第十四章 Android 内核驱动——内存管理

14.1 Low Memory Killer

基本原理

Android 的 Low Memory Killer 是在标准 linux kernel 的 OOM 基础上修改而来的一种内存管理机制，当系统内存不足时，杀死 Bad 进程释放其内存。Bad 进程的选择标准有两个：oom_adj 和占用内存的大小。oom_adj 代表进程的优先级，数值越大，优先级越高，对应每个 oom_adj 都有一个空闲内存的阈值。Android Kernel 每隔一段时间会检查当前空闲内存是否低于某个阈值，如果是，则杀死 oom_adj 最大的 Bad 进程，如果有两个以上 Bad 进程 oom_adj 相同，则杀死其中占用内存最多的进程。

Low Memory Killer 与 OOM 的区别

OOM 即 Out of Memory 是标准 linux Kernel 的一种内存管理机制，Low Memory Killer 在它基础上作了改进：

- OOM 基于多个标准给每个进程打分，分最高的进程将被杀死；Low Memory Killer 则用 oom_adj 和占用内存的大小来选择 Bad 进程
- OOM 在内存分配不足时调用，而 Low Memory Killer 每隔一段时间就会检查，一旦发现空闲内存低于某个阈值，则杀死 Bad 进程。

Low Memory Killer 的实现

Low Memory Killer 的源代码在 drivers/staging/android/lowmemorykiller.c 中，它是通过注册 Cache Shrinker 来实现的。Cache Shrinker 是标准 linux kernel 回收内存页面的一种机制，它由内核线程 kswapd 监控，当空闲内存页面不足时，kswapd 会调用注册的 Shrinker 回调函数，来回收内存页面。

Low Memory Killer 是在模块初始化时注册 Cache Shrinker 的，代码如下：

```
static int __init lowmem_init(void){
    register_shrinker(&lowmem_shrinker); // 注册 Cache Shrinker
    return 0;
}
```

lowmem_shrinker 的定义如下：

```
static struct shrinker lowmem_shrinker = {
    .shrink = lowmem_shrink,
    .seeks = DEFAULT_SEEKS * 16
};
```

register_shrinker 会将 lowmem_shrink 加入 Shrinker List 中，被 kswapd 在遍历 Shrinker List 时调用，而 Low Memory Killer 的功能就是在 lowmem_shrink 中实现的。

lowmem_shrink 用两个数组作为选择 Bad 进程的依据，这两个数组的定义如下：

```
static int lowmem_adj[6] = {
    0,
    1,
    6,
    12,
};

static int lowmem_adj_size = 4;

static size_t lowmem_minfree[6] = {
    3*512, // 6MB
    2*1024, // 8MB
    4*1024, // 16MB
    16*1024, // 64MB
};
```

lowmem_minfree 保存空闲内存的阈值，单位是一个页面 4K，lowmem_adj 保存每个阈值对应的优先级。

lowmem_shrink 首先计算当前空闲内存的大小，如果小于某个阈值，则以该阈值对应的优先级为基准，遍历各个进程，计算每个进程占用内存的大小，找出优先级大于基准优先级的进程，在这些进程中选择优先级最大的杀死，如果优先级相同，则选择占用内存最多的进程。

lowmem_shrink 杀死进程的方法是向进程发送一个不可以忽略或阻塞的 SIGKILL 信号：

```
force_sig(SIGKILL, selected);
```

用户接口

设置空闲内存阈值的接口：/sys/module/lowmemorykiller/parameters/minfree，设置对应优先级的接口：/sys/module/lowmemorykiller/parameters/adj，设置各个进程优先级的接口：/proc/<进程 pid>/oom_adj。

Android 启动时读取的配置文件/init.rc 中定义了相应的属性供 AP 使用并有设置这些参数：

```
# Define the oom_adj values for the classes of processes that can be
# killed by the kernel.  These are used in ActivityManagerService.
setprop ro.FOREGROUND_APP_ADJ 0
setprop ro.VISIBLE_APP_ADJ 1
setprop ro.SECONDARY_SERVER_ADJ 2
setprop ro.BACKUP_APP_ADJ 2
setprop ro.HOME_APP_ADJ 4
setprop ro.HIDDEN_APP_MIN_ADJ 7
setprop ro.CONTENT_PROVIDER_ADJ 14
setprop ro.EMPTY_APP_ADJ 15
# Define the memory thresholds at which the above process classes will
# be killed.  These numbers are in pages (4k).
setprop ro.FOREGROUND_APP_MEM 1536
setprop ro.VISIBLE_APP_MEM 2048
setprop ro.SECONDARY_SERVER_MEM 4096
setprop ro.BACKUP_APP_MEM 4096
setprop ro.HOME_APP_MEM 4096
setprop ro.HIDDEN_APP_MEM 5120
setprop ro.CONTENT_PROVIDER_MEM 5632
```

```
setprop ro.EMPTY_APP_MEM 6144
# Write value must be consistent with the above properties.
# Note that the driver only supports 6 slots, so we have HOME_APP at the
# same memory level as services.
write /sys/module/lowmemorykiller/parameters/adj 0,1,2,7,14,15
write                          /sys/module/lowmemorykiller/parameters/minfree
1536,2048,4096,5120,5632,6144

# Set init its forked children's oom_adj.
write /proc/1/oom_adj -16
```

