《操作系统》

实验八报告

目录

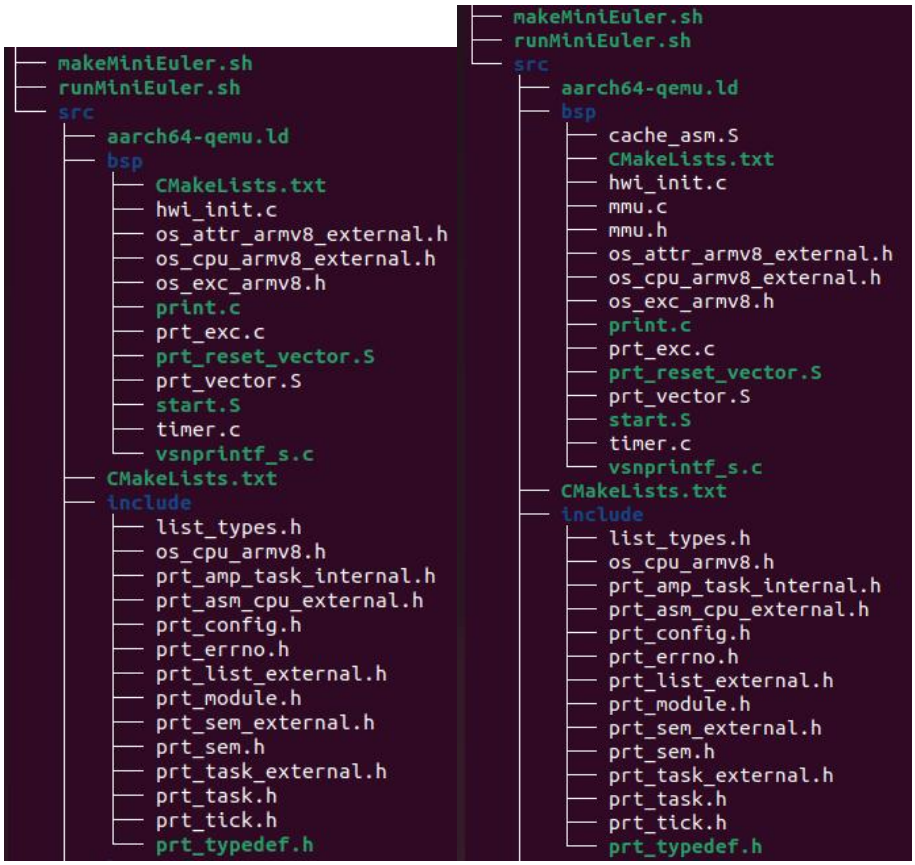
1 实验代码分析 3
1.1 实验目的 3
1.2 代码结构 3
1.3 Armv8 的地址转换 4
1.4 MMU 管理 5
1.5 构建系统 7
1.6 执行结果 7
2 实验任务 9
2.1 作业 1 9

