Android的lowmemorykiller演变分析

在学习Android的lowmemorykiller机制过程中，发现从KK到L再到M有一些新的变化，因此有必要进行一下总结。在开始分析前先厘清一些基础概念，便于描述的展开和结论的形成。本文内容并没涉及太多具体实现细节，目的只是为实际的开发提供纲领性指导意见。

一、基本概念

1、PageCache：Linux内核为加快文件预取而采用的特有机制(参考引用一)，就是尽可能的把空闲内存用于缓存最近访问过的磁盘文件数据。由于会占用大量的空闲内存，在某种情况下就会导致OOM的发生。Linux内核的内存管理是很重要的部分，需要花很多时间和精力来学习(参考引用二)。

2、OOM：即“Out of Memory”的缩写，这个其实是标准Linux内核的一种内存管理机制，在内存不足或无法分配(大)内存的时候会触发此机制，来完成内存的回收。具体到代码级别，内存页面回收机制是Cache Shrinker，通过内核线程kswapd监控内存页面情况和触发Shrinker回调函数进行内存页面回收。

3、lowmemorykiller(简称LMK，下同)：是Android基于标准Linux内核OOM机制修改而来，最初的方式是通过在内核实现一个staging形式的驱动，在此驱动中注册Shrinker回调函数和实现应用策略，剩下的工作就是定时检查及判断是否需要回收(或定时检测和内核触发相结合???)。关于OOM和LMK详细信息可参考引用三。

4、CGroup：简单的说就是对进程线程使用各种系统资源的分组管理和控制，包括如CPU、IO、MEMORY，等等。

20160308补充：

关于Linux的OOM和Android的LMK的差异对比在引用八有详细介绍。

二、LMK实现演变

1、在Android的KK(或之前)版本中，LMK机制和策略都是在内核空间驱动中实现的，驱动提供两个sys文件结点给AMS用于设置触发内存回收的阀值。

2、在Android的L(或之后)版本中，引入了一个处于用户空间的守护进程lmkd(参考引用四)，LMK机制和策略的实现多了一种选择---即可在用户空间完成。但不管是旧的还是新的方式，原来由AMS完成的设置工作都通过socket方式传递参数给lmkd来实际完成。这样一来AMS与内核的耦合性进一步降低，把精力关注于activity的管理。

三、lmkd进程分析

关于lmkd的作用在修改日志中很明确：

Author: Todd Poynor <toddpoynor@google.com>

Date: Tue Jul 9 19:35:14 2013 -0700

Add lmkd low memory killer daemon

Move kernel low memory killer logic to new daemon lmkd. ActivityManager

communicates with this daemon over a named socket.

为什么导入lmkd，根据我的经验分析一来在用户空间实现策略更容易优化和管控，二来LMK驱动是Android特有的东西不太容易进入Linux内核的正式驱动列表(一直是staging形式)。在google groups的一个讨论组(android-platform)的贴子中也提到了类似的观点(引用五，需自备楼梯)：

Dual supported in L. For example N9 released with lmk due to its advanced architecture giving an edge to lmk over lmkd. Low memory killing is an important feature on all Android Devices and its tuning and optimization is a requirement to provide the best and balanced user experience. The option of either improves the tuning options of the device manufacturers. Support for lmk scales because of its inclusion upstream in the kernel. Our goal is to improve lmkd as an optimized replacement with algorithms that live well in the android ecosystem.

考虑到使用Android版本内核的差异性，lmkd进程的实现是兼容了新旧两种不同方式的：

(1) 使用LMK驱动(旧的方式)，配置阀值参数保存在驱动中，LMK策略由驱动实现，内核通过Shrinker机制触发策略执行；

(2) 不使用LMK驱动(新的方式)，配置阀值参数保存在lmkd进程中，LMK策略由lmkd进程实现。但为了触发策略的执行，必须定制内核加入memory cgroup，通过监控memory pressure event来触发LMK策略执行；

根据lmkd的实现意图，理论上来说应该使用第二种方式才对；

20160308补充： 在引用七中，对于引入LMK守护进程的意图及其相比于LMK驱动的优势(比如优雅地结束进程和回收资源，更快的响应和处理low memory状况避免linux的oom触发，正确判断cache和计算free memory)有详细的介绍。有意思的是，这个邮件是由memory pressure event的提交人Anton Vorontsov发出的，让我不禁觉得他就是lmkd的发起者。