从以上设置可以看出，将 init 进程 oom_adj 设置为-16，从而保证 init 进程永远不会被杀掉。

14.2 Ashmem

基本原理

Android 的 Ashmem 是一种共享内存的机制，它基于 mmap 系统调用，不同进程可以将同一段物理内存映射到各自的虚拟地址控制，从而实现共享。

Ashmem 与 mmap 的区别

mmap 通过映射同一个普通文件实现进程间共享内存，普通文件被映射到进程地址空间后，进程可以像访问普通内存一样对文件进行访问，不必再调用 read，write 等操作。进程在映射空间对共享内存的改变并不直接写回到磁盘文件中，在调用 munmap 后才执行此操作。可以通过调用 msync 实现磁盘上文件内存与共享内存区的内容一致。

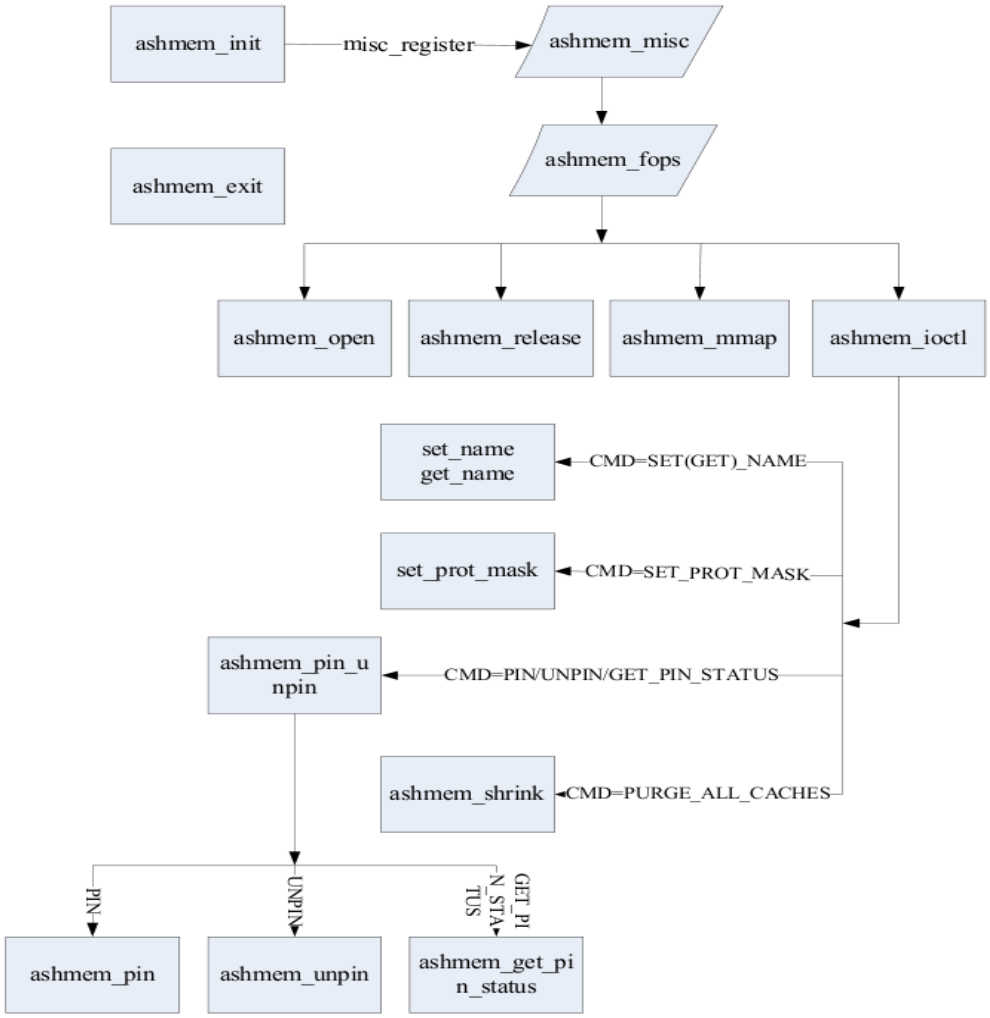
Ashmem 与 mmap 的区别在于 Ashmem 与 cache shrinker 关联起来，可以控制 cache shrinker 在适当时机回收这些共享内存。

