

实验案例六：内核子系统—物理虚拟内存

实验案例六：内核子系统—物理虚拟内存

- 一、实验简介
- 二、实验内容及要求
 - 任务一：虚实映射测试
- 三、实验原理
 - 1. 物理内存
 - (1). 概念
 - (2). 数据结构
 - (3). 伙伴算法
 - (4). 物理内存使用
 - 2. 虚拟内存
 - (1). 概述
 - (2). 实现原理
 - (3). 页表映射
 - (4). 内存映射过程
- 四、实验流程
 - 任务一：虚实映射测试
 - 1. 流程
- 五、参考资料

一、实验简介

物理内存和虚拟内存是现代计算机系统的重要概念，在 OpenHarmony 的学习过程中同样至关重要。物理内存是计算机中直接与 CPU 交互的硬件存储，用于存储正在运行的程序 and 数据的临时区域。虚拟内存是一种内存管理技术，它为每个进程创建独立的虚拟地址空间，该空间可大于实际物理内存。操作系统通过虚拟内存管理和分配物理内存资源，确保每个进程拥有足够的内存空间，即使物理内存不足。此机制通过页表映射和页面调度实现，可将不常用的数据或代码移至磁盘，从而释放物理内存供其他进程使用。

本节实验基于 LiteOS-A 内核，旨在学习 OpenHarmony 中物理内存与虚拟内存的概念、重要性及实现方法，并通过在内核中增添代码的实践，深化对该知识点的理解。

二、实验内容及要求

本次实验需要依次完成下列实验要求，并提交在 QEMU 中运行的最终结果截图。

任务一：虚实映射测试

请参考函数 `LOS_vmalloc` 的实现尝试在内核空间中申请一个物理页 `ptr` 与两个虚拟页 `region1` 和 `region2`，最后将上述虚拟地址 `region1` 和 `region2` 映射到同一个物理页 `ptr` 上。在完成虚实映射任务之后，尝试运行如下代码：

```
VADDR_T va1 = region1->range.base, va2 = region2->range.base;
int *va11 = (int*)va1, *va12 = (int*)va2;
PRINTK("virtual address of va1: %p\n", va11);
PRINTK("value of va1: %d\n", *va11);
PRINTK("virtual address of va2: %p\n", va12);
PRINTK("value of va2: %d\n", *va12);
*va11 = 2025;
PRINTK("Assigning a value to va1: %d\n", *va12);
PRINTK("virtual address of va1: %p\n", va11);
PRINTK("value of va1: %d\n", *va11);
PRINTK("virtual address of va2: %p\n", va12);
PRINTK("value of va2: %d\n", *va12);
```

若实现正确，打印输出应如下所示，虽然va1和va2指向着不同的虚拟地址，但是改变其中任意一个变量，都会影响到另一方。

```
virtual address of va1: 0xc0000000
value of va1: 0
virtual address of va2: 0xc0001000
value of va2: 0
Assigning a value to va1: 2025
virtual address of va1: 0xc0000000
value of va1: 2025
virtual address of va2: 0xc0001000
value of va2: 2025
```

本实验旨在加深对 LiteOS 内核中虚拟地址与物理地址映射机制的理解，并掌握相关的开发方法。

三、实验原理

1. 物理内存

(1). 概念

物理内存是计算机上最重要的资源之一，指的是实际的内存设备提供的、可以通过CPU总线直接进行寻址的内存空间，其主要作用是为用户提供临时存储空间。OpenHarmony的LiteOS内核对于物理内存采用段页式管理，除了内核堆占用的一部分内存外，其余可用内存均以4KB为单位划分成页帧，内存分配和内存回收便是以页帧为单位进行操作，而其中每一个页帧又根据情况分属于不同的段。

(2). 数据结构

OpenHarmony采用段页式管理，内核代码中找到与之相关的物理段与物理页帧的数据结构体定义于`//kernel/liteos_a/kernel/base/include`下的`los_vm_phys.h`与`los_vm_page.h`之中，用于储存划分的各个内存区域的信息。其中相关代码展示如下：

物理页帧

```
typedef struct VmPage {
    LOS_DL_LIST    node;           /**< 链表节点，连接在物理段的伙伴链表上 */
    PADDR_T        physAddr;       /**< 物理页帧起始物理地址 */
    Atomic          refCounts;      /**< 引用次数 */
    UINT32          flags;          /**< 标志位 */
    UINT8           order;          /**< 伙伴算法级别 */
    UINT8           segID;          /**< 所属物理段ID */
    UINT16          nPages;         /**< 分配页数 */
} LosVmPage;
```

