低内存终止守护程序

Android 低内存终止守护程序（`lmkd`），全称 Low Memory Killer Daemon，该进程可以监控运行中 Android 系统的内存状态，并通过终止最不必要的进程来应对内存压力大的问题，使系统以可接受的性能水平运行。

内存压力简介

并行运行多个进程的 Android 系统可能会遇到系统内存耗尽，需要更多内存的进程出现明显延迟的情况。内存压力是系统内存不足的一种状态，它需要 Android 通过限制或终止不必要的进程、请求进程释放非关键缓存资源等方式来释放内存（以缓解这种压力）。

压力失速信息

Android 10 及更高版本支持新的 `lmkd` 模式，它使用内核压力失速信息（PSI）监视器来检测内存压力。上游内核中的 PSI 补丁程序集（已向后移植到 4.9 和 4.14 内核）可测量由于内存不足导致任务延迟的时间。由于这些延迟会直接影响用户体验，隐藏他们确定了内存压力严重性的便捷指标。上游内核还包括 PSI 监视器，这类监视器允许特权用户空间进程（例如 `lmkd`）指定这些延迟的阈值，并在突破阈值时从内核订阅事件。

PSI 监视器与 vmpressure 信号

由于`vmpressure`信号（由内核产生，用于检测内存压力并由`lmkd`使用）通常包含大量误报，因此`lmkd`必须执行过滤以确定是否真的存在内存压力。这会导致不必要的`lmkd`唤醒并使用额外的计算资源。使用 PSI 监视器可以实现更精确的内存压力检测，并最大限度地减少过滤开销。

使用PSI监视器

如需使用 PSI 监视器（而不是 `vmpressure` 事件），请配置 `ro.lmk.use_psi` 属性。默认值为 `true`，这会以 PSI 监视器作为 `lmkd` 内存压力检测的默认机制。由于 PSI 监视器需要内核支持，因此内核必须包含 PSI 向后移植补丁程序，并在启用 PSI 支持 (`CONFIG_PSI=y`) 的情况下进行编译。

内核中 LMK 驱动程序的缺点

由于存在大量问题，Android 启动了 LMK 驱动程序，问题包括：

- 对于低内存设备，必须主动进行调整，即便如此，在处理涉及支持大文件的活跃页面缓存的工作负载时，性能比较差，性能不良会导致出现抖动，但是不会终止该进程。
- LMK 内核驱动程序一栏与可用内存限制，不会根据内存压力进行扩缩。
- 由于设计的严格性，合作伙伴通常会自定义该驱动程序，使其可以在自己的设备上使用。
- LMK 驱动程序已挂接到 Slab Shrinker API，该 API 并非为了执行繁重操作（例如搜索并终止目标）而设计，这类操作会导致 `vmscan` 进程变慢。

用户空间 lmkd

用户空间 `lmkd` 可实现与内核中的驱动程序相同的功能，但它使用现有的内核机制检测和评估内存压力。这些机制包括使用内核生成的 `vmpressure` 事件或压力失速信息（PSI）监视器来获取关于内存压力水平的通知，以及使用内存 cgroup 功能，限制那些根据进程的重要性分配的内存资源。

在 Android 10 中使用用户空间 lmkd

在 Android 9 及更高版本中，用户空间 `lmkd` 会在未检测到内核中的 LMK 驱动程序时激活。由于用户空间 `lmkd` 要求内核支持内存 cgroup，因此必须使用以下配置设置编译内核：

```
CONFIG_ANDROID_LOW_MEMORY_KILLER=n
CONFIG_MEMCG=y
CONFIG_MEMCG_SWAP=y
```

终止策略

用户空间 `lmkd` 支持基于以下各项的终止策略：`vmpressure` 事件或 PSI 监视器、其严重性以及交换利用率等其他提示。低内存设备和高性能设备的终止策略有所不同：

- 对于内存不足的设备，一般情况下，系统会选择承受较大的内存压力。
- 对于高性能设备，如果出现内存压力，则会视为异常情况，应及时修复，以免影响整体性能。

您可以使用 `ro.config.low_ram` 属性配置终止策略。如需了解详情，请参阅低 RAM 配置。

用户空间 `lmkd` 还支持一种旧模式，在该模式下，它使用与内核中的 LMK 驱动程序相同的策略（即可用内存和文件缓存阈值）做出终止决策。要启用旧模式，请将 `ro.lmk.use_minfree_levels` 属性设置为 `true`。

配置 lmkd

使用以下属性为特定设备配置 `lmkd`。

属性	使用	默认
`ro.config.low_ram`	指定设备是低内存设备还是高性能设备。	`false`
`ro.lmk.use_psi`	使用 PSI 监视器（而不是 `vmpressure` 事件）。	`true`
`ro.lmk.use_minfree_levels`	使用可用内存和文件缓存阈值来做出进程终止决策（即与内核中的 LMK 驱动程序的功能一致）。	`false`
`ro.lmk.low`	在低 `vmpressure` 水平下可被终止的进程的最低 `oom_adj` 得分。	`1001`（停用）
`ro.lmk.medium`	在中等 `vmpressure` 水平下可被终止的进程的最低 `oom_adj` 得分。	`800`（已缓存或非必要服务）
`ro.lmk.critical`	在临界 `vmpressure` 水平下可被终止的进程的最低 `oom_adj` 得分。	`0`（任何进程）
`ro.lmk.critical_upgrade`	支持升级到临界水平。	`false`
`ro.lmk.upgrade_pressure`	由于系统交换次数过多，将在该水平执行水平升级的 `mem_pressure` 上限。	`100`（停用）

属性	使用	默认
`ro.lmk.downgrade_pressure`	由于仍有足够的可用内存，将在该水平忽略 `vmpressure` 事件的 `mem_pressure` 下限。	`100`（停用）
`ro.lmk.kill_heaviest_task`	终止符合条件的最繁重任务（最佳决策）与终止符合条件的任何任务（快速决策）。	`true`
`ro.lmk.kill_timeout_ms`	从某次终止后到其他终止完成之前的持续时间（以毫秒为单位）。	`0`（停用）
`ro.lmk.debug`	启用 `lmkd` 调试日志。	`false`

注意：`mem_pressure` = 内存使用量/RAM_and_swap 使用量（以百分比的形式表示）。

`oom_adj` 值越高，代表进程越不重要

设备配置示例：

```
PRODUCT_PROPERTY_OVERRIDES += \
    ro.lmk.low=1001 \
    ro.lmk.medium=800 \
    ro.lmk.critical=0 \
    ro.lmk.critical_upgrade=false \
    ro.lmk.upgrade_pressure=100 \
    ro.lmk.downgrade_pressure=100 \
    ro.lmk.kill_heaviest_task=true
```

Android 11 中的用户空间 lmkd

Android 11 通过引入新的终止策略改进了 `lmkd`。该终止策略使用 PSI 机制来执行 Android 10 中引入的内存压力检测。Android 11 中的 `lmkd` 会根据内存资源使用情况和抖动来防止出现内存不足和性能下降。这一终止策略取代了以前的策略，可同时用于高性能设备和低内存 (Android Go) 设备。

内核要求

对于 Android 11 设备，`lmkd` 需要以下内核功能：

- 添加 PSI 补丁程序并启用 PSI（Android 通用内核 4.9、4.14 和 4.19 中提供向后移植）。
- 添加 PIDFD 支持补丁程序（Android 通用内核 4.9、4.14 和 4.19 中提供向后移植）。
- 对于低内存设备，添加内存 cgroup。

必须使用以下配置设置编译内核：

```
CONFIG_PSI=y
```

在 Android 11 中配置 lmkd

Android 11 中的内存终止策略支持下面列出的调节旋钮和默认值。这些功能在高性能设备和低内存设备上都可使用。

属性	使用	默认	
		高性能	低内存
`ro.lmk.psi_partial_stall_ms`	部分 PSI 失速阈值（以毫秒为单位），用于触发内存不足通知。如果设备收到内存压力通知的时间太晚，可以降低此值以在较早的时间触发通知。如果在不必要的情况下触发了内存压力通知，请提高此值以降低设备对噪声的敏感度。	`70`	`200`
`ro.lmk.psi_complete_stall_ms`	完全 PSI 失速阈值（以毫秒为单位），用于触发关键内存通知。如果设备收到关键内存压力通知的时间太晚，可以降低该值以在较早的时间触发通知。如果在不必要的情况下触发了关键内存压力通知，可以提高该值以降低设备对噪声的敏感度。	`700`	
`ro.lmk.thrashing_limit`	工作集 refaults 数量的最大值，占文件支持的页面缓存总大小的百分比表示。如果工作集refaults的数量超过该值，则视为系统对其页面缓存造成抖动。如果设备性能在内存压力期间受到影响，请降低该值以限制抖动。如果因抖动原因而导致设备性能不必要地降低，请提高该值以允许更多抖动。	`100`	`30`
`ro.lmk.thrashing_limit_decay`	抖动阈值衰减，表示为在系统无法恢复时（甚至是终止后）用于降低阈值的原始阈值的百分比。其实就是原始阈值的百分比。如果持续抖动导致不必要的终止，请降低该值。如果终止后对持续抖动的响应速度过慢，请提高该值。	`10`	`50`
`ro.lmk.swap_util_max`	最大交换内存量，以占可交换内存总量的百分比表示。如果交换的内存量超过此上限，则表示系统在交换了其大部分可交换内存后仍然存在压力。 当内存压力是由不可交换内存的分配导致时，就可能会发生这种情况，原因在于大部分可交换内存已经交换，所以无法通过交换来缓解这一压力。默认值为 100，这实际上会停用此检查。如果设备的性能在交换利用率较高且可用交换水平未降至 `ro.lmk.swap_free_low_percentage` 的内存压力期间受到影响，请降低该值以限制交换利用率。	`100`	`100`

