Linux 内存管理

--伙伴系统初始化

一、伙伴系统管理的内存

如果用一句话来描述伙伴系统管理的内存的话，那就是在系统初始化阶段通过__free_pages 函数释放的内存就是伙伴系统管理的内存。在伙伴系统初始化之前，由 memblock 来完成内存管理的工作。但是在 memblock 向伙伴系统移交内存的时候，并不是把全部内存都移交过去，而是把不在 reserved 区域且不带 MEMBLOCK_NOMAP 标记的内存释放到伙伴系统。除了 memblock 移交的内存，系统初始化阶段还有其他通过__free_pages 接口释放的内存都会被添加到伙伴系统，这些内存包括__init_begin 到__init_end 之间的代码段和数据段，以及带有 reusable 属性的 reserved-memory 的子节点定义的内存。

1.1 内存总大小

Linux 内核看到的内存总大小并不一定是 DDR 的总大小，而是从设备树的 memory 节点的 reg 参数中解析出来的。由于 hypervisor 的存在，会修改该节点，挖掉自己使用的一些内存，之后在 bootloader 中进行设备树的叠加。这样 linux 看到的内存起始是被 hyp 挖掉之后的内存。

1.2 设备树 reserved-memory 节点

在设备树的 reserved-memory 节点定义了很多子节点，这些是用来给 modem、adsp 等子系统使用的内存，这些内存会在 memblock 初始化的时候被添加到 memblock 的 reserved 区域中。

1.2.1no-map 属性

子节点如果带有 no-map 属性的话，这段内存会从 memblock 中移除，在 kernel 中不会创建页表，也就意味着无法被 CPU 访问。其他不带有该属性的节点定义的内存都会被添加到 memblock.reserved 区域。

1.2.2reusable 属性

子节点如果带有 reusable 属性的话，这段内存虽然在 memblock 的 reserved 区域，但是依然会被释放到伙伴系统，也就意味着不仅仅是特定的子系统可以使用这段内存。在启动阶段的打印中可以看到类似如下的打印：

[0.000000] Reserved memory: created CMA memory pool at 0x00000000ff400000, size 8 MiB

[0.000000] OF: reserved mem: initialized node sdsp_region, compatible id shared-dma-pool

如果没有 reusable 属性，那么这段内存就不会被释放到伙伴系统。同样可以看到类似如下的打印：

[0.000000] Reserved memory: created DMA memory pool at 0x00000000df700000, size 8 MiB

[0.000000] OF: reserved mem: initialized node memshare_region, compatible id shared-dma-pool

1.3 memblock_reserve 指定的内存

在系统启动阶段以及启动之后，有些内存需要一直占用，这种情况可以使用 memblock_reserve 函数，把一段内存添加到 memblock 的 reserved 区域，这段内存就不会被伙伴系统管理。以下是不完全统计的调用 memblock_reserve 函数指定 reserved 的内存：

1.3.1 内核的代码段和数据段

memblock_reserve(__pa_symbol(_text), _end - _text);但是在系统初始化完成之后会释放__init_begin 到__init_end 之间的内存，也就是__init、__exit 等标记的代码段和__initdata、__exit_data 等标记的数据段。参考启动阶段 log：xxK kernel code, xxK rwdata, xxK rodata, xxK init, xxK bss。

1.3.2 设备树占用的内存

memblock_reserve(dt_phys, size);kernel 的 dtb 和 userspace 的 dtbo 文件头部中保存有设备树的大小，叠加之后的设备树总大小会被 reserve。

1.4memblock_alloc 申请没释放的内存

在启动阶段通过 memblock 申请内存的函数是 memblock_alloc，这个函数会把一段申请得到的内存添加到 memblock 的 reserve 的区域。与之对应的 memblock_free 函数会把内存从 reserved 区域去掉。如果调用 memblock_alloc 而没有调用 memblock_free 的话，那么这段内存就会一直在 reserved 区域。

1.4.1cmdline 占用的内存

在启动阶段 linux 从设备树中解析 chosen 节点获取 bootargd 参数保存到字符数组 boot_command_line[COMMAND_LINE_SIZE]中，然后会把字符串拷贝到 saved_command_line 字符型指针中。因为在启动阶段是无法预知 cmdline 所占用内存的大小的，所以先定义 COMMAND_LINE_SIZE 为 4096（arm64 平台），用 4096 长度的数组先临时保存一下，然后把实际的字符串拷贝到 saved_command_line，这里占用的内存就是 cmdline 字符串实际占用的内存，不会造成浪费。最后会把 boot_command_line 内存释放掉。Saved_command_line 这段内存也是 reserved 内存，不会释放到伙伴系统中。