物理页帧是物理内存分配的基本单位，LiteOS中的大部分内存都被划分为大小4KB的物理页，而每个物理页都需要上述数据结构管理其信息，其中的重要数据成员作用如下：

- node: 链表节点，将相同伙伴算法级别的各个空闲物理页管理节点连接在一起，并挂在物理段的 freeList 之上。
- physAddr: 描述物理页帧在内存中的起始物理地址，每个物理页帧大小为4KB。
- flags: 物理页标志位，包括**FILE_PAGE_SHARED(共享)**、**FILE_PAGE_FREE(空闲)**、**FILE_PAGE_REFERENCED(被引用)**等标注，描述物理页的状态，
- order: 伙伴算法相关，表示该物理页在伙伴算法中所属级别，其被挂在哪一个级的空闲链表上。
- segID: 表示该物理页所属的物理段ID
- nPages: 许多时候需要申请连续的物理空间，也意味着需要使用到连续的物理页帧。nPages就代表着自该页开始，后续连续的nPages-1页也同样属于被分配的物理页。

物理段

```
#define VM_LIST_ORDER_MAX    9    //伙伴算法最大分组级别
#define VM_PHYS_SEG_MAX     32    //LiteOS最大支持物理段数量

typedef struct VmPhysSeg {
    PADDR_T start;                /* 物理内存起始地址 */
    size_t size;                  /* 物理内存大小 */
    LosVmPage *pageBase;          /* 物理页帧管理区域起始地址 */

    SPIN_LOCK_S freeListLock; /* 伙伴链表自旋锁 */
    struct VmFreeList freeList[VM_LIST_ORDER_MAX]; /* 伙伴链表空闲页 */

    SPIN_LOCK_S lruLock;
    size_t lruSize[VM_NR_LRU_LISTS];
    LOS_DL_LIST lruList[VM_NR_LRU_LISTS];
} LosVmPhysSeg;
```

物理段中包含多个物理页，每个物理段都有着自己的伙伴链表，用于管理其拥有的物理页。在上述物理段数据结构中部分重要成员作用如下：

- start、size: 描述了该物理段在LiteOS内存中的起始物理地址，以及其区域大小。
- pageBase: 指明该物理段中所有物理页帧管理区域的起始虚拟地址。
- freeList: 每个物理段都使用伙伴算法进行物理页帧的管理，LiteOS共有VM_LIST_ORDER_MAX即9级的划分，freeList也为大小为9的链表，用于链接根据伙伴算法划分的各个级别大小的物理页帧。

- lruList、lruSize: 物理页帧LRU(最近最少使用)置换算法相关，数组大小为5，分别描述5种不同类型的链表。当物理页不足需要进行置换时用于指导需要替换物理页。

(3). 伙伴算法

伙伴算法来管理段的物理页帧，其注重物理内存的**连续性**，用于缓解多次申请连续的物理页帧之后，导致已经分配的块内充斥大量小而分散的内存块，而导致后续很难再申请大块的连续页帧。

LiteOS实现的伙伴算法之中将所有连续的物理页块根据其大小分为0-8共9个等级。等级*i*的链表序列连接着具有连续 2^i 个物理页帧的头页帧。每次申请内存首先从空闲的内存中搜索比申请的内存大的最小的内存块。如果这样的内存块存在，直接将该内存块从链表上摘下并修改页表节点相关信息。而如果这样的内存不存在，则操作系统将依次寻找更大的连续空闲内存，然后将这块内存平分成几部分，一部分返回给程序使用，另一部分重新放到链表的相应位置之中。

图1展示了LiteOS在伙伴算法的基础之上申请12KB内存的示意图，由于对于12KB内存的申请相当于申请3个连续页帧，可以看到系统将原本具有8个连续页帧的空闲块拆分为两个大小为4的页帧，并将其中之一的3个页帧分配出去，而最后剩下的大小为4和1的页帧将被重新插入到链表之中。LiteOS中对于物理内存的申请和释放的具体操作可以查看函数 `LOS_PhysPagesAllocContiguous` 与 `LOS_PhysPagesFreeContiguous`。