以下旧的调节旋钮也可用于新的终止策略。

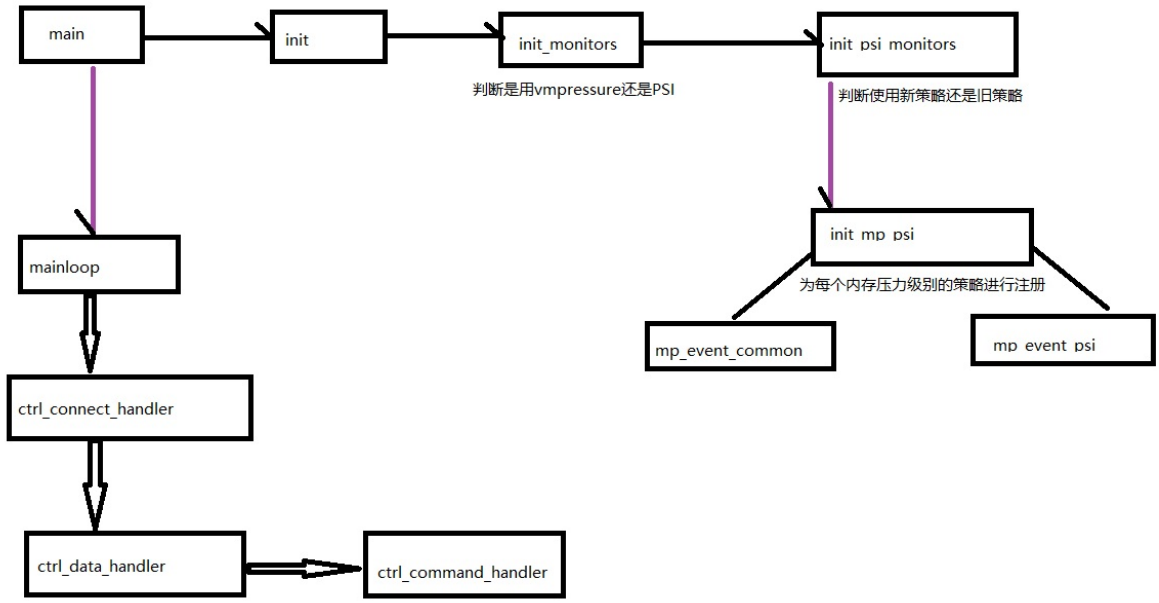
属性	使用	默认	
		高性能	低内存
	可用交换水平，可用交换区占全部交换空间的百分比。“lmkd”使用该值作为阈值来判断何时将系		

`ro.lmk.swap_free_low_percentage`	统视为交换空间不足。如果使用“lmkd”因交换空间过多而终止，请降低该百分比。如果“lmkd”终止得太晚，从而导致 OOM 终止，请提高该百分比。	`20`	`10`
`ro.lmk.debug`	这会启用“lmkd”调试日志。在调节时启用调试。	`false`	

oom_adj 是代表进程优先级，越高越容易被杀

oom_adj_score 是优先级得分，根据oom_adj 计算得出

lmkd 相关代码解读



lmkd是系统一个非常重要的服务，开机是由init进程启动，相关代码如下所示：

```
system/core/lmkd/lmkd.rc
service lmkd /system/bin/lmkd
    class core
    user lmkd
    group lmkd system readproc
    capabilities DAC_OVERRIDE KILL IPC_LOCK SYS_NICE SYS_RESOURCE BLOCK_SUSPEND
    critical
    socket lmkd seqpacket 0660 system system
    writepid /dev/cpuset/system-background/tasks
```

服务启动后，入口在system/core/lmkd/lmkd.c文件的main函数中，主要做了如下几件事：

1. 读取配置参数
2. 初始化 epoll 事件监听
3. 锁住内存页
4. 设置进程调度器
5. 循环处理事件

以下的代码解读主要是在代码中添加便于理解的注释，按照调用顺序进行排序。

main 函数解读

主要是看代码中的注释

```
int main(int argc, char **argv) {
    ...
    update_props(); //step 1，进程最初，需要获取所有的lmkd 的prop，为init 做准备

    ctx = create_android_logger(KILLINFO_LOG_TAG);

    if (!init()) { //step 2，init 处理所有核心的初始化工作
        if (!use_inkernel_interface) { //step 3，如果不再使用旧的LMK 驱动程序
            ...
            //step4，给虚拟空间上锁，防止内存交换
            if (mlockall(MCL_CURRENT | MCL_FUTURE | MCL_ONFAULT) && (errno !=
EINVAL)) {
                ALOGW("mlockall failed %s", strerror(errno));
            }

            //step 4，添加调度策略，即先进先出
            struct sched_param param = {
                    .sched_priority = 1,
            };
            if (sched_setscheduler(0, SCHED_FIFO, &param)) {
                ALOGW("set SCHED_FIFO failed %s", strerror(errno));
            }
        }

        mainloop(); //step 5，  进入循环，等待polling
    }

    android_log_destroy(&ctx);

    ALOGI("exiting");
    return 0;
}
```

基本的流程见上面代码的注释部分，可以看到 lmkd 的核心部分在step 2 和step 5。

mlockall 锁空间解析

```
 if (mlockall(MCL_CURRENT | MCL_FUTURE | MCL_ONFAULT) && (errno != EINVAL)) {
        ALOGW("mlockall failed %s", strerror(errno));
    }
```

- mlockall 函数将调用进程的全部虚拟地址空间加锁。防止出现内存交换，将该进程的地址空间交换到外存上。
- mlockall 将所有映射到进程地址空间的内存上锁。这些页包括： 代码段，数据段，栈段，共享库，共享内存，user space kernel data,memory-mapped file。当函数成功返回的时候，所有的被映射的页都在内存中。
- flags可取两个值：MCL_CURRENT, MCL_FUTURE
    - MCL_CURRENT: 表示对所有已经映射到进程地址空间的页上锁

    - MCL_FUTURE：表示对所有将来映射到进程地空间的页都上锁。
- 函数返回：成功返回0，失败返回-1
- 这个函数有两个重要的应用： real-time algorithms(实时算法) 和 high-security data processing(机密数据的处理)
- 如果进程执行了一个execve类函数，所有的锁都会被删除。
- 内存锁不会被子进程继承。
- 内存锁不会叠加，即使多次调用mlockall函数，只调用一次munlock就会解锁

init 函数解读

fd 是 file descriptor 即文件描述符，是内核为了高效访问文件建立的索引。

step 1. 创建epoll

```
    epollfd = epoll_create(MAX_EPOLL_EVENTS);
    if (epollfd == -1) {
        ALOGE("epoll_create failed (errno=%d)", errno);
        return -1;
    }
```

整个lmkd 都是依赖epoll 机制，这里创建了 9 个event：

```
/*
 * 1 ctrl listen socket, 3 ctrl data socket, 3 memory pressure levels,
 * 1 lmk events + 1 fd to wait for process death
 */
#define MAX_EPOLL_EVENTS (1 + MAX_DATA_CONN + VMPRESS_LEVEL_COUNT + 1 + 1)
```

step 2. 初始化socket lmkd

```
    ctrl_sock.sock = android_get_control_socket("lmkd");
    if (ctrl_sock.sock < 0) {
        ALOGE("get lmkd control socket failed");
        return -1;
    }

    ret = listen(ctrl_sock.sock, MAX_DATA_CONN);
    if (ret < 0) {
        ALOGE("lmkd control socket listen failed (errno=%d)", errno);
        return -1;
    }

    epev.events = EPOLLIN;
//当 socket lmkd 有客户链接时，执行下面的回调函数进行数据处理
    ctrl_sock.handler_info.handler = ctrl_connect_handler;
    epev.data.ptr = (void *)&(ctrl_sock.handler_info);
    if (epoll_ctl(epollfd, EPOLL_CTL_ADD, ctrl_sock.sock, &epev) == -1) {
        ALOGE("epoll_ctl for lmkd control socket failed (errno=%d)", errno);
        return -1;
    }
    maxevents++;
```

ctrl_sock 主要存储的是socket lmkd 的fd 和handle info，主要注意这里的ctrl_connect_handler()