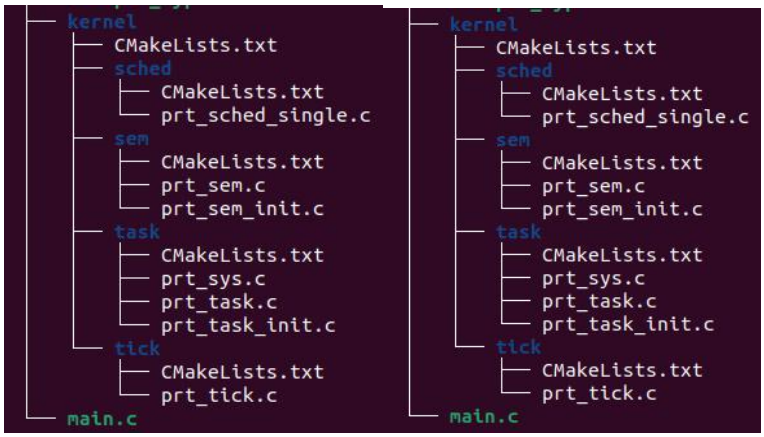
1 实验代码分析

1.1 实验目的

实现分页内存管理。

1.2 代码结构



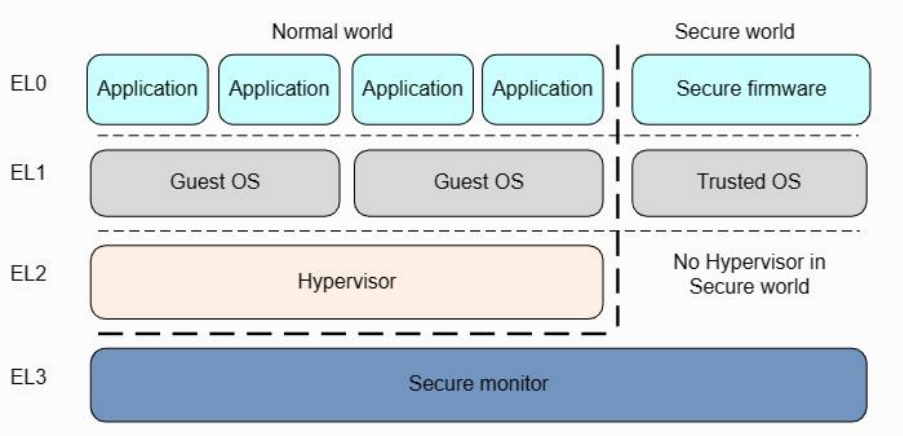


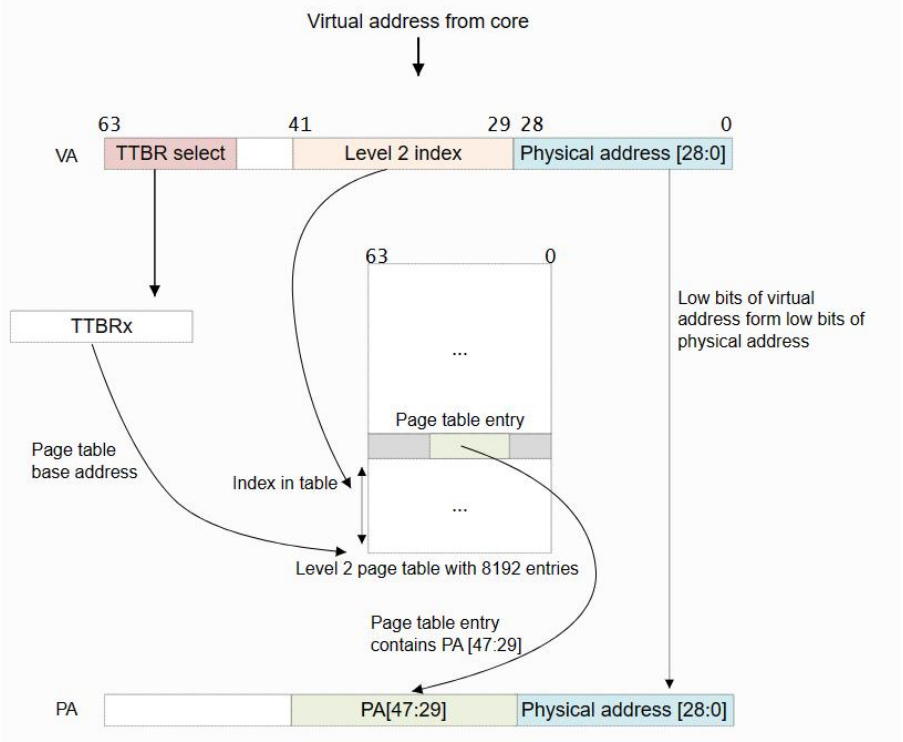
对应 lab7 的结构来看，在 bsp 中添加了 mmu.c 、mmu.h、cache_asm.S 三个文件。

1.3 Armv8 的地址转换

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。（因为 EL1 为操作系统层。）

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.





