amp linux内存

一、参考引用

这里先把一些需要引用的值和定义列出，方便后文参考。

```
/* ddr在物理空间地址的偏移，在这里为0，取决于ddr在amba中的布局 */
#define PHYS_OFFSET __pv_phys_offset
__pv_phys_offset:
    .long   0
    .size   __pv_phys_offset, . - __pv_phys_offset

 //PAGE_OFFSET - the virtual address of the start of the kernel image(我认为是space)
/* 内核空间起始的虚拟地址，一般为0xc0000000，但在amp中为0x80000000 */
#define PAGE_OFFSET     UL(CONFIG_PAGE_OFFSET)
#define CONFIG_PAGE_OFFSET 0x80000000 //include/generated/autoconf.h

//The byte offset of the kernel image in RAM from the start of RAM
/* 内核镜像在ram中相对ram的偏移 */
textofs-y   := 0x00008000
TEXT_OFFSET := $(textofs-y)  //arch/arm/Makefile

/* 一级页表(页表目录)的大小16kb */
#define PG_DIR_SIZE 0x4000

/* 内核镜像起始的虚拟地址（内核镜像的起始就是.text段） */
#define KERNEL_RAM_VADDR    (PAGE_OFFSET + TEXT_OFFSET)

/* 内核一级页表initial page tables的虚拟地址*/
.globl  swapper_pg_dir
.equ    swapper_pg_dir, KERNEL_RAM_VADDR - PG_DIR_SIZE


/* rd = phys + #TEXT_OFFSET - PG_DIR_SIZE
 * 计算内核一级页表的物理地址
*/
.macro  pgtbl, rd, phys
add \rd, \phys, #TEXT_OFFSET - PG_DIR_SIZE
.endm
```

```c
/*  */
typedef u32 pteval_t;
typedef u32 pmdval_t;

/*  */
typedef pteval_t pte_t;
typedef pmdval_t pmd_t;
typedef pmdval_t pgd_t[2];
typedef pteval_t pgprot_t;

/*  */
#define pte_val(x)      (x)
#define pmd_val(x)      (x)
#define pgd_val(x)   ((x)[0])
#define pgprot_val(x)   (x)

/*  */
#define __pte(x)        (x)
#define __pmd(x)        (x)
#define __pgprot(x)     (x)


#define pgd_index(addr)     ((addr) >> PGDIR_SHIFT)
#define pgd_offset(mm, addr)    ((mm)->pgd + pgd_index(addr))

/* to find an entry in a kernel page-table-directory
 * 找到虚拟地址addr对应的pgd页表项的地址
*/
#define pgd_offset_k(addr)  pgd_offset(&init_mm, addr)

/* 根据上一级目录项地址pud和虚拟地址addr
 * 返回对应的pmd页表项的地址
 */
static inline pmd_t *pmd_offset(pud_t *pud, unsigned long addr)
{
    return (pmd_t *)pud;
}

/* 返回虚拟地址virt所对应的pmd页表项的地址 */
static inline pmd_t *pmd_off_k(unsigned long virt)
{
    return pmd_offset(pud_offset(pgd_offset_k(virt), virt), virt);
}

/* 将pmdp指向的一级页表项清0 */
#define pmd_clear(pmdp)         \
    do {                \
        pmdp[0] = __pmd(0); \
        pmdp[1] = __pmd(0); \
        clean_pmd_entry(pmdp);  \
    } while (0)
```

```
 *
 *    pgd               pte
 * |        |
 * +--------+
 * |        |       +------------+ +0
 * +- - - - +       | Linux pt 0 |
 * |        |       +------------+ +1024
 * +--------+ +0    | Linux pt 1 |
 * |        |-----> +------------+ +2048
 * +- - - - + +4    |  h/w pt 0  |
 * |        |-----> +------------+ +3072
 * +--------+ +8    |  h/w pt 1  |
 * |        |       +------------+ +4096
 *
```