该函数时socket /dev/socket/lmkd 有信息时的处理函数，lmkd 的客户端**AMS.mProcessList** 会通过socket /dev/socket/lmkd 与lmkd 进行通信。

step 3. 确定是否用LMK 驱动程序

```
 #define INKERNEL_MINFREE_PATH "/sys/module/lowmemorykiller/parameters/minfree"

   has_inkernel_module = !access(INKERNEL_MINFREE_PATH, W_OK);
   use_inkernel_interface = has_inkernel_module;
//如果还存在就使用，不存在就不用啦
```

通过函数access 确认旧的节点是否还存在，用以确认kernel 是否还在用LMK 驱动程序。

之所以有这样的处理，应该是Android 为了兼容旧版本kernel。

step 4. init_monitors

该函数是init 函数中的核心了，这里用来注册PSI 的监视器策略或者是common 的adj 策略，并将其添加到epoll 中。

```
static bool init_monitors() {
    /* Try to use psi monitor first if kernel has it */
    use_psi_monitors = property_get_bool("ro.lmk.use_psi", true) &&
        init_psi_monitors();
    /* Fall back to vmpressure */
    if (!use_psi_monitors &&
        (!init_mp_common(VMPRESS_LEVEL_LOW) ||
        !init_mp_common(VMPRESS_LEVEL_MEDIUM) ||
        !init_mp_common(VMPRESS_LEVEL_CRITICAL))) {
        ALOGE("Kernel does not support memory pressure events or in-kernel low
memory killer");
        return false;
    }
    ...
    return true;
}
```

变量use_psi_monitors 用以确认是使用 PSI 还是vmpressure

- 如果使用vmpressure，则通过init_mp_common 来初始化kill 策略；
- 如果使用PSI，则通过init_psi_monitors 来初始化kill 策略；

所以lmkd 中如果使用 PSI，要求 **ro.lmk.use_psi** 为 true。

另外，lmkd 支持旧模式的kill 策略，只要 **ro.lmk.use_new_strategy** 设为false，或者将 **ro.lmk.use_minfree_levels** 设为true（针对非低内存设备，即**ro.config.low_ram** 不为true）：

```
static bool init_psi_monitors() {
    bool use_new_strategy =
        property_get_bool("ro.lmk.use_new_strategy", low_ram_device ||
!use_minfree_levels);
```

继续深入分析init_psi_monitors：

```
static bool init_psi_monitors() {
    /*
     * When PSI is used on low-ram devices or on high-end devices without
memfree levels
     * use new kill strategy based on zone watermarks, free swap and thrashing
stats
     */
    //判断是不是新策略
    bool use_new_strategy =
        property_get_bool("ro.lmk.use_new_strategy", low_ram_device ||
!use_minfree_levels);

    //是新策略就要重设覆盖之前的阈值数组
    /* In default PSI mode override stall amounts using system properties */
    if (use_new_strategy) {
        /* Do not use low pressure level */
        psi_thresholds[VMPRESS_LEVEL_LOW].threshold_ms = 0;
        psi_thresholds[VMPRESS_LEVEL_MEDIUM].threshold_ms =
psi_partial_stall_ms;
        psi_thresholds[VMPRESS_LEVEL_CRITICAL].threshold_ms =
psi_complete_stall_ms;
    }
    //通过init_mp_psi为每个级别的策略进行注册

    if (!init_mp_psi(VMPRESS_LEVEL_LOW, use_new_strategy)) {
        return false;
    }
    if (!init_mp_psi(VMPRESS_LEVEL_MEDIUM, use_new_strategy)) {
        destroy_mp_psi(VMPRESS_LEVEL_LOW);
        return false;
    }
    if (!init_mp_psi(VMPRESS_LEVEL_CRITICAL, use_new_strategy)) {
        destroy_mp_psi(VMPRESS_LEVEL_MEDIUM);
        destroy_mp_psi(VMPRESS_LEVEL_LOW);
        return false;
    }
    return true;
}
```

函数比较简单的，最开始的变量use_new_strategy 用以确认是使用PSI 策略还是vmpressure。如果是使用PSI 策略，psi_thresholds数组中的threshold_ms 需要重新赋值为prop 指定的值（也就是说支持动态配置）。最后通过init_mp_psi 为每个级别的strategy 进行最后的注册，当然对于PSI，只有some 和full 等级，所以与level 中的medium 和 critical 分别对应。

这里的psi_thresholds 数组中threshold_ms 通过prop：

- **ro.lmk.psi_partial_stall_ms low_ram** 默认为200ms，PSI 默认为70ms；
- **ro.lmk.psi_complete_stall_ms** 默认700ms；

接下来看下init_mp_psi 到底做了些什么：

```
static bool init_mp_psi(enum vmpressure_level level, bool use_new_strategy) {
    int fd;

    /* Do not register a handler if threshold_ms is not set */
    if (!psi_thresholds[level].threshold_ms) {
        return true;
    }

    //step1: 传入level值，后期如果超过了阈值就会触发epoll
    fd = init_psi_monitor(psi_thresholds[level].stall_type,
        psi_thresholds[level].threshold_ms * US_PER_MS,
        PSI_WINDOW_SIZE_MS * US_PER_MS);

    if (fd < 0) {
        return false;
    }
    //step2: 在这里判断是新策略还是之前的旧策略，分别对应 mp_event_psi 和
mp_event_common
    vmpressure_hinfo[level].handler = use_new_strategy ? mp_event_psi :
mp_event_common;
    vmpressure_hinfo[level].data = level;
    //step3: 通过register_psi_monitor 将节点/proc/pressure/memory 添加到epoll 中监听
    if (register_psi_monitor(epollfd, fd, &vmpressure_hinfo[level]) < 0) {
        destroy_psi_monitor(fd);
        return false;
    }
    maxevents++;
    mpevfd[level] = fd;

    return true;
}
```

函数比较简单，主要分三步：

- 通过init_psi_monitor 将不同level 的值写入节点/proc/pressure/memory，后期阈值如果超过了设定就会触发一次epoll；
- 根据use_new_strategy，选择是新策略mp_event_psi，还是旧模式mp_event_common，详细的策略见第**8** 节和第**10** 节；
- 通过register_psi_monitor 将节点/proc/pressure/memory 添加到epoll 中监听；

step 5. 标记进入lmkd 流程

```
property_set("sys.lmk.reportkills", "1");
```

step 6. 其他初始化

```
    memset(killcnt_idx, KILLCNT_INVALID_IDX, sizeof(killcnt_idx));

    if (reread_file(&file_data) == NULL) {
        ALOGE("Failed to read %s: %s", file_data.filename, strerror(errno));
    }

    pidfd = TEMP_FAILURE_RETRY(sys_pidfd_open(getpid(), 0));
    if (pidfd < 0) {
        pidfd_supported = (errno != ENOSYS);
    } else {
        pidfd_supported = true;
        close(pidfd);
    }
```

这里主要是reread_file 函数，用来占坑。通过读取 /proc/zoneinfo，创建一个最大size 的buffer，后面的其他节点都直接使用该buffer，而不用再重新malloc。详细看reread_file() 中的buf 变量。

另外，通过sys_pidfd_open，确定是否支持pidfd_open 的syscall。

至此，init 基本剖析完成，主要：

- 创建epoll，用以监听 9 个event；
- 初始化socket /dev/socket/lmkd，并将其添加到epoll 中；
- 根据prop ***ro.lmk.use_psi**** 确认是否使用PSI 还是vmpressure；
- 根据prop **ro.lmk.use_new_strategy** 或者通过 prop **ro.lmk.use_minfree_levels** 和 prop **ro.config.low_ram** 使用PSI 时的新策略还是旧策略；
- 新、旧策略主要体现在mp_event_psi 和mp_event_common 处理，而本质都是通过节点 /proc/pressure/memory 获取内存压力是否达到some/full 指定来确认是否触发event；
- 后期epoll 触发主要的处理函数是mp_event_psi 或 mp_event_common；

mainloop 解读

主要是使用epoll 机制，通过epoll_wait 阻塞等待触发，如下：

```
nevents = epoll_wait(epollfd, events, maxevents, -1);
```