四、memory pressure event

这个其实也是标准Linux内核在2013年4月就已经实现的功能(提交者为Anton Vorontsov，patch的标题是“memcg: Add memory.pressure\_level events”)。基于memory cgroup，内核可以分析当前内存的使用情况压力等级(low、medium、critical)，并把相关信息通过eventfd方式通知给用户空间的监听者执行相关动作(如内存回收)。

memory pressure event的实现(引用六)经过四次review(多人参与讨论，包括Linus Torvalds、Andrew Morton等大boss)才进入mainline，不得不配服国外开源环境的活跃及严谨治学态度。在这一系列讨论中也有人提及Android相关考虑，作者的想法就是实现一个通用的方式(包括对Android等移动设备支持)。

从lmkd是2013年7月提交时间来看，借助memory pressure event方式的实现，google终于如愿以标准Linux内核方式消除了staging形式LMK驱动这个“鱼刺”。至于使用memory pressure event方式的内存回收效率及实际效果如何，未进行相关测试前就不得而知了。

五、引用

1、linux的page cache策略

2、linux内存管理各文件简介

3、android Low Memory Killer介绍

4、Android——内存管理-lowmemorykiller 机制

5、why lmkd and logd are created to move log and lmk from kernel to userspace in L?

6、memcg: Add memory.pressure\_level events 1 2 3 4

7、Userspace low memory killer daemon

8、What is the difference between low memory killer and out of memory killer?

Source URL

Lmkd pressure值计算(Android lmkd计算核心)

Pressure=memory.usage\_in\_bytes\*100/memory.memsw.usage\_in\_bytes

注： memory.usage\_in\_bytes=cached+anonrss

memory.memsw.usage\_in\_bytes=cached+anonrss+swapuse

kernel会通过计算vmpressure的值当该值发生变化时（一般为low，当其由low变为medium或者有medium变为critical）会触发lmkd回调函数来执行进程清理工作。Kernel计算pressure的公式为：

pressure= [1 – (reclaimed/scanned)] \* 100

Scanned应该是固定时间段扫描的memory page个数，reclaim表示当前扫描的内存页数中可回收的memorypage个数，从公式可以看出压力值就是当前已扫描的内存中不可回收内存的比例。

然后kernel会根据得到的当前压力值来得出memory的压力状态时low还是medium或者critical：

static enumvmpressure\_levelsvmpressure\_level(unsignedlong pressure)

{

if(pressure >= vmpressure\_level\_critical) //vmpressure\_level\_critical=95

returnVMPRESSURE\_CRITICAL;

elseif (pressure >= vmpressure\_level\_med) //vmpressure\_level\_med=60

returnVMPRESSURE\_MEDIUM;

returnVMPRESSURE\_LOW;

}

Source URL

Android O&Go lmkd执行流程

我们知道android lowmemorykiller机制有两套执行方案，在N之前的版本都是采用的kernel的lowmemorykiller.c里面的方式。最近查看了Android Go的代码结构发现，Android Go采用的是native的lmkd service的方式来起到lowmemorykiller的作用。具体实现流程如下：

我们知道kernel lowmemorykiller启动杀进程的条件是file\_page-shmem-swapcache(-unevictable)的值低于AMS所预设的minfree的各档的阈值。这里如果是采用的lmkd的方式，则采用的是另外一套机制启动lmkd来杀进程。

首先讲一下lmkd的执行流程，其主要代码位于/system/core/lmkd/lmkd.c,其执行流程如下：

int main(int argc \_\_unused, char \*\*argv \_\_unused) {

struct sched\_param param = {

.sched\_priority = 1,

};

//………………

if (!init())//启动init()初始化函数

mainloop();//进入mainloop，检测是否有所监听的事件发生变化，如有，则调用event poll回调函数

ALOGI("exiting");

return 0;

static int init(void) {

struct epoll\_event epev;

int i;

int ret;

//………………

ctrl\_lfd = android\_get\_control\_socket("lmkd");//lmkd socket通信，获取AMS设置的minfree和adj的值

if (ctrl\_lfd < 0) {

ALOGE("get lmkd control socket failed");

return -1;

}

ret = listen(ctrl\_lfd, 1);

if (ret < 0) {

ALOGE("lmkd control socket listen failed (errno=%d)", errno);

return -1;

}

epev.events = EPOLLIN;

epev.data.ptr = (void \*)ctrl\_connect\_handler;//epoll callback，当socket里面有数据变化是调用此函数

if (epoll\_ctl(epollfd, EPOLL\_CTL\_ADD, ctrl\_lfd, &epev) == -1) {//将ctrl\_lfd添加到epollfd中去，当其有变化是，启东callback函数

