第二十章　实验 2: 内存管理

20.1　简介

在实验 2 中，需要实现 ChCore 的内存管理功能，包括以下三个部分：

第一部分是**物理内存管理**。ChCore 使用了伙伴系统作为内存分配器，并且以页的粒度对物理内存进行组织与管理。本实验已经给出了物理内存管理相关的基本数据结构、物理内存布局、slab 分配器等。需要实现伙伴系统的相关功能，主要包括向伙伴系统中添加物理页、删除物理页面、伙伴块的合并与分裂等。

第二部分是**虚拟内存映射**。ChCore 支持**内存管理单元（Memory Management Unit，MMU**）和虚拟内存，在实现物理内存管理的基础上，可以通过设置页表实现虚拟内存到物理内存的映射。在程序执行的过程中，MMU 会负责将虚拟内存翻译成为真正的物理内存。本实验需要学习 ARM 中的页表配置，MMU 翻译页表机制等相关知识，然后完成 ChCore 中关于页表映射与解映射的相关函数。

第三部分是**内核地址空间**。在完成虚拟内存映射之后，你需要为内核映射一段新的虚拟内存。同时需要了解在 ChCore 是如何实现内核空间和用户空间的隔离。

20.1.1　评分

实验 2 中，代码部分的总成绩为 100 分。可使用如下命令检查当前得分：

```
chcore$ make grade
…
Score: 100/100
```

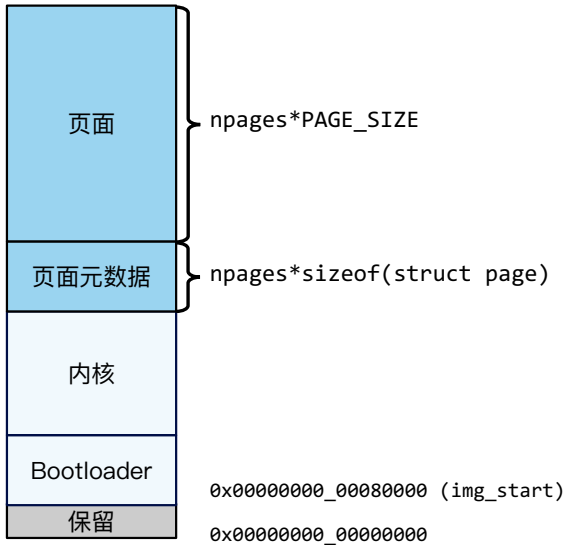

图 20.1: ChCore 中物理内存布局

20.2 第一部分：物理内存管理

为了提高内存资源的利用率，ChCore 以 4KB（PAGE_SIZE）为页的粒度对物理内存进行管理，同时采用了主流物理内存管理机制：**伙伴系统（buddy system）**。

20.2.1 物理内存布局

图 20.1展示了 ChCore 的物理内存布局。其中，物理地址img_start以下是保留的。img_start~img_end（img_start被硬编码为 0x80000）被分为了两个区域：其中的底部区域作为 bootloader 代码、数据和 CPU 栈，每个 CPU 栈的大小为 4KB；顶部区域用于内核的代码、数据和 CPU 栈，每个 CPU 栈的大小为 8KB。img_end以上空闲的物理内存由物理页分配器管理。分配器将内存区域划分为两个范围：元数据范围和页面范围，它们的大小与页面数（npages）有关。页面元数据包括list_node和flags等。

问题 1

请简单解释，在哪个文件或代码段中指定了 ChCore 物理内存布局。你可以从两个方面回答这个问题: 编译阶段和运行时阶段。

20.2.2 伙伴系统

ChCore 基于以上的物理内存布局，实现物理内存分配器：每个物理页面对应一个`struct page`对象维护页面信息，并且通过该对象的链表跟踪哪些页面是空闲的。

一种主流的内存管理机制——伙伴系统可以用于组织物理页面。伙伴系统中的每个内存块都有一个**阶（order）**，阶是从 0 到指定上限`buddy_max_order`的整数。一个 n 阶的块的大小为 $2^n * \text{PAGE\_SIZE}$，因此这些内存块的大小正好是比它小一个阶的内存块的大小的两倍。内存块的大小是 2 次幂对齐，使地址计算变得简单。当一个较大的内存块被分割时，它被分成两个较小的内存块，这两个小内存块相互成为唯一的伙伴。一个分割内存块也只能与它唯一的伙伴块进行合并（合并成他们分割前的块）。