等待唤醒后主要做了两件事情，确认是否有connect 断开，执行handler：

```
static void mainloop(void) {
......
    while (1) {
        nevents = epoll_wait(epollfd, events, maxevents, -1);
        for (i = 0, evt = &events[0]; i < nevents; ++i, evt++) {
            if ((evt->events & EPOLLHUP) && evt->data.ptr) {
                ALOGI("lmkd data connection dropped");
                handler_info = (struct event_handler_info*)evt->data.ptr;
                ctrl_data_close(handler_info->data);
            }
        }

    /* Second pass to handle all other events */
        for (i = 0, evt = &events[0]; i < nevents; ++i, evt++) {
```

```
            if (evt->events & EPOLLERR) {
                ALOGD("EPOLLERR on event #%d", i);
            }
            if (evt->events & EPOLLHUP) {
                /* This case was handled in the first pass */
                continue;
            }
            if (evt->data.ptr) {
                handler_info = (struct event_handler_info*)evt->data.ptr;
                call_handler(handler_info, &poll_params, evt->events);
                //这里会回调ctrl_connect_handler
            }
        }

        ......
    }
}
```

```
static void ctrl_data_handler(int data, uint32_t events) {
    if (events & EPOLLIN) {
        ctrl_command_handler(data);
    }
}
```

```
// lmkd进程的客户端是ActivityManager，通过socket(dev/socket/lmkd)跟 lmkd 进行通信，
// 当有客户连接时，就会回调ctrl_connect_handler函数。
static void ctrl_connect_handler(int data __unused, uint32_t events __unused) {
.........
 // ctrl_sock上调用accept接收客户端的连接
data_sock[free_dscock_idx].sock = accept(ctrl_sock.sock, NULL, NULL);
ALOGI("lmkd data connection established");
......
/* use data to store data connection idx */
data_sock[free_dscock_idx].handler_info.data = free_dscock_idx;
// 客户连接对应的处理函数
data_sock[free_dscock_idx].handler_info.handler = ctrl_data_handler;
......
}
```

从init 中可以知道 epoll 主要监听了 9 个event，不同的event fd 对应不同的handler 处理逻辑。这些 handler 大致分为：

- **一个**socket listener fd 监听，主要是/dev/socket/lmkd，在init() 中添加到epoll；
- **三个**客户端socket data fd 的数据通信，在ctrl_connect_handler() 中添加到epoll，**在step2中有具体的回调函数：ctrl_sock.handler_info.handler = ctrl_connect_handler;**
- **三个**presurre 状态的监听，在init_psi_monitors() -> init_mp_psi() 中添加到epoll；（或者 init_mp_common 的旧策略）**后面会提到。**
- **一个**是LMK event kpoll_fd 监听，在init() 中添加到epoll，目前新的lmkd **不再使用这个监听**；
- **一个**是wait 进程death 的pid fd 监听，在start_wait_for_proc_kill() 中添加到epoll；

ctrl listener fd 的处理流程 ctrl_connect_handler

首先，在init 中得知，socket lmkd 在listen 之后会将fd 添加到epoll 中，用以监听socket 从上一节mainloop 得知epoll 触发后会调用event 对应的handler 接口，对于 lmkd，如果connect 成功后会触发ctrl_connect_handler。

AMS 中会尝试连接 lmkd，如果无法connect 会每隔 1 s 去retry，一直到connect。

frameworks/base/servcies/core/java/com/android/server/am/ProcessList.java

```
    // lmkd reconnect delay in msecs
    private static final long LMKD_RECONNECT_DELAY_MS = 1000;

    ...

    final class KillHandler extends Handler {
        ...
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case KILL_PROCESS_GROUP_MSG:
                   ...
                case LMKD_RECONNECT_MSG:
                    if (!sLmkdConnection.connect()) {
                        Slog.i(TAG,"Failed to connect to lmkd, retry after " +
                                LMKD_RECONNECT_DELAY_MS + " ms");
                        // retry after LMKD_RECONNECT_DELAY_MS

 sKillHandler.sendMessageDelayed(sKillHandler.obtainMessage(
                                KillHandler.LMKD_RECONNECT_MSG),
LMKD_RECONNECT_DELAY_MS);
                    }
                    break;
                default:
                    super.handleMessage(msg);
            }
        }
    }
```

通过代码可以得知，AMS 会通过sLmkdConnection.connect() 尝试连接lmkd，如果connect 失败会一直retry。

当LmkdConnection 连接lmkd 成功后，会进行notify，而 lmkd 会通过epoll 收到消息，并调用ctrl_connect_handler：

```
static void ctrl_connect_handler(int data __unused, uint32_t events __unused,
                                 struct polling_params *poll_params __unused) {
    struct epoll_event epev;
    int free_dscock_idx = get_free_dsock();

    if (free_dscock_idx < 0) {
        for (int i = 0; i < MAX_DATA_CONN; i++) {
            //通过这个断开连接
            ctrl_data_close(i);
        }
        free_dscock_idx = 0;
    }
```

```
    //通过accept添加连接
    data_sock[free_dscock_idx].sock = accept(ctrl_sock.sock, NULL, NULL);
    if (data_sock[free_dscock_idx].sock < 0) {
        ALOGE("lmkd control socket accept failed; errno=%d", errno);
        return;
    }
//和lmkd的连接建立
    ALOGI("lmkd data connection established");
    /* use data to store data connection idx */
    //存储连接数据的idx
    data_sock[free_dscock_idx].handler_info.data = free_dscock_idx;
    //这个就是data的处理函数
    data_sock[free_dscock_idx].handler_info.handler = ctrl_data_handler;
    data_sock[free_dscock_idx].async_event_mask = 0;
    //加上标记
    epev.events = EPOLLIN;
    epev.data.ptr = (void *)&(data_sock[free_dscock_idx].handler_info);
    if (epoll_ctl(epollfd, EPOLL_CTL_ADD, data_sock[free_dscock_idx].sock,
&epev) == -1) {       //通信失败
        ALOGE("epoll_ctl for data connection socket failed; errno=%d", errno);
        ctrl_data_close(free_dscock_idx);
        return;
    }
    maxevents++;
}
```

对于 lmkd 会提供最大 3 个的客户端连接，如果超过3个后要进行ctrl_data_close() 以断开epoll 和 socket。

如果没有超过的话，会通过accept 创建个新的data socket，并将其添加到epoll 中。

主要注意的是data 的交互函数ctrl_data_handler().

ctrl data fd 的处理流程 ctrl_data_handler

```
static void ctrl_data_handler(int data, uint32_t events,
                              struct polling_params *poll_params __unused) {
    if (events & EPOLLIN) {
        ctrl_command_handler(data);
    }
}
```

客户端建立连接后，通过socket给lmkd发送命令，命令的执行操作在函数ctrl_data_handler中处理的。当时添加到epoll 时是以EPOLLIN 添加的，所以这里接着会调用ctrl_command_handler，主要处理从ProcessList.java 中发出的几个 lmk command：

```
enum lmk_cmd {
    LMK_TARGET = 0, // 将minfree与oom_adj_score关联起来

    LMK_PROCPRIO,   //  注册进程并设置oom_adj_score
    LMK_PROCREMOVE, //  注销进程
    LMK_PROCPURGE,  //  清除所有已注册的进程
    LMK_GETKILLCNT, //  获取被杀次数
    LMK_SUBSCRIBE,  /* Subscribe for asynchronous events */
    LMK_PROCKILL,   /* Unsolicited msg to subscribed clients on proc kills */
    LMK_UPDATE_PROPS, /* Reinit properties */
};
```

lmkd 中ctrl_command_handler 函数根据cmd 解析出相应的指令，调用相应的函数

```
/* LMK_TARGET packet payload */
struct lmk_target {
    int minfree;
    int oom_adj_score;
};

/* LMK_PROCPRIO packet payload */
struct lmk_procprio {
    pid_t pid;
    uid_t uid;
    int oomadj;
};

/* LMK_PROCREMOVE packet payload */
struct lmk_procremove {
    pid_t pid;
};

/* LMK_GETKILLCNT packet payload */
struct lmk_getkillcnt {
    int min_oomadj;
    int max_oomadj;
};

static void ctrl_command_handler(int dsock_idx) {
 ......
    switch(cmd) {
        case LMK_TARGET:
             // 解析socket packet里面传过来的数据，写入lowmem_minfree和lowmem_adj两个
数组中，
             // 用于控制low memory的行为；
             // 设置sys.lmk.minfree_levels，比如属性值：
             // [sys.lmk.minfree_levels]:
[18432:0,23040:100,27648:200,85000:250,191250:900,241920:950]
              cmd_target(targets, packet);
        case LMK_PROCPRIO:
        // 设置进程的oomadj，把oomadj写入对应的节点(/proc/pid/oom_score_adj)中；
        // 将oomadj保存在一个哈希表中。
        // 哈希表 pidhash 是以 pid 做 key，proc_slot 则是把 struct proc 插入到以
oomadj 为 key 的哈希表 procadjslot_list 里面
              cmd_procprio(packet);
        case LMK_PROCREMOVE:
```

```
        //  解析socket传过来进程的pid,
        // 通过pid_remove 把这个 pid 对应的 struct proc 从 pidhash 和
procadjslot_list 里移除
                cmd_procremove(packet);
        case LMK_PROCPURGE:
                cmd_procpurge();
        case LMK_GETKILLCNT:
                kill_cnt = cmd_getkillcnt(packet);
........
}
```

