# Sunxi ION 使用文档

文档版本号：V1.0

发布日期：2017.09.01

**商标声明**


、全志和其他全志商标均为珠海全志科技股份有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

**注意**

您购买的产品、服务或特性等应受全志公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，全志公司对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保

# 前 言

## 概述

介绍 linux/android 连续内存使用方法，方便相关驱动和应用开发人员.

## 产品版本

| 产品名称 | 产品版本 |
| --- | --- |
| | |
| | |
| | |

## 读者对象

本文档（本指南）主要适用于以下工程师：

相关驱动和应用开发人员

## 修订记录

| 版本号 | 修订日期 | 修订内容 |
| --- | --- | --- |
| V0.1 | 2013-06-25 | 建立初始版本 |
| V0.2 | 2013-07-31 | 修改内核态映射函数 |
| V0.3 | 2013-09-09 | 增加打印 ion 内存分配状态章节 |
| V1.0 | 2016-01-20 | 内核版本升级到 linux-3.10，修改不一致的描述 |
| | | |

# 目 录

# 1. 概述

## 1.1. 编写目的

介绍 linux/android 连续内存使用方法, 方便相关驱动和应用开发人员.

## 1.2. 适用范围

适用于 linux3.4/linux-3.10/linux-4.4 平台的所有芯片.

# 2. 模块介绍

## 2.1. 相关术语

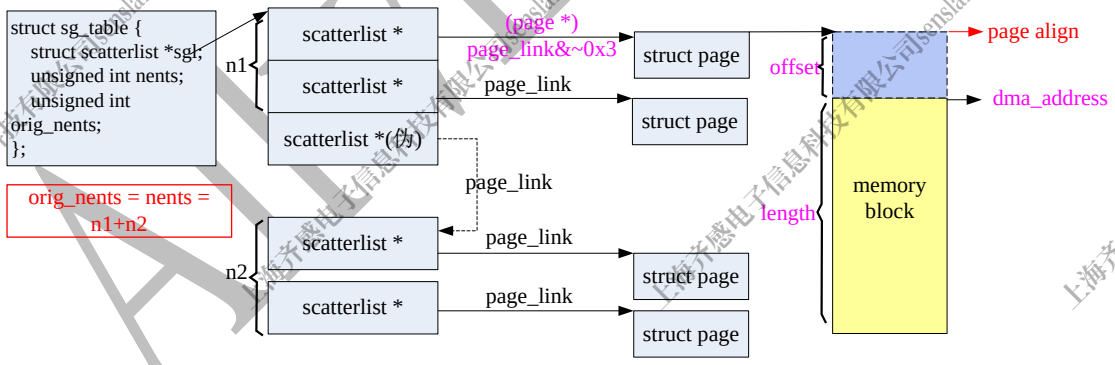
### 2.1.1. sg_table/scatterlist

scatterlist: 描述一块连续物理内存.

```
struct scatterlist {
        unsigned long page_link;
        unsigned int    offset;
        unsigned int    length;
        dma_addr_t      dma_address;
};
```

sg_table: scatterlist 数组.

```
struct sg_table {
        struct scatterlist *sgl;
        unsigned int nents;
        unsigned int orig_nents;
};
```




### 2.1.2. CMA

Contiguous Memory Allocator. 连续内存分配器.

linux 内核分配物理内存使用的伙伴分配算法，每次最大只能分配 4M，无论 kmalloc, dma_alloc_coherent 均不能分配超过 4M.

为了解决驱动对大块物理内存的需求，linux3.5 在内存管理中，引入的连续物理内存分配机制，

即 CMA.

### 2.1.3. ION

ION 是 google 在 Android4.0 引入的内存管理框架.

它不是内存分配算法，而是一个框架，为应用层，内核层, dma 硬件使用，提供了接口.

对于应用层，它提供了字符设备命令;

对于内核层，它提供了将若干块连续物理内存，映射成连续内核虚拟空间的机制;

对于 dma 设备(泛指需要使用连续物理内存的外设，比如 de, ve)，它提供了将若干块连续物理内存，组合成散列表(sg_table)的机制;

## 2.2. 模块配置

CONFIG_ION=y

CONFIG_ION_SUNXI=y

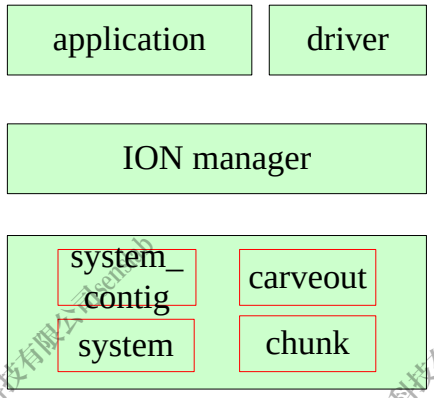
## 2.3. 源码结构介绍

linux-3.4\drivers\gpu\ion

linux-3.10\drivers\staging\android\ion

# 3. ION 框架解析

## 3.1. 框架层次图




应用程序和驱动通过 ion 管理器，调用底层堆的接口，进行内存分配，释放，映射给用户空间，映射给内核，映射给 dma 访问；

## 3.2. 关键数据结构

### 3.2.1. ion_device

ion 设备，一个平台只能创建一个，属于 misc 设备，sunxi_ion_probe 中调用 ion_device_create 创建.

管理不同应用程序创建的 ion_client，应用程序通过 ION_IOC_ALLOC 创建的 ion_buffer，驱动直接调用 ion_buffer_create 创建的 ion_buffer.

```
struct ion_device {
        struct miscdevice dev;
        struct rb_root buffers;
        struct mutex buffer_lock;
        struct rw_semaphore lock;
        struct plist_head heaps;
        long (*custom_ioctl) (struct ion_client *client,);
        struct rb_root clients;
        struct dentry *debug_root;
};
```

dev: miscdevice 设备成员；

buffers: 应用层和驱动层通过该 ion_device 创建的所有 ion_buffer 链表；

heaps: 该 ion_device 支持的所有堆，在调用 ion_device_create 时指定；

custom_ioctl: 用户指定的 ioctl 函数，当应用层调用 ION_IOC_CUSTOM 时，ion_ioctl 将具体操作委托给 custom_ioctl；

clients: 注册在该设备下的所有 ion_client 链表. ion_client 可以由应用层打开 misc 设备时, ion_open 中创建; 也可以由驱动层直接调用 ion_client_create 创建;

### 3.2.2. ion_heap

ion 堆, 具体内存的分配, 释放, 映射由它完成.

不同类型的内存, 其分配函数, 映射操作也不同, 因此 ion 用堆来管理不同类型的内存操作.

```
struct ion_heap {
        struct plist_node node;
        struct ion_device *dev;
        enum ion_heap_type type;
        struct ion_heap_ops *ops;
        unsigned long flags;
        unsigned int id;
        const char *name;
        struct shrinker shrinker;
        struct list_head free_list;
        size_t free_list_size;
        struct rt_mutex lock;
        wait_queue_head_t waitqueue;
        struct task_struct *task;
        int (*debug_show)(struct ion_heap *heap, struct seq_file *, void *);
};
```

有以下几种类型堆:

ION_HEAP_TYPE_SYSTEM: 系统非物理连续类型堆, 比如上层要求分配 10M 内存, 实际会得到 4M + 4M + 2M 共一个内存块, 组成散列数组(sg_table);

这个散列数组可直接给 dma 使用, 用户态和内核态 cpu 访问时, 需要通过 map_user, map_kernel 映射; 映射后的是连续虚拟空间;

ION_HEAP_TYPE_SYSTEM_CONTIG: 系统物理连续堆, 分配由 kmalloc 完成, 一次最多不能分配超过 4M(linux 系统限制);

ION_HEAP_TYPE_CARVEOUT: 用户预留一块连续大物理内存(预留区不能在 bank 内, 不能用 memblock_reserve 预留, 可用 memblock_remove 预留), 通过 gen_pool_alloc 进行分配, 一次能分配的大小不受系统 4M 限制, 由预留量和使用情况决定;

这种堆能满足 de/ve 对连续大块(超过 4M)物理内存的需求; 但缺点是预留后, 系统不能使用, 造成浪费;

ION_HEAP_TYPE_CHUNK: 和 carveout 类似, 先预留一块大物理内存, 再用 gen_pool_alloc 进行分配. 不同是, 只能按指定单元大小(chunk_size)来分配, 比如要求分配 10M 内存, 指定单元大小为 4M, 则最终得到的是 3 块内存区, 每块大小 4M, 两块之间可能不连续;

ION_HEAP_TYPE_CUSTOM: 作为与用户自定义的堆的分界线, 不实际使用; 比如 sunxi 新增的 ION_HEAP_TYPE_SUNXI 就定义在它之后;

```
enum ion_heap_type {
        ION_HEAP_TYPE_SYSTEM,
        ION_HEAP_TYPE_SYSTEM_CONTIG,
        ION_HEAP_TYPE_CARVEOUT,
        ION_HEAP_TYPE_CHUNK,
        ION_HEAP_TYPE_CUSTOM,
        ION_NUM_HEAPS = 16,
};
```

### 3.2.3. ion_client

a process/hw block local address space.

进程调用 open("ion", )打开 ion 设备时, 就创建一个 ion_client, 以后分配的内存由 ion_client 进行管理.

```
struct ion_client {
        struct rb_node node;
        struct ion_device *dev;
        struct rb_root handles;
        struct mutex lock;
        const char *name;
        struct task_struct *task;
        pid_t pid;
        struct dentry *debug_root;
};
```

node: 挂在 ion_device->clients 链表上的节点;

dev: 回指向所属的 ion_device;

handles: 包含的所有 ion_handle 链表头;

task: 创建此 ion_client 的当前进程;

### 3.2.4. ion_handle

访问 ion_buffer 的句柄.

一个 ion_buffer 可能被不同 ion_client 共享, ion_client 通过 ion_handle 来访问 ion_buffer.

```
struct ion_handle {
        struct kref ref;
        struct ion_client *client;
        struct ion_buffer *buffer;
        struct rb_node node;
        unsigned int kmap_cnt;
};
```

client: 回指向所属的 ion_client;

buffer: 关联的 ion_buffer;

node: ion_client->handles 的节点;

kmap_cnt: 该 handle map 到 kernel 的次数. 实际是对 ion_buffer->kmap_cnt 的封装;

ion_client 为什么不直接访问 ion_buffer, 而要通过 ion_handle?

因为 ion_buffer 作为底层资源, 可被其他 client 共享,

ion_handle 作为 client 和 buffer 的中间层, 起到了缓冲作用.