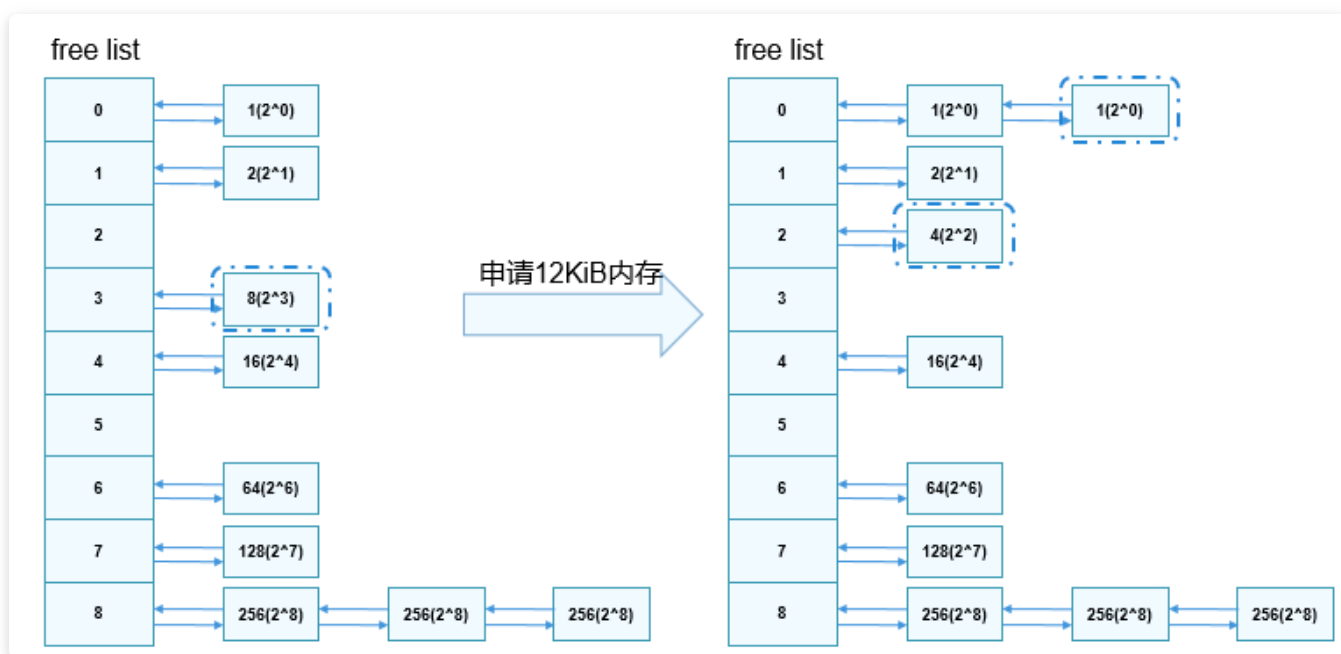


图1.伙伴算法内存申请[1]

(4). 物理内存使用

操作系统对物理内存真正进行管理之前首先需要进行包括确定物理空间、物理页分割、物理段初始化和物理页帧初始化等操作。在此之后，操作系统才能正常地为各个进程分配合适的内存资源以保证程序正常运行。在LiteOS之中物理内存的初始化操作在 `OsMain` -> `OsSysMemInit` -> `OsVmPageStartup` 中完成，其中大致流程如下：

1. 确定初始化物理空间。图2中展示了LiteOS对于物理内存空间使用方式，其中Kernel.bin为内核镜像其中包括内核代码以及相关变量数据等等，Heap为内核堆空间也就是在第五节实验中动态内存管理所负责的空间，而Page frames则是本节实验的重点，它占据了剩余的所有物理内存。而下列代码中的**g_physArea**是LiteOS预先设置好的变量，其中定义了整个操作系统运行过程中所能使用的物理内存区域。因为Kernel.bin以及Heap的存在，在 `OsVmPageStartup` 的开始，需要先校正**g_physArea**的大小。

```

/* Physical memory area array */
STATIC struct VmPhysArea g_physArea[] = {
    {
        .start = SYS_MEM_BASE,
        .size = SYS_MEM_SIZE_DEFAULT,
    },
};

```



图2.物理内存使用分布[1]

2. 为物理页管理节点 `vmPage` 分配空间,并将剩余物理内存按照4KB大小进行分割。如下代码所示, 由于每个物理页帧都需要一个`vmPage`管理, 因而在剩余的物理内存中还需要根据所分割的物理页帧数量建立等量的`vmPage`管理节点。这些节点同样需要占据一定的物理空间, LiteOS就将所有的物理页管理节点`vmPage`放在内核堆空间以及物理页帧之间。

```

UINT32 pageNum = OsVmPhysPageNumGet(); // 未包括LosVmPage管理节点的页帧总数
nPage = pageNum * PAGE_SIZE / (sizeof(LosVmPage) + PAGE_SIZE); // 包括LosVmPage节点的页帧总数
g_vmPageArraySize = nPage * sizeof(LosVmPage);
g_vmPageArray = (LosVmPage *)OsVmBootMemAlloc(g_vmPageArraySize); // 在内核堆空间Heap之后申请空间存储LosVmPage
OsVmPhysAreaSizeAdjust(ROUNDUP(g_vmPageArraySize, PAGE_SIZE)); // 调整剩余内核空间大小

```

3. 初始化物理段与物理页管理节点, 并且计算可供管理的物理页帧。在确定好剩余待管理的物理空间以及物理页节点后, LiteOS首先进行物理段的初始化, 使用剩余的`g_physArea`指向的物理内存建立第一个物理段并进行初始化, 确定物理段所包含的物理页管理节点`vmPage`。

```

OsVmPhysSegAdd();
OsVmPhysInit();