cmd_procprio

例如，进程的oom_adj_score 发生变化时，AMS 会调用setOomAdj 通知到lmkd：

frameworks/base/servcies/core/java/com/android/server/am/ProcessList.java

```
    public static void setOomAdj(int pid, int uid, int amt) {
        ...

        long start = SystemClock.elapsedRealtime();
        ByteBuffer buf = ByteBuffer.allocate(4 * 4);
        buf.putInt(LMK_PROCPRIO);
        buf.putInt(pid);
        buf.putInt(uid);
        buf.putInt(amt);
        writeLmkd(buf, null);
        long now = SystemClock.elapsedRealtime();
        ...
```

lmkd 中ctrl_command_handler 函数根据cmd 解析出LMK_PROCPRIO，最终调用cmd_procprio()：

```
    case LMK_PROCPRIO:
        /* process type field is optional for backward compatibility */
        if (nargs < 3 || nargs > 4)
            goto wronglen;
        cmd_procprio(packet, nargs, &cred);
        break;
```

---->接着来看cmd_procprio 的处理：

```
static void cmd_procprio(LMKD_CTRL_PACKET packet, int field_count, struct ucred
*cred) {
    struct proc *procp;
    char path[LINE_MAX];
    char val[20];
    int soft_limit_mult;
    struct lmk_procprio params;
    bool is_system_server;
    struct passwd *pwdrec;
    int tgid;
// AMS 中传下来的进程的oom_score_adj 写入到节点/proc/pid/oom_score_adj：
    lmkd_pack_get_procprio(packet, field_count, &params);

    ...
```

```
    snprintf(path, sizeof(path), "/proc/%d/oom_score_adj", params.pid);
    snprintf(val, sizeof(val), "%d", params.oomadj);
    if (!writefilestring(path, val, false)) {
        ALOGW("Failed to open %s; errno=%d: process %d might have been killed",
              path, errno, params.pid);
        /* If this file does not exist the process is dead. */
        return;
    }

    ...
// 查找进程是否已经存在
    procp = pid_lookup(params.pid);
    // 判断是否是新进程
    if (!procp) {
        int pidfd = -1;

        if (pidfd_supported) {
            pidfd = TEMP_FAILURE_RETRY(sys_pidfd_open(params.pid, 0));
            ...
        }
        // 是新进程，开辟空间，并且给procp赋值
        procp = static_cast<struct proc*>(calloc(1, sizeof(struct proc)));
        if (!procp) {
            // Oh, the irony.  May need to rebuild our state.
            return;
        }

        procp->pid = params.pid;
        procp->pidfd = pidfd;
        procp->uid = params.uid;
        procp->reg_pid = cred->pid;
        procp->oomadj = params.oomadj;
        // 通过proc_insert将procp插入到进程list中
        proc_insert(procp);
    } else {
        // 是从前的进程，出队，更新oomadj，再重新进队
        ...
        proc_unslot(procp);
        procp->oomadj = params.oomadj;
        proc_slot(procp)
    }
}
```

- 首先是将AMS 中传下来的进程的oom_score_adj 写入到节点/proc/pid/oom_score_adj；
- 通过pid_lookup 查找是否已经存在的进程；
- 如果是新的进程，将通过sys_pidfd_open 获取pidfd，并通过proc_insert 添加到 procadjslot_list 数组链表中；
- 如果是已经存在的进程，则更新oomadj 属性，重新添加到 procadjslot_list 数组链表中；

cmd_procremove

同上，当应用进程不再启动时，会通过 ProcessList.remove() 发送命令 LMK_PROCREMOVE 通知 lmkd，并最终调用到cmd_procremove：

```
static void cmd_procremove(LMKD_CTRL_PACKET packet, struct ucred *cred) {
    ...

    procp = pid_lookup(params.pid);
    if (!procp) {
        return;
    }

    ...

    pid_remove(params.pid);
}
```

代码比较简单，如果proc 存在，则通过pid_remove 进行移除工作。

cmd_procpurge

一般是在AMS 构造的时候会对 lmkd 进行connect，如果connect 成功，则会发命令LMK_PROCPURGE 通知lmkd 先进行环境的打扫工作，最终调用 cmd_procpurge：

```
static void cmd_procpurge(struct ucred *cred) {
    ...

    for (i = 0; i < PIDHASH_SZ; i++) {
        procp = pidhash[i];
        while (procp) {
            next = procp->pidhash_next;
            /* Purge only records created by the requestor */
            if (claim_record(procp, cred->pid)) {
                pid_remove(procp->pid);
            }
            procp = next;
        }
    }
}
```

代码比较简单，就是将所有的proc 都清理一遍。

cmd_subscribe

在AMS 通过ProcessList connect 到 lmkd 之后，会发送命令LMK_SUBSCRIBE：

frameworks/base/servcies/core/java/com/android/server/am/ProcessList.java

```
public boolean onLmkdConnect(OutputStream ostream) {
    try {
        ...
            // Subscribe for kill event notifications
            buf = ByteBuffer.allocate(4 * 2);
        buf.putInt(LMK_SUBSCRIBE);
        buf.putInt(LMK_ASYNC_EVENT_KILL);
        ostream.write(buf.array(), 0, buf.position());
    } catch (IOException ex) {
        return false;
    }
    return true;
}
```

用以接受 lmkd 在kill 进程后的通知，lmkd 在kill 进程需要根据client 是否有subscribe 决定是否通知，如果向 lmkd 发送subscribe：

```
static void cmd_subscribe(int dsock_idx, LMKD_CTRL_PACKET packet) {
    struct lmk_subscribe params;

    lmkd_pack_get_subscribe(packet, &params);
    data_sock[dsock_idx].async_event_mask |= 1 << params.evt_type;
}
```

会将对应的客户端信息数组 data_sock 中对应的async_event_mask 标记为 LMK_ASYNC_EVENT_KILL，在 lmkd kill 一个进程后会调用：

```
static void ctrl_data_write_lmk_kill_occurred(pid_t pid, uid_t uid) {
    LMKD_CTRL_PACKET packet;
    size_t len = lmkd_pack_set_prockills(packet, pid, uid);

    for (int i = 0; i < MAX_DATA_CONN; i++) {
        if (data_sock[i].sock >= 0 && data_sock[i].async_event_mask & 1 <<
LMK_ASYNC_EVENT_KILL) {
            ctrl_data_write(i, (char*)packet, len);
        }
    }
}
```

通过ctrl_data_write 通知 AMS 中的ProcessList：

frameworks/base/servcies/core/java/com/android/server/am/ProcessList.java

```
sLmkdConnection = new LmkdConnection(sKillThread.getLooper().getQueue(),
                    new LmkdConnection.LmkdConnectionListener() {
                        ...

                        @Override
                        public boolean handleUnsolicitedMessage(ByteBuffer
dataReceived,
                                int receivedLen) {
                            ...
                        }
```

cmd_target

从ProcessList.java 中得知在ProcessList 构造时会初始化一次，另外会在ATMS.updateConfiguration 是会触发：

frameworks/base/services/core/java/com/android/server/wm/ActivityTaskManagerService.java

```
   public boolean updateConfiguration(Configuration values) {
        mAmInternal.enforceCallingPermission(CHANGE_CONFIGURATION,
"updateConfiguration()");

        synchronized (mGlobalLock) {
            ...

            mH.sendMessage(PooledLambda.obtainMessage(
                    ActivityManagerInternal::updateOomLevelsForDisplay,
mAmInternal,
                    DEFAULT_DISPLAY));

            ...
        }
    }
```

感兴趣的可以跟一下源码，最终会调用到ProcessList.updateOomLevels()

frameworks/base/servcies/core/java/com/android/server/am/ProcessList.java

```
    private void updateOomLevels(int displayWidth, int displayHeight, boolean
write) {
        ...

        if (write) {
            ByteBuffer buf = ByteBuffer.allocate(4 * (2 * mOomAdj.length + 1));
            buf.putInt(LMK_TARGET);
            for (int i = 0; i < mOomAdj.length; i++) {
                buf.putInt((mOomMinFree[i] * 1024)/PAGE_SIZE);
                buf.putInt(mOomAdj[i]);
            }

            writeLmkd(buf, null);
            ...
        }
    }
```