### 3.2.5. ion_buffer

ion 分配的 buffer 的元数据.

ion_device 分配的所有 buffer, 记录在 ion_device->buffers 链上.

```
struct ion_buffer {
        struct kref ref;
        union {
                struct rb_node node;
                struct list_head list;
        };
        struct ion_device *dev;
        struct ion_heap *heap;
        unsigned long flags;
        size_t size;
        union {
                void *priv_virt;
                ion_phys_addr_t priv_phys;
        };
```

```
    struct mutex lock;
    int kmap_cnt;
    void *vaddr;
    int dmap_cnt;
    struct sg_table *sg_table;
    unsigned long *dirty;
    struct list_head vmas;
    /* used to track orphaned buffers */
    int handle_count;
    char task_comm[TASK_COMM_LEN];
    pid_t pid;
};
```

ref: buffer 引用计数;

node: ion_device->buffers 链表节点;

dev: 回指向所属的 ion_device;

heap: 回指向所属的 ion_heap;

flags: 用户传入, ION_FLAG_CACHED 表示用户希望 buffer 被映射成 cached 的(映射到用户或内核空间时);

size: buffer 大小, 字节;

priv_virt: 映射到内核时的虚拟地址;

priv_phys: 分配的 buffer 的物理地址; buffer 必须物理地址连续, 比如 carveout 类型堆分配;

kmap_cnt: 映射给内核的次数. 第一次时, 才真正调用 heap->ops->map_kernel, 此后只增加 kmap_cnt 计数;

vaddr: 映射给内核的虚拟地址;

dmap_cnt: dma 映射次数;

sg_table: dma 映射得到的散列表;

dirty: 对于自动刷 cache 的用户态映射, 为 buffer 的每个 page 分配的位图;

vmas: 对于自动刷 cache 的用户态映射, 记录映射的区域链表, 每个区域占一个 page;

handle_count: 对于共享 buffer 情形, 记录有多少个 ion_handle 关联到了该 buffer;

### 3.2.6. ion_heap_ops

ion 堆操作函数.

allocate: 分配内存; free: 释放内存;

phys: 得到物理内存大小和长度(仅对连续物理内存);

map_dma: 映射给 dma 用，返回散列数组;

unmap_dma: 解除 dma 映射;

map_kernel: 映射给 kernel，得到内核态连续虚拟地址;

unmap_kernel: 解除 kernel 映射;

map_user: 映射给用户空间，得到用户态连续虚拟地址;

为什么没有 unmap_user? 用户空间通过 unmap 会自动释放虚拟空间，不用内核态做什么;

```
struct ion_heap_ops {
    int (*allocate) (struct ion_heap *heap,
                struct ion_buffer *buffer, unsigned long len,
                unsigned long align, unsigned long flags);
    void (*free) (struct ion_buffer *buffer);
    int (*phys) (struct ion_heap *heap, struct ion_buffer *buffer,
                ion_phys_addr_t *addr, size_t *len);
    struct sg_table *(*map_dma) (struct ion_heap *heap,
                        struct ion_buffer *buffer);
    void (*unmap_dma) (struct ion_heap *heap, struct ion_buffer *buffer);
    void * (*map_kernel) (struct ion_heap *heap, struct ion_buffer *buffer);
    void (*unmap_kernel) (struct ion_heap *heap, struct ion_buffer *buffer);
    int (*map_user) (struct ion_heap *mapper, struct ion_buffer *buffer,
                struct vm_area_struct *vma);
};
```

### 3.2.7. ion_page_pool

描述一个内存池，包含指定大小(1<<order 个 page)的连续物理内存块.

ION_HEAP_TYPE_SYSTEM 堆用它来提高分配/释放效率.

当用户请求从 ION_HEAP_TYPE_SYSTEM 堆中分配 uncached 的内存时，就会从 ion_page_pool 中分配， 否则用 alloc_pages 来分配.

```
struct ion_page_pool {
    int high_count;
    int low_count;
    struct list_head high_items;
    struct list_head low_items;
    struct mutex mutex;
    gfp_t gfp_mask;
```

```
        unsigned int order;
        struct plist_node list;
    };
```

high_count: 处于 HIGHMEM 区的内存块个数;

low_count: 处于 LOWMEM 区的内存块个数;

high_items: HIGHMEM 区的内存块组成的链表;

low_items: LOWMEM 区的内存块组成的链表;

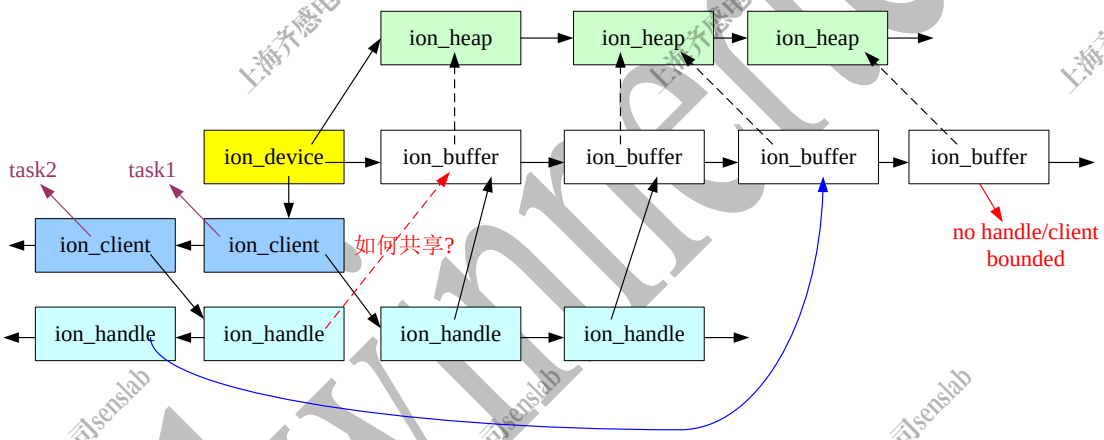
gfp_mask: 池为空时，调用 alloc_pages 使用该 flags 从系统分配内存;

order: 内存块的大小;

list: 系统全局 pools 链上的节点，按 order 大小作为优先级排序.

## 3.3. 数据结构关系

下图是 linux-3.4 ion 的数据结构关系图:




系统全局只能创建一个 ion_device，设备名为”ion”;

当不同进程(task)通过 open(“ion”,) 打开 ion 设备时，创建对应的 ion_client;

ion_client 通过 ion_handle 来管理 ion_buffer;

当应用层调用 ION_IOC_ALLOC 时，创建 ion_buffer 和 ion_handle，并建立 ion_handle 与 ion_buffer, ion_handle 与 ion_client 的对应关系;

ion_buffer 是元数据，从 ion_heap 里面分配;

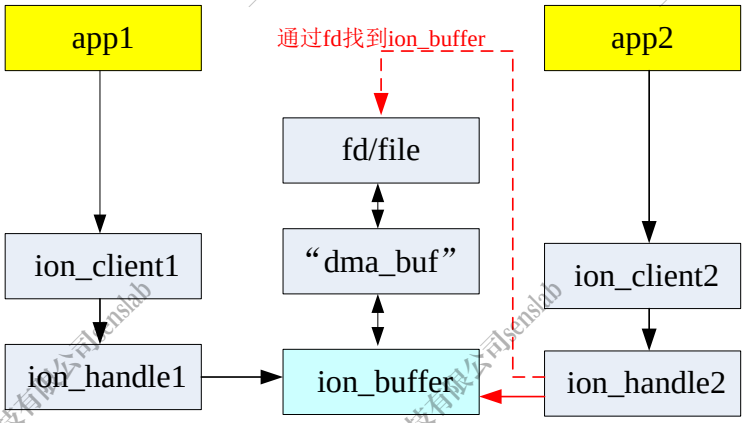
ion_buffer 可以由应用层通过 ION_IOC_ALLOC 分配，也可以内核层直接调用 ion_buffer_create 分配；前一种情况下, ion_buffer 与 ion_handle, ion_client 绑定；后一种情况, ion_buffer 没有绑定，如上图最右的 buffer;

同一 ion_client 的 ion_handle 中，不会有多个 ion_handle 对应一个 ion_buffer;

不同的 ion_client 可能共享同一个 ion_buffer，这种情况下，可能有多个 ion_handle 对应一个 buffer，如上图最左的 ion_buffer.

## 3.4. ion_buffer 的共享

假设应用程序 1 向 ion 申请了 buffer，应用程序 2 如何共享呢?




应用程序 1 调用 ION_IOC_MAP/ION_IOC_SHARE，将 ion_buffer 通过 dma_buf 导出，创建 fd 与之关联;

应用程序 2 得到 fd 号，调用 ION_IOC_IMPORT，通过 fd 找到 dma_buf，再找到 ion_buffer，并创建 ion_handle 与之关联;

这样，程序 1 和程序 2 就共享了 ion_buffer;

ION_IOC_MAP/ION_IOC_SHARE 对应操作主要是 ion_share_dma_buf_fd 函数:

```
struct dma_buf *ion_share_dma_buf(struct ion_client *client,
                                  struct ion_handle *handle)
{
    struct ion_buffer *buffer;
    struct dma_buf *dmabuf;
    bool valid_handle;

    mutex_lock(&client->lock);
    valid_handle = ion_handle_validate(client, handle);
    mutex_unlock(&client->lock);
    if (!valid_handle) {
        WARN(1, "%s: invalid handle passed to share.\n", __func__);
        return ERR_PTR(-EINVAL);
    }

    buffer = handle->buffer;
```

```
        ion_buffer_get(buffer);  增加 buffer 计数
        dmabuf = dma_buf_export(buffer, &dma_buf_ops, buffer->size, O_RDWR);
        if (IS_ERR(dmabuf)) {  创建 dmabuf，与 ion_buffer 关联
            ion_buffer_put(buffer);
            return dmabuf;
        }

        return dmabuf;
    }

    int ion_share_dma_buf_fd(struct ion_client *client, struct ion_handle *handle)
    {
        struct dma_buf *dmabuf;
        int fd;

        dmabuf = ion_share_dma_buf(client, handle);
        if (IS_ERR(dmabuf))
            return PTR_ERR(dmabuf);

        fd = dma_buf_fd(dmabuf, O_CLOEXEC);  创建 fd，与 dmabuf 关联
        if (fd < 0)
            dma_buf_put(dmabuf);

        return fd;
    }
```

增加的计数在哪里还原?

当用户调用 close(fd)时，上层会通过 dma_buf_release 回调 dmabuf->ops->release，即 ion_dma_buf_release，后者调用 ion_buffer_put 还原引用计数.

ION_IOC_IMPORT 主要操作是 ion_import_dma_buf:

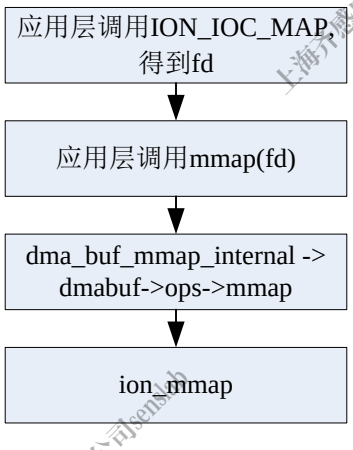
```
    struct ion_handle *ion_import_dma_buf(struct ion_client *client, int fd)
    {
        struct dma_buf *dmabuf;
```

```
    struct ion_buffer *buffer;
    struct ion_handle *handle;

    dmabuf = dma_buf_get(fd);  由 fd 得到 dmabuf
    if (IS_ERR_OR_NULL(dmabuf))
        return ERR_PTR(PTR_ERR(dmabuf));
    /* if this memory came from ion */

    if (dmabuf->ops != &dma_buf_ops) {
        pr_err("%s: can not import dmabuf from another exporter\n",
                __func__);
        dma_buf_put(dmabuf);
        return ERR_PTR(-EINVAL);
    }
    buffer = dmabuf->priv;  由 dmabuf 得到 ion_buffer

    mutex_lock(&client->lock);
    /* if a handle exists for this buffer just take a reference to it */
    handle = ion_handle_lookup(client, buffer);
    if (!IS_ERR_OR_NULL(handle)) {
        ion_handle_get(handle);
        goto end;
    }
    handle = ion_handle_create(client, buffer);  创建 ion_handle，与 buffer 关联
    if (IS_ERR_OR_NULL(handle))
        goto end;
    ion_handle_add(client, handle);  添加 ion_handle 到 ion_client(client2)
end:
    mutex_unlock(&client->lock);
    dma_buf_put(dmabuf);
    return handle;
}
```

## 3.5. ion buffer 映射给用户空间的 cache 管理

应用层通过 ION_IOC_MAP 获得 ion_buffer 对应的 dmabuf fd，再调用 mmap(fd) 将 ion_buffer

映射到用户空间.




到 ion_mmap 函数时，分三种情况处理，自动刷 cache 的情形，手动刷 cache，和 uncache 情形.