```

4. 物理页节点初始化。根据伙伴算法计算每个物理页帧的等级, 也就是代码里所指的`order`, 并将其挂载到物理段节点对应等级的`freeList`链表上, 以便后续按照伙伴算法分配内存与释放内存。

```

for (segID = 0; segID < g_vmPhysSegNum; segID++) { // 依次对每个物理段中的物理页表初始化, 事实上此时只有1个物理段
    seg = &g_vmPhysSeg[segID];
    nPage = seg->size >> PAGE_SHIFT;
    UINT32 count = nPage >> 3;
    UINT32 left = nPage & 0x7;

    // 每次循环初始化8个物理页帧, 这样处理是为了优化性能
    for (page = seg->pageBase, pa = seg->start; count > 0; count--) {
        VMPAGEINIT(page, pa, segID);
        VMPAGEINIT(page, pa, segID);
        VMPAGEINIT(page, pa, segID);
        VMPAGEINIT(page, pa, segID);
    }
}

```

```

        VMPAGEINIT(page, pa, segID);
        VMPAGEINIT(page, pa, segID);
        VMPAGEINIT(page, pa, segID);
        VMPAGEINIT(page, pa, segID);
    }
    for (; left > 0; left--) {
        VMPAGEINIT(page, pa, segID);
    }
    OsVmPageOrderListInit(seg->pageBase, nPage);
}

```

完成上述操作后，LiteOS 的物理内存初始化即告结束。在后续系统运行过程中，如 `vmalloc`、大于 16KB 的 `kmalloc` 以及用户进程所使用的内存资源均来自该内核堆空间。此外，物理页帧的申请与释放均基于伙伴算法实现，具体细节不再赘述，可自行查阅源码进行学习。

2. 虚拟内存

(1). 概述

物理内存是计算机中实际安装的RAM（随机存取存储器），是计算机硬件的一部分，直接与CPU通信，用于存储正在运行的程序和数据。物理内存的速度快，可以直接被CPU访问，但其容量通常有限。并且随着内存的分配增多，可用的内存块地址也大多不再连续，所有的进程也共享着一个物理内存地址空间，这会给操作系统的管理带来很大的麻烦。而与之相对的虚拟内存管理是计算机系统管理内存的一种技术，为程序提供一个抽象的、统一的内存视图，用于解决上述提到的问题。虚拟内存管理提供了一种新的地址空间，虚拟地址空间(VA)。不同于物理内存所使用的物理地址(PA)受限于硬件设备，虚拟地址是由编译器和链接器在定位程序时分配的，对于每个进程而言，这个地址空间似乎是连续的，并且独立于任何其他进程。但在实际上这些虚拟内存地址空间实际上分别映射到不同的实际物理内存空间上。虚拟地址不仅能够提供了隔离性，每个进程都有自己的虚拟地址空间，这有助于避免进程间的冲突，允许进程使用比实际物理内存更大的地址空间，还能实现延迟加载使得只有真正使用数据时才将其从硬盘中加载进内存。

(2). 实现原理

虚拟内存管理

需要注意的是LiteOS通过进程空间 `vmSpace` 中的成员 `regionRbTree` 使用红黑树技术来管理线性区 `VmMapRegion`，也就是虚拟地址。

```

typedef struct VmSpace {
    LOS_DL_LIST      node;           /**< 进程空间节点 */
    LosRbTree        regionRbTree;  /**< 红黑树，管理虚拟地址空间 */
    LosMux           regionMux;     /**< 红黑树锁 */
    VADDR_T          base;          /**< 虚拟空间基址 */
    UINT32           size;          /**< 虚拟空间大小 */
    VADDR_T          heapBase;      /**< 虚拟空间堆基址 */
    VADDR_T          heapNow;       /**< 虚拟空间堆当前地址 */
    LosArchMmu       archMmu;      /**< mmu信息 */
    ...
} LosVmSpace;

```


LiteOS中提供了一系列的接口帮助管理虚拟内存，如 `LOS_RegionAlloc` 和 `LOS_RegionFree` 分别是申请虚拟内存空间和释放虚拟内存空间。一般来说所有的内核进程共享着同样的虚拟地址空间，其虚拟地址的分配是随着物理空间的分配和虚实映射一同进行的，分配了虚拟地址的同时，所用的数据也已经一同加载进了物理内存之中。但是用户态进程则不同，每个用户进程都拥有着各自独立的虚拟地址空间，而且其所会占用的内存一般情况也都很大，因而用户进程的虚拟地址空间的分配是和数据加载到物理内存中是分离的，也就是说虽然进程已经分配了一段虚拟内存，但是其对应的数据并不一定已经加载到物理内存中，真正的加载需要等到对这段数据进行访问，产生缺页中段时才会进行。在加载到物理内存后，虚实映射关系才会真正地在页表之中更新保存。