上图是地址转换的过程（VA→PA），有一个二级页表。

TTBR0_ELn 和 TTBR1_ELn 是页表基地址寄存器。

在一个简单的地址翻译中，只涉及一级查找。它假设我们使用一个 64KB 的颗粒和一个 42 位的虚拟地址。MMU 对虚拟地址进行如下转换。

1.如果 VA[63:42] 均为 1，则 TTBR1 用于第一页表的基址。当 VA[63:42] 均为 0 时，TTBR0 被用作第一页表的基址。

2.页表包含 8192 个 64 位页表条目，使用 VA[41:29]进行索引，MMU 从表中读取相关的 2 级页表项。

3.MMU 检查页表条目的有效性，以及是否允许请求的内存访问。假设它是有效的，内存访问是允许的。

4.在上面的图中，页表项指向一个 512MB 的页(它是块描述符)。

5.[47:29]从这个页表条目中取出，形成物理地址的位[47:29]。

6.因为我们的页大小为 512MB，VA 的[28:0]作为偏移被转换成 PA[28:0]。参见颗粒大小对平移表的影响。

7.返回完整的 PA[47:0]，以及来自页表条目的附加信息。

在实践中，如此简单的翻译过程严重限制了您划分地址空间的精细程度。与仅使用这个第一级翻译表不同，第一级表项还可以指向第二级页表。

1.4 MMU 管理

新建 src/bsp/mmu.c 文件。包含头文件。定义宏 g_mmu_page_begin 和 g_mmu_page_end 用于表示 MMU 管理的页表的开始和结束地址。

进行了外部函数声明：

os_asm_invalidate_dcache_all(),os_asm_invalidate_icache_all(),os_asm_invalidate_tlb_all() 用于清空数据缓存（DCache）、指令缓存（ICache）和翻译后备缓冲区（TLB）。

定义了两个内存映射区域：g_mem_map_info[]是一个数组，其中每个区域都由一个 mmu_mmap_region_s 结构体表示。其中包括虚拟地址、物理地址、大小、最大级别、属性。我们这里的内存映射区域，设备的区域从 0x0 开始，虚拟地址与物理地址相同，大小为 1G，最大级别为 2，内存的区域从 0x4000000 开始，虚拟地址与物理地址相同，大小为 1G。

定义一个 mmu_ctrl_s 类型的变量 g_mmu_ctrl，用于存储与 MMU 控制相关的信息，初

始化为 0。

mmu_get_tcr：根据 g_mem_map_info 数组中定义的内存映射区域来生成并返回一个 TCR 的值，pva_bits 返回虚拟地址位数。TCR 用于配置 ARM 架构中的 MMU。

mmu_get_pte_type：接受一个指向页表条目（PTE）的指针，并返回 PTE 的类型。它使用了一个掩码 PTE_TYPE_MASK 来从 PTE 中提取类型信息。

mmu_level2shift：根据页表项级别计算当个页表项表示的范围（位数）。

mmu_find_pte：根据虚拟地址找到对应级别的页表项。

mmu_create_table：根据页表粒度在页表区域新建一个页表，返回页表起始位置。

mmu_set_pte_table：设置页表条目，它将 PTE 配置为一个指向另一个页表的指针。

mmu_add_map_pte_process：依据 mmu_mmap_region_s 填充 PTE.

mmu_add_map：依据 mmu_mmap_region_s 的定义，生成 mmu 映射。

mmu_set_ttbr_tcr_mair：设置 TTBR0_EL1、TCR_EL1 和 MAIR_EL1 寄存器。

mmu_setup_pgtables：设置 MMU 的页表，并配置相关的寄存器。设置了一些全局配置，然后调用了以上函数，获取虚拟地址的位数、计算了地址转换的起始级别过高则返回错误、创建一个新的顶级页表并遍历 mem_map 中的内存映射将其添加到页表中，最后调用 mmu_set_ttbr_tcr_mair 函数进行配置设置。

mmu_setup：配置 MMU。调用 mmu_setup_pgtables。

mmu_init：初始化 MMU，调用 mmu_setup，无效化缓存和 TLB，设置系统控制寄存器（SCTLR）。

新建 src/bsp/mmu.h，定义了一些宏和内嵌函数。

新建 src/bsp/cache_asm.S，汇编代码，os_asm_dcache_level 处理特定级别的数据缓存；os_asm_dcache_all 函数遍历所有的缓存级别，并对每个级别执行由 os_asm_dcache_level 定义的特定操作；os_asm_invalidate_dcache_all、os_asm_flush_dcache_all、os_asm_clean_dcache_all、os_asm_invalidate_icache_all、os_asm_invalidate_tlb_all 用于无效化。

