i)) & 0x1FF;
        } else {
            idx = (addr >> mmu_level2shift(i)) & 0x1FFF;
        }

        pte += idx;                           // 定位到具体页表项

        // 找到目标层级则返回
        if (i == level) return pte;
```

```
        // 检查页表项有效性
        if (mmu_get_pte_type(pte) != PTE_TYPE_TABLE) return NULL;

        // 获取下级页表地址
        if (g_mmu_ctrl.granule == MMU_GRANULE_4K) {
            pte = (U64 *)(*pte & PTE_TABLE_ADDR_MARK_4K);
        } else {
            pte = (U64 *)(*pte & PTE_TABLE_ADDR_MARK_64K);
        }
    }
    return NULL;
}

/**
 * 创建新页表
 * @return 新页表地址指针（失败返回NULL）
 */
static U64 *mmu_create_table(void)
{
    U32 pt_len;
    U64 *new_table = (U64 *)g_mmu_ctrl.tlb_fillptr;

    // 计算页表长度
    if (g_mmu_ctrl.granule == MMU_GRANULE_4K) {
        pt_len = MAX_PTE_ENTRIES_4K * sizeof(U64);
    } else {
        pt_len = MAX_PTE_ENTRIES_64K * sizeof(U64);
    }

    // 检查空间是否足够
    g_mmu_ctrl.tlb_fillptr += pt_len;
    if (g_mmu_ctrl.tlb_fillptr - g_mmu_ctrl.tlb_addr > g_mmu_ctrl.tlb_size)
{
        return NULL;
    }

    // 初始化页表（清零）
    U64 *tmp = new_table;
    for(int i = 0; i < pt_len; i+=sizeof(U64)){
        *tmp++ = 0;
    }

    return new_table;
```

```
}

/**
 * 设置页表项为表类型
 * @param pte 目标页表项地址
 * @param table 下级页表地址
 */
static void mmu_set_pte_table(U64 *pte, U64 *table)
{
    *pte = PTE_TYPE_TABLE | (U64)table;      // 设置为表类型并关联下级页表
}

/**
 * 处理页表映射关系
 * @param map 内存映射区域描述
 * @param pte 当前页表项
 * @param phys 物理地址
 * @param level 当前页表层级
 * @return 操作结果（0成功/负数错误码）
 */
static S32 mmu_add_map_pte_process(mmu_mmap_region_s const *map, U64 *pte,
U64 phys, U32 level)
{
    U64 *new_table = NULL;

    // 上级页表处理
    if (level < map->max_level) {
        // 无效页表项则创建新表
        if (mmu_get_pte_type(pte) == PTE_TYPE_FAULT) {
            new_table = mmu_create_table();
            if (!new_table) return -1;
            mmu_set_pte_table(pte, new_table); // 关联新页表
        }
    }
    // 叶子页表处理
    else if (level == MMU_LEVEL_3) {
        *pte = phys | map->attrs | PTE_TYPE_PAGE; // 设置物理页映射
    }
    // 中间页表处理
    else {
        *pte = phys | map->attrs | PTE_TYPE_BLOCK; // 设置块映射
    }
    return 0;
```

```
}

/**
 * 建立内存映射
 * @param map 内存映射区域描述
 * @return 操作结果（0成功/负数错误码）
 */
static S32 mmu_add_map(mmu_mmap_region_s const *map)
{
    U64 virt = map->virt;
    U64 phys = map->phys;
    U64 max_level = map->max_level;
    U64 start_level = g_mmu_ctrl.start_level;
    U64 block_size = 0;
    U64 map_size = 0;
    U32 level;
    U64 *pte = NULL;
    S32 ret;

    // 参数校验
    if (map->max_level <= start_level) return -2;

    // 分块处理内存映射
    while (map_size < map->size) {
        // 遍历页表层级
        for (level = start_level; level <= max_level; ++level) {
            pte = mmu_find_pte(virt, level);
            if (!pte) return -3;

            ret = mmu_add_map_pte_process(map, pte, phys, level);
            if (ret) return ret;

            if (level != start_level) {
                block_size = 1ULL << mmu_level2shift(level); // 计算块大小
            }
        }

        // 移动到下一个内存块
        virt += block_size;
        phys += block_size;
        map_size += block_size;
    }
    return 0;
```

```
}

/**
 * 初始化页表系统
 * @param mem_map 内存映射表
 * @param mem_region_num 映射区域数量
 * @param tlb_addr TLB基地址
 * @param tlb_len TLB大小
 * @param granule 页表粒度
 * @return 初始化结果
 */
static U32 mmu_setup_pgtables(mmu_mmap_region_s *mem_map, U32
mem_region_num, U64 tlb_addr, U64 tlb_len, U32 granule)
{
    U32 i, ret;
    U64 tcr;