虚实映射

在实现虚拟内存的程序运行过程中，程序所使用的地址事实上都是虚拟地址，若想要访问位于内存上的数据就必须将原先的虚拟地址转换为数据在物理内存上的真实地址，并进而从该地址上提取所需要的数据。这个由虚拟地址转换到物理地址的过程一般被称为虚实映射。这个转换的过程一般是由内存管理单元(MMU)来完成，正如图3所示。

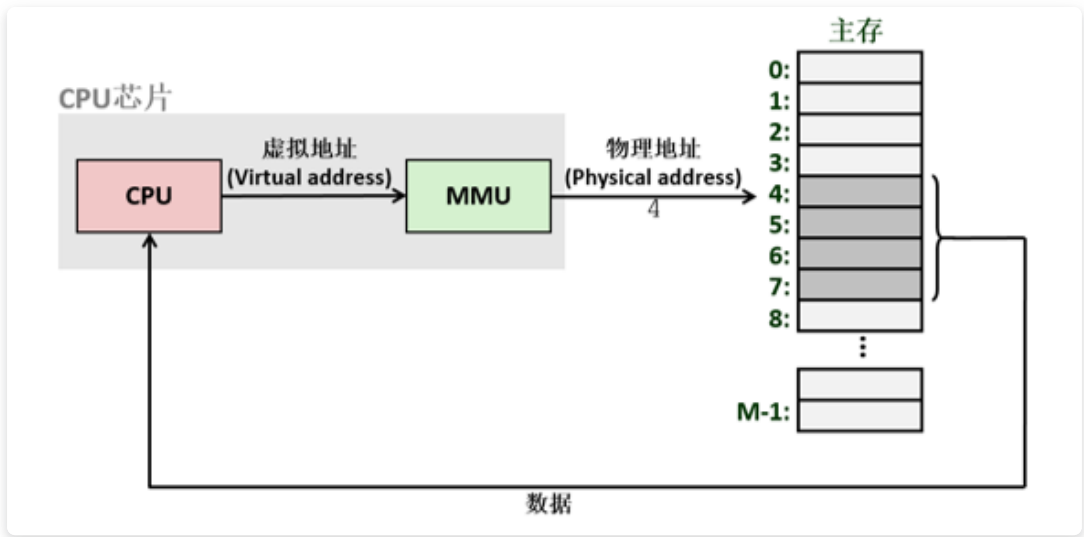


图3.CPU、MMU和内存之间的关系

MMU虽然能够完成虚拟地址到物理地址的映射工作，但是将虚拟地址映射到哪里的物理地址则是由页表(Page Table)来描述。页表中保存着一个进程空间中虚拟地址和物理地址的映射关系，这个关系每个进程都不一样，MMU完成地址转换工作也就需要借助于页表中的信息来完成。因此一个内存访问过程应该是CPU发出访问虚拟地址A的请求，MMU拦截虚拟地址A，并访问页表逐级查找对应的页表项，读取页表项中储存着的物理地址，最后将CPU请求的虚拟地址A替换为映射后的物理地址B并根据需求从内存的对应该位置读取数据返回给CPU。

虚实地址转换工作由MMU硬件自动完成，无需通过软件来实现。但是正如上文所述，MMU想要完成虚实映射就必须要知道映射关系，也就是需要知道使用哪里的页表，因而在操作系统中需要手动配置相关寄存器，以告诉MMU从哪里读取页表完成虚实地址的转换。ARM处理器使用协处理器15(CP15)的寄存器来控制caches、MMU、保护系统等。其中的c2寄存器就描述着MMU所使用的页表基址，MMU每次实现虚实地址映射就是从c2寄存器中读取映射关系完成地址转换。

在LiteOS之中的 `/kernel/liteos_a/arch/arm/arm/include/los_arch_mmu.h` 之中定义的 `ArchMmu` 管理着mmu的相关信息，代码如下：

```
typedef struct ArchMmu {
    LOSMUX          mtx;          /**< arch mmu page table entry modification mutex
lock */
    VADDR_T         *virtTtb;     /**< translation table base virtual addr */
    PADDR_T         physTtb;      /**< translation table base phys addr */
    UINT32          asid;         /**< TLB asid */
    LOS_DL_LIST     ptList;       /**< page table vm page list */
} LosArchMmu;
```