`struct global_mem`是用来描述物理内存的数据结构，保存了伙伴系统中一组空闲内存块的链表`free_lists`，每组链表中使用`list_head`链接所有同`order`的内存块。ChCore 提供了一些有用的函数和宏来操作`struct list_head`:

- `init_list_head(struct list_head * head)`：初始化列表头
- `list_add(struct list_head *new, struct list_head *head)`：向列表的头部添加新节点
- `list_del(struct list_head *entry)`：删除列表中的这个节点
- `list_entry(ptr, type, member)`：使用给定的`ptr`获取相应的对象，`member`是对应对象中`struct list_head`的变量名

练习 1

实现`kernel/mm/buddy.c`中的四个函数：`buddy_get_pages()`，`split_page()`，`buddy_free_pages()`，`merge_page()`。
请参考伙伴块索引等功能的辅助函数:`get_buddy_chunk()`。

在实验 2 中我们使用单元测试工具 minunit 对伙伴系统以及之后的页表映射做测试。`test_buddy()`用于测试伙伴系统是否正确实现。你可以使用以下命令在`tests/mm/buddy`目录下创建测试文件:

```
chcore$ make docker
chcore$ cd ./tests/mm/buddy
```

```
chcore$ cmake ./
chcore$ make
chcore$ ./test_buddy
```

如果通过了全部的测试, 会得到以下的结果：

```
1 tests, 2621578 assertions, 0 failures
```

如果在测试时候出现`failure`，请详细检查你所写的代码，考虑可能存在的边界情况，在调试的时候，可以适当添加`mu_check()`来辅助验证实现是否正确。另外，可以通过阅读`tests/README.md`进一步了解 minunit 的使用方法与说明。如果测试时候没有出现`failure`，那么恭喜你, 实验 2 的第一部分已经完成了。

20.3 第二部分：虚拟内存映射

在做实验 2 第二部分之前，需要先学习一下 AArch64 的地址翻译机制。本部分内容可以通过参阅《Arm 架构参考手册》[1]（链接）的 D5 节，了解更多的关于 AArch64 中和内存管理相关的知识。

20.3.1 AArch64 地址翻译

由于程序中的数据或指令的虚拟地址，无法直接被处理器用于访问物理内存，需要一套虚拟内存翻译为物理内存的机制。MMU 通过遍历内存中的页表，将程序中的虚拟地址翻译为物理地址。

20.3.2 内核与用户地址空间分离

为了保证进程间的隔离性，不同进程之间，以及用户态与内核态之间，所使用的页表是不同的，操作系统在进行上下文切换的时候会进行页表的切换。但是，大部分内存只由内核使用（在内核中保留了所有内存的映射，并且为了便于管理，内核往往固定了虚拟地址到物理地址的偏移），因此该部分的页表项很少需要更改。AArch64 体系结构提供了几个特性来有效地处理该需求。

在 ARM 中拥有两个页表的基地址寄存器：TTBR0_EL1 和 TTBR1_EL1。这两个寄存器所翻译的虚拟地址范围可以通过 TCR_EL1 进行配置。通常，操作系统会将 TTBR1_EL1 寄存器用于存储内核映射的页表，将 TTBR0_EL1 寄存器用于存储用户态程序的映射的页表。

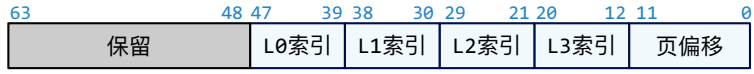

图 20.2: AArch64 中 4KB 页的虚拟地址解析方式

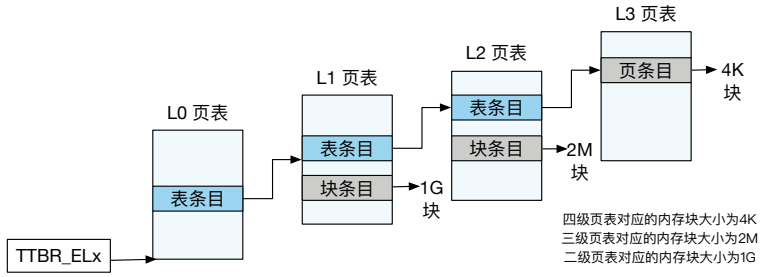

图 20.3: AArch64 页表的组织结构

问题 2

AArch64 采用了两个页表基地址寄存器，相较于 x86-64 架构中只有一个页表基地址寄存器，这样的好处是什么？请从性能与安全两个角度做简要的回答。

20.3.3 虚拟地址翻译与组成

对于 n 位物理地址空间中的虚拟地址（AArch64 支持的物理内存地址空间的大小为 48 位），前 64-n 位 [63:n] 必须全是 0 或 1，否则地址将触发异常错误，MMU 使用剩余 n 位进行页表的遍历。