需要在 start.S 中 B OsEnterMain 前启用 MMU。

1.5 构建系统

修改，将新建文件加入构建系统。

src/bsp/CMakeLists.txt：

```
set(SRCS start.S prt_reset_vector.S print.c vsnprintf_s.c prt_exc.c prt_vector.S os_exc_armv8
.h  os_attr_armv8_external.h  os_cpu_armv8_external.h  hwi_init.c  timer.c mmu.h  cache_asm.S
 mmu.c)
add_library(bsp OBJECT ${SRCS})  # OBJECT类型只编译生成.o目标文件，但不实际链接成库
```

无需修改 main 函数。

1.6 执行结果

本实验逻辑为：

新建 src/bsp/mmu.c 文件实现大部分页表操作，初始化 MMU。

新建 src/bsp/mmu.h，定义了一些宏和内嵌函数。

新建 src/bsp/cache_asm.S 使用汇编代码执行特定处理。

在 start.S 中 B OsEnterMain 前启用 MMU。

编译执行：

```
guoruiling@guoruiling-virtual-machine:~/lab8$ sh makeMiniEuler.sh
mkdir: cannot create directory ‘build’: File exists
-- The C compiler identification is GNU 11.2.1
-- The ASM compiler identification is GNU
-- Found assembler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarch64-none-elf/
bin/aarch64-none-elf-gcc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /home/guoruiling/aarch64-none-elf/gcc-arm-11.2-2022.02-x86_64-aarc
h64-none-elf/bin/aarch64-none-elf-gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/guoruiling/lab8/build
[  5%] Building C object kernel/sem/CMakeFiles/sem.dir/prt_sem_init.c.obj
[ 10%] Building C object kernel/sem/CMakeFiles/sem.dir/prt_sem.c.obj
[ 10%] Built target sem
Scanning dependencies of target bsp
[ 15%] Building ASM object bsp/CMakeFiles/bsp.dir/start.S.obj
[ 21%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_reset_vector.S.obj
[ 26%] Building C object bsp/CMakeFiles/bsp.dir/print.c.obj
[ 31%] Building C object bsp/CMakeFiles/bsp.dir/vsnprintf_s.c.obj
[ 36%] Building C object bsp/CMakeFiles/bsp.dir/prt_exc.c.obj
[ 42%] Building ASM object bsp/CMakeFiles/bsp.dir/prt_vector.S.obj
[ 47%] Building C object bsp/CMakeFiles/bsp.dir/hwi_init.c.obj
[ 52%] Building C object bsp/CMakeFiles/bsp.dir/timer.c.obj
[ 57%] Building ASM object bsp/CMakeFiles/bsp.dir/cache_asm.S.obj
[ 63%] Building C object bsp/CMakeFiles/bsp.dir/mmu.c.obj
[ 63%] Built target bsp
[ 68%] Building C object kernel/tick/CMakeFiles/tick.dir/prt_tick.c.obj
[ 68%] Built target tick
[ 73%] Building C object kernel/task/CMakeFiles/task.dir/prt_task.c.obj
[ 78%] Building C object kernel/task/CMakeFiles/task.dir/prt_sys.c.obj
[ 84%] Building C object kernel/task/CMakeFiles/task.dir/prt_task_init.c.obj
[ 84%] Built target task
[ 89%] Building C object kernel/sched/CMakeFiles/sched.dir/prt_sched_single.c.obj
[ 89%] Built target sched
[ 94%] Building C object CMakeFiles/miniEuler.dir/main.c.obj
[100%] Linking C executable miniEuler
[100%] Built target miniEuler
```

```
guoruiling@guoruiling-virtual-machine:~/lab8$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/m
iniEuler -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 ...
QEMU: Terminated
```