上图是arm和linux的页表布局，在arm中每个页表项都为4个字节，无论是一级页表还是二级页表。每个一级页表项表示1M的内存，因此4g空间需要4096个，所以一级页表大小为4096*4字节，即16KB。如果是页式映射，则每个一级页表项指向一个二级页表的首地址，每个二级页表项表示4KB内存，因此一个2级页表具有1M/4KB=256个页表项，因此一个二级页表大小为256*4字节，只占用了四分之一页的大小，而且这些页表项并没有记录所有这些页表项所表示的内存的状态(比如是否为脏)。linux为了实现记录硬件没有支持的状态和充分利用内存，将PGDIR_SHIFT从20换为21，所以一级页表项变为8字节，二级页表从256个项变为512个项，代表了2M的内存。因此最后21到31位相同但20位不同的虚拟地址对应的是同一个8字节条目中相邻的两个，并且这两个条目指向二级页表在内存上连续，然后剩下的半页留给linux二级页表项。

二、第一阶段的页表设置和开启mmu

这个阶段主要负责设置早期的页表，主要是内核代码和数据的映射，然后开启mmu，之后内核便可以运行于虚拟地址，在此之前内核运行于物理地址，但是在此之前是位置无关码所以没有问题。还有就是在使能mmu和跳转到mmap_switched的虚拟链接地址处运行之间还有一小段代码，在这之间cpu发出的地址会经过mmu转化，所以必须要有一个关键代码的一对一的相同映射。也就是说内核镜像部分(turn_mmu_on到__turn_mmu_end)有两个映射，最后那个映射会在paging_init()中去掉。