系统通过这个函数计算oom adj 的minfree，并将各个级别的 minfree和oom_adj_score 传入到 lmkd 中，至于adj minfree 的算法，后续会补充，这里继续跟lmkd 的cmd_target：

```
static void cmd_target(int ntargets, LMKD_CTRL_PACKET packet) {
    int i;
    struct lmk_target target;
    char minfree_str[PROPERTY_VALUE_MAX];
    char *pstr = minfree_str;
    char *pend = minfree_str + sizeof(minfree_str);
    static struct timespec last_req_tm;
    struct timespec curr_tm;
```

```
    ...

    for (i = 0; i < ntargets; i++) {
        lmkd_pack_get_target(packet, i, &target);
        lowmem_minfree[i] = target.minfree;
        lowmem_adj[i] = target.oom_adj_score;

        pstr += snprintf(pstr, pend - pstr, "%d:%d,", target.minfree,
            target.oom_adj_score);
        if (pstr >= pend) {
            /* if no more space in the buffer then terminate the loop */
            pstr = pend;
            break;
        }
    }

    lowmem_targets_size = ntargets;

    /* Override the last extra comma */
    pstr[-1] = '\0';
    property_set("sys.lmk.minfree_levels", minfree_str);

    ...
}
```

代码比较简单，将minfree 和oom_adj_score 进行组装，然后将组装的字符串存入到prop sys.lmk.minfree_levels。

这里的prop 其实应该是为了后面debug 时查看的，而最终的是两个数组变量：

```
static int lowmem_adj[MAX_TARGETS];
static int lowmem_minfree[MAX_TARGETS];
```

这里是将AMS 中设置的oom adj 都存放起来，后面需要kill 进程时会根据内存的使用情况、内存的mem pressure计算出最合适的min_score_adj，然后根据这个min_score_adj，kill 所有大于此值的进程。

至此， ctrl data fd 的处理流程 ctrl_data_handler 基本已剖析完成了，主要是配合第 6 节，AMS 在构造的时候会通过ProcessList 进行相对 lmkd 的初始化，包括connect 和 lmkd kill 进程后的通知监听。

- 在AMS 初始化时connect lmkd，并发送命令LMK_PROCPURGE 进行环境清理；
- 同上，在AMS 发送完LMK_PROCPURGE 后，会紧接着发送LMK_SUBSCRIBE 用以接受 lmkd kill 进程后的通知；
- 在AMS 停掉某个进程时，会发送命令LMK_PROCREMOVE；
- 在AMS 更新oom_score_adj 时，会通过接口setOomAdj 发送命令LMK_PROCPRIO；
- 在更新oom level 时，会通过updateOomLevels 发送命令LMK_TARGET；

mp_event_common解读

在Android R 中lmkd 是支持旧模式的，在init_mp_psi 的时候，会通过之前确认的是否为new strategy 来确认最终lmkd 处理部分采用的是PSI 监视器策略还是旧模式。

总流程

```
static void mp_event_common(...) {
...
 if (meminfo_parse(&mi) < 0 || zoneinfo_parse(&zi) < 0) {
        ALOGE("Failed to get free memory!");
        return;
    }
...
 if (use_minfree_levels) {       //系统属性值，使用系统剩余的内存页和文件缓存阈值作为判断依据。
        int i;
         //other_free 表示系统可用的内存页的数目，从meminfo和zoneinfo中参数计算
        // nr_free_pages为proc/meminfo中MemFree，当前系统的空闲内存大小，是完全没有被使用的内存
        // totalreserve_pages为proc/zoneinfo中max_protection+high，其中max_protection在android中为0
        other_free = mi.field.nr_free_pages - zi.field.totalreserve_pages;
        //nr_file_pages = cached + swap_cached + buffers;有时还会有多余的页(other_file就是多余的)，需要减去
        if (mi.field.nr_file_pages > (mi.field.shmem + mi.field.unevictable + mi.field.swap_cached)) {
            //other_file 基本就等于除 tmpfs 和 unevictable 外的缓存在内存的文件所占用的 page 数
            other_file = (mi.field.nr_file_pages - mi.field.shmem -
mi.field.unevictable - mi.field.swap_cached);
        } else {
            other_file = 0;
        }   //由此计算出 other_free 和 other_file

        //遍历oomadj和minfree数组，找出other_free对应的minfree和adj，作为min_score_adj
        min_score_adj = OOM_SCORE_ADJ_MAX + 1;          //综合other_free，other_file 和 lowmem_minfree计算
        for (i = 0; i < lowmem_targets_size; i++) {
        //根据 lowmem_minfree 的值来确定 min_score_adj，oomadj小于 min_score_adj 的进程在这次回收过程中不会被杀死
            minfree = lowmem_minfree[i];
            if (other_free < minfree && other_file < minfree) {
                min_score_adj = lowmem_adj[i];
                // Adaptive LMK
                if (enable_adaptive_lmk && level == VMPRESS_LEVEL_CRITICAL && i > lowmem_targets_size-4) {
                    min_score_adj = lowmem_adj[i-1];
                }
                break;
            }
        }
        if (min_score_adj == OOM_SCORE_ADJ_MAX + 1) {
            if (debug_process_killing) {
                ALOGI("Ignore %s memory pressure event "
                    "(free memory=%ldkB, cache=%ldkB, limit=%ldkB)",
                    level_name[level], other_free * page_k, other_file * page_k,
                    (long)lowmem_minfree[lowmem_targets_size - 1] * page_k);
            }
            return;
        }
```

```
        goto do_kill;
    }
...
do_kill:
..
pages_freed = find_and_kill_process(min_score_adj, -1, NULL, &mi, &curr_tm);
..
}
```

下面是init_mp_psi 中注册的策略处理选择：

```
static bool init_mp_psi(enum vmpressure_level level, bool use_new_strategy) {
    ...

    vmpressure_hinfo[level].handler = use_new_strategy ? mp_event_psi :
mp_event_common;
    vmpressure_hinfo[level].data = level;
```

本节主要来分析下Android R 中旧模式的处理函数mp_event_common。因为代码比较多，下面分段来剖析。

step1.初始化工作

从代码中看到 Android R 中的lmkd 还支持非psi 监听器的策略，那应该还旧的 kenerl 驱动程序策略。

```
    if (!use_psi_monitors) {
        /*
         * Check all event counters from low to critical
         * and upgrade to the highest priority one. By reading
         * eventfd we also reset the event counters.
         */
        for (int lvl = VMPRESS_LEVEL_LOW; lvl < VMPRESS_LEVEL_COUNT; lvl++) {
            if (mpevfd[lvl] != -1 &&
                TEMP_FAILURE_RETRY(read(mpevfd[lvl],
                                        &evcount, sizeof(evcount))) > 0 &&
                evcount > 0 && lvl > level) {
                level = static_cast<vmpressure_level>(lvl);
            }
        }
    }
```

step2.解析meminfo

通过meminfo_parse 函数来解析节点/proc/meminfo，感兴趣可以看源码，这里主要注意的是：

```
static int meminfo_parse(union meminfo *mi) {
    ...
    mi->field.nr_file_pages = mi->field.cached + mi->field.swap_cached +
        mi->field.buffers;

    return 0;
}
```

会根据meminfo 统计nr_file_pages。

step3.解析zoneinfo

通过zoneinfo_parse 函数来解析节点/proc/zoneinfo，主要是：

```
static int zoneinfo_parse(struct zoneinfo *zi) {
    ...
    node->zone_count = zone_idx + 1;
    zi->node_count = node_idx + 1;

    /* calculate totals fields */
    for (node_idx = 0; node_idx < zi->node_count; node_idx++) {
        node = &zi->nodes[node_idx];
        for (zone_idx = 0; zone_idx < node->zone_count; zone_idx++) {
            struct zoneinfo_zone *zone = &zi->nodes[node_idx].zones[zone_idx];
            zi->totalreserve_pages += zone->max_protection + zone-
>fields.field.high;
        }
        zi->total_inactive_file += node->fields.field.nr_inactive_file;
        zi->total_active_file += node->fields.field.nr_active_file;
        zi->total_workingset_refault += node->fields.field.workingset_refault;
    }
    return 0;
}
```

显示分别解析zone 的各个node，然后再统一计算zone (水平，存放水位)的：

- totalreserve_pages;
- total_inactive_file;
- total_active_file;
- total_workingset_refault

step4.关键：找到此时的min_score_adj

```
static int lowmem_adj[MAX_TARGETS];
static int lowmem_minfree[MAX_TARGETS];
```