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
__text_start () at /home/guoruiling/lab8/src/bsp/start.S:11
11          MRS     x6, CurrentEL // 把系统寄存器 CurrentEL 的值读入到通用寄存器 x6 中
(gdb) si
12          MOV     x2, #0x4 // CurrentEL EL1： bits [3:2] = 0b01
(gdb)
13          CMP     w6, w2
(gdb) si
15          BEQ Start // 若 CurrentEl 为 EL1 级别，跳转到 Start 处执行，否则死循环。
(gdb) si
21          LDR     x1, =__os_sys_sp_end // ld文件中定义，堆栈设置
(gdb) si
22          BIC     sp, x1, #0xf
(gdb) si
Enable_FPU () at /home/guoruiling/lab8/src/bsp/start.S:26
26          MRS X1, CPACR_EL1
(gdb) si
27          ORR X1, X1, #(0x3 << 20)
(gdb) si
28          MSR CPACR_EL1, X1
(gdb) si
29          ISB
(gdb) si
32      BL      mmu_init
(gdb) si
mmu_init () at /home/guoruiling/lab8/src/bsp/mmu.c:346
346         ret = mmu_setup();
(gdb) si
0x0000000400035b8       346         ret = mmu_setup();
```

通过调试确定已启用 MMU（从 start.s 中跳转到 mmu_init 执行。）

```
19 Start:
20     LDR     x1, =__os_sys_sp_end // ld文件中定义，堆栈设置
21     BIC     sp, x1, #0xf
22
23 //参考： https://developer.arm.com/documentation/den0024/a
   re-detail
24 Enable_FPU:
25    MRS X1, CPACR_EL1
26    ORR X1, X1, #(0x3 << 20)
27    MSR CPACR_EL1, X1
28    ISB
29
30 // 启用 MMU
31 BL      mmu_init
32 // 进入 main 函数
33  B       OsEnterMain
```

2 实验任务

2.1 作业 1

题目：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。修改后有：

后： (1) src/bsp/print.c中

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

(2)src/bsp/hwi_init.c 中

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

使得程序可以正常运行（GIC_DIST_BASE 和 GIC_CPU_BASE 的高位多少个 f 与你对 MMU 的配置有关）。

在修改题目给定的相应配置后，我们需要修改 TCR_EL1，启用 TTBR1，相应调整 g_mem_map_info 的配置，使得程序可以正常运行。

修改过一些参数 U32 改为 U64，以消除警告，防止溢出后错误的发生。

调用过程为：S32 mmu_init → static S32 mmu_setup → static U32 mmu_setup_pgtables → mmu_get_tcr 和 mmu_set_ttbr_tcr_mair。

根据之前的分析，我们可以知道，函数 static inline void mmu_set_ttbr_tcr_mair(U64 table, U64 tcr, U64 attr)用于设置 TTBR0_EL1、TCR_EL1 和 MAIR_EL1 的值。

```
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  }
```

263 行设置了一个“数据同步屏障”，dsb sy 表示一个系统级的数据同步屏障，它会确保在此之前的所有写操作在屏障之后的所有操作（包括其他 CPU 上的操作）之前对系统可见。这通常用于确保内存操作的顺序性和一致性。

265 行是对 TTBR0_EL1 的设置。这个寄存器存储了页表的基地址，它将 table 参数的值写入该寄存器。

266 行是对 TTBR1_EL1 的设置，它被注释掉了，也是将 table 参数的值写入。

267 行是对 TCR_EL1 的设置，将 tcr 参数的值写入。

267 行是对 MAIR_EL1 的设置，将 attr 参数的值写入。

270 行设置了一个 isb 指令同步屏障，确保所有之前的指令都已完成，并且后续指令的执行将看到前面指令的效果。

所以我们将注释取消，将地址传给 TTBR1_EL1 寄存器，而非 TTBR0_EL1。

```
4 static inline void mmu_set_ttbr_tcr_mair(U64 table, U64 tcr, U64 attr)
5 {
6     OS_EMBED_ASM("dsb sy");
7
8    // OS_EMBED_ASM("msr ttbr0_el1, %0" : : "r" (table) : "memory");
9     OS_EMBED_ASM("msr ttbr1_el1, %0" : : "r" (table) : "memory");
10    OS_EMBED_ASM("msr tcr_el1, %0" : : "r" (tcr) : "memory");
11    OS_EMBED_ASM("msr mair_el1, %0" : : "r" (attr) : "memory");
12
13    OS_EMBED_ASM("isb");
14 }
```

我们可以看到 **static U32 mmu_setup_pgtables(mmu_mmap_region_s *mem_map, U32 mem_region_num, U64 tlb_addr, U64 tlb_len, U32 granule)** 函数中在最后调用了这个函数，作为对寄存器的配置的设置。