- virtTtb: mmu所使用的页表在内存中的虚拟地址。该地址主要提供给内核程序操作使用，例如在其中更改虚实映射关系。
- physTtb: mmu所使用的页表在内存中的物理地址。专门用于提供给CP15的c2寄存器，也就是告诉mmu从哪里读取页表。
- asid: 进程标识。由于每个进程都由自己独立的虚拟空间，当mmu使用高速缓存TLB时需要对照此id，以避免使用错误的缓存。
- ptList: 页表链接节点。由于每个进程都有一个虚拟空间，因而也需要有独立的页表，每个页表也同样需要占用物理内存空间。该链表上链接的就算页表所占用的物理空间所属的物理页帧的vmPage节点。例如当需要销毁某个进程的虚拟空间时，就可以通过该链表依次释放页表所占用的内存空间。

上述的 ArchMmu 结构体与MMU硬件功能强相关，LiteOS中每当需要切换进程使用一个新的虚拟地址空间时，都需要使用函数 `LOS_ArchMmuContextSwitch(LosArchMmu *archMmu)`，其作用就是使用给定的 ArchMmu 结构体中的信息更新协处理器CP15的寄存器值，其中包括有页表基地址和asid等信息。这也是LiteOS实现虚拟内存管理的重要组成部分。

(3). 页表映射

页表承担着保存虚拟地址与物理地址之间的映射关系，而这个映射关系以什么样的形式保存在页表中也是一个重要课题。一般而言，页表的映射方式一般需要考虑两个问题，分别是虚拟页的大小以及页表分级形式。

操作系统一般会将虚拟内存分割为称为虚拟页的内存块，根据硬件设计不同，支持的虚拟页大小也不相同。虚拟页如果设置太小，会显著增加页表项的数量，占用更多的内存，并且缺页中断也更可能发生。但是虚拟页设置太大，也会导致页面换入换出效率低，内部碎片太大浪费内存。

页表分级也是需要考虑的问题之一。对于32位的系统而言，其所支持的寻址范围达到4GB的大小。而假设仅使用4KB的虚拟页，这就意味着需要在页表中设置 2^{20} 个页表项才能覆盖整个4GB的寻址范围，这无疑会占用太多内存空间。因而大部分的mmu都会支持多级页表，即mmu需要通过依次访问不同级别的页表最后才能得到所需要的物理地址。这样虽然会增加内存访问次数，消耗更多的时间，但是这也能节省空间。

以本次实验所使用的arm32架构硬件为例，其支持一级和二级的映射结构。其中一级映射分别支持1M的Section映射以及16M的Super section映射。而二级映射中则支持虚拟页大小为4KB和64KB的映射。而在LiteOS内核之中，默认使用的是1M的Section一级映射和4KB的虚拟页二级映射方式。在LiteOS代码中，一级映射的L1页表和二级映射的L2页表项描述格式定义如下：


```
//kernel/liteos_a/arch/arm/arm/include/los_mmu_descriptor_v6.h
/* L1 descriptor type */
#define MMU_DESCRIPTOR_L1_TYPE_INVALID (0x0 << 0) // 无效页表
#define MMU_DESCRIPTOR_L1_TYPE_PAGE_TABLE (0x1 << 0) // 二级页表
#define MMU_DESCRIPTOR_L1_TYPE_SECTION (0x2 << 0) // 1M Section映射
#define MMU_DESCRIPTOR_L1_TYPE_MASK (0x3 << 0) // 页表掩模
/* L2 descriptor type */
#define MMU_DESCRIPTOR_L2_TYPE_INVALID (0x0 << 0) // 无效页表
#define MMU_DESCRIPTOR_L2_TYPE_LARGE_PAGE (0x1 << 0) // 64KB大页表，未实现
#define MMU_DESCRIPTOR_L2_TYPE_SMALL_PAGE (0x2 << 0) // 4KB小页表
#define MMU_DESCRIPTOR_L2_TYPE_SMALL_PAGE_XN (0x3 << 0) // 小页表，暂时与4KB小页表没有实现差别
#define MMU_DESCRIPTOR_L2_TYPE_MASK (0x3 << 0) // 页表掩模
```