    // 初始化控制结构体
    g_mmu_ctrl.tlb_addr = tlb_addr;
    g_mmu_ctrl.tlb_size = tlb_len;
    g_mmu_ctrl.tlb_fillptr = tlb_addr;
    g_mmu_ctrl.granule = granule;
    g_mmu_ctrl.start_level = 0;

    // 获取TCR配置
    tcr = mmu_get_tcr(NULL, &g_mmu_ctrl.va_bits);

    // 确定起始层级
    if (g_mmu_ctrl.granule == MMU_GRANULE_4K) {
        g_mmu_ctrl.start_level = (g_mmu_ctrl.va_bits < MMU_BITS_39) ?
MMU_LEVEL_1 : MMU_LEVEL_0;
    } else {
        if (g_mmu_ctrl.va_bits <= MMU_BITS_36) {
            g_mmu_ctrl.start_level = MMU_LEVEL_2;
        } else {
            g_mmu_ctrl.start_level = MMU_LEVEL_1;
            return 3; // 不支持的配置
        }
    }

    // 创建顶级页表
    U64 *new_table = mmu_create_table();
    if (!new_table) return 1;
```

```
    // 配置所有内存区域
    for (i = 0; i < mem_region_num; ++i) {
        ret = mmu_add_map(&mem_map[i]);
        if (ret) return ret;
    }

    // 应用配置到MMU
    mmu_set_ttbr_tcr_mair(g_mmu_ctrl.tlb_addr, tcr, MEMORY_ATTRIBUTES);
    return 0;
}

/**
 * MMU硬件初始化
 * @return 初始化结果
 */
static S32 mmu_setup(void)
{
    S32 ret;
    U64 page_addr, page_len;

    // 获取内存映射区域
    page_addr = (U64)&g_mmu_page_begin;
    page_len = (U64)&g_mmu_page_end - page_addr;

    // 初始化页表系统
    ret = mmu_setup_pgtables(g_mem_map_info,

sizeof(g_mem_map_info)/sizeof(mmu_mmap_region_s),
                             page_addr, page_len,
                             MMU_GRANULE_4K);
    return ret;
}

/**
 * MMU初始化入口函数
 */
S32 mmu_init(void)
{
    S32 ret;

    // 执行MMU初始化
    ret = mmu_setup();
```

```
    if (ret) return ret;

    // 刷新缓存和TLB
    os_asm_invalidate_dcache_all();
    os_asm_invalidate_icache_all();
    os_asm_invalidate_tlb_all();

    // 启用MMU和缓存
    set_sctlr(get_sctlr() | CR_C | CR_M | CR_I); // 设置控制寄存器

    return 0;
}
```

新建 src/bsp/mmu.h， 该文件可从 这里 下载

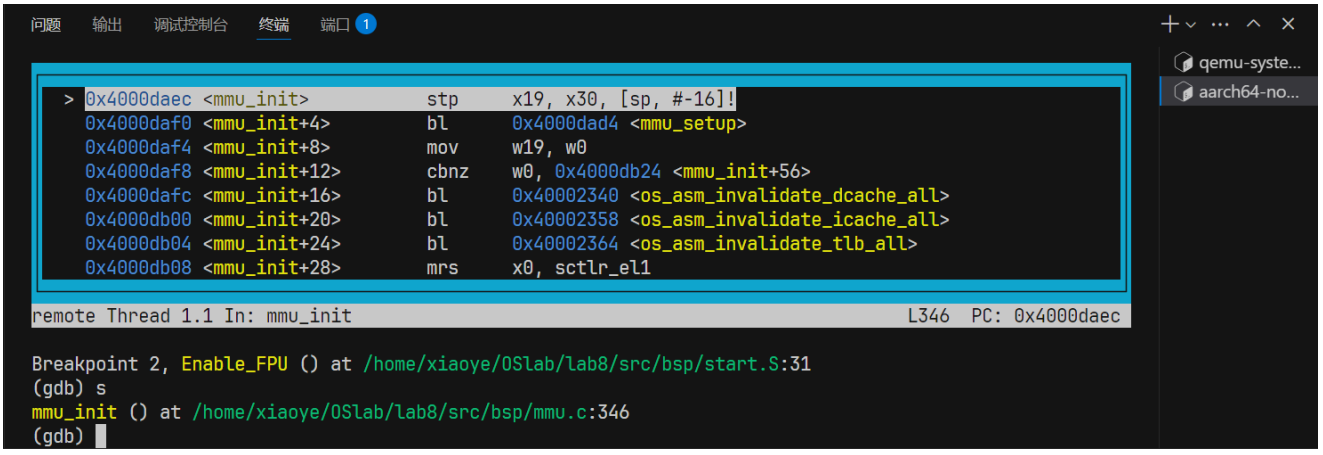
新建 src/bsp/cache_asm.S， 该文件可从 这里 下载

启用 mmu

start.S 中在 B OsEnterMain 之前启用 MMU