用户调用 ION_IOC_ALLOC 时，指定 flags:

| flgas | buffer 映射类型 |
|---|---|
| ION_FLAG_CACHED | cached，自动刷 cache |
| ION_FLAG_CACHED \| ION_FLAG_CACHED_NEEDS_SYNC | cached，手动刷 cache |
| 0 | uncached 映射 |

### 3.5.1. uncache 映射

当 ion_buffer->flags 没有指定 ION_FLAG_CACHED 标记时.

ion_mmap 中将 vma->vm_page_prot 置 pgprot_writecombine 标记，再调用 buffer->heap->ops->map_user; 进行 uncached 映射;

### 3.5.2. 手动刷 cache

当 ion_buffer->flags 指定 ION_FLAG_CACHED 和 ION_FLAG_CACHED_NEEDS_SYNC 标记时.

ion_mmap 中直接调用 buffer->heap->ops->map_user; 由用户层手动刷 cache.

### 3.5.3. 自动刷 cache

当 ion_buffer->flags 指定了 ION_FLAG_CACHED,

同时未指定 ION_FLAG_CACHED_NEEDS_SYNC 标记时.

(以下个人理解，待完善)

ion_mmap 中不进行映射，只是调用 ion_vm_open 新建 vma_list，并添加到 buffer->vmas 链;

由于没有映射，应用层访问时会缺页，此时 ion_vm_fault 建立该页的映射，将该页在

buffer->dirty 的位写 1，并调用 dma_sync_sg_for_cpu 将该页 cache 无效；此后应用层可访问该页.

用户访问完，ion_map_dma_buf 中调用 ion_buffer_sync_for_device，对 buffer 中所有 dirty 页刷 cache(调用 dma_sync_sg_for_device)，将 cpu cache 同步到 memory，这样 dma 设备可以访问；

ion_buffer_sync_for_device 还会清除所有 dirty 标记，清除 buffer->vmas 链上的所有用户页表；

至此，一次访问结束；

下次用户访问时，又出现缺页，由 ion_vm_fault 重新建立页表；

疑问: ion_map_dma_buf 由谁调用?

## 3.6. 几种堆的操作比较

| 堆类型 | 分配 | map dma | map kernel | map user |
|---|---|---|---|---|
| SYSTEM_CONTIG | kmalloc，buffer->priv_virt 为 vaddr | sg_table 包含一项 scatterlist | ion_heap_map_kernel -> vmap | remap_pfn_range，virt_to_phys 得到 page 号 |
| SYSTEM | uncached: 通过 ion_page_pool 分配;<br>cached: 通过 alloc_pages，多次分配. buffer->priv_virt 为 sg_table | sg_table 包含多个 scatterlist | ion_heap_map_kernel -> vmap | ion_heap_map_user，分段映射. |
| CARVEOUT | gen_pool_alloc 一次分配. | sg_table 包含一项 scatterlist | __arm_ioremap | remap_pfn_range 一次性映射 |
| CHUNK | gen_pool_alloc 多次分配,每笔大小固定，buffer->priv_virt 为 sg_table. | sg_table 包含多个 scatterlist | ion_heap_map_kernel -> vmap | ion_heap_map_user，分段映射. |

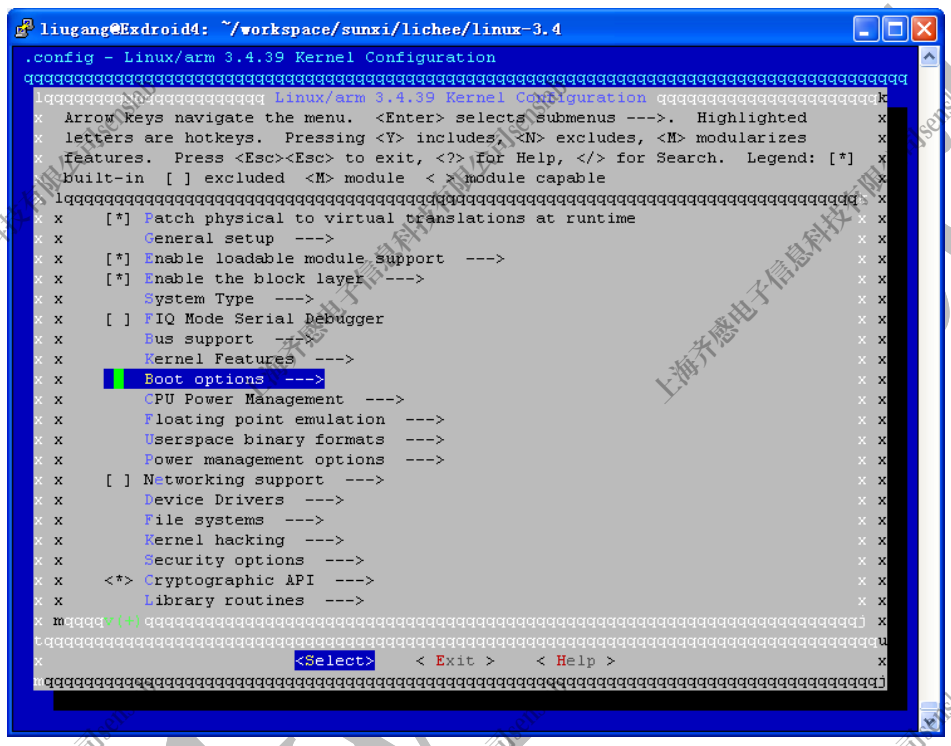
# 4. ION 预留内存的设置

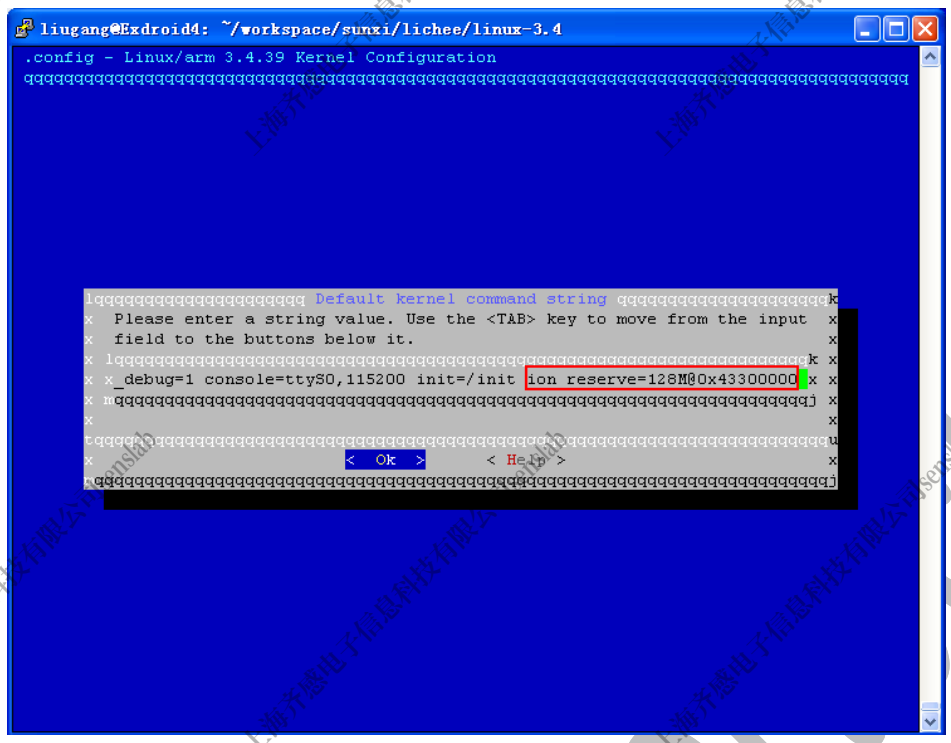
## 4.1. 预留方式

### 4.1.1. linux-3.4 平台

支持两种预留方式.

(1) 当命令行参数指定了 early_param "ion_reserve"时，由配置项值决定 ion 预留位置和大小:



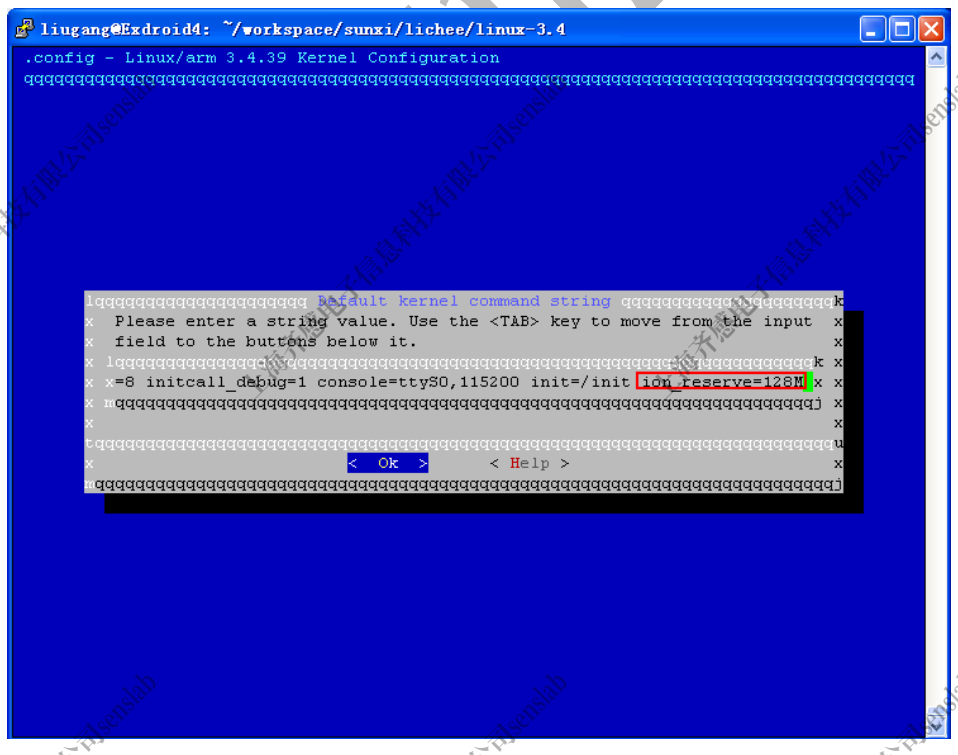



比如上述为 ION 预留空间的起始地址为 0x43300000，大小为 128M;

若未指定起始地址，则 ION 占用系统内存空间的最后端，此时:

| ION 预留区起始地址 = 系统内存起始物理地址 + 系统总内存大小 - ION 预留大小 |
|---|

比如下面情况:




对于 1650 fpga 平台，ION 起始地址为:

| ION start addr = 0x40000000 + SZ_256M - SZ_128M = 0x48000000 |
|---|

0x40000000 为 fpga 起始物理内存地址, SZ_256M 为 fpga 内存总大小, SZ_128M 为 cmdline 指定的 ION 预留大小.