ALOGE("epoll\_ctl for lmkd control socket failed (errno=%d)", errno);

return -1;

}

maxevents++;

has\_inkernel\_module = !access(INKERNEL\_MINFREE\_PATH, W\_OK);

use\_inkernel\_interface = has\_inkernel\_module && !is\_go\_device;//后面用来判断是否采用kernel的lowmemorykiller机制

if (use\_inkernel\_interface) {

ALOGI("Using in-kernel low memory killer interface");

} else {

ret = init\_mp\_medium();//如果不采用kernel lowmemorykiller机制，则执行此处初始化进程

ret |= init\_mp\_critical();

if (ret)

ALOGE("Kernel does not support memory pressure events or in-kernel low memory killer");

}

//………………

return 0;

}

如果使用kernel的lowmemorykiller机制，则此时监听的lmkd socket会传输AMS设置的minfree和adj值，设置函数如下：

static void cmd\_target(int ntargets, int \*params) {

//………………

for (i = 0; i < ntargets; i++) {//将参数赋给本地变量

lowmem\_minfree[i] = ntohl(\*params++);

lowmem\_adj[i] = ntohl(\*params++);

}

//…………

if (has\_inkernel\_module) {

char minfreestr[128];

char killpriostr[128];

minfreestr[0] = '\0';

killpriostr[0] = '\0';

for (i = 0; i < lowmem\_targets\_size; i++) {

char val[40];

if (i) {

strlcat(minfreestr, ",", sizeof(minfreestr));

strlcat(killpriostr, ",", sizeof(killpriostr));

}

snprintf(val, sizeof(val), "%d", use\_inkernel\_interface ? lowmem\_minfree[i] : 0);//通过use\_inkernel\_interface决定是否采用kernel lowmemorykiller机制

strlcat(minfreestr, val, sizeof(minfreestr));

snprintf(val, sizeof(val), "%d", use\_inkernel\_interface ? lowmem\_adj[i] : 0);

strlcat(killpriostr, val, sizeof(killpriostr));

}

writefilestring(INKERNEL\_MINFREE\_PATH, minfreestr);//往kernel节点里面写值

writefilestring(INKERNEL\_ADJ\_PATH, killpriostr);

}

而本文讲的是在Android Go上面采用另外的lmkd的方式来实现lowmemorykiller机制，下面是他的初始化：

static int init\_mp\_common(char \*levelstr, void \*event\_handler, bool is\_critical)

{

int mpfd;

int evfd;

int evctlfd;

char buf[256];

struct epoll\_event epev;

int ret;

int mpevfd\_index = is\_critical ? CRITICAL\_INDEX : MEDIUM\_INDEX;

mpfd = open(MEMCG\_SYSFS\_PATH "memory.pressure\_level", O\_RDONLY | O\_CLOEXEC);//memory.presure\_level句柄

if (mpfd < 0) {

ALOGI("No kernel memory.pressure\_level support (errno=%d)", errno);

goto err\_open\_mpfd;

}

evctlfd = open(MEMCG\_SYSFS\_PATH "cgroup.event\_control", O\_WRONLY | O\_CLOEXEC);//cgroup.event\_control句柄

if (evctlfd < 0) {

ALOGI("No kernel memory cgroup event control (errno=%d)", errno);

goto err\_open\_evctlfd;

}

evfd = eventfd(0, EFD\_NONBLOCK | EFD\_CLOEXEC);//创建事件描述符，后面需要启动lmkd会通过往此文件描述符发信号来实现

if (evfd < 0) {

ALOGE("eventfd failed for level %s; errno=%d", levelstr, errno);

goto err\_eventfd;

}

ret = snprintf(buf, sizeof(buf), "%d %d %s", evfd, mpfd, levelstr);//将关心的事件描述符，文件节点句柄和当前内存缺少状态写到buf中

if (ret >= (ssize\_t)sizeof(buf)) {

ALOGE("cgroup.event\_control line overflow for level %s", levelstr);

goto err;

}

ret = write(evctlfd, buf, strlen(buf) + 1);//将上面buf的内容写到cgroup.event\_control,调用这个函数会触发后面要讲到的event\_control的memcg\_write\_event\_control函数执行

if (ret == -1) {

ALOGE("cgroup.event\_control write failed for level %s; errno=%d",

levelstr, errno);

goto err;