从内核解压后的入口ENTRY(stext)开始(如果有解压，没有的话uboot直接到这里)。这里只看内存设置相关。

```
/*
 * MMU = off
 * D-cache = off
 * r0 = 0
 * r1 = machine number
 * r2 = dtb pointer （物理地址）
 */
ENTRY(stext)
    .....
    /* r8 = ddr在物理地址空间的偏移，这里为0 */
    ldr r8, =PHYS_OFFSET        @ always constant in this case
    /*
     * r1 = machine no, r2 = atags or dtb,
     * r8 = phys_offset, r9 = cpuid, r10 = procinfo
     */
    .....
    /* 设置部分(内核镜像所在物理区域的部分)一级页表的条目，设置为section，
     * 返回一级页表的物理地址给r4
```

```
    */
    bl  __create_page_tables
    /*
     * The following calls CPU specific code in a position independent
     * manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
     * xxx_proc_info structure selected by __lookup_processor_type
     * above.  On return, the CPU will be ready for the MMU to be
     * turned on, and r0 will hold the CPU control register value.
     */
    /* 以下代码直到开启mmu跳转到__mmap_switched的虚拟地址处运行之前都是位置无关码 */
    ldr r13, =__mmap_switched       @ address to jump to after mmu has been enabled
    ...
    /* r8 = 页表的物理地址 */
    mov r8, r4
    ARM(    add pc, r10, #PROCINFO_INITFUNC )   //设置TTBR1指向swapper_pg_dir页表物理地址
    ...
    /* 设置TTBR0指向swapper_pg_dir页表物理地址，然后使能mmu并跳到__mmap_switched的虚拟地址 */
1:  b   __enable_mmu
ENDPROC(stext)
```

```
/*
 * Setup the initial page tables.  We only setup the barest
 * amount which are required to get the kernel running, which
 * generally means mapping in the kernel code.
 *
 * r8 = phys_offset, r9 = cpuid, r10 = procinfo
 *
 * Returns:
 *  r0, r3, r5-r7 corrupted
 *  r4 = physical page table address
 */
__create_page_tables:
    pgtbl   r4, r8          //r4 = 内核一级页表的物理地址

    /* Clear the swapper page table
     * 将内核一级页表清0
     */
    mov r0, r4      //r0 = 内核一级页表的起始物理地址
    mov r3, #0      //r3 = 0
    add r6, r0, #PG_DIR_SIZE    // r6 = 内核一级页表的结束物理地址后一个字节
1:  str r3, [r0], #4        //将0写入r0，然后r0 = r0 + 4
    str r3, [r0], #4        //将0写入r0，然后r0 = r0 + 4
    str r3, [r0], #4        //将0写入r0，然后r0 = r0 + 4
    str r3, [r0], #4        //将0写入r0，然后r0 = r0 + 4
    teq r0, r6              //测试r0和r6是否相等
    bne 1b                  //如果不等，说明还没清完，back to clear

    ...

    //r7 = proc_info_list__cpu_结构体中的__cpu_mm_mmu_flags成员变量
    ldr r7, [r10, #PROCINFO_MM_MMUFLAGS]

    /* 创建__turn_mmu_on到__turn_mmu_on_end的等价映射，因为在开mmu之后
```

```
     * 到跳转到__mmap_switched虚拟地址处运行还有一小段路，这部分映射
     * 在paging_init()中去除
    */
    adr r0, __turn_mmu_on_loc    //r0 = __turn_mmu_on_loc的物理地址
    ldmia   r0, {r3, r5, r6}     //r3 = __turn_mmu_on_loc的虚拟地址
                               //r5 = __turn_mmu_on的虚拟地址
                               //r6 = __turn_mmu_on_end的虚拟地址
    sub r0, r0, r3           //r0 = 物理地址减虚拟地址的偏移
    add r5, r5, r0           //r5 = __turn_mmu_on的物理地址
    add r6, r6, r0           //r6 = __turn_mmu_on_end的物理地址
    mov r5, r5, lsr #SECTION_SHIFT  //r5 = __turn_mmu_on的物理段号（r5 >> 20）
    mov r6, r6, lsr #SECTION_SHIFT  //r6 = __turn_mmu_on_end的物理段号（r6 >> 20）

    /*  r3 = 一个物理段号为__turn_mmu_on的物理段号的一级页表条目，
     *  r3 = __cpu_mm_mmu_flags | (__turn_mmu_on的物理段号 << 20)
     */
1:  orr r3, r7, r5, lsl #SECTION_SHIFT
    /* 将r3中的页表条目写到r5代表的虚拟地址对应的地址，
     * 因此这是一个虚拟和物理相同的映射
     */
    str r3, [r4, r5, lsl #PMD_ORDER]    //identity mapping
    cmp r5, r6                          //比较r5-r6
    addlo   r5, r5, #1          //如果r5比r6小则r5 = r5 + 1,指向下一个段
    blo 1b                      //如果r5比r6小则继续设置下一个段的一级页表

    /*
     * Map our RAM from the start to the end of the kernel .bss section.
     * 将ddr起始到内核的.bss段结束映射到虚拟连接地址
     */
     /* r0 = r4 + 0xc0000000 >> (20 - 2)
      * r0 = 虚拟地址0xc0000000后一个段对应的内核一级页表表项的物理地址
      */
    add r0, r4, #PAGE_OFFSET >> (SECTION_SHIFT - PMD_ORDER)
    ldr r6, =(_end - 1) //r6 = bss段虚拟结束地址
    orr r3, r8, r7      //r3 = phys_offset | __cpu_mm_mmu_flags
    /* r6 = r4 + (r6 >> (20 - 2))
     * r6 = bss段虚拟结束地址对应的内核一级页表表项的物理地址
     */
    add r6, r4, r6, lsr #(SECTION_SHIFT - PMD_ORDER)
1:  str r3, [r0], #1 << PMD_ORDER//将该页表项设置为r3，即映射到物理地址phys_offset，同时r0更新
    add r3, r3, #1 << SECTION_SHIFT
    cmp r0, r6
    bls 1b

    ...

    /*
     * Then map boot params address in r2 if specified.
     * 映射设备树到内核虚拟地址
     */
    mov     r0, r2, lsr #SECTION_SHIFT  //r0 = dtb的物理地址段号
    movs    r0, r0, lsl #SECTION_SHIFT  //r0 = r0 << 20
    subne   r3, r0, r8              //如果r0不为0则r3 = r0 - phys_offset
```

```
    addne   r3, r3, #PAGE_OFFSET    //如果r0不为0则r3 = r0 - phys_offset + PAGE_OFFSET
                                  //即dtb的虚拟地址
    addne   r3, r4, r3, lsr #(SECTION_SHIFT - PMD_ORDER)
    orrne   r6, r7, r0
    strne   r6, [r3]        //线性映射dtb

    ...

    /* 返回__create_page_tables下一条指令，最终先跑到__enable_mmu */
    mov pc, lr
ENDPROC(__create_page_tables)
    .ltorg
    .align
__turn_mmu_on_loc:
    .long   .                   //__turn_mmu_on_loc的虚拟连接地址
    .long   __turn_mmu_on       //__turn_mmu_on虚拟连接地址
    .long   __turn_mmu_on_end   //__turn_mmu_on_end虚拟连接地址
```

```
__enable_mmu:

    mov r5, #(domain_val(DOMAIN_USER, DOMAIN_MANAGER) | \
              domain_val(DOMAIN_KERNEL, DOMAIN_MANAGER) | \
              domain_val(DOMAIN_TABLE, DOMAIN_MANAGER) | \
              domain_val(DOMAIN_IO, DOMAIN_CLIENT))
    mcr p15, 0, r5, c3, c0, 0       @ load domain access register
    mcr p15, 0, r4, c2, c0, 0       // 设置ttbr0指向swapper_pg_dir页表物理地址

    b   __turn_mmu_on       //使能mmu，并跳到__mmap_switched的虚拟地址处
ENDPROC(__enable_mmu)
```

```
ENTRY(__turn_mmu_on)
    mov r0, r0
    instr_sync
    mcr p15, 0, r0, c1, c0, 0       @ write control reg //使能mmu
    mrc p15, 0, r3, c0, c0, 0       @ read id reg
    instr_sync
    mov r3, r3
    mov r3, r13     //r3 = __mmap_switched的虚拟连接地址
    mov pc, r3      //跳转到__mmap_switched的虚拟连接地址运行
__turn_mmu_on_end:
ENDPROC(__turn_mmu_on)
```

```
__mmap_switched:
    ...
    b   start_kernel     //跳到start_kernel
ENDPROC(__mmap_switched)
```