(2) 当没有指定 early_param "ion_reserve" 时, ION 预留区起始地址为 ION_CARVEOUT_MEM_BASE, 大小为 ION_CARVEOUT_MEM_SIZE.

arch/arm/mach-sunxi/include/mach/sun8i/memory-sun8iw3p1.h 文件:

```
#if defined(CONFIG_ION) || defined(CONFIG_ION_MODULE)
#define ION_CARVEOUT_MEM_BASE (0x43100000)
#define ION_CARVEOUT_MEM_SIZE (CONFIG_ION_SUNXI_CARVEOUT_SIZE * SZ_1M)
#endif
```

### 4.1.2. linux-3.10/linux-4.4 平台

linux-3.10/linux-4.4 平台通过 dts 配置 ION 预留位置和大小

```
ion {
    compatible = "allwinner,sunxi-ion";
    system_contig{
        type = <1>;
        name = "system_contig";
    };
    carvout{
        type = <2>;
        name = "carvout";
        base = <0x78800000>;
        size = <0x07800000>;
    };
    cma{
        type = <4>;
        name = "cma";
    };
};
```

比如上述为 ION 预留空间的起始地址为 0x78800000, 大小为 0x07800000;

## 4.2. 预留内存大小的确定

不同的平台预留量也不一样, 可通过典型场景的实际消耗量, 来确定 ION 预留内存大小. 可按以下步骤:

(1) 修改 \linux-3.3\kernel\printk.c

修改前:

```
asmlinkage int vprintk(const char *fmt, va_list args)
{
...
#ifdef      CONFIG_DEBUG_LL
#if 0
        printascii(printk_buf);
#endif /* add by shuge */
#endif
...
}
```

修改后:

```
asmlinkage int vprintk(const char *fmt, va_list args)
{
...
#ifdef      CONFIG_DEBUG_LL
#if 1
        printascii(printk_buf);
#endif /* add by shuge */
#endif
...
}
```

(2) 在 ion heap 分配函数中，增加统计信息.

修改 \linux-3.3\drivers\gpu\ion\ion_carveout_heap.c, 用下面内容替换:

```
/*
 * drivers/gpu/ion/ion_carveout_heap.c
 *
 * Copyright (C) 2011 Google, Inc.
 *
 * This software is licensed under the terms of the GNU General Public
 * License version 2, as published by the Free Software Foundation, and
 * may be copied, distributed, and modified under those terms.
 *
```

```
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 */
#include <linux/spinlock.h>

#include <linux/err.h>
#include <linux/genalloc.h>
#include <linux/io.h>
#include <linux/ion.h>
#include <linux/mm.h>
#include <linux/scatterlist.h>
#include <linux/slab.h>
#include <linux/vmalloc.h>
#include "ion_priv.h"

#include <asm/mach/map.h>

#define DEBUG_HEAP

#ifdef DEBUG_HEAP
u32 total_alloc = 0;
u32 alloc_cnt = 0, free_cnt = 0;
u32 max_single_len = 0, max_total_alloc = 0;
static DEFINE_RWLOCK(heap_lock);
#endif

struct ion_carveout_heap {
        struct ion_heap heap;
        struct gen_pool *pool;
        ion_phys_addr_t base;
};
```

```
ion_phys_addr_t ion_carveout_allocate(struct ion_heap *heap,
                                      unsigned long size,
                                      unsigned long align)
{
        struct ion_carveout_heap *carveout_heap =
                container_of(heap, struct ion_carveout_heap, heap);
        unsigned long offset = gen_pool_alloc(carveout_heap->pool, size);
#ifdef DEBUG_HEAP
        u32 tmp_a_c, tmp_f_c, tmp_t_a, max_s_l, max_t_a;
        unsigned long flags;
#endif

        if (!offset) {
#ifdef DEBUG_HEAP
                read_lock_irqsave(&heap_lock, flags);
                tmp_a_c = alloc_cnt;
                tmp_f_c = free_cnt;
                tmp_t_a = total_alloc;
                read_unlock_irqrestore(&heap_lock, flags);
                printk("%s(%d) err: size 0x%08x, align 0x%08x. alloc_cnt %d, free_cnt %d, total_alloc
0x%08x\n",
                        __func__, __LINE__, (int)size, (int)align, tmp_a_c, tmp_f_c, tmp_t_a);
#endif
                return ION_CARVEOUT_ALLOCATE_FAIL;
        }
#ifdef DEBUG_HEAP
        else {
                write_lock_irqsave(&heap_lock, flags);
                alloc_cnt++;
                total_alloc += size;
                if(max_single_len < size)
                        max_single_len = size;
                if(max_total_alloc < total_alloc)
```

```
                max_total_alloc = total_alloc;
            tmp_a_c = alloc_cnt;
            tmp_f_c = free_cnt;
            tmp_t_a = total_alloc;
            max_s_l = max_single_len;
            max_t_a = max_total_alloc;
            write_unlock_irqrestore(&heap_lock, flags);
            printk("%s(%d) success: size 0x%08x, align 0x%08x, ret 0x%08x, alloc_cnt %d,
free_cnt %d,"
                "max_single_len 0x%08x, total_alloc 0x%08x, max_total_alloc 0x%08x\n",
__func__, __LINE__,
                (int)size, (int)align, (int)offset, tmp_a_c, tmp_f_c, max_s_l, tmp_t_a, max_t_a);
        }
#endif

        return offset;
}

void ion_carveout_free(struct ion_heap *heap, ion_phys_addr_t addr, unsigned long size)
{
        struct ion_carveout_heap *carveout_heap =
                container_of(heap, struct ion_carveout_heap, heap);
#ifdef DEBUG_HEAP
        u32 tmp_a_c, tmp_f_c, tmp_t_a, max_s_l, max_t_a;
        unsigned long flags;
#endif

        if (addr == ION_CARVEOUT_ALLOCATE_FAIL)
                return;
        gen_pool_free(carveout_heap->pool, addr, size);
#ifdef DEBUG_HEAP
        write_lock_irqsave(&heap_lock, flags);
        free_cnt++;
        total_alloc -= size;
```

```
	tmp_a_c = alloc_cnt;
	tmp_f_c = free_cnt;
	tmp_t_a = total_alloc;
	max_s_l = max_single_len;
	max_t_a = max_total_alloc;
	write_unlock_irqrestore(&heap_lock, flags);
	printk("%s(%d): addr 0x%08x, size 0x%08x, alloc_cnt %d, free_cnt %d, max_single_len
0x%08x,"
		"total_alloc 0x%08x, max_total_alloc 0x%08x\n", __func__, __LINE__,
		(int)addr, (int)size, tmp_a_c, tmp_f_c, max_s_l, tmp_t_a, max_t_a);
#endif
}

static int ion_carveout_heap_phys(struct ion_heap *heap,
				  struct ion_buffer *buffer,
				  ion_phys_addr_t *addr, size_t *len)
{
	*addr = buffer->priv_phys;
	*len = buffer->size;
	return 0;
}

static int ion_carveout_heap_allocate(struct ion_heap *heap,
				      struct ion_buffer *buffer,
				      unsigned long size, unsigned long align,
				      unsigned long flags)
{
	buffer->priv_phys = ion_carveout_allocate(heap, size, align);
	return buffer->priv_phys == ION_CARVEOUT_ALLOCATE_FAIL ? -ENOMEM : 0;
}

static void ion_carveout_heap_free(struct ion_buffer *buffer)
{
	struct ion_heap *heap = buffer->heap;
```

```
	ion_carveout_free(heap, buffer->priv_phys, buffer->size);
	buffer->priv_phys = ION_CARVEOUT_ALLOCATE_FAIL;
}

struct sg_table *ion_carveout_heap_map_dma(struct ion_heap *heap,
                                           struct ion_buffer *buffer)
{
	struct sg_table *table;
	int ret;

	table = kzalloc(sizeof(struct sg_table), GFP_KERNEL);
	if (!table)
		return ERR_PTR(-ENOMEM);
	ret = sg_alloc_table(table, 1, GFP_KERNEL);
	if (ret) {
		kfree(table);
		return ERR_PTR(ret);
	}
	sg_set_page(table->sgl, phys_to_page(buffer->priv_phys), buffer->size,
		    0);
	return table;
}

void ion_carveout_heap_unmap_dma(struct ion_heap *heap,
                                 struct ion_buffer *buffer)
{
	sg_free_table(buffer->sg_table);
	/* liugang add */
	kfree(buffer->sg_table);
	buffer->sg_table = NULL;
}

void *ion_carveout_heap_map_kernel(struct ion_heap *heap,
```

```
                        struct ion_buffer *buffer)
{
    int mtype = MT_MEMORY_NONCACHED;

    if (buffer->flags & ION_FLAG_CACHED)
        mtype = MT_MEMORY;

    return __arm_ioremap(buffer->priv_phys, buffer->size, mtype);
}

void ion_carveout_heap_unmap_kernel(struct ion_heap *heap,
                        struct ion_buffer *buffer)
{
    __iounmap(buffer->vaddr);
    buffer->vaddr = NULL;
    return;
}

int ion_carveout_heap_map_user(struct ion_heap *heap, struct ion_buffer *buffer,
                        struct vm_area_struct *vma)
{
    return remap_pfn_range(vma, vma->vm_start,
                        __phys_to_pfn(buffer->priv_phys) + vma->vm_pgoff,
                        vma->vm_end - vma->vm_start,
                        __pgprot_modify(vma->vm_page_prot,          L_PTE_MT_MASK,
L_PTE_MT_BUFFERABLE));
                        //pgprot_noncached(vma->vm_page_prot));

}

static struct ion_heap_ops carveout_heap_ops = {
    .allocate = ion_carveout_heap_allocate,
    .free = ion_carveout_heap_free,
    .phys = ion_carveout_heap_phys,
```

```
    .map_dma = ion_carveout_heap_map_dma,
    .unmap_dma = ion_carveout_heap_unmap_dma,
    .map_user = ion_carveout_heap_map_user,
    .map_kernel = ion_carveout_heap_map_kernel,
    .unmap_kernel = ion_carveout_heap_unmap_kernel,
};

struct ion_heap *ion_carveout_heap_create(struct ion_platform_heap *heap_data)
{
    struct ion_carveout_heap *carveout_heap;

    carveout_heap = kzalloc(sizeof(struct ion_carveout_heap), GFP_KERNEL);
    if (!carveout_heap)
        return ERR_PTR(-ENOMEM);

    carveout_heap->pool = gen_pool_create(12, -1);
    if (!carveout_heap->pool) {
        kfree(carveout_heap);
        return ERR_PTR(-ENOMEM);
    }
    carveout_heap->base = heap_data->base;
    gen_pool_add(carveout_heap->pool, carveout_heap->base, heap_data->size, -1);
    carveout_heap->heap.ops = &carveout_heap_ops;
    carveout_heap->heap.type = ION_HEAP_TYPE_CARVEOUT;

    return &carveout_heap->heap;
}

void ion_carveout_heap_destroy(struct ion_heap *heap)
{
    struct ion_carveout_heap *carveout_heap =
        container_of(heap, struct    ion_carveout_heap, heap);

    gen_pool_destroy(carveout_heap->pool);
```

```
    kfree(carveout_heap);
    carveout_heap = NULL;
}
```

(3) 编译带卡打印的 andorid 镜像(pack -d)

(4) 烧写 android 镜像，插入 tf 打印子板

(5) 启动后，测试下面场景，填写表格:

| 场景 | 单笔最大申请(bytes) | 总(净)最大申请量(bytes) |
|---|---|---|
| 主界面 | | |
| 视频 | | |
| 视频 + 游戏(神庙逃亡) | | |
| 视频 + 游戏(N.O.V.A.3) | | |
| 视频 + 拍照 | | |
| 视频 + 拍照 + 游戏(神庙逃亡) | | |
| 视频 + 拍照 + 游戏(N.O.V.A.3) | | |
| 视频 + 网页 | | |
| 视频 + 网页 + 设置 + 地图 | | |
| 视频 + 网页 + 设置 + 地图 + 游戏(神庙逃亡) | | |
| 视频 + 网页 + 设置 + 地图 + 拍照 + 游戏(神庙逃亡) | | |
| 视频 + 网页 + 设置 + 地图 + 游戏(N.O.V.A.3) | | |
| 视频 + 网页 + 设置 + 地图 + 拍照 | | |

测试时，每次 ion 分配时串口打印: "ion_carveout_allocate success: size 0x00c00000, align 0x00001000, ret 0x51503000, alloc_cnt 9, free_cnt 0, max_single_len 0x00c00000, total_alloc 0x04803000, max_total_alloc 0x04803000 "

max_single_len 即"单笔最大申请(bytes)", max_total_alloc 即"总(净)最大申请量(bytes)"

(6) 上述表格中"总(净)最大申请量(bytes)"栏的最大值，将作为 ion 预留内存的参考.