如图 20.2所示，以四级页表为例：在四级页表中，虚拟地址大小为 48 位，48 位地址对于每个页表级别有 9 个比特位（即每一级页表有 512 个条目）用于索引，最后的 12 位（页偏移）用于选择 4KB 内的一个字节。虚拟地址的 [47:39] 位作为 L0 页表的索引，L0 页表每个表条目的范围为 512GB，并指向一个 L1 表。在 L1 表中也有 512 个条目，[38:30] 位被用作索引来选择一个条目，每个条目都指向一个 1GB 页或一个 L2 表。[29:21] 位用于索引 L2 表中的条目（512 个），每个条目指向一个 2MB 页或一个 L1 页表。在最后一级（L3）页表中，将 [20:12] 位索引到有 512 条目 L3 表中，每个条目指向一个 4KB 页。L3 页表中存储了物理页的**页帧号（Page Frame Number，PFN）**，将 PFN 和原本虚拟地址中的偏移量组合起来，就能够得到真正的物理地址。

20.3.4 内存属性

图 20.3展示了页表的组织结构，在 AArch64 中页表的项被称为**描述符**（**descriptor**），共有以下三种：

- **表描述符（table descriptor）**：包含下一级页表的地址 (`next_table_address`) 和相应的属性，[1:0] 位为 0x0b11。
- **块描述符（block descriptor）**：包含下一级页表的地址或 PFN 和相应的属性，[1:0] 位为 0b01。实验 2 中作为 L0 级页表中的项用于管理内核态页表；
- **页描述符（page descriptor）**：包含 PFN 和相应的属性，[1:0] 位为 0x0b11。实验 2 中用于管理用户态页表。

如图 20.4所示，块描述符和页描述符中指定了内存属性。AArch64 MMU 架构定义了块/页条目中每个区域的属性位的含义：

- UXN：置为 1 表示非特权程序无法执行（Unprivileged eXecute Never）。
- PXN：置为 1 表示特权程序无法执行（Privileged eXecute Never）。
- AF：访问标志。
- SH：可共享属性标志，见表 20.1。
- AP：访问权限，见表 20.2。
- NS：安全位，仅在 EL3 和安全世界的 EL1 中有效。
- Indx：MAIR_ELn 的索引，置为 0 表示强序设备内存，置为 4 表示正常内存。

页/块描述符和表描述符之间有一些区别，主要在于高位页表属性的不同:

NSTable：当 CPU 处在安全状态下发起内存访问请求，该标志位表示后续页表查询的安全状态。如果表标识符位于安全的物理空间，则 NSTable 为 0；位于非安全的内存区域，NSTable 为 1。特别的，如果 NSTable 为 1，则在后续查找时，页或块描述符上的 NS 位的值都将被忽略，所引用的块或页也位于非安全内存中。

APTable：该标志位限制了后续页表查询的访问权限。具体描述请见 20.3

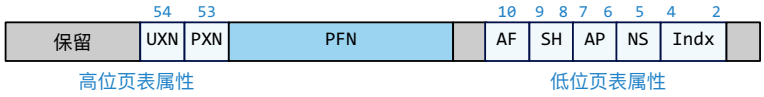

图 20.4: ARM 中页表属性

表 20.1: SH 位标识

SH0 位 [13:12]	可缓存属性
00	正常内存，内部不可缓存
01	正常内存，内部写回，写分配，可缓存
10	正常内存，内部写直达，可缓存
11	正常内存，内部写回，不写分配，可缓存

表 20.2: AP 位标识

AP	EL0	EL1,2,3
00	不可访问	可读可写
01	可读可写	可读可写
10	不可访问	只读
11	只读	只读

表 20.3: APTable 位标识

APTable[1:0]	描述
00	保留，对权限检查无影响
01	写操作在任何异常级别中均被禁止，并且忽视后续的页表中 ap 位的值
10	对内存的访问在 EL0 中被禁止，并且忽视后续页表中的权限设置
11	写操作在任何异常级别中均被禁止，读操作在 EL0 中被禁止