三、start_kernel之后的内存设置

start_kernel之后的内存设置主要在**paging_init**中，路径为：

> start_kernel --> setup_arch --> paging_init

```
asmlinkage void __init start_kernel(void)
{
    ...
    setup_arch(&command_line);
    ...
}
```

```
void __init setup_arch(char **cmdline_p)
{
    ...
    /* 通过设备树地址找到机器描述符，并提取命令行，内存信息等信息设置到相应的数据结构 */
    mdesc = setup_machine_fdt(__atags_pointer);
    ...
    /* 设置好内核虚拟地址vmalloc_min和arm_lowmem_limit*/
    sanity_check_meminfo();
    /* 通过提取到meminfo中的内存信息设置初始化早期的内存分配器memblock */
    arm_memblock_init(&meminfo, mdesc);
     /* 设置初始化页表，建立区域的映射，迁移内存分配器 */
    paging_init(mdesc);
    ...
}
```

```
/*
 * paging_init() sets up the page tables, initialises the zone memory
 * maps, and sets up the zero page, bad page and bad page tables.
 */
void __init paging_init(struct machine_desc *mdesc)
{
    void *zero_page;

    memblock_set_current_limit(arm_lowmem_limit);   /* 设置lowmem的结束地址 */

    build_mem_type_table();     /* 设置mem_types数组 */
    prepare_page_table();       /* 清页表 */
    map_lowmem();               /* 映射lowmem，即线性映射区 */
    dma_contiguous_remap();     /* 映射dma区 */
    devicemaps_init(mdesc);     /* 映射设备区域，在amp中这里还映射amp相关区域 */
    kmap_init();               /* 分配pkmap区域的页表 */

    top_pmd = pmd_off_k(0xffff0000);    /* 获取0xffff0000对应的pmd页表项地址 */

    /* allocate the zero page. */
    zero_page = early_alloc(PAGE_SIZE);     /* 分配并初始化一个0页 */

    bootmem_init();         /* 将内存分配器从memblock迁移到buddy */
```

```
    empty_zero_page = virt_to_page(zero_page);  //
    __flush_dcache_page(NULL, empty_zero_page); //
}
```

1、prepare_page_table

```
static inline void prepare_page_table(void)
{
    unsigned long addr;
    phys_addr_t end;

    /* 将0到模块起始之前的一级页表目录项都清0
     * 内核镜像起始从模块开始，在内核空间前面一点
    */
    for (addr = 0; addr < MODULES_VADDR; addr += PMD_SIZE)
        pmd_clear(pmd_off_k(addr));

    /* 将模块到内核空间起始的一级页表目录项都清0 */
    for ( ; addr < PAGE_OFFSET; addr += PMD_SIZE)
        pmd_clear(pmd_off_k(addr));

    /* 找到lowmem中的第一个region的结尾
     * 第一个region应该包含内核代码和数据
    */
    end = memblock.memory.regions[0].base + memblock.memory.regions[0].size;
    if (end >= arm_lowmem_limit)
        end = arm_lowmem_limit;

    /* 除了第一个region，将剩下的到vmalloc起始的内核空间的一级页表目录项清0
     * 因为第一个region包含内核代码和数据，清了就无法运行了
    */
    for (addr = __phys_to_virt(end);
         addr < VMALLOC_START; addr += PMD_SIZE)
        pmd_clear(pmd_off_k(addr));
}
```