这两个数组变量的初始化是在 cmd_target 中从 AMS 中设置下来的。

```
    if (use_minfree_levels) {
        int i;
```

```
        other_free = mi.field.nr_free_pages - zi.totalreserve_pages;
        if (mi.field.nr_file_pages > (mi.field.shmem + mi.field.unevictable +
mi.field.swap_cached)) {
            other_file = (mi.field.nr_file_pages - mi.field.shmem -
                          mi.field.unevictable - mi.field.swap_cached);
        } else {
            other_file = 0;
        }

        min_score_adj = OOM_SCORE_ADJ_MAX + 1;
        for (i = 0; i < lowmem_targets_size; i++) {
            minfree = lowmem_minfree[i];
            if (other_free < minfree && other_file < minfree) {
                min_score_adj = lowmem_adj[i];
                break;
            }
        }

        if (min_score_adj == OOM_SCORE_ADJ_MAX + 1) {
            if (debug_process_killing) {
                ALOGI("Ignore %s memory pressure event "
                      "(free memory=%ldkB, cache=%ldkB, limit=%ldkB)",
                      level_name[level], other_free * page_k, other_file *
page_k,
                      (long)lowmem_minfree[lowmem_targets_size - 1] * page_k);
            }
            return;
        }

        goto do_kill;
    }
```

代码变量解析

1. 变量user_minfree_levels 是根据prop ro.lmk.use_minfree_levels 的值两种方式：

- 使用minfree_level （true）
- 不使用minfree_level （false）

如果是用minfree_level，则通过判断当前的other_fire 和other_file 是否满足lowmem_minfree，如果达到了水位，则会记录min_score_adj，最终去kill 符合oom_adj > min_score_adj 的进程。

2. other_free 是meminfo 中free size 转换为页数，再与zoneinfo 中**totalreserve_pages** 的差值；

```
 other_free = mi.field.nr_free_pages - zi.field.totalreserve_pages;
//other_free 表示系统可用的内存页的数目，从meminfo和zoneinfo中参数计算
// nr_free_pages为proc/meminfo中MemFree，当前系统的空闲内存大小，是完全没有被使用的内存
// totalreserve_pages为proc/zoneinfo中max_protection+high，其中max_protection在
android中为0
```

3. other_file是除去share mem、swap cache 和unevictable（不能被回收的意思） 之后的剩余page；

```
//other_file 基本就等于除 tmpfs 和 unevictable 外的缓存在内存的文件所占用的 page 数
other_file = (mi.field.nr_file_pages - mi.field.shmem - mi.field.unevictable -
mi.field.swap_cached);
```

如果 other_free 和 other_file 两者都小于 minfree 中的某个元素，证明出现了 low memory，需要记录 min_score_adj 然后进行 kill，否则认为没有触发 low memory。

如果不是使用minfree_level，测使用旧模式的策略，通过mem pressure 计算出最终的level（low、mdeium、critical），然后根据level 从level_oomadj 数组中确定最终的min_score_adj：

```
        if (!use_minfree_levels) {
            ...
            min_score_adj = level_oomadj[level];
        }
```

step5.find and kill process

不管是否使用 minfree_level，最终都会根据 min_score_adj 去 kill process

```
pages_freed = find_and_kill_process(min_score_adj, -1, NULL, &mi, &wi,
&curr_tm);
```

mi 是 meminfo，wi 是 wakeup_info。

```
static int find_and_kill_process(int min_score_adj, enum kill_reasons
kill_reason,
                                 const char *kill_desc, union meminfo *mi,
                                 struct wakeup_info *wi, struct timespec *tm) {
    int i;
    int killed_size = 0;
    bool lmk_state_change_start = false;
    bool choose_heaviest_task = kill_heaviest_task;

    for (i = OOM_SCORE_ADJ_MAX; i >= min_score_adj; i--) {
        struct proc *procp;

        if (!choose_heaviest_task && i <= PERCEPTIBLE_APP_ADJ) {
            /*
             * If we have to choose a perceptible process, choose the heaviest
one to
             * hopefully minimize the number of victims.
             */
            choose_heaviest_task = true;
        }

        while (true) {
            procp = choose_heaviest_task ?
                proc_get_heaviest(i) : proc_adj_lru(i);
            //这里是两种不同的选择方案
            if (!procp)
                break;

            killed_size = kill_one_process(procp, min_score_adj, kill_reason,
kill_desc,
                                           mi, wi, tm);
            if (killed_size >= 0) {
                ...
                break;
            }
        }
```

```
        if (killed_size) {
            break;
        }
    }

    ...
    return killed_size;
}
```

代码的逻辑很清晰，通过 mp_event_common 或者 mp_event_psi 处理后会根据内存使用情况确定 oom_adj 的限度，并根据该 omm_adj 会从 oom_score_adj_max 开始选择一个进程进行 kill 用以释放内存，而该进程 procp 的选择有两种方式（LRU 或 heaviest），确定最终的 procp 后会调用 kill_one_process() 进行最后的 kill 操作。

在选择进程的时候，会有两种方式进行选择，凭借 choose_heavest_task

有两种方式确定：

- 根据prop ***ro.lmk.kill_heaviest_task*** 的值；
- oom adj 是否为perceptible app（adj 为200），对于mp_event_psi 尤其突出这个oom adj，应该是在这里用以确定是否kill heaviest task；

如果选择heaviest 方式会从procadjslot_list 找到占用**内存最多(rss)** 的proc，其中结构体数组 procadjslot_list 是在cmd_procprio 中添加，可以去看 **cmd_proprio**。另外需要注意的是，进程proc 的内存是根据procadjslot_list 中proc 进程的pid 查看节点 **/proc/pid/statm**：

```
    snprintf(path, PATH_MAX, "/proc/%d/statm", pid);
    fd = open(path, O_RDONLY | O_CLOEXEC);
    if (fd == -1)
        return -1;

    ret = read_all(fd, line, sizeof(line) - 1);
    ...

    sscanf(line, "%d %d ", &total, &rss);
    close(fd);
    return rss;
```

如果不是选择heaviest，则使用LRU 方式：

```
            procp = choose_heaviest_task ?
                proc_get_heaviest(i) : proc_adj_lru(i);
```

这里补充下/proc/pid/statm：

```
frost:/ # cat /proc/1822/statm
3568153     52137       38465       7       0       1311617     0
field       size        resident    trs     lrs     drs         dt
```

分别对应：

> Field Content
>
> size total program size (pages) (same as **VmSize** in status)
>
> resident size of memory portions (pages) (same as **VmRSS** in status)

> trs number of pages that are 'code' (i.e. backed by a file, same as RssFile+RssShmem in status)
>
> lrs number of pages of library (always 0 on 2.6)
>
> drs number of pages of data/stack (including libs; broken, includes library text)
>
> dt number of dirty pages (always 0 on 2.6)

step6.kill one process

代码这里就不贴了，所有信息都确定了，只需要进行kill 操作即可，只不过在lmkd 中kill 进程会有一定的阻塞：

```
    if (pidfd < 0) {
        start_wait_for_proc_kill(pid);
        r = kill(pid, SIGKILL);
    } else {
        start_wait_for_proc_kill(pidfd);
        r = sys_pidfd_send_signal(pidfd, SIGKILL, NULL, 0);
    }
```

函数start_wait_for_proc_kill 参数是将要kill 的进程的pid **fd**，将其添加到epoll 中进行监听。

并通过sys_pidfd_send_signal 发送SIGKILL 信号进行kill 操作。

mp_event_psi解读

总流程

```
mp_event_psi(..) {
    //判断last_kill_pid_or_fd节点是否存在，存在则为true
    bool kill_pending = is_kill_pending();
    //进程已死或杀死超时结束，停止等待。 如果支持pidfds，并且死亡通知已经导致等待停止，这将没有影响
    stop_wait_for_proc_kill(!kill_pending);
    // 解析/proc/vmstat
    vmstat_parse(...);
      // 解析/proc/meminfo并匹配各个字段的信息，获取可用内存页信息：
    meminfo_parse(...)
    ...
  // 计算
    if (swap_free_low_percentage) {
        // 计算swap_low_threshold=SwapTotal*10/100
        //swap_free_low_percentage从ro.lmk.swap_free_low_percentage获取，默认为10
        if (!swap_low_threshold) {
            swap_low_threshold = mi.field.total_swap * swap_free_low_percentage / 100;
        }
        //当swap可用空间低于ro.lmk.swap_free_low_percentage属性定义的百分比时，设置swap_is_low =           true
        swap_is_low = mi.field.free_swap < swap_low_threshold;
    }
    // 通过判断pgscan_direct/pgscan_kswapd字段较上一次的变化，
    //确定内存回收的状态是直接回收(DIRECT_RECLAIM)还是通过swap回收(KSWAPD_RECLAIM)，
```

```
    // 如果都不是(NO_RECLAIM)，说明内存压力不大，不进行kill，否则获取thrashing值（通过判断refault页所占比例）
    if (vs.field.pgscan_direct > init_pgscan_direct) {
     ...
    }
    ...
    in_reclaim = true;
    // 解析/proc/zoneinfo并匹配相应字段信息，
    // 获取保留页的大小：zi->field.totalreserve_pages += zi->field.high;（获取可用内存）
    //并计算min/low/hight水位线，zmi->nr_free_pages - zmi->cma_free和watermarks比较
    zoneinfo_parse(...)
    calc_zone_watermarks(...);
    //判断当前所处水位
    wmark = get_lowest_watermark(&mi, &zone_mem_info);
    //根据水位线、thrashing值、压力值、swap_low值、内存回收模式等进行多种场景判断，并添加不同的kill原因
    if (cycle_after_kill && wmark <= WMARK_LOW) {
    ...
    } }else if (level >= VMPRESS_LEVEL_CRITICAL && (events != 0 || wmark <= WMARK_HIGH)) {
    ...
    }
    ...
    // 如果任意条件满足，则进行kill操作
    pages_freed = find_and_kill_process(min_score_adj, kill_reason, kill_desc, &mi,
                                        &curr_tm);
}
```