}

epev.events = EPOLLIN;

epev.data.ptr = event\_handler;//设置回调函数

ret = epoll\_ctl(epollfd, EPOLL\_CTL\_ADD, evfd, &epev);//将当前的监听的事件添加到监听事件列表中

if (ret == -1) {

ALOGE("epoll\_ctl for level %s failed; errno=%d", levelstr, errno);

goto err;

}

maxevents++;

mpevfd[mpevfd\_index] = evfd;

return 0;

err:

close(evfd);

err\_eventfd:

close(evctlfd);

err\_open\_evctlfd:

close(mpfd);

err\_open\_mpfd:

return -1;

}

当往cgroup.event\_control函数里面写event\_fd,memory.pressure\_level,memory缺少状态时，会执行以下函数：

static ssize\_t memcg\_write\_event\_control(struct kernfs\_open\_file \*of,

char \*buf, size\_t nbytes, loff\_t off)

{

//………………

efd = simple\_strtoul(buf, &endp, 10);//也就是上文的evfd

if (\*endp != ' ')

return -EINVAL;

buf = endp + 1;

cfd = simple\_strtoul(buf, &endp, 10);//要监听的文件节点的fd

if ((\*endp != ' ') && (\*endp != '\0'))

return -EINVAL;

buf = endp + 1;

//………………

efile = fdget(efd);//获取当前fd对应的文件结构体

if (!efile.file) {

ret = -EBADF;

goto out\_kfree;

}

event->eventfd = eventfd\_ctx\_fileget(efile.file);//将当前efd赋值给event->eventfd

if (IS\_ERR(event->eventfd)) {

ret = PTR\_ERR(event->eventfd);

goto out\_put\_efile;

}

cfile = fdget(cfd);//获取cfd对应的file，也就是memory.pressure\_level

if (!cfile.file) {

ret = -EBADF;

goto out\_put\_eventfd;

}

//………………

} else if (!strcmp(name, "memory.pressure\_level")) {

event->register\_event = vmpressure\_register\_event;//event注册

event->unregister\_event = vmpressure\_unregister\_event;

//………………

ret = event->register\_event(memcg, event->eventfd, buf);执行注册函数

if (ret)

goto out\_put\_css;

//………………

spin\_lock(&memcg->event\_list\_lock);

list\_add(&event->list, &memcg->event\_list);//将当前的event添加到memcg的event\_list里面

spin\_unlock(&memcg->event\_list\_lock);

//………………

}

int vmpressure\_register\_event(struct mem\_cgroup \*memcg,

struct eventfd\_ctx \*eventfd, const char \*args)

{

//………………

ev->efd = eventfd;将上文提到的eventfd赋值给efd

ev->level = level;

mutex\_lock(&vmpr->events\_lock);

list\_add(&ev->node, &vmpr->events);//将当前ev->node添加到vmpr->event链表

mutex\_unlock(&vmpr->events\_lock);

//………………

}

以上就是lmkd的完整初始化流程，从以上我们可以看出，主要是监听memory.pressure\_level节点的变化，那么合适会触发监听的回调函数的执行呢？往下看： 在shrink\_zone中有如下操作：

static bool shrink\_zone(struct zone \*zone, struct scan\_control \*sc,

bool is\_classzone)

{

//………………

vmpressure(sc->gfp\_mask, sc->target\_mem\_cgroup,

sc->nr\_scanned - nr\_scanned,

sc->nr\_reclaimed - nr\_reclaimed);//vmpressure函数执行，往下看

//……………………

}

void vmpressure(gfp\_t gfp, struct mem\_cgroup \*memcg,

unsigned long scanned, unsigned long reclaimed)

{

//………………

schedule\_work(&vmpr->work);//启动work的执行函数vmpressure\_work\_fn

}

static void vmpressure\_work\_fn(struct work\_struct \*work)

{

//………………

vmpressure\_event(vmpr, scanned, reclaimed)

//………………

}

static bool vmpressure\_event(struct vmpressure \*vmpr,

unsigned long scanned, unsigned long reclaimed)

{

//………………

level = vmpressure\_calc\_level(scanned, reclaimed);//vmpressure level计算

list\_for\_each\_entry(ev, &vmpr->events, node) {

if (level >= ev->level) {

eventfd\_signal(ev->efd, 1);//触发lmkd回调函数开始工作

signalled = true;

}

}

//………………

}

以上就是lmkd整个执行流程，部分可能理解有误，还请多多指正。从目前的代码来看，Android Go用到了当前的lmkd的执行。

Source URL

Other reference resources

Android 6.0的lowmemorykiller机制

Android——内存管理-lowmemorykiller 机制