1.4.2per-cpu 变量区域

```
/*
* Always reserve area for module percpu variables.  That's
* what the legacy allocator did.
*/
rc = pcpu_embed_first_chunk(PERCPU_MODULE_RESERVE,
```

PERCPU_DYNAMIC_RESERVE, PAGE_SIZE, NULL,

pcpu_dfl_fc_alloc, pcpu_dfl_fc_free);

1.4.3 设备树展开的 device_node 参数

/* Allocate memory for the expanded device tree */

mem = dt_alloc(size + 4, __alignof__(struct device_node));

1.4.4hash 表

if (flags & HASH_EARLY) {

if (flags & HASH_ZERO)

table = memblock_alloc(size, SMP_CACHE_BYTES);

else

table = memblock_alloc_raw(size, SMP_CACHE_BYTES);

1.4.5log_buf 内存

内核日志保存在字符指针 log_buf 处，默认的情况下 log_buf 指向数组 __log_buf[__LOG_BUF_LEN]，数组的长度由配置参数 CONFIG_LOG_BUF_SHIFT 决定：log_buf_len=1<<CONFIG_LOG_BUF_SHIFT。由于全局数组是在 kernel 的数据段，所以这部分内存起始已经在 1.2.1 中被 reserve 过了。但是在 cmdline 中可以通过 log_buf_len 参数重新定义内核日志的环形缓冲区的大小，当设置的新的缓冲区的大小大于默认值的时候，会为重新申请需要的内存，并且把 log_buf 指向新申请的内存，同时拷贝__log_buf 数组中的数据到 log_buf，这时候就会多出来内核日志缓冲区 reserved 内存，这部分内存也不会添加到伙伴系统中。

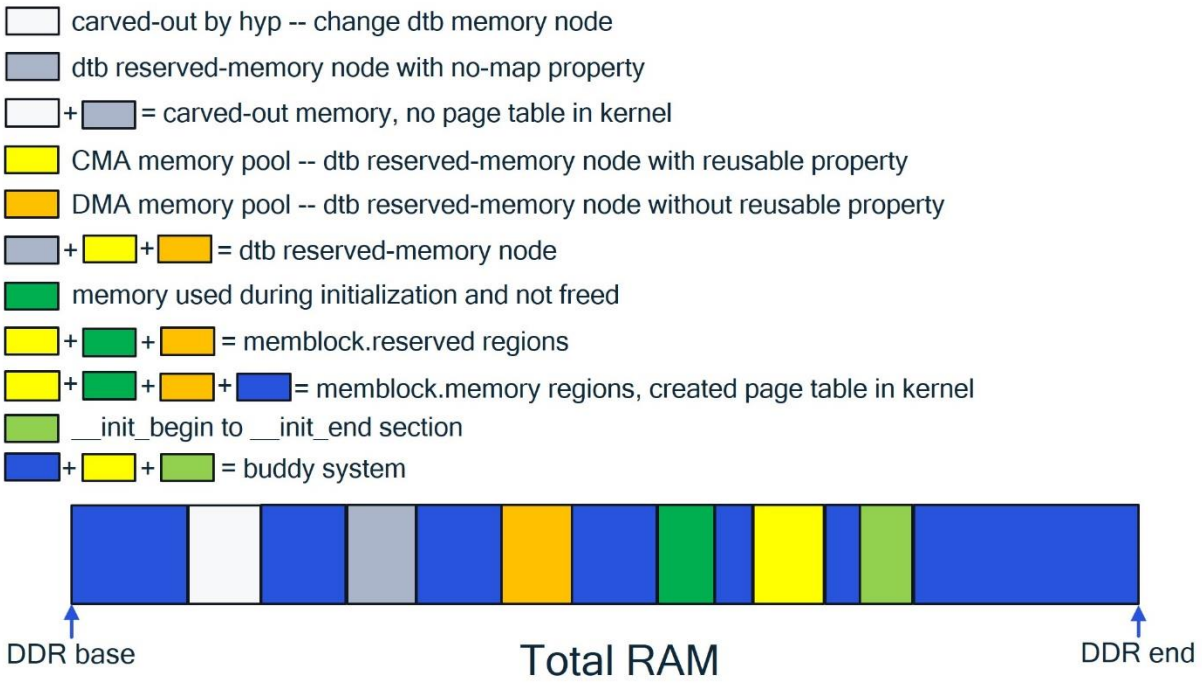
1.4.6struct page 占用的内存

每个物理页框都有对应的 struct page 来描述，所有的 struct page 保存在 vmemmap 起始的数组中。这段内存是不会释放到伙伴系统的。

1.5 小结

在系统初始化阶段，每次调用__free_pages 之后会更新_totalram_pages 的值，这个变量记录的就是伙伴系统中 page 的数量。当通过 cat /proc/meminfo 命令来查看内存信息的时候，MemTotal 的值其实就是通过读_totalram_pages 的值计算得来的。