Ashmem 的实现

Ashmem 的源代码在 mm/ashmem.c 中，它通过注册 Cache Shrinker 回收内存，通过注册 misc 设备提供 open，mmap 等接口，mmap 则通过 tmpfs 创建文件来分配内存，tmpfs 将一块内存虚拟为一个文件，这样操作共享内存就相当于操作一个文件。

Ashmem 用两个结构体 ashmem_area 和 ashmem_range 来维护分配的内存，ashmem_area 代表共享的内存区域，ashmem_range 则将这段区域以页为单位分为多个 range。ashmem_area 有个 unpinned_list 成员，挂在这个 list 上的 range 可以被回收。ashmem_range 有一个 LRU 链表，在 cache shrink 回收一个 ashmem_area 的某段内存时候，是根据 LRU 的原则来选择哪些页面优先被回收的。

Ashmem 的基本结构如下图所示。



下面依次简单分析主要函数功能：

- ashmem_init

这是 module 初始化函数，Ashmem 是作为一个模块实现的。该函数主要功能：

- 调用 kmem_cache_create 分别创建 struct ashmem_area 和 struct ashmem_range 的 slab cache
- 调用 misc_register 注册 ahsmem driver
- 调用 register_shrinker 注册 Ashmem 的 Cache Shrinker

- ashmem_open

标准 misc 设备的 open 函数。它调用 kmem_cache_zalloc 分配一个 ashmem_area，并初始化各成员变量。

- ashmem_release

做与 ashmem_open 相反工作，释放 tmpfs 文件，ashmem_area 及其 ashmem_range。

- ashmem_mmap

mmap 操作，主要就是调用 shmem_file_setup 从 tmpfs 文件系统中创建一个文件（实际上就

是一段 RAM）给 ashmem_area 用，该文件代表着这段被共享的内存。Ashmem 真正实现进程共享内存的机制是靠 shmem 这个 linux 标准机制提供的。

- ashmem_shrink

即 Ashmem 的 cache shrink 函数。它被 mm/vmscan.c：：shrink_slab 调用，或者被用户的 ioctl 命令调用。这个函数从 LRU 链表上回收指定数目的 unpinned ashmem_range。

- ashmem_ioctl

这个函数提供 ioctl 接口，它实现了如下命令：

序号	命令	功能	调用的内部函数
1	ASHMEM_SET_NAME	设置 ashmem_area->name	set_name
2	ASHMEM_GET_NAME	获取 ashmem_area->name	get_name
3	ASHMEM_SET_SIZE	设置 ashmem_area->size	
4	ASHMEM_GET_SIZE	获取 ashmem_area->size	
5	ASHMEM_SET_PROT_MASK	设置 ashmem_area->prot_mask	set_prot_mask
6	ASHMEM_GET_PROT_MASK	获取 ashmem_area->prot_mask	
7	ASHMEM_PIN	Pin 一段 range	ashmem_pin_unpin→ashmem_pin
8	ASHMEM_UNPIN	unpin 一段 range	ashmem_pin_unpin→ashmem_unpin
9	ASHMEM_GET_PIN_STATUS	获取一个 range 是否被 pin	ashmem_pin_unpin→ashmem_get_pin_status
10	ASHMEM_PURGE_ALL_CACHES	Purge 一个 ashmem_area 里的所有 ashmem_range	ashmem_shrink

- ashmem_unpin

unpin 一段内存。实现的方法很简单，就是分配一个 ashmem_range，把它挂到 ashmem_area -> unpinned_list 上，并加到 LRU 链表上。

ashmem_pin

pin 一段内存，从 ashmem_area->unpinned_list 上拿下这个 ashmem_range，由此可知，被 unpin 的 range 才能被回收，pin 的 range 则不能回收。

用户接口

Ashmem 驱动创建了/dev/ashmem 设备文件，进程 A 可通过 open 打开该文件，用 ioctl 命令

ASHMEM_SET_NAME 和 ASHMEM_SET_SIZE 设置共享内存块的名字和大小，并将得到的 handle 传给 mmap，来获得共享的内存区域，进程 B 通过将相同的 handle 传给 mmap，获得同一块内存，handle 在进程间的传递可通过 Binder 来实现。

14.3 Pmem

基本原理

Android Pmem 是为了实现共享大尺寸连续物理内存而开发的一种机制，该机制对 dsp，gpu 等部件非常有用。Pmem 相当于把系统内存划分出一部分单独管理，即不被 linux mm 管理，实际上 linux mm 根本看不到这段内存。