考虑内存碎片，ion 实际预留量要适当比它大，比如上述最大值为 204.5M，则建议 ion 预留内存 256M.

# 5. 内核态使用说明

## 5.1. 申请

函数: unsigned int sunxi_mem_alloc(unsigned int size);

参数: size: 字节为单位，建议 4K 对齐;

返回值: 成功返回非 0，为起始物理地址;

失败返回 0;

## 5.2. 释放

函数: void sunxi_mem_free(unsigned int phys_addr, unsigned int size)

参数: phys_addr: 起始物理地址，必须与 sunxi_mem_alloc 返回的地址相同;

size: 字节为单位的大小，必须与 sunxi_mem_alloc 的 size 相同;

## 5.3. 映射

通过 ioremap 宏进行映射.

```
void *pvirt_addr;
unsigned int start_phys_addr = 0x43300000;
unsigned int size = SZ_1M;
pvirt_addr = (void *)ioremap(start_phys_addr, size);
```

start_phys_addr: 待映射区间的起始物理地址; 建议 4K 对齐;

size: 待映射区间的大小;

ioremap 不一定要对 sunxi_mem_alloc 的整块内存进行映射，可以映射某一部分;

2013-7-31 11:26 修改:

不再用 ioremap 映射，改用 sunxi_map_kernel 映射，参数和 ioremap 一样;

修改前:

```
void *pvirt_addr;
unsigned int start_phys_addr = 0x43300000;
unsigned int size = SZ_1M;
pvirt_addr = (void *)ioremap(start_phys_addr, size);
```

修改后:

```
void *pvirt_addr;
unsigned int start_phys_addr = 0x43300000;
unsigned int size = SZ_1M;
```

```
pvirt_addr = (void *)sunxi_map_kernel(start_phys_addr, size);
```

## 5.4. 解除映射

通过 iounmap 宏解除映射.

```
void *pvirt_addr;
iounmap((void* __iomem)pvirt_addr);
```

pvirt_addr 必须为 ioremap 的返回值.

2013-7-31 11:26 修改:

不再用 iounmap 解除映射，改用 sunxi_unmap_kernel 解除映射，参数和 ioremap 一样;

修改前:

```
void *pvirt_addr;
iounmap((void* __iomem)pvirt_addr);
```

修改后:

```
void *pvirt_addr;
sunxi_unmap_kernel((void* __iomem)pvirt_addr);
```

## 5.5. demo

```
#include <linux/kernel.h>
#include <linux/io.h>
#include <linux/types.h>
#include <linux/random.h>
#include <linux/ion_sunxi.h>

    unsigned int size = SZ_1M;
    unsigned int phys_addr;
    void *virt_addr = NULL;
    int i;

    /* alloc buffer */
    phys_addr = sunxi_mem_alloc(size);
    if(!phys_addr) {
        printk("%s(%d) err: alloc 0x%08x failed!\n", __func__, __LINE__, size);
```

```
        goto end;
    }

    /* map kernel */
    virt_addr = sunxi_map_kernel(phys_addr, size);
    if(!virt_addr) {
        printk("%s(%d) err: ioremap 0x%08x failed!\n", __func__, __LINE__, phys_addr);
        goto end;
    }

    /* now you can access buffer via virt_addr */
    for(i = 0; i < 100; i++)
        *((char *)virt_addr + i) = get_random_int()%0xff;

end:
    /* free resource */
    if(virt_addr)
        sunxi_unmap_kernel(virt_addr);
    if(phys_addr)
        sunxi_mem_free(phys_addr, size);
    return;
```

# 6. 用户态使用说明

## 6.1. cache 映射情形

cache 映射指的是，用户态申请得到 ion 物理内存后，以 cache 方式映射到用户空间.

优点是效率高，缺点是需要手动刷 cache. 当用户修改了 ion buffer 内容，需要手动刷 cache，将修改回写到 dram 中；当硬件(比如 dma)更新了 ion buffer 内容后，需要手动刷 cache，将 dcache 无效，以便用户可以访问到 buffer 最新内容;

手动刷 cache 可以通过 ION_IOC_SUNXI_FLUSH_RANGE 命令来完成.

一般情况下，用户态采用 cached 映射方式.

### 6.1.1. 申请

步骤:

(1) 打开/dev/ion 设备;

(2) ION_IOC_ALLOC 申请内存;

传入 ION_FLAG_CACHED | ION_FLAG_CACHED_NEEDS_SYNC 标记

(3) ION_IOC_MAP 获取 dma_buf 文件描述符;

(4) mmap 映射到用户空间; cached 的映射.

```
#define ION_DEV_NAME "/dev/ion"
#define ION_ALLOC_SIZE  (SZ_4M + SZ_1M - SZ_64K)
#define ION_ALLOC_ALIGN     (SZ_1M)

int test_ion()
{
    struct ion_allocation_data alloc_data;
    struct ion_handle_data handle_data;
    struct ion_fd_data fd_data;
    void *user_addr;
    int fd, ret = 0;

    /* 1. open ion device */
    fd = open(ION_DEV_NAME, O_RDONLY);
    if(fd < 0) {
        printf("err open %s\n", ION_DEV_NAME);
        return -1;
    }
```

```
        /* 2. alloc buffer */
        alloc_data.len = ION_ALLOC_SIZE;
        alloc_data.align = ION_ALLOC_ALIGN;
        alloc_data.heap_id_mask = ION_HEAP_CARVEOUT_MASK;
        alloc_data.flags = ION_FLAG_CACHED | ION_FLAG_CACHED_NEEDS_SYNC;
        ret = ioctl(fd, ION_IOC_ALLOC, &alloc_data);
        if(ret) {
            printf("ION_IOC_ALLOC err, ret %d, handle 0x%08x\n", ret, (unsigned int)alloc_data.handle);
            goto out1;
        }

        /* 3. get dmabuf fd */
        fd_data.handle = alloc_data.handle;
        ret = ioctl(fd, ION_IOC_MAP, &fd_data);
        if(ret) {
            printf("ION_IOC_MAP err, ret %d, dmabuf fd 0x%08x\n", ret, (unsigned int)fd_data.fd);
            goto out2;
        }

        /* 4. mmap to user space */
        user_addr = mmap(NULL, alloc_data.len, PROT_READ|PROT_WRITE, MAP_SHARED, fd_data.fd, 0);
        if(MAP_FAILED == user_addr) {
            printf("mmap err, ret %d\n", (unsigned int)user_addr);
            goto out3;
        }

        /* now you can access ion buffer via user_addr */
        ...
    out4:
        /* unmmap user buffer */
```

```
        ret = munmap(user_addr, alloc_data.len);
        if(ret)
                printf("munmap err, ret %d\n", ret);
out3:
        /* close dmabuf fd */
        close(fd_data.fd);
out2:
        /* free buffer */
        handle_data.handle = alloc_data.handle;
        ret = ioctl(fd, ION_IOC_FREE, &handle_data);
        if(ret)
                printf("ION_IOC_FREE err, ret %d\n", ret);
out1:
        /* close ion device */
        close(fd);
        return ret;
}
```

### 6.1.2. 释放

步骤:

(1) 将之前 map 的 user buffer 进行 unmap;

(2) 关闭 dma_buf 文件描述符;

(3) ION_IOC_FREE 释放内存;

(4) 关闭/dev/ion 设备;

```
#define ION_DEV_NAME "/dev/ion"
#define ION_ALLOC_SIZE  (SZ_4M + SZ_1M - SZ_64K)
#define ION_ALLOC_ALIGN     (SZ_1M)

int test_ion()
{
        struct ion_allocation_data alloc_data;
        struct ion_handle_data handle_data;
        struct ion_fd_data fd_data;
        void *user_addr;
        int fd, ret = 0;
```

```
	/* 1. open ion device */
	fd = open(ION_DEV_NAME, O_RDONLY);
	if(fd < 0) {
		printf("err open %s\n", ION_DEV_NAME);
		return -1;
	}

	/* 2. alloc buffer */
	alloc_data.len = ION_ALLOC_SIZE;
	alloc_data.align = ION_ALLOC_ALIGN;
	alloc_data.heap_id_mask = ION_HEAP_CARVEOUT_MASK;
	alloc_data.flags = ION_FLAG_CACHED | ION_FLAG_CACHED_NEEDS_SYNC;
	ret = ioctl(fd, ION_IOC_ALLOC, &alloc_data);
	if(ret) {
		printf("ION_IOC_ALLOC err, ret %d, handle 0x%08x\n", ret, (unsigned int)alloc_data.handle);
		goto out1;
	}

	/* 3. get dmabuf fd */
	fd_data.handle = alloc_data.handle;
	ret = ioctl(fd, ION_IOC_MAP, &fd_data);
	if(ret) {
		printf("ION_IOC_MAP err, ret %d, dmabuf fd 0x%08x\n", ret, (unsigned int)fd_data.fd);
		goto out2;
	}

	/* 4. mmap to user space */
	user_addr = mmap(NULL, alloc_data.len, PROT_READ|PROT_WRITE, MAP_SHARED, fd_data.fd, 0);
	if(MAP_FAILED == user_addr) {
		printf("mmap err, ret %d\n", (unsigned int)user_addr);
```

```
        goto out3;
    }

    /* now you can access ion buffer via user_addr */
    ...
out4:
    /* 1. unmmap user buffer */
    ret = munmap(user_addr, alloc_data.len);
    if(ret)
        printf("munmap err, ret %d\n", ret);
out3:
    /* 2. close dmabuf fd */
    close(fd_data.fd);
out2:
    /* 3. free buffer */
    handle_data.handle = alloc_data.handle;
    ret = ioctl(fd, ION_IOC_FREE, &handle_data);
    if(ret)
        printf("ION_IOC_FREE err, ret %d\n", ret);
out1:
    /* 4. close ion device */
    close(fd);
    return ret;
}
```

### 6.1.3. 刷 cache

通过 ION_IOC_SUNXI_FLUSH_RANGE 命令完成.