```
316
317         mmu_set_ttbr_tcr_mair(g_mmu_ctrl.tlb_addr, tcr, MEMORY_ATTRIBUTES);
318
```

在先前的函数和这个函数中，有一个结构体经常被使用，mmu_ctrl_s，它的声明在 mmu.h 文件中，在函数的起始有一个实际的变量 g_mmu_ctrl。

```
typedef struct {
    U64 tlb_addr;
    U64 tlb_size;
    U64 tlb_fillptr;
    U32 granule;
    U32 start_level;
    U32 va_bits;
} mmu_ctrl_s;
```

U64 的 tlb_addr 代表 TLB 的基地址；U64 的 tlb_size 代表 TLB 的大小；U64 的 tlb_fillptr 是一个指针或索引，指向 TLB 中下一个将被填充或覆盖的条目；U32 的 granule 代表页粒度；U32 的 start_level 代表页表的起始级别；U32 的 va_bits 代表虚拟地址位数。

我们回来再看对 mmu_setup_pgtables 函数的调用过程，将我们定义的内存映射区域 g_mem_map_info[]数组、数组元素个数、起始地址、大小、颗粒度传给了该函数。

```
328         page_addr = (U64)&g_mmu_page_begin;
329         page_len = (U64)&g_mmu_page_end - (U64)&g_mmu_page_begin;
```

```
331     ret = mmu_setup_pgtables(g_mem_map_info, (sizeof(g_mem_map_info) / sizeof(mmu_mmap_region_s)),
332                              page_addr, page_len, MMU_GRANULE_4K);
```

在函数的执行过程中，它将各个参数赋给了变量 g_mmu_ctrl，包括 tlb_addr、tlb_size、tlb_fillptr、granule；调用 mmu_get_tcr 函数生成 tcr 的值，pva_bits 返回虚拟地址位数赋给 va_bits；根据粒度和虚拟地址位数计算起始级别赋给 start_level；创建一个顶级页表；调用 mmu_add_map 函数将我们的两个内存映射区域加入。

最终将 g_mmu_ctrl.tlb_addr, tcr, MEMORY_ATTRIBUTES 三个参数传给 mmu_set_ttbr_tcr_mair 函数。g_mmu_ctrl.tlb_addr 作为参数 table，赋给 TTBR0_EL1 或 TTBR1_EL1，tcr 作为参数 tcr 赋给 TCR_EL1，MEMORY_ATTRIBUTE 作为参数 attr 赋给 MAIR_EL1。

```
317         mmu_set_ttbr_tcr_mair(g_mmu_ctrl.tlb_addr, tcr, MEMORY_ATTRIBUTES);
```

计算 TCR_EL1 的值的函数为 static U64 mmu_get_tcr(U32 *pips, U32 *pva_bits)，前半部分为根据 g_mem_map_info 表计算所使用的虚拟地址的最大值和虚拟地址所需的位数，后半部分构建 TCR 的值，如下图所示。

```
68      // 构建Translation Control Register寄存器的值,tcr可控制TTBR0_EL1和TTBR1_EL1的影响
69      tcr = TCR_EL1_RSVD | TCR_IPS(ips);
70
71      if (g_mmu_ctrl.granule == MMU_GRANULE_4K) {
72          tcr |= TCR_TG0_4K | TCR_SHARED_INNER | TCR_ORGN_WBWA | TCR_IRGN_WBWA;
73      } else {
74          tcr |= TCR_TG0_64K | TCR_SHARED_INNER | TCR_ORGN_WBWA | TCR_IRGN_WBWA;
75      }
76
77      tcr |= TCR_T0SZ(va_bits);   // Memory region 2^(64-T0SZ)
78
79      if (pips != NULL) {
80          *pips = ips;
81      }
82
83      if (pva_bits != NULL) {
84          *pva_bits = va_bits;
85      }
86
87      return tcr;
88  }
```

为了读懂这段函数的意思以修改它，我们需要根据宏的定义查找手册中 TCR_EL1 的相关描述。

69 行是一个保留位的宏和一个安全性宏按位或。