Pmem 与 Ashmem 的区别

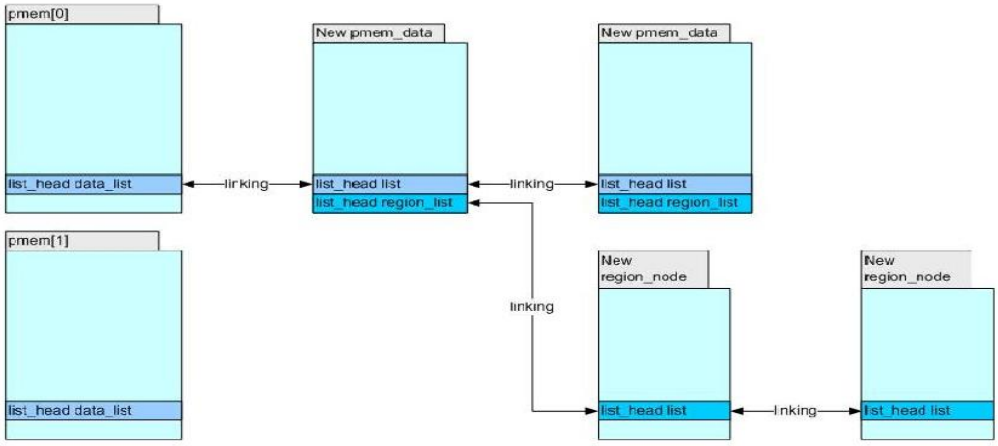
Pmem 和 Ashmem 都通过 mmap 来实现共享内存，其区别在于 Pmem 的共享区域是一段连续的物理内存，而 Ashmem 的共享区域在虚拟空间是连续的，物理内存却不一定连续。dsp 和某些设备只能工作在连续的物理内存上，这样 cpu 与 dsp 之间的通信就需要通过 Pmem 来实现。

Pmem 的实现

Pmem 的源代码在 drivers/misc/pmem.c 中，Pmem 驱动依赖于 linux 的 misc device 和 platform driver 框架，一个系统可以有多个 Pmem，默认的是最多 10 个。Pmem 暴露 4 组操作，分别是 platform driver 的 probe 和 remove 操作； misc device 的 fops 接口和 vm_ops 操作。模块初始化时会注册一个 platform driver，在之后 probe 时，创建 misc 设备文件，分配内存，完成初始化工作。

Pmem 通过 pmem_info，pmem_data，pmem_region 三个结构体维护分配的共享内存，其中 pmem_info 代表一个 Pmem 设备分配的内存块，pmem_data 代表该内存块的一个子块，pmem_region 则把每个子块分成多个区域。 pmem_data 是分配的基本单位，即每次应用层要分配一块 Pmem 内存，就会有一个 pmem_data 来表示这个被分配的内存块，实际上在 open 的时候，并不是 open 一个 pmem_info 表示的整个 Pmem 内存块，而是创建一个 pmem_data 以备使用。一个应用可以通过 ioctl 来分配 pmem_data 中的一个区域，并可以把它 map 到进程空间；并不一定每次都要分配和 map 整个 pmem_data 内存块。

上面三个数据结构的关系可以用下面的图来表示



Pmem 驱动会创建/dev/pme、/dev/adsp，实现了 pmem_open，pmem_mmap，pmem_release 和 pmem_ioctl，应用层可以通过 open，mmap，close，ioctl 来操作 Pmem 设备文件。其中 ioctl 支持的命令如下：

- PMEM_GET_PHYS 获取物理地址
- PMEM_MAP 映射一段内存
- PMEM_GET_SIZE 返回 pmem 分配的内存大小
- PMEM_UNMAP unmap 一段内存
- PMEM_ALLOCATE 分配 pmem 空间，len 是参数，如果已分配则失败
- PMEM_CONNECT 将一个 pmem file 与其他相连接
- PMEM_GET_TOTAL_SIZE 返回 pmem device 内存的大小

用户接口

一个进程首先打开 Pmem 设备，通过 ioctl(PMEM_ALLOCATE)分配内存，它 mmap 这段内存到自己的进程空间后，该进程成为 master 进程。其他进程可以重新打开这个 pmem 设 备，通过调用 ioctl(PMEM_CONNECT)将自己的 pmem_data 与 master 进程的 pmem_data 建立连接关系，这个进程就成为 client 进程。Client 进程可以通过 mmap 将 master Pmem 中的一段或全部重新映射到自己的进程空间，这样就实现了共享 Pmem 内存。如果是 GPU 或 DSP 则可以通过 ioctl(PMEM_GET_PHYS)获取物理地址进行操作。