下图展示了 memblock 和伙伴系统之间的关系：



二、伙伴系统框架

伙伴系统把物理内存按照页个数分为 11(MAX_ORDER)个组，分别对应 11 种大小不同的连续内存块，每组的编号 0~10 叫阶次(order)，每组中的内存块数量都相等，为 2 的 order 幂次个物理页。也就是说系统中就存在 2^0*4KB~2^10*4KB 大小不同的内存块。对应的有 11 个链表 zone->free_area[order]来管理相同大小的内存区域。

伙伴系统示意图

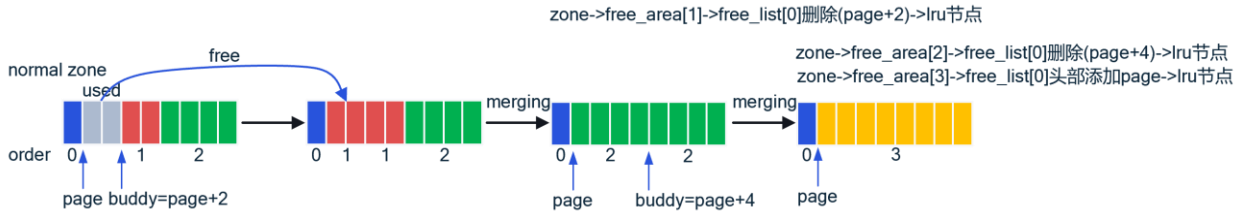
分配内存的时候就根据所申请的内存阶次（大小），从对应的 free_area 链表中获取即可，如果对应的阶次空闲内存不足的话，会把更高一阶 free_area 链表中的一块内存进行，添加到对应的 free_area 链表中。同样的，在释放内存的时候，会检查相邻的内存块是否可以合并成高阶的内存块，这样使得系统中内存尽可能是大块的内存。

2.1 伙伴系统合并

分配内存的时候是从伙伴系统中取走内存，相反的释放内存的时候就是把内存放回伙伴系统的过程。在释放内存到伙伴系统的时候会检测这块内存是否有“伙伴”，如果有就会与伙伴合并成更高一阶的内存，然后会进一步检测合并之后的内存块是否有“伙伴”，进一步合并，直到没有伙伴为止。与待释放内存构成“伙伴”的条件是：1. 在伙伴系统

中，也就是空闲内存，2.相邻且阶次相同的物理内存块，3.同一个 zone。同时满足以上条件的内存就是伙伴。

现在看一下各个条件是怎么检测的。对每个物理页框都有对应的 struct page 来描述，所有的 struct page 保存在 vmemmap 起始的数组中，根据每个物理页框的编号 pfn 就可以找到对应的 struct page 的地址 page=vmemmap+pfn。如果内存已经被申请，也就是不在伙伴系统中了，page->page_type 的 PG_buddy 标志位会置 1，空闲内存的 page->page_type 的 PG_buddy 标志位是 0，而且空闲内存的 page->private 保存的是 order 值，ZONE 的编号保存在 page->flags 中。



伙伴系统合并

2.2gfp_mask

在调用 alloc_pages 函数从伙伴系统中申请内存的时候会传入 gfp_mask 参数，这个参数是一个 32bit 的变量，其中每一位都有特定的意义，bit0~bit3 是 zone 类型，bit3~bit4 是迁移类型。

2.2.1zone 类型

伙伴系统中 zone 的类型按 index 从低到高依次为 ZONE_DMA，ZONE_DMA 32，ZONE_NORMAL，ZONE_HIGHMEM，ZONE_MOVABLE，ZONE_DEVICE。系统中并不是所有的 zone 都支持的，其中 ZONE_NORMAL 和 ZONE_MOVABLE 是基本的，而对于 ARM64，没有 ZONE_HIGHMEM 和 ZONE_DMA。ARM64 如果想用单独的 zone 管理 DMA 内存的话可以打开 CONFIG_ZONE_DMA32 配置选择从而使能 ZONE_DMA32。常见的系统使用的 zone 是 ZONE_NORMAL+ ZONE_MOVABLE 和 ZONE_DMA 32+ZONE_NORMAL+ ZONE_MOVABLE。系统初始化阶段会把 zone 添加到 contig_page_data->node_zonelists->_zonerefs[MAX_ZONES_PER_ZONELIST+1]（UMA 架构）数组中.