该命令将用户对 ion buffer 的修改同步到 dram, 并将 data cache 无效.

```
#define ION_DEV_NAME "/dev/ion"
#define ION_ALLOC_SIZE  (SZ_4M + SZ_1M - SZ_64K)
#define ION_ALLOC_ALIGN     (SZ_1M)

int test_ion()
{
    struct ion_allocation_data alloc_data;
```

```
    struct ion_handle_data handle_data;
    struct ion_fd_data fd_data;
    sunxi_cache_range range;
    void *user_addr;
    int fd, ret = 0;

    /* 1. open ion device */
    fd = open(ION_DEV_NAME, O_RDONLY);
    if(fd < 0) {
        printf("err open %s\n", ION_DEV_NAME);
        return -1;
    }

    /* 2. alloc buffer */
    alloc_data.len = ION_ALLOC_SIZE;
    alloc_data.align = ION_ALLOC_ALIGN;
    alloc_data.heap_id_mask = ION_HEAP_CARVEOUT_MASK;
    alloc_data.flags = ION_FLAG_CACHED | ION_FLAG_CACHED_NEEDS_SYNC;
    ret = ioctl(fd, ION_IOC_ALLOC, &alloc_data);
    if(ret) {
        printf("ION_IOC_ALLOC err, ret %d, handle 0x%08x\n", ret, (unsigned
int)alloc_data.handle);
        goto out1;
    }

    /* 3. get dmabuf fd */
    fd_data.handle = alloc_data.handle;
    ret = ioctl(fd, ION_IOC_MAP, &fd_data);
    if(ret) {
        printf("ION_IOC_MAP err, ret %d, dmabuf fd 0x%08x\n", ret, (unsigned
int)fd_data.fd);
        goto out2;
    }
```

```
        /* 4. mmap to user space */
        user_addr = mmap(NULL, alloc_data.len, PROT_READ|PROT_WRITE,
MAP_SHARED, fd_data.fd, 0);
        if(MAP_FAILED == user_addr) {
            printf("mmap err, ret %d\n", (unsigned int)user_addr);
            goto out3;
        }

        /* user access ion buffer */
        for(i = 0; i < 256; i++)
            *((char *)user_addr + i) = 0x55;

        /* write back and invalid user cache */
        range.start = (unsigned long)user_addr;
        range.end = (unsigned long)user_addr + 256;
        custom_data.cmd = ION_IOC_SUNXI_FLUSH_RANGE;
        custom_data.arg = (unsigned long)&range;
        ret = ioctl(fd, ION_IOC_CUSTOM, &custom_data);
        if(ret) {
            printf("ION_IOC_CUSTOM err, ret %d\n", ret);
            goto out4;
        }

        ...
    out4:
        /* unmmap user buffer */
        ret = munmap(user_addr, alloc_data.len);
        if(ret)
            printf("munmap err, ret %d\n", ret);
    out3:
        /* close dmabuf fd */
        close(fd_data.fd);
    out2:
        /* free buffer */
```

```
        handle_data.handle = alloc_data.handle;
        ret = ioctl(fd, ION_IOC_FREE, &handle_data);
        if(ret)
            printf("ION_IOC_FREE err, ret %d\n", ret);
out1:
        /* close ion device */
        close(fd);
        return ret;
}
```

### 6.1.4. demo

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <time.h>
#include <sys/mman.h>
#include <sys/mman.h>
#include <asm-generic/ioctl.h>

#define ION_DEV_NAME    "/dev/ion"

/*
 * structures define from linux kernel
 */
#define SZ_64M          0x04000000
#define SZ_4M           0x00400000
#define SZ_1M           0x00100000
#define SZ_64K          0x00010000
#define ION_ALLOC_SIZE  (SZ_4M + SZ_1M - SZ_64K)
#define ION_ALLOC_ALIGN     (SZ_1M)
```

```
struct ion_allocation_data {
        size_t len;
        size_t align;
        unsigned int heap_id_mask;
        unsigned int flags;
        void *handle;
};

struct ion_handle_data {
        void *handle;
};

struct ion_fd_data {
        void *handle;
        int fd;
};

struct ion_custom_data {
        unsigned int cmd;
        unsigned long arg;
};

enum ion_heap_type {
        ION_HEAP_TYPE_SYSTEM,
        ION_HEAP_TYPE_SYSTEM_CONTIG,
        ION_HEAP_TYPE_CARVEOUT,
        ION_HEAP_TYPE_CHUNK,
        ION_HEAP_TYPE_CUSTOM, /* must be last so device specific heaps always
                                 are at the end of this enum */
        ION_NUM_HEAPS = 16,
};

#define ION_IOC_MAGIC           'I'
```

```
#define ION_IOC_ALLOC       _IOWR(ION_IOC_MAGIC, 0, struct ion_allocation_data)
#define ION_IOC_FREE        _IOWR(ION_IOC_MAGIC, 1, struct ion_handle_data)
#define ION_IOC_MAP         _IOWR(ION_IOC_MAGIC, 2, struct ion_fd_data)
#define ION_IOC_SHARE       _IOWR(ION_IOC_MAGIC, 4, struct ion_fd_data)
#define ION_IOC_IMPORT      _IOWR(ION_IOC_MAGIC, 5, struct ion_fd_data)
#define ION_IOC_SYNC        _IOWR(ION_IOC_MAGIC, 7, struct ion_fd_data)
#define ION_IOC_CUSTOM      _IOWR(ION_IOC_MAGIC, 6, struct ion_custom_data)

#define ION_FLAG_CACHED 1       /* mappings of this buffer should be
                                   cached, ion will do cache
                                   maintenance when the buffer is
                                   mapped for dma */
#define ION_FLAG_CACHED_NEEDS_SYNC 2    /* mappings of this buffer will created
                                   at mmap time, if this is set
                                   caches must be managed manually */

typedef struct {
    long    start;
    long    end;
}sunxi_cache_range;

typedef struct {
    void *handle;
    unsigned int phys_addr;
    unsigned int size;
}sunxi_phys_data;

#define ION_IOC_SUNXI_FLUSH_RANGE       5
#define ION_IOC_SUNXI_FLUSH_ALL         6
#define ION_IOC_SUNXI_PHYS_ADDR         7
#define ION_IOC_SUNXI_DMA_COPY          8

#define ION_HEAP_SYSTEM_MASK    (1 << ION_HEAP_TYPE_SYSTEM)
#define ION_HEAP_SYSTEM_CONTIG_MASK     (1                  <<
```

```
ION_HEAP_TYPE_SYSTEM_CONTIG)
    #define ION_HEAP_CARVEOUT_MASK        (1 << ION_HEAP_TYPE_CARVEOUT)

    int test_ion()
    {
        struct ion_allocation_data alloc_data;
        struct ion_fd_data fd_data;
        struct ion_handle_data handle_data;
        struct ion_custom_data custom_data;
        sunxi_cache_range range;
        sunxi_phys_data phys_data;
        void *user_addr;
        int fd, ret = 0, i;

        /* open ion device */
        fd = open(ION_DEV_NAME, O_RDONLY);
        if(fd < 0) {
            printf("err open %s\n", ION_DEV_NAME);
            return -1;
        }
        printf("open %s success\n", ION_DEV_NAME);

        /* alloc buffer */
        alloc_data.len = ION_ALLOC_SIZE;
        alloc_data.align = ION_ALLOC_ALIGN;
        alloc_data.heap_id_mask = ION_HEAP_CARVEOUT_MASK;
        alloc_data.flags = ION_FLAG_CACHED | ION_FLAG_CACHED_NEEDS_SYNC;
        ret = ioctl(fd, ION_IOC_ALLOC, &alloc_data);
        if(ret) {
            printf("ION_IOC_ALLOC err, ret %d, handle 0x%08x\n", ret, (unsigned
int)alloc_data.handle);
            goto out1;
        }
        printf("ION_IOC_ALLOC succes, handle 0x%08x\n", (unsigned int)alloc_data.handle);
```

```
        /* optional: get buffer phys_addr */
        custom_data.cmd = ION_IOC_SUNXI_PHYS_ADDR;
        phys_data.handle = alloc_data.handle;
        custom_data.arg = (unsigned long)&phys_data;
        ret = ioctl(fd, ION_IOC_CUSTOM, &custom_data);
        if(ret) {
            printf("ION_IOC_SUNXI_PHYS_ADDR err, ret %d\n", ret);
            goto out1;
        }
        printf("ION_IOC_SUNXI_PHYS_ADDR succes, phys_addr 0x%08x, size 0x%08x\n",
phys_data.phys_addr, phys_data.size);

        /* get dmabuf fd */
        fd_data.handle = alloc_data.handle;
        ret = ioctl(fd, ION_IOC_MAP, &fd_data);
        if(ret) {
            printf("ION_IOC_MAP err, ret %d, dmabuf fd 0x%08x\n", ret, (unsigned
int)fd_data.fd);
            goto out2;
        }
        printf("ION_IOC_MAP succes, get dmabuf fd 0x%08x\n", (unsigned int)fd_data.fd);

        /* mmap to user */
        user_addr = mmap(NULL, alloc_data.len, PROT_READ|PROT_WRITE,
MAP_SHARED, fd_data.fd, 0);
        if(MAP_FAILED == user_addr) {
            printf("mmap err, ret %d\n", (unsigned int)user_addr);
            goto out3;
        }
        printf("mmap succes, get user_addr 0x%08x\n", (unsigned int)user_addr);

        /* access buffer */
        for(i = 0; i < 100; i++)
```

```
            *((char *)user_addr + i) = rand()&0xff;
        for(i = 0; i < 100; i++)
            *((char *)user_addr + alloc_data.len - i) = rand()&0xff;
        printf("random access user buf succes\n");

        /* clean and invalid user cache */
        range.start = (unsigned long)user_addr;
        range.end = (unsigned long)user_addr + alloc_data.len;
        custom_data.cmd = ION_IOC_SUNXI_FLUSH_RANGE;
        custom_data.arg = (unsigned long)&range;
        printf("start flush user cache 0x%08x~0x%08x via
ION_IOC_SUNXI_FLUSH_RANGE\n", range.start, range.end);
        ret = ioctl(fd, ION_IOC_CUSTOM, &custom_data);
        if(ret) {
            printf("ION_IOC_CUSTOM err, ret %d\n", ret);
            goto out4;
        }
        printf("flush cache succes\n");

        /* now device can access the buffer... */

    out4:
        /* unmmap */
        ret = munmap(user_addr, alloc_data.len);
        if(ret)
            printf("munmap err, ret %d\n", ret);
        printf("munmap succes\n");
    out3:
        /* close dmabuf fd */
        close(fd_data.fd);
        printf("close dmabuf fd succes\n");
    out2:
        /* free buffer */
        handle_data.handle = alloc_data.handle;
```

```
    ret = ioctl(fd, ION_IOC_FREE, &handle_data);
    if(ret)
        printf("ION_IOC_FREE err, ret %d\n", ret);
    printf("ION_IOC_FREE succes\n");
out1:
    /* close ion device */
    close(fd);
    return ret;
}

int main()
{
    test_ion();
}
```

## 6.2. uncache 映射情形

cache 映射指的是，用户态申请得到 ion 物理内存后，以 uncache 方式映射到用户空间.

优点是用户对 ion buffer 内容的修改能直通到 dram，不用人为刷 cache；缺点是效率低;

某些场合需要用户对 ion buffer 的修改能直接写到内存中，此时只能用 uncached 映射.比如显示驱动的 frambuffer，若按 cached 方式映射，用户写 framebuffer 时，会在 lcd 屏上看到条纹.

### 6.2.1. 申请

与 cached 方式的唯一区别是, ION_IOC_ALLOC 的 alloc_data.flags 参数设为 0.

步骤:

(1) 打开/dev/ion 设备;

(2) ION_IOC_ALLOC 申请内存; alloc_data.flags 标记设为 0;

(3) ION_IOC_MAP 获取 dma_buf 文件描述符;

(4) mmap 映射到用户空间; ion 驱动会进行 uncached 的映射.