```
// 启用 MMU
BL      mmu_init
// 进入 main 函数
B       OsEnterMain
```

提示: 将新增文件加入构建系统
提示: 通过调试确保你真的启动了 MMU



作业

作业1

启用 TTBR1 ，将地址映射到虚拟地址的高半部分，使用高地址访问串口 修改后：（1）src/bsp/print.c中

```
#define UART_0_REG_BASE (0xffffffff00000000 + 0x09000000)
//0xffffffff00000000 是虚拟地址高位标记（由MMU页表配置决定）
//0x09000000 是物理地址偏移量
```

(2)src/bsp/hwi_init.c 中

```
#define GIC_DIST_BASE               (0xffffffff00000000 + 0x08000000)
#define GIC_CPU_BASE                (0xffffffff00000000 + 0x08010000)
//此处定义了dist和cpu的基址，GIC_DIST_BASE 和 GIC_CPU_BASE 的高位多少个f与MMU的配置有关，此处设置为8个f
```

（虽然此时程序已经可以正常运行，但是我们还是要继续做一些操作。）
（3）更改g_mem_map_info，使得进行映射时，将地址映射到高半部分。

```
static mmu_mmap_region_s g_mem_map_info[] = {

    {

        .virt      = 0xffffffff00000000,//改动后

        .phys      = 0x0,

        .size      = 0x40000000, // 1G size

        .max_level = 0x2,  // 不应大于3

        .attrs     = MMU_ATTR_DEVICE_NGNRNE | MMU_ACCESS_RWX, // 设备

    }, {

        .virt      = 0xffffffff40000000,//改动后

        .phys      = 0x40000000,

        .size      = 0x40000000, // 1G size

        .max_level = 0x2, // // 不应大于3

        .attrs     = MMU_ATTR_CACHE_SHARE | MMU_ACCESS_RWX, // 内存
```

```
    }
};
```

（4）取消mmu_set_ttbr_tcr_mair函数中的注释, 启用ttbr1_el1。

> **Note**
>
> ttbr1_el1 是 ARM 架构中的一个系统寄存器，全称为 “Translation Table Base Register 1 - EL1”，用于存放转换表基址，在内存管理单元（MMU）中用来控制地址空间的转换。

```
static inline void mmu_set_ttbr_tcr_mair(U64 table, U64 tcr, U64 attr)
{
    OS_EMBED_ASM("dsb sy");


    OS_EMBED_ASM("msr ttbr0_el1, %0" : : "r" (table) : "memory");
     OS_EMBED_ASM("msr ttbr1_el1, %0" : : "r" (table) : "memory");//取消这一行原本的注释
    OS_EMBED_ASM("msr tcr_el1, %0" : : "r" (tcr) : "memory");
    OS_EMBED_ASM("msr mair_el1, %0" : : "r" (attr) : "memory");


    OS_EMBED_ASM("isb");
}
```

（5）最后，Init函数、write函数以及read函数中的基址被更改为新定义的地址
程序可以正常运行

结果如下：

```
(base) xiaoye@localhost:~/OSlab/lab8$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
                _       _ _____      _              _              _    _ _   _ _   _
  _ __ ___ (_)_ __ (_) ____|   _| | ___ _ __  | |__   _   _  | | | | \ | | | | ___ _ __
 | '_ ` _ \| | '_ \| |  _|| | | |/ _ \ '__| | '_ \| | | | | |_| |  \| | | | |/ _ \ '__|
 | | | | | | | | | | | |__| |_| |  __/ |    | |_) | |_| | |  _  | |\  | |_| |  __/ |
 |_| |_| |_|_|_| |_|_|_____\__,_|_|\___|_|    |_.__/ \__, | |_| |_|_| \_|\___/ \___|_|
                                                       |___/
ctr-a h: print help of qemu emulator. ctr-a x: quit emulator.

task 2 run ...
task 1 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 2 run ...
task 1 run ...
task 1 run ...
task 1 run ...
task 1 run ...
task 1 run ...
```