UXNTable：表示非特权不能执行, 如果该位被置上，UXN 位在所有后续级别查找中都被视为置上，而与之后页或者块描述符中的实际值无关

PXNTable：表示特权无法执行，如果该位被置上，对于所有后续级别的查找，PXN 位都被视为置上，而与页或者块描述符中设置的实际值无关

ChCore 运行在支持虚拟内存 ARM 核心中，一旦 CPU 进入页模式 (在`bootloader/arch64/init/init_c.c`中使能了该页表模式)，就无法直接使用物理地址。所有内存引用被解释为虚拟地址，并由 MMU 翻译，这意味着 C 代码的所有指针都是虚拟地址。然而 ChCore 内核经常需要将地址作为整数进行操作 (例如指针加减)，而不需要对指针进行解引用。例如在虚拟内存映射以及页表配置的时候，有时候需要使用虚拟地址，有时候需要使用物理地址。

为了更好地区分虚拟地址与物理地址，ChCore 源代码通过`typedef`区分了两种情况：`vaddr_t`表示虚拟地址，而`paddr_t`表示物理地址。这两种类型都是 64 位整数 (u64) 的别名，因此可以将其中一种类型转换为另一种类型而不会报错。但是不论是`paddr_t`还是 `vaddr_t`都不是指针的类型，因此不能够对这两个类型的变量做解引用。

这两个类型的区别是，将`vaddr_t`转换为指针类型后，可以通过解引用操作来读取该虚拟地址中的内容。然而，不能通过此方法读取`paddr_t`的内容。如果将`paddr_t` 强制转换为指针并取消引用它，硬件仍将其解释为虚拟地址，而非物理地址。因此访问的并不是该物理地址，而只是虚拟地址为这个值的内容。**实验的过程中，请特别注意这一点。**

问题 3

1. 请问在页表条目中填写的下一级页表的地址是物理地址还是虚拟地址?
2. 在 ChCore 中检索当前页表条目的时候，使用的页表基地址是虚拟地址还是物理地址?

实验中，有时只能通过物理地址读取或修改内存。例如，将映射添加到页表可能需要分配物理内存来存储页目录，然后将该页设置为也标的格式。但是，内核和用户态程序一样，都不能绕过虚拟内存转换，因此不能直接加载和存储数据到物理地址。ChCore 从虚拟地址 0xffffff00_00000000（对应物理地址 0）开始线性地映射所有物理内存。

为了将物理地址翻译成内核实际上可以读写的虚拟地址，内核必须向物理地址增加 0xffffff00_00000000 作为偏移量，以便在线性映射的区域中找到它对应的虚拟地址。反之，有时还需要根据虚拟地址查找物理地址，此时只需减去偏移量该值。`phys_to_virt(pa)` 宏和`virt_to_phys(va)`宏即通过加上或减去这个偏移量进行地址转换。该偏移量可以在`boot/image.h`和`kernel/common/mmu.h`中通过重新定义`KERNEL_VADDR`和`KBASE`进行修改。

问题 4

1. 如果我们有 4G 物理内存，管理内存需要多少空间开销? 这个开销是如何降低的?
2. 总结一下 x86-64 和 AArch64 地址翻译机制的区别，AArch64 MMU 架构设计的优点是什么?

20.3.5 页表管理

本部分实验需要实现页表管理机制，包括通过四级页表插入和删除虚拟到物理的映射，以及需要时创建页表页。

练习 2

在文件`kernel/mm/page_table`中，实现`map_range_in_pgtbl()`，`unmap_range_in_pgtbl()`和`query_in_pgtbl()` 。可以调用辅助函数：`set_pte_flags()`,`get_next_ptp()`,`flush_tlb()`。

和第一部分的测试一样，对于页表的映射也使用minunit单元测试工具。test_map_unmap_page()会测试页表相关操作，你可以使用以下脚本在tests/mm/page_table下创建测试文件:

```
make docker
cd ./tests/mm/page_table
cmake ./
make
./test_aarch64_page_table
```