2、devicemaps_init

```
/*
 * Set up the device mappings.  Since we clear out the page tables for all
 * mappings above VMALLOC_START, we will remove any debug device mappings.
 * This means you have to be careful how you debug this function, or any
 * called function.  This means you can't use any function or debugging
 * method which may touch any device, otherwise the kernel _will_ crash.
 */
static void __init devicemaps_init(struct machine_desc *mdesc)
{
    struct map_desc map;
```

```
    unsigned long addr;
    void *vectors;

    /*
     * Allocate the vector page early.
     */
    /* 分配一页的内存，返回映射后得到的虚拟地址 */
    vectors = early_alloc(PAGE_SIZE);

    /* 将异常向量表和处理函数拷贝到vectors */
    early_trap_init(vectors);

    /* 将VMALLOC_START以上的区域的一级页表项都清0 */
    for (addr = VMALLOC_START; addr; addr += PMD_SIZE)
        pmd_clear(pmd_off_k(addr));

    ...

    /*
     * Create a mapping for the machine vectors at the high-vectors
     * location (0xffff0000).  If we aren't using high-vectors, also
     * create a mapping at the low-vectors virtual address.
     */
     /* 将vectors对应的4kb物理区域映射到0xffff0000 */
    map.pfn = __phys_to_pfn(virt_to_phys(vectors));
    map.virtual = 0xffff0000;
    map.length = PAGE_SIZE;
    map.type = MT_HIGH_VECTORS;
    create_mapping(&map);

    if (!vectors_high()) {
        map.virtual = 0;
        map.type = MT_LOW_VECTORS;
        create_mapping(&map);
    }


    /*
    ***************************************************************************
    *                        Embest Tech co., ltd
    *                         www.embest-tech.com
    ***************************************************************************
    */
    //add extra ampmaping yejc
    /* 将物理地址0x1E000000-0x1FE00000映射到虚拟地址0xfc600000-0xfe400000(30MB)
     * 这是amp共享内存区域
    */
    map.virtual         = AMPMAP_START,
    map.pfn             = __phys_to_pfn(AMPPHY_START),
    map.length          = AMPMAP_SIZE - DMABUF_SIZE,
    map.type            = MT_MEMORY,
    create_mapping(&map);
```

```
    /* 将物理地址0x1FE00000-0x20000000映射到虚拟地址0xfe400000-0xfe600000(2MB,非缓存)
     * 这是bm镜像区域
    */
    map.pfn = __phys_to_pfn(AMPPHY_START + AMPMAP_SIZE - DMABUF_SIZE);
    map.virtual = AMP_SHARE_DMABUF_START;
    map.length = DMABUF_SIZE;
    //map.type = MT_MEMORY_DMA_READY;
    map.type = MT_MEMORY_NONCACHED;
    create_mapping(&map);

    /* 将物理地址0xFFFF0000 - 0xFFFFFFFF映射到虚拟地址0xfe700000-0xfe710000(64kb)
     * 这是外设区域
    */
    map.virtual         = SRAMMAP_START,
    map.pfn             = __phys_to_pfn(SRAMPHY_START),
    map.length          = SRAMMAP_SIZE,
    map.type            = MT_MEMORY,
    create_mapping(&map);

    /*
     * Ask the machine support to map in the statically mapped devices.
     */
    if (mdesc->map_io)
        mdesc->map_io();
    fill_pmd_gaps();

    /* Reserve fixed i/o space in VMALLOC region */
    pci_reserve_io();

    /*
     * Finally flush the caches and tlb to ensure that we're in a
     * consistent state wrt the writebuffer.  This also ensures that
     * any write-allocated cache lines in the vector page are written
     * back.  After this point, we can start to touch devices again.
     */
    local_flush_tlb_all();
    flush_cache_all();
}
```