```
#define ION_DEV_NAME "/dev/ion"
#define ION_ALLOC_SIZE  (SZ_4M + SZ_1M - SZ_64K)
#define ION_ALLOC_ALIGN     (SZ_1M)

int test_ion()
{
    struct ion_allocation_data alloc_data;
```

```
    struct ion_handle_data handle_data;
    struct ion_fd_data fd_data;
    void *user_addr;
    int fd, ret = 0;

    /* 1. open ion device */
    fd = open(ION_DEV_NAME, O_RDONLY);
    if(fd < 0) {
        printf("err open %s\n", ION_DEV_NAME);
        return -1;
    }

    /* 2. alloc buffer */
    alloc_data.len = ION_ALLOC_SIZE;
    alloc_data.align = ION_ALLOC_ALIGN;
    alloc_data.heap_id_mask = ION_HEAP_CARVEOUT_MASK;
    alloc_data.flags = 0;
    ret = ioctl(fd, ION_IOC_ALLOC, &alloc_data);
    if(ret) {
        printf("ION_IOC_ALLOC err, ret %d, handle 0x%08x\n", ret, (unsigned int)alloc_data.handle);
        goto out1;
    }

    /* 3. get dmabuf fd */
    fd_data.handle = alloc_data.handle;
    ret = ioctl(fd, ION_IOC_MAP, &fd_data);
    if(ret) {
        printf("ION_IOC_MAP err, ret %d, dmabuf fd 0x%08x\n", ret, (unsigned int)fd_data.fd);
        goto out2;
    }

    /* 4. mmap to user space */
```

```
        user_addr = mmap(NULL, alloc_data.len, PROT_READ|PROT_WRITE,
MAP_SHARED, fd_data.fd, 0);
        if(MAP_FAILED == user_addr) {
            printf("mmap err, ret %d\n", (unsigned int)user_addr);
            goto out3;
        }

        /* now you can access ion buffer via user_addr */
        ...
    out4:
        /* unmmap user buffer */
        ret = munmap(user_addr, alloc_data.len);
        if(ret)
            printf("munmap err, ret %d\n", ret);
    out3:
        /* close dmabuf fd */
        close(fd_data.fd);
    out2:
        /* free buffer */
        handle_data.handle = alloc_data.handle;
        ret = ioctl(fd, ION_IOC_FREE, &handle_data);
        if(ret)
            printf("ION_IOC_FREE err, ret %d\n", ret);
    out1:
        /* close ion device */
        close(fd);
        return ret;
    }
```

### 6.2.2. 释放

与 cached 映射完全一样.

步骤:

(5) 将之前 map 的 user buffer 进行 unmap;

(6) 关闭 dma_buf 文件描述符;

(7) ION_IOC_FREE 释放内存;

(8) 关闭/dev/ion 设备;

```
#define ION_DEV_NAME "/dev/ion"
#define ION_ALLOC_SIZE  (SZ_4M + SZ_1M - SZ_64K)
#define ION_ALLOC_ALIGN     (SZ_1M)

int test_ion()
{
    struct ion_allocation_data alloc_data;
    struct ion_handle_data handle_data;
    struct ion_fd_data fd_data;
    void *user_addr;
    int fd, ret = 0;

    /* 1. open ion device */
    fd = open(ION_DEV_NAME, O_RDONLY);
    if(fd < 0) {
        printf("err open %s\n", ION_DEV_NAME);
        return -1;
    }

    /* 2. alloc buffer */
    alloc_data.len = ION_ALLOC_SIZE;
    alloc_data.align = ION_ALLOC_ALIGN;
    alloc_data.heap_id_mask = ION_HEAP_CARVEOUT_MASK;
    alloc_data.flags = ION_FLAG_CACHED | ION_FLAG_CACHED_NEEDS_SYNC;
    ret = ioctl(fd, ION_IOC_ALLOC, &alloc_data);
    if(ret) {
        printf("ION_IOC_ALLOC err, ret %d, handle 0x%08x\n", ret, (unsigned
int)alloc_data.handle);
        goto out1;
    }

    /* 3. get dmabuf fd */
    fd_data.handle = alloc_data.handle;
```

```
        ret = ioctl(fd, ION_IOC_MAP, &fd_data);
        if(ret) {
            printf("ION_IOC_MAP err, ret %d, dmabuf fd 0x%08x\n", ret, (unsigned int)fd_data.fd);
            goto out2;
        }

        /* 4. mmap to user space */
        user_addr = mmap(NULL, alloc_data.len, PROT_READ|PROT_WRITE, MAP_SHARED, fd_data.fd, 0);
        if(MAP_FAILED == user_addr) {
            printf("mmap err, ret %d\n", (unsigned int)user_addr);
            goto out3;
        }

        /* now you can access ion buffer via user_addr */
        ...
    out4:
        /* 1. unmmap user buffer */
        ret = munmap(user_addr, alloc_data.len);
        if(ret)
            printf("munmap err, ret %d\n", ret);
    out3:
        /* 2. close dmabuf fd */
        close(fd_data.fd);
    out2:
        /* 3. free buffer */
        handle_data.handle = alloc_data.handle;
        ret = ioctl(fd, ION_IOC_FREE, &handle_data);
        if(ret)
            printf("ION_IOC_FREE err, ret %d\n", ret);
    out1:
        /* 4. close ion device */
        close(fd);
```

```
    return ret;
}
```

### 6.2.3. demo

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <time.h>
#include <sys/mman.h>
#include <sys/mman.h>
#include <asm-generic/ioctl.h>

#define ION_DEV_NAME    "/dev/ion"

/*
 * structures define from linux kernel
 */
#define SZ_64M          0x04000000
#define SZ_4M           0x00400000
#define SZ_1M           0x00100000
#define SZ_64K          0x00010000
#define ION_ALLOC_SIZE  (SZ_4M + SZ_1M - SZ_64K)
#define ION_ALLOC_ALIGN     (SZ_1M)

struct ion_allocation_data {
    size_t len;
    size_t align;
    unsigned int heap_id_mask;
    unsigned int flags;
    void *handle;
```

```
};

struct ion_handle_data {
        void *handle;
};

struct ion_fd_data {
        void *handle;
        int fd;
};

struct ion_custom_data {
        unsigned int cmd;
        unsigned long arg;
};

enum ion_heap_type {
        ION_HEAP_TYPE_SYSTEM,
        ION_HEAP_TYPE_SYSTEM_CONTIG,
        ION_HEAP_TYPE_CARVEOUT,
        ION_HEAP_TYPE_CHUNK,
        ION_HEAP_TYPE_CUSTOM, /* must be last so device specific heaps always
                                 are at the end of this enum */
        ION_NUM_HEAPS = 16,
};

#define ION_IOC_MAGIC           'I'
#define ION_IOC_ALLOC           _IOWR(ION_IOC_MAGIC, 0, struct ion_allocation_data)
#define ION_IOC_FREE            _IOWR(ION_IOC_MAGIC, 1, struct ion_handle_data)
#define ION_IOC_MAP         _IOWR(ION_IOC_MAGIC, 2, struct ion_fd_data)
#define ION_IOC_SHARE           _IOWR(ION_IOC_MAGIC, 4, struct ion_fd_data)
#define ION_IOC_IMPORT          _IOWR(ION_IOC_MAGIC, 5, struct ion_fd_data)
#define ION_IOC_SYNC            _IOWR(ION_IOC_MAGIC, 7, struct ion_fd_data)
#define ION_IOC_CUSTOM          _IOWR(ION_IOC_MAGIC, 6, struct ion_custom_data)
```

```
#define ION_FLAG_CACHED 1        /* mappings of this buffer should be
                                    cached, ion will do cache
                                    maintenance when the buffer is
                                    mapped for dma */
#define ION_FLAG_CACHED_NEEDS_SYNC 2      /* mappings of this buffer will created
                                    at mmap time, if this is set
                                    caches must be managed manually */

typedef struct {
    long        start;
    long        end;
}sunxi_cache_range;

typedef struct {
    void *handle;
    unsigned int phys_addr;
    unsigned int size;
}sunxi_phys_data;

#define ION_IOC_SUNXI_FLUSH_RANGE           5
#define ION_IOC_SUNXI_FLUSH_ALL             6
#define ION_IOC_SUNXI_PHYS_ADDR             7
#define ION_IOC_SUNXI_DMA_COPY              8

#define ION_HEAP_SYSTEM_MASK       (1 << ION_HEAP_TYPE_SYSTEM)
#define ION_HEAP_SYSTEM_CONTIG_MASK      (1                              <<
ION_HEAP_TYPE_SYSTEM_CONTIG)
#define ION_HEAP_CARVEOUT_MASK        (1 << ION_HEAP_TYPE_CARVEOUT)

int test_ion()
{
    struct ion_allocation_data alloc_data;
    struct ion_fd_data fd_data;
```

```
struct ion_handle_data handle_data;
struct ion_custom_data custom_data;
sunxi_phys_data phys_data;
void *user_addr;
int fd, ret = 0, i;

/* open ion device */
fd = open(ION_DEV_NAME, O_RDONLY);
if(fd < 0) {
    printf("err open %s\n", ION_DEV_NAME);
    return -1;
}
printf("open %s success\n", ION_DEV_NAME);

/* alloc buffer */
alloc_data.len = ION_ALLOC_SIZE;
alloc_data.align = ION_ALLOC_ALIGN;
alloc_data.heap_id_mask = ION_HEAP_CARVEOUT_MASK;
alloc_data.flags = 0;
ret = ioctl(fd, ION_IOC_ALLOC, &alloc_data);
if(ret) {
    printf("ION_IOC_ALLOC err, ret %d, handle 0x%08x\n", ret, (unsigned int)alloc_data.handle);
    goto out1;
}
printf("ION_IOC_ALLOC succes, handle 0x%08x\n", (unsigned int)alloc_data.handle);

/* optional: get buffer phys_addr */
custom_data.cmd = ION_IOC_SUNXI_PHYS_ADDR;
phys_data.handle = alloc_data.handle;
custom_data.arg = (unsigned long)&phys_data;
ret = ioctl(fd, ION_IOC_CUSTOM, &custom_data);
if(ret) {
    printf("ION_IOC_SUNXI_PHYS_ADDR err, ret %d\n", ret);
```

```
        goto out1;
    }
    printf("ION_IOC_SUNXI_PHYS_ADDR succes, phys_addr 0x%08x, size 0x%08x\n",
phys_data.phys_addr, phys_data.size);

    /* get dmabuf fd */
    fd_data.handle = alloc_data.handle;
    ret = ioctl(fd, ION_IOC_MAP, &fd_data);
    if(ret) {
        printf("ION_IOC_MAP err, ret %d, dmabuf fd 0x%08x\n", ret, (unsigned
int)fd_data.fd);
        goto out2;
    }
    printf("ION_IOC_MAP succes, get dmabuf fd 0x%08x\n", (unsigned int)fd_data.fd);

    /* mmap to user */
    user_addr = mmap(NULL, alloc_data.len, PROT_READ|PROT_WRITE,
MAP_SHARED, fd_data.fd, 0);
    if(MAP_FAILED == user_addr) {
        printf("mmap err, ret %d\n", (unsigned int)user_addr);
        goto out3;
    }
    printf("mmap succes, get user_addr 0x%08x\n", (unsigned int)user_addr);

    /* access buffer */
    for(i = 0; i < 100; i++)
        *((char *)user_addr + i) = rand()&0xff;
    for(i = 0; i < 100; i++)
        *((char *)user_addr + alloc_data.len - i) = rand()&0xff;
    printf("random access user buf succes\n");

out4:
    /* unmmap */
    ret = munmap(user_addr, alloc_data.len);
```

```
    if(ret)
        printf("munmap err, ret %d\n", ret);
    printf("munmap succes\n");
out3:
    /* close dmabuf fd */
    close(fd_data.fd);
    printf("close dmabuf fd succes\n");
out2:
    /* free buffer */
    handle_data.handle = alloc_data.handle;
    ret = ioctl(fd, ION_IOC_FREE, &handle_data);
    if(ret)
        printf("ION_IOC_FREE err, ret %d\n", ret);
    printf("ION_IOC_FREE succes\n");
out1:
    /* close ion device */
    close(fd);
    return ret;
}

int main()
{
    test_ion();
}
```

## 6.3. 获取 ION 内存物理地址

用户调用 ION_IOC_ALLOC 申请得到的是 ion handle，不包含物理地址信息；

如何通过 handle 获取 buffer 的起始物理地址呢?