申请内存的时候首先会解析 gfp_mask，得到本次申请内存可以使用的 zone 的最大的 index，然后从 zonelist 数组中索引小于 index 的 zone 里获取内存。 如果以最基本的 ZONE_NORMAL+ ZONE_MOVABLE 系统为例的话，只有 gfp_mask 包含了 __GFP_MOVABLE 的时候，会先从 normal zone 获取内存，没有空闲内存的话会继续尝试从 movable zone 获取内存。而其他情况下都是只从 normal zone 获取内存。

2.2.2 迁移类型

关于迁移类型的概念是针对内存页的，在系统初始化阶段就会把不太的 page 设置为不同的迁移类型，并分别管理在 zone->free_area[order]->free_list[migrate]数组中。在 gfp_mask 中也有对应的标志位 bit3~bit4 来指定本次申请的是哪种迁移类型的内存，具体有 MIGRATE_UNMOVABLE，MIGRATE_MOVABLE，MIGRATE_RECLAIMABLE，MIGRATE_CMA。每一种迁移类型还有 fallback 数组，会依次从这几种迁移类型的内存里获取内存。各迁移类型的 fallback 如下：

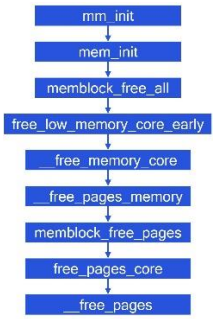
```
2224 static int fallbacks[MIGRATE_TYPES][4] = {
2225   [MIGRATE_UNMOVABLE]     = { MIGRATE_RECLAIMABLE, MIGRATE_MOVABLE,     MIGRATE_TYPES },
2226   [MIGRATE_MOVABLE]       = { MIGRATE_RECLAIMABLE, MIGRATE_UNMOVABLE, MIGRATE_TYPES },
2227   [MIGRATE_RECLAIMABLE] = { MIGRATE_UNMOVABLE,     MIGRATE_MOVABLE,     MIGRATE_TYPES },
2228 #ifdef CONFIG_CMA
2229   [MIGRATE_CMA]             = { MIGRATE_TYPES }, /* Never used */
2230 #endif
2231 #ifdef CONFIG_MEMORY_ISOLATION
2232   [MIGRATE_ISOLATE]       = { MIGRATE_TYPES }, /* Never used */
2233 #endif
2234 };
```

2.3zone 的初始化

2.3.1normal zone 的初始化

在 memblock 初始化完成之后就会对 zone 进行初始化，具体的操作就是把 memblock 管理的内存移交到 zone 中。对于 UMA 架构(CONFIG_NUMA 没定义)只使能了 normal zone 和 movable zone 的系统，memblock 管理的内存会全部释放到 normal zone 中。而 movable zone 的初始化是由内存热插拔驱动来实现的。

normal zone 初始化



释放内存到伙伴系统的 normal zone

2.3.2movable zone 初始化

在 movable zone 初始化之前会有一个隐藏的操作，就是在 memblock 扫描完 moemory 节点把内存添加到 memblock 中之后还会解析 mem-offline 节点，解析出其中定义的 offline-sizes 内存。以下图所以为例，DDR 起始地址为 0x80000000，那么这个属性的含义是当 DDR size<3GB 时，offline_size=0；当 3GB<=DDR size<5GB 时，offline_size=1GB；5GB<=DDR size<9GB 时，offline_size=2GB；DDR size>9GB 时，offline_size=5GB。然后从 memblock 中管理的最大地址开始移除 offline_size 大小的内存。

```
mem-offline {
        compatible = "qcom,mem-offline";
        offline-sizes = <0x1 0x40000000 0x0 0x40000000>,
                        <0x1 0xc0000000 0x0 0x80000000>,
                        <0x2 0xc0000000 0x1 0x40000000>;
        granule = <512>;
        mboxes = <&qmp_aop 0>;
};
```

mem-offline 节点

解析 mem-offline 节点的 offline-sizes 属性

后续在内存热插拔的驱动中会重新 online 这些内存页，重新添加回 memblock 中，同时释放到 movable zone 中。到这里，伙伴系统的初始化工作已经完成，之后就可以愉快的申请内存了。