`LOS_ArchMmuMap` 中实现了生成L1页表和L2页表，并保存虚拟地址和物理地址之间映射关系到 `ArchMmu` 中的过程，其中重要代码如下：

```
status_t LOS_ArchMmuMap(LosArchMmu *archMmu, VADDR_T vaddr, PADDR_T paddr, size_t count,
UINT32 flags)
{
    PTE_T l1Entry;
    ...
    MmuMapInfo mmuMapInfo = { // 映射信息
        .archMmu = archMmu,
        .vaddr = &vaddr,
        .paddr = &paddr,
        .flags = &flags,
    };
    ...
    /* see what kind of mapping we can use */
    while (count > 0) {
        if (MMU_DESCRIPTOR_IS_L1_SIZE_ALIGNED(vaddr) &&
            MMU_DESCRIPTOR_IS_L1_SIZE_ALIGNED(paddr) &&
            count >= MMU_DESCRIPTOR_L2_NUMBERS_PER_L1) {
            /* compute the arch flags for L1 sections cache, r, w, x, domain and type */
            saveCounts = OsmMapSection(archMmu, flags, &vaddr, &paddr, &count);
            //虚拟地址和物理地址对齐 0x100000(1M)则采用1M的Section映射
        } else {
            /* have to use a L2 mapping, we only allocate 4KB for L1, support 0 ~ 1GB */
            l1Entry = OsGetPte1(archMmu->virtTtb, vaddr); //获取页表基址中获取L1页表项
            if (OsIsPte1Invalid(l1Entry)) { //若标志位无效说明需要建立第二级页表，并保存映射关系
                saveCounts = OsmMapL1PTE(&mmuMapInfo, l1Entry, &count);
            } else if (OsIsPte1PageTable(l1Entry)) { //存在第二级页表
                saveCounts = OsmMapL2PageContinuous(l1Entry, flags, &vaddr, &paddr,
&count);
            } else {
                // 64KB大页表未实现
                LOS_Panic("%s %d, unimplemented tt_entry %x\n", __FUNCTION__, __LINE__,
l1Entry);
            }
        }
    }
}
```

```

    }
    ...
}
return mapped;
}

```

由上述代码可知，LiteOS建立页表映射关系时对内存地址对齐于1M的地址映射采用1M的Section方式，否则就按照4KB虚拟页的二级页表形式映射。

(4). 内存映射过程

OpenHarmony的运行启动过程之中，每个用户态进程创建之时系统都会为其创建一个独立的页表，使其拥有独立的虚拟地址空间，而内核态进程则不同，所有的内核态进程都共享使用着同一个页表，使用着同样的虚拟地址空间。

这里从OpenHarmony的启动开始逐步讲解LiteOS的虚实映射关系的初始化过程：

1. 关闭MMU。LiteOS的启动始于 `reset_vector_up.s` 的 `reset_vector` 段,在启动的最开始，LiteOS事实上还没有初始化页表，因而无法使用虚实地址的转换，故而在启动的第一步需要先关闭mmu的映射功能以及缓存。

```

mrc    p15, 0, r0, c1, c0, 0
bic    r0, #(1 << 12)          /* i cache */
bic    r0, #(1 << 2)           /* d cache */
bic    r0, #(1 << 0)           /* mmu */
mcr    p15, 0, r0, c1, c0, 0

```

2. 重定位系统镜像。LiteOS希望能够将内核镜像移动到指定的物理内存地址 `SYS_MEM_BASE` 处，但是加载内核镜像并不能保证这一点，因此在内核启动时需要手动运行代码完成这一点，将内核镜像重定向到指定位置。关于 `SYS_MEM_BASE` 地址的具体定义，事实上并不属于内核代码的一部分，因为不同的硬件架构有着不同的内存大小和分布，因此需根据开发板配置确定。如这次实验所使用的开发板代码位于 `//device/board/hisilicon/liteos_a/board/include/board.h`。
3. 配置内核页表 `g_firstPageTable`。内核页表大小位16KB，刚好能够覆盖4GB的地址空间，其声明位于 `kernel\liteos_a\arch\arm\arm\src\los_arch_mmu.c` 之中。在内核的汇编启动阶段，为了之后开启MMU使用虚拟地址，需要提前配着内核页表，在其中保存虚实映射关系。其中保存的映射关系同样属于开发板的配置，具体映射可以在 `//device/board/hisilicon/liteos_a/board/board.c` 中查看。
4. 开启MMU，使用内核页表 `g_firstPageTable`。最进入到C语言启动阶段之前，liteos还需要开启mmu，正式使用内核的虚拟地址空间。而由于此时的PC寄存器中储存的还是物理地址，如果直接使用内核页表开启mmu会导致错乱，因而这一步需要通过构建临时页表帮助转换来完成。
5. 在这一步中内核正式进入了C语言启动阶段，此时也已经开启了MMU虚实映射。但此时的内核页表其实仍然没有完成配置，这是因为在汇编阶段在内核页表中所写的虚实映射都是不加区分的1M段Section映射，这会导致大量的内存浪费。因而在 `OsMain->OsSysMemInit->OsInitMappingStartUp` 中需要按照上一小节的1M Section与4KB虚拟页结合的方式重新设置内核页表的映射关系。

以上就是内核启动过程中对于内核页表初始化的大致流程，不过在上述页表映射初始化其实只完成了内核页表的配置，所有的内核进程共享这同一个页表同一个虚拟空间，其内存空间分布如图4所示。