```
#define TCR_IPS(x)          ((x) << 32)     // Intermediate Physical Address Size

#define TCR_EL1_RSVD            (1UL << 35)     //原定义错误
#define TCR_EL2_RSVD            (1UL << 31 | 1UL << 23)
#define TCR_EL3_RSVD            (1UL << 31 | 1UL << 23)
```

Bit [35]

Reserved, RES0.

35 位为保留位，没有规定功能。

TG1, bits [31:30]

Granule size for the TTBR1_EL1.

TG1	Meaning
0b01	16KB.
0b10	4KB.
0b11	64KB.

Other values are reserved.

If the value is programmed to either a reserved value or a size that has not been implemented, then the hardware will treat the field as if it has been programmed to an IMPLEMENTATION DEFINED choice of the sizes that has been implemented for all purposes other than the value read back from this register.

It is IMPLEMENTATION DEFINED whether the value read back is the value programmed or the value that corresponds to the size chosen.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Related c

31 位为设置 **TTBR1_EL1 的颗粒度**的两位的高位。

EPD1, bit [23]

Translation table walk disable for translations using TTBR1_EL1. This bit controls whether a translation table walk is performed on a TLB miss, for an address that is translated using TTBR1_EL1. The encoding of this bit is:

EPD1	Meaning
0b0	Perform translation table walks using TTBR1_EL1.
0b1	A TLB miss on an address that is translated using TTBR1_EL1 generates a Translation fault. No translation table walk is performed.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

23 位设置了在 TLB 不命中的时候是否遍历转换表。

71-75 行用于设置颗粒度。77 行用于设置虚拟空间大小。且注释中提到了 TCR_EL1_RSVD 原定义错误，仅定义了一个保留值（无用），故我们需要修改该宏的定义，使得启用 TTBR1_EL1.

A1, bit [22]

Selects whether TTBR0_EL1 or TTBR1_EL1 defines the ASID. The encoding of this bit is:

A1	Meaning
0b0	TTBR0_EL1.ASID defines the ASID.
0b1	TTBR1_EL1.ASID defines the ASID.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

AS, bit [36]

ASID Size.

AS	Meaning
0b0	8 bit - the upper 8 bits of TTBR0_EL1 and TTBR1_EL1 are ignored by hardware for every purpose except reading back the register, and are treated as if they are all zeros for when used for allocation and matching entries in the TLB.
0b1	16 bit - the upper 16 bits of TTBR0_EL1 and TTBR1_EL1 are used for allocation and matching in the TLB.

If the implementation has only 8 bits of ASID, this field is RES0.

The reset behavior of this field is:

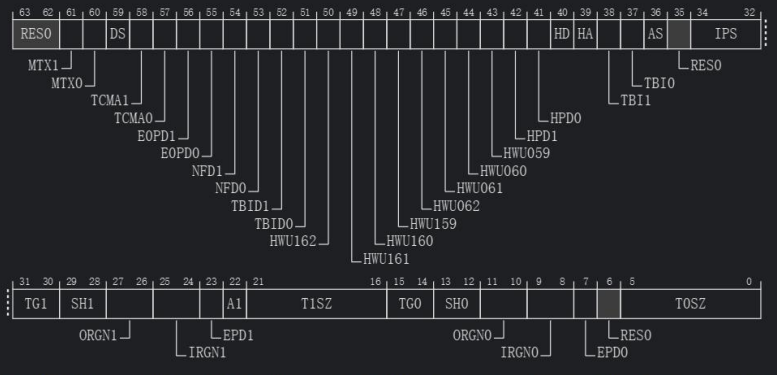
- On a Warm reset, this field resets to an architecturally UNKNOWN value.

由于第 22 位的作用是控制使用 TTBR0_EL1 还是 TTBR1_EL1 定义 ASID。第 36 位的作用是控制 ASID 大小。我们尝试修改宏为（仍然保留 4KB 的颗粒度，启用 TTBR1_EL1）：