如果测试通过将会得到一下的输出

```
1 tests, 82529 assertions, 0 failures
```

如果测试结束没有产生failure，那么恭喜你，实验 2 第二部分已经顺利的完成。如果产生了failure，可以参考第一部分调试的方式来完善代码。

20.4 第三部分：内核地址空间

ChCore 将虚拟地址空间分为两部分：用户态地址空间与内核态地址空间。内核始终保持对高地址空间的完全控制，低地址用户态地址空间将在实验 3 中实现。高地址与低地址的分界线由kernel/common/mmu.h中的KBASE宏定义。

为保证隔离性，ChCore 使用页表中的权限位来保证用户代码只访问用户态地址空间。否则可能会造成崩溃以及安全隐患，如用户数据覆盖内核数据、恶意的用户态进程修改内核数据等。用户态程序对内核态的任何内存都没有权限，而内核将能够读写这些内存。

问题 5

在 AArch64 MMU 架构中，使用了两个 TTBR 寄存器，ChCore 使用一个 TTBR 寄存器映射内核地址空间，另一个寄存器映射用户态的地址空间，那么是否还需要通过设置页表位的属性来隔离内核态和用户态的地址空间?

表 20.4: 内存映射

TTBR	虚拟地址	物理地址	用途
TTBR1_EL1	KBASE+512M~KBASE+4G	512M~4G	设备空间
	KBASE+256M~KBASE+512M	256M~512M	内核空洞
	KBASE~KBASE+256M	0~256M	内核空间
.....			
TTBR0_EL1	2G~4G	2G~4G	设备空间
	0~2G	0~2G	正常空间

20.4.1 映射内核地址空间

如表 20.4所示，启动时，ChCore 已经将虚拟地址（KBASE~KBASE + 256M）映射到物理地址（0~256M），在本实验将额外的 256M 虚拟地址（KBASE + 256M~KBASE + 512M）映射到物理地址（256M~512M）。KBASE + 512M 之上是为设备预留虚拟内存。

ChCore 使用块（2MB）来管理内核内存，回顾你在上一部分中学到的东西。每个块包含 1G / 2M 物理内存，因此 MMU 在将内核空间中的虚拟地址翻译为物理地址时，只需要遍历 L2/L3 页表 (1G 块 L2 页表，2M 块 L3 页表)。L0 页表地址存储在 TTBR1_EL1 寄存器中，L1 页表地址存储在 L0 页表条目中，类似地，L2 页表地址存储在 L1 页表条目中。

问题 6

1. ChCore 为什么要使用块条目组织内核内存? 哪些虚拟地址空间在 Boot 阶段必须映射，哪些虚拟地址空间可以在内核启动后延迟?
2. 为什么用户程序不能读写内核内存? 保护内核内存的具体机制是什么?

EL0 没有读取/写入/执行内核空间内存的权限，所以块条目 (pud / pmd) 中的属性位为:

- UXN = 1
- AF = 1
- SH = 3
- Indx = 4

- bits[1:0] = 1 (块条目)

你可以读取 TTBR1_EL1 寄存器获取第一级页表的基地址

```
1 unsigned long get_ttbr1()
2 {
3   unsigned long pgd;
4   __asm__ ("mrs %0,ttbr1_el1" : "=r"(pgd));
5   return pgd;
6 }
```

练习 3

完善kernel/mm/mm.c中的map_kernel_space()函数，实现对内核空间的映射，并且可以通过kernel_space_check()的检查。

如果你对 ChCore 内存管理有更多的兴趣，我们给出两个挑战，你可以在 ChCore 中实现以下的两个挑战，注意：无论是否完成挑战，都不会影响之后实验的完成。

挑战！

1. 以页粒度（4KB）映射内核空间。默认情况下（即实验 2 第二部分中），只有用户进程可以以页粒度中映射虚拟地址，因此需要修改函数set_pte_flags()，来支持页粒度内核空间映射。
2. 支持以块粒度（2MB）来管理用户态低空空间，修改page_table.c中的map_range_in_pgtbl()以区分需要映射的页的大小。

当你完成所有的任务后，你可以在目录中键入make grade进行测试。如果你通过了所有的考试，你可以看到你的成绩报告如下:

```
running chcore: (0.2s)
  buddy: OK
  page table: OK
  kernel space check: OK
Score: 100/100
```

参考文献

[1] ARM. Arm architecture reference manual. https://static.docs.arm.com/ddi0487/fa/DDI0487F_a_armv8_arm.pdf?_ga=2.181644388.2107974726.1583153879-1487747685.1581514464, 2020.

实验 **2**：扫码反馈