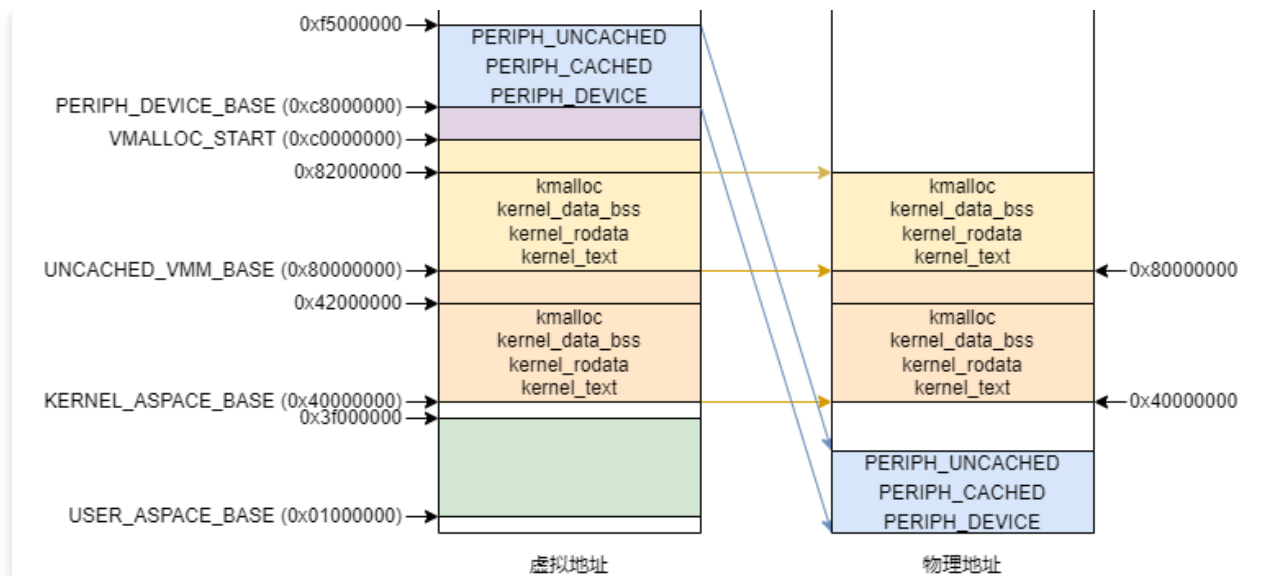


图4.LiteOS-a映射全景图[2]

四、实验流程

任务一：虚实映射测试

本节任务需要在 `//kernel/liteos_a/arch/arm/arm/src/los_arch_mmu.c` 中实现虚实映射测试，学习LiteOS中虚实映射的实现原理。

1. 流程

1. 阅读 `LOS_VmAlloc` 的实现，学习LiteOS中如何申请虚拟空间与物理空间以及如何完成虚实映射
2. 在 `//kernel/liteos_a/arch/arm/arm/src/los_arch_mmu.c` 的函数 `OsInitMappingStartup` 的末尾，即调用 `OsKSectionNewAttrEnable` 函数使得MMU启用内核页表之后，添加下述代码

```
{
    PRINTK("\nkernel page table initialization completed!\n");
    /*
    任务一：
    1. 申一份请物理页和两份虚拟页va1与va2
    2. 将两份虚拟页映射到同一个物理页上
    */
    int *va11 = (int*)va1, *va12 = (int*)va2;
    PRINTK("virtual address of va1: %p\n", va11);
    PRINTK("value of va1: %d\n", *va11);
    PRINTK("virtual address of va2: %p\n", va12);
    PRINTK("value of va2: %d\n", *va12);
    *va11 = 2025;
    PRINTK("Assigning a value to va1: %d\n", *va12);
}
```

```

    PRINTK("virtual address of va1: %p\n", va1);
    PRINTK("value of va1: %d\n", *va1);
    PRINTK("virtual address of va2: %p\n", va2);
    PRINTK("value of va2: %d\n", *va2);
    /*
    任务一：
    3. 释放申请的物理页和虚拟页
    */
}

```

3. 补全其中剩余内容，并重新编译与运行OHOS，如果正常运行的话结果应当如下所示：

```

virtual address of va1: 0xc0000000
value of va1: 0
virtual address of va2: 0xc0001000
value of va2: 0
Assigning a value to va1: 2025
virtual address of va1: 0xc0000000
value of va1: 2025
virtual address of va2: 0xc0001000
value of va2: 2025

```

五、参考资料

- [1]. 虚实映射文档:<https://docs.openharmony.cn/pages/v4.1/zh-cn/device-dev/kernel/kernel-small-basic-inner-reflect.md>
- [2]. LiteOS-A 内核页表和地址映射:<https://forums.openharmony.cn/forum.php?mod=viewthread&tid=925>