```
15 #define TCR_EL1_RSVD        (1UL << 31 | 1UL << 22| 1UL << 23|1UL << 36)    //原定义错误
16 #define TCR_EL2_RSVD        (1UL << 31 | 1UL << 23)
17 #define TCR_EL3_RSVD        (1UL << 31 | 1UL << 23)
```

TCR_EL1 is a 64-bit register.

Field descriptions



Any of the bits in TCR_EL1, other than the EPDx bits when they have the value 1, and the A1 bit are permitted to be cached in a TLB.

接下来修改 g_mem_map_info 的配置。

根据题目中给的修改地址，我们修改为：

```
13 static mmu_mmap_region_s g_mem_map_info[] = {
14     {
15         .virt      = 0xffffffff00000000,
16         .phys      = 0x0,
17         .size      = 0x40000000, // 1G size
18         .max_level = 0x2,  // 不应大于3
19         .attrs     = MMU_ATTR_DEVICE_NGNRNE | MMU_ACCESS_RWX, // 设备
20     }, {
21         .virt      = 0x40000000,
22         .phys      = 0x40000000,
23         .size      = 0x40000000, // 1G size
24         .max_level = 0x2, // // 不应大于3
25         .attrs     = MMU_ATTR_CACHE_SHARE | MMU_ACCESS_RWX, // 内存
26     }
27 };
```

或

```
13 static mmu_mmap_region_s g_mem_map_info[] = {
14     {
15         .virt      = 0xffffffff00000000,
16         .phys      = 0x0,
17         .size      = 0x40000000, // 1G size
18         .max_level = 0x2,  // 不应大于3
19         .attrs     = MMU_ATTR_DEVICE_NGNRNE | MMU_ACCESS_RWX, // 设备
20     }, {
21         .virt      = 0xffffffff40000000,
22         .phys      = 0x40000000,
23         .size      = 0x40000000, // 1G size
24         .max_level = 0x2, // // 不应大于3
25         .attrs     = MMU_ATTR_CACHE_SHARE | MMU_ACCESS_RWX, // 内存
26     }
27 };
```

此时我已完成按照我的思路进行修改，但仍然没有输出，说明没有成功，我尝试用 GDB 调试查看我的执行过程。

```
(gdb) info locals
ret = <optimized out>
page_addr = <optimized out>
page_len = <optimized out>
```

变量值无法查看。

```
0x0000000040003274      43              max_addr = MAX(max_addr, g_mem_map_info[i].virt + g_mem_map_info
[i].size);
(gdb) si
0x0000000040003278      43              max_addr = MAX(max_addr, g_mem_map_info[i].virt + g_mem_map_info
[i].size);
(gdb) si
0x000000004000327c      43              max_addr = MAX(max_addr, g_mem_map_info[i].virt + g_mem_map_info
[i].size);
(gdb) si
0x0000000040003280      43              max_addr = MAX(max_addr, g_mem_map_info[i].virt + g_mem_map_info
[i].size);
(gdb) si
```

输出中有一些重复的行为，即 GDB 在相同的源代码行上多次“停留”。查阅资料得知这是由于汇编指令与源代码不匹配。

```
(gdb) n
15          BEQ Start // 若 CurrentEl 为 EL1 级别，跳转到 Start 处执行，否则死循环。
(gdb) n
21          LDR     x1, =__os_sys_sp_end // ld文件中定义，堆栈设置
(gdb) n
22          BIC     sp, x1, #0xf
(gdb) n

n
```

如果使用 n 指令就会发现在 BIC 执行的下一步陷入死循环。

最终没有成功实现，总的来说这是一次失败的尝试，不知道是哪里的问题。

这个实验和实验六一样，都很难，花了很多时间也没有成功，很失望很难受。最困难的地方主要是没有丝毫的基础，所有代码都是一点点读资料一点点查，很久才能拼起一个完整的框架，此时再去事无巨细地考虑有哪些方面需要修改真的有点难。以上是我的全部尝试思路，做了也挺久的。