通过 ION_IOC_SUNXI_PHYS_ADDR 命令.

```
...
typedef struct {
    void *handle;
    unsigned int phys_addr;
    unsigned int size;
}sunxi_phys_data;
```

```
int test_ion()
{
    struct ion_allocation_data alloc_data;
    struct ion_fd_data fd_data;
    struct ion_handle_data handle_data;
    struct ion_custom_data custom_data;
    sunxi_phys_data phys_data;
    void *user_addr;
    int fd, ret = 0, i;

    /* open ion device */
    fd = open(ION_DEV_NAME, O_RDONLY);
    if(fd < 0) {
        printf("err open %s\n", ION_DEV_NAME);
        return -1;
    }
    printf("open %s success\n", ION_DEV_NAME);

    /* alloc buffer */
    alloc_data.len = ION_ALLOC_SIZE;
    alloc_data.align = ION_ALLOC_ALIGN;
    alloc_data.heap_id_mask = ION_HEAP_CARVEOUT_MASK;
    alloc_data.flags = 0;
    ret = ioctl(fd, ION_IOC_ALLOC, &alloc_data);
    if(ret) {
        printf("ION_IOC_ALLOC err, ret %d, handle 0x%08x\n", ret, (unsigned
int)alloc_data.handle);
        goto out1;
    }
    printf("ION_IOC_ALLOC succes, handle 0x%08x\n", (unsigned int)alloc_data.handle);

    /* optional: get buffer phys_addr */
    custom_data.cmd = ION_IOC_SUNXI_PHYS_ADDR;
```

```
        phys_data.handle = alloc_data.handle;
        custom_data.arg = (unsigned long)&phys_data;
        ret = ioctl(fd, ION_IOC_CUSTOM, &custom_data);
        if(ret) {
            printf("ION_IOC_SUNXI_PHYS_ADDR err, ret %d\n", ret);
            goto out1;
        }
        printf("ION_IOC_SUNXI_PHYS_ADDR succes, phys_addr 0x%08x, size 0x%08x\n",
phys_data.phys_addr, phys_data.size);
    ...
```

# 7. 打印 ion 内存状态

## 7.1. 功能

ion 内存的频繁申请和释放过程中，会导致碎片产生，为了了解碎片化情况，增加了打印内存状态的功能.

打印信息类似:

```
sunxi_ion_dump_mem(259): memory 0x43100000~0x4b100000, layout(+: free, -: busy, unit: 0x00010000bytes):
-------+++-----++++----------++++++++
```

其中"+"表述空闲内存，"-"表述被占用的内存，即已经被分配的内存，每个"+/-"默认代表 64K 空间，按 memory 由低到高排列; ion 预留区间物理地址范围: 0x43100000 ~ 0x4b10000.

若需要修改"+/-"代表的大小，可以修改 dump_unit 节点值，比如:

```
echo 0x100000 > /sys/module/ion_carveout_heap/parameters/dump_unit
```

这样，以后每次打印时，"+/-"代表 1M 空间，打印长度会减少许多. 实际可根据需要调整.

## 7.2. 内核态使用

在需要打印的地方，调用 sunxi_ion_dump_mem 函数.

```
int sunxi_ion_dump_mem(void);
```

示例:

```
#include <linux/ion_sunxi.h>

void demo(void)
{
...
        sunxi_ion_dump_mem();
...
}
```

## 7.3. 用户态使用

### 7.3.1. 命令行中使用

在 shell 界面输入:

```
cat   /proc/sunxi_ion
```

### 7.3.2. 代码中使用

通过 ION_IOC_SUNXI_DUMP 命令调用.

示例:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <time.h>
#include <sys/mman.h>
#include <sys/mman.h>
#include <asm-generic/ioctl.h>

#define ION_DEV_NAME    "/dev/ion"

/*
 * structures define from linux kernel
 */
#define SZ_64M          0x04000000
#define SZ_4M           0x00400000
#define SZ_1M           0x00100000
#define SZ_64K          0x00010000
#define ION_ALLOC_SIZE  (SZ_4M + SZ_1M - SZ_64K)
#define ION_ALLOC_ALIGN     (SZ_1M)

struct ion_allocation_data {
        size_t len;
        size_t align;
        unsigned int heap_id_mask;
        unsigned int flags;
```

```
        //struct ion_handle *handle; /* note: user space not realize ion_handle, so use void* here
*/
        void *handle;
    };

    struct ion_handle_data {
        //struct ion_handle *handle;
        void *handle;
    };

    struct ion_fd_data {
        //struct ion_handle *handle;
        void *handle;
        int fd;
    };

    struct ion_custom_data {
        unsigned int cmd;
        unsigned long arg;
    };

    enum ion_heap_type {
        ION_HEAP_TYPE_SYSTEM,
        ION_HEAP_TYPE_SYSTEM_CONTIG,
        ION_HEAP_TYPE_CARVEOUT,
        ION_HEAP_TYPE_CHUNK,
        ION_HEAP_TYPE_CUSTOM, /* must be last so device specific heaps always
                                 are at the end of this enum */
        ION_NUM_HEAPS = 16,
    };

    #define ION_IOC_MAGIC         'I'
    #define ION_IOC_ALLOC         _IOWR(ION_IOC_MAGIC, 0, struct ion_allocation_data)
    #define ION_IOC_FREE          _IOWR(ION_IOC_MAGIC, 1, struct ion_handle_data)
```

```
#define ION_IOC_MAP        _IOWR(ION_IOC_MAGIC, 2, struct ion_fd_data)
#define ION_IOC_SHARE      _IOWR(ION_IOC_MAGIC, 4, struct ion_fd_data)
#define ION_IOC_IMPORT     _IOWR(ION_IOC_MAGIC, 5, struct ion_fd_data)
#define ION_IOC_SYNC       _IOWR(ION_IOC_MAGIC, 7, struct ion_fd_data)
#define ION_IOC_CUSTOM     _IOWR(ION_IOC_MAGIC, 6, struct ion_custom_data)

#define ION_FLAG_CACHED 1        /* mappings of this buffer should be
                                    cached, ion will do cache
                                    maintenance when the buffer is
                                    mapped for dma */
#define ION_FLAG_CACHED_NEEDS_SYNC 2     /* mappings of this buffer will created
                                    at mmap time, if this is set
                                    caches must be managed manually */

typedef struct {
    long    start;
    long    end;
}sunxi_cache_range;

#define ION_IOC_SUNXI_FLUSH_RANGE        5
#define ION_IOC_SUNXI_FLUSH_ALL          6
#define ION_IOC_SUNXI_PHYS_ADDR          7
#define ION_IOC_SUNXI_DMA_COPY           8
#define ION_IOC_SUNXI_DUMP               9

#define ION_HEAP_SYSTEM_MASK       (1 << ION_HEAP_TYPE_SYSTEM)
#define ION_HEAP_SYSTEM_CONTIG_MASK      (1                              <<
ION_HEAP_TYPE_SYSTEM_CONTIG)
#define ION_HEAP_CARVEOUT_MASK        (1 << ION_HEAP_TYPE_CARVEOUT)

typedef struct {
    void *handle;
    unsigned int phys_addr;
    unsigned int size;
```

```
}sunxi_phys_data;

int test_ion()
{
    struct ion_allocation_data alloc_data;
    struct ion_fd_data fd_data;
    struct ion_handle_data handle_data;
    struct ion_custom_data custom_data;
    sunxi_cache_range range;
    sunxi_phys_data phys_data;
    void *user_addr;
    int fd, ret = 0, i;

    /* open ion device */
    fd = open(ION_DEV_NAME, O_RDONLY);
    if(fd < 0) {
        printf("err open %s\n", ION_DEV_NAME);
        return -1;
    }
    printf("open %s success\n", ION_DEV_NAME);

    /* alloc buffer */
    alloc_data.len = ION_ALLOC_SIZE;
    alloc_data.align = ION_ALLOC_ALIGN;
    alloc_data.heap_id_mask = ION_HEAP_CARVEOUT_MASK;
    alloc_data.flags = ION_FLAG_CACHED | ION_FLAG_CACHED_NEEDS_SYNC;
    ret = ioctl(fd, ION_IOC_ALLOC, &alloc_data);
    if(ret) {
        printf("ION_IOC_ALLOC err, ret %d, handle 0x%08x\n", ret, (unsigned
int)alloc_data.handle);
        goto out1;
    }
    printf("ION_IOC_ALLOC succes, handle 0x%08x\n", (unsigned int)alloc_data.handle);
```

```
        /* dump ion memory layout */
        custom_data.cmd = ION_IOC_SUNXI_DUMP;
        custom_data.arg = 0;
        printf("start dump ion memory layout:\n");
        ret = ioctl(fd, ION_IOC_CUSTOM, &custom_data);
        if(ret) {
            printf("ION_IOC_SUNXI_DUMP err, ret %d\n", ret);
            goto out1;
        }
        printf("dump ion memory success!\n");

        /* optional: get buffer phys_addr */
        custom_data.cmd = ION_IOC_SUNXI_PHYS_ADDR;
        phys_data.handle = alloc_data.handle;
        custom_data.arg = (unsigned long)&phys_data;
        ret = ioctl(fd, ION_IOC_CUSTOM, &custom_data);
        if(ret) {
            printf("ION_IOC_SUNXI_PHYS_ADDR err, ret %d\n", ret);
            goto out1;
        }
        printf("ION_IOC_SUNXI_PHYS_ADDR succes, phys_addr 0x%08x, size 0x%08x\n",
phys_data.phys_addr, phys_data.size);

        /* get dmabuf fd */
        fd_data.handle = alloc_data.handle;
        ret = ioctl(fd, ION_IOC_MAP, &fd_data);
        if(ret) {
            printf("ION_IOC_MAP err, ret %d, dmabuf fd 0x%08x\n", ret, (unsigned
int)fd_data.fd);
            goto out2;
        }
        printf("ION_IOC_MAP succes, get dmabuf fd 0x%08x\n", (unsigned int)fd_data.fd);

        /* mmap to user */
```

```
        user_addr = mmap(NULL, alloc_data.len, PROT_READ|PROT_WRITE,
MAP_SHARED, fd_data.fd, 0);
        if(MAP_FAILED == user_addr) {
            printf("mmap err, ret %d\n", (unsigned int)user_addr);
            goto out3;
        }
        printf("mmap succes, get user_addr 0x%08x\n", (unsigned int)user_addr);

        /* access buffer */
        for(i = 0; i < 100; i++)
            *((char *)user_addr + i) = rand()&0xff;
        for(i = 0; i < 100; i++)
            *((char *)user_addr + alloc_data.len - i) = rand()&0xff;
        printf("random access user buf succes\n");

        /* clean and invalid user cache */
        range.start = (unsigned long)user_addr;
        range.end = (unsigned long)user_addr + alloc_data.len;
        custom_data.cmd = ION_IOC_SUNXI_FLUSH_RANGE;
        custom_data.arg = (unsigned long)&range;
        printf("start flush user cache 0x%08x~0x%08x via
ION_IOC_SUNXI_FLUSH_RANGE\n", range.start, range.end);
        ret = ioctl(fd, ION_IOC_CUSTOM, &custom_data);
        if(ret) {
            printf("ION_IOC_CUSTOM err, ret %d\n", ret);
            goto out4;
        }
        printf("flush cache succes\n");

        /* now device can access the buffer... */

    out4:
        /* unmmap */
        ret = munmap(user_addr, alloc_data.len);
```

```
    if(ret)
        printf("munmap err, ret %d\n", ret);
    printf("munmap succes\n");
out3:
    /* close dmabuf fd */
    close(fd_data.fd);
    printf("close dmabuf fd succes\n");
out2:
    /* free buffer */
    handle_data.handle = alloc_data.handle;
    ret = ioctl(fd, ION_IOC_FREE, &handle_data);
    if(ret)
        printf("ION_IOC_FREE err, ret %d\n", ret);
    printf("ION_IOC_FREE succes\n");
out1:
    /* close ion device */
    close(fd);
    return ret;
}

int main()
{
    test_ion();
}
```

# 8. 总结

# 9. 调试问题记录