与上述提到的 mp_event_common 相对应，新策略的 event 处理时通过 mp_event_psi。接下来进行分步解析。

static 变量

```
  static int64_t init_ws_refault;
    static int64_t prev_workingset_refault;
    static int64_t base_file_lru;
    static int64_t init_pgscan_kswapd;
    static int64_t init_pgscan_direct;
    static int64_t swap_low_threshold;
    static bool killing;
    static int thrashing_limit = thrashing_limit_pct;
    static struct zone_watermarks watermarks;
    static struct timespec wmark_update_tm;
    static struct wakeup_info wi;
    static struct timespec thrashing_reset_tm;
    static int64_t prev_thrash_growth = 0;
```

整个lmkd 处理都是持续记录的，对于PSI 策略处理过程，这些static 起到了至关重要的作用。

- ***init_ws_refault*** 初始的工作集 refault 值。每次event 触发时都会重新读取/proc/vmstat 节点中部分属性值，其中就有工作集refault，读取节点后都会记录在这个变量中；

- ***prev_workingset_refault*** 上一次工作集refault 值，用以确认两次event 是否存在workingset_refault值是一样的；
- ***base_file_lru*** 从vmstat 节点读取的inactive file 和 active file 之和；
- ***init_pgscan_kswaped*** 上一次vmstat 节点中pgscan_kswaped 值，用以下一次event 时确认reclaim 状态，详细看 **step 3**；
- ***init_pgscan_direct*** 上一次vmstat 节点中pgscan_direct 值，用以下一次event 时确认reclaim 状态，与上面的init_pgscan_kswaped 组合使用，详细看 **step 3**；
- ***swap_low_threshold*** 用以记录swap 分区预留的内存大小。用以确认从 /proc/meminfo 节点中读取的free_swap 小于此预留值，详细看 **step2** 和 **step 6**；
- ***killing*** 用以记录上一次event 正在处理，已经找到process 并处于killing 状态；
- ***thrashing_limit*** PSI event处理的重要变量，用以记录抖动界限。如上面代码，正常情况下thrashing_limit 的值等同于prop ro.lmk.thrashing_limit（详细看 lmkd机制一），每一次reset thrashing时也会重置该值为prop ro.lmk.thrashing_limit。但是，当内存紧张时，短时间内可能会触发多次event，此时抖动比较厉害，抖动值thrashing 有可能会超过thrashing_limit，选择process kill后，会对该值进行衰减处理，衰减百分比为 prop ro.lmk.thrashing_limit_decay 的值，详细看 **step 7**；
- ***watermarks*** 记录水位值，没分钟都会读取/proc/zoneinfo 的水位，会记录在此变量中，详细看 **step 5**；
- ***wmark_update_tm*** 记录上一次更新水位的时间，详细看 **step 5**；
- ***wi*** 用以记录event 被wake up 的时间；
- ***thrashing_reset_tm*** 记录thrashing 值reset 的时间，详细看 **step 4**；
- ***prev_thrash_growth*** 记录两次vmstat 节点读取的workingset_refault 增长幅度，详细看setp 4；

step1.解析 vmstat 和 meminfo

```
    if (vmstat_parse(&vs) < 0) {
        ALOGE("Failed to parse vmstat!");
        return;
    }

    if (meminfo_parse(&mi) < 0) {
        ALOGE("Failed to parse meminfo!");
        return;
    }
```

step2.确定 swap 是否足够

```
    if (swap_free_low_percentage) {
        if (!swap_low_threshold) {
            swap_low_threshold = mi.field.total_swap * swap_free_low_percentage
/ 100;
        }
        swap_is_low = mi.field.free_swap < swap_low_threshold;
    }
```

变量 swap_free_low_percentage 是通过 prop **ro.lmk.swap_free_low_percentage** 来标记 swap 可预留的最低空间百分比，取值 0 ~ 100。

如果当前 free 的 swap 低于 swap 的最低空间大小，则标记 swap 处于 low 状态。

step3.确定 reclaim 状态

```
    if (vs.field.pgscan_direct > init_pgscan_direct) {
        init_pgscan_direct = vs.field.pgscan_direct;
        init_pgscan_kswapd = vs.field.pgscan_kswapd;
        reclaim = DIRECT_RECLAIM;
    } else if (vs.field.pgscan_kswapd > init_pgscan_kswapd) {
        init_pgscan_kswapd = vs.field.pgscan_kswapd;
        reclaim = KSWAPD_RECLAIM;
    } else if (vs.field.workingset_refault == prev_workingset_refault) {
        /* Device is not thrashing and not reclaiming, bail out early until we
see these stats changing*/
        goto no_kill;
    }

    prev_workingset_refault = vs.field.workingset_refault;
```

> 通过当前 pgscan_direct 和 pgscan_kswapd 与上一次对应的值进行比较，确认当前 kswapd 处于 reclaim 状态。

> 确定内存回收的状态是直接回收 (DIRECT_RECLAIM) 还是通过swap回收 (KSWAPD_RECLAIM)，如果都不是 (NO_RECLAIM)，说明内存压力不大，不进行kill，否则获取thrashing值（通过判断refault页所占比例）

> 内核在应对这两类回收的需求下，分别实现了两种不同的机制。一个是使用kswapd进程对内存进行周期检查，以保证平常状态下剩余内存尽可能够用。另一个是直接内存回收（direct page reclaim），就是当内存分配时没有空闲内存可以满足要求时，触发直接内存回收。

step4.thrashing 计算

```
    since_thrashing_reset_ms = get_time_diff_ms(&thrashing_reset_tm, &curr_tm);
    if (since_thrashing_reset_ms > THRASHING_RESET_INTERVAL_MS) {
        long windows_passed;
        /* Calculate prev_thrash_growth if we crossed
THRASHING_RESET_INTERVAL_MS */
        /* 当我们超过THRASHING_RESET_INTERVAL_MS的时候计算prev_thrash_growth。*/
        prev_thrash_growth = (vs.field.workingset_refault - init_ws_refault) *
100
                             / (base_file_lru + 1);
        windows_passed = (since_thrashing_reset_ms /
THRASHING_RESET_INTERVAL_MS);
        /*
          Decay prev_thrashing unless over-the-limit thrashing was registered in
the              window we just crossed, which means there were no eligible
processes to kill.           We preserve the counter in that case to ensure a
kill if a new eligible                  process appears.
         */
        /*
        衰减 prev_trashing 除非在我们刚刚穿过的窗口注册了超过限制的抖动，这意味着并没有合适
的进行可以            被杀，在这种情况下，我们会保留计数器以确保出现合适的进程可以将其终止。
        */
        if (windows_passed > 1 || prev_thrash_growth < thrashing_limit) {
            prev_thrash_growth >>= windows_passed;
        }

        /* Record file-backed pagecache size when crossing
THRASHING_RESET_INTERVAL_MS */
        /* 在超过THRASHING_RESET_INTERVAL_MS时，记录文件支持的缓存大小*/
```

```
        base_file_lru = vs.field.nr_inactive_file + vs.field.nr_active_file;
        init_ws_refault = vs.field.workingset_refault;
        thrashing_reset_tm = curr_tm;
        thrashing_limit = thrashing_limit_pct;
    } else {
        /* Calculate what % of the file-backed pagecache refaulted so far */
        /* 计算到目前为止文件支持的页面缓存 refault 值的百分比，refault 值就是用来反应抖动
情况的值*/
        thrashing = (vs.field.workingset_refault - init_ws_refault) * 100 /
(base_file_lru + 1);
    }
    /* Add previous cycle's decayed thrashing amount */
    thrashing += prev_thrash_growth;
```

本段代码总的来说就是重置 **thrashing** 值。从代码来看，如果距离上一次重置超过了 **THRASHING_RESET_INTERVAL_MS（**默认是1000，即1s**）**，那么thrashing 相关的值都需要重置。

主要是计算工作集refault 值占据 file-backed 页面缓存的抖动百分比：

```
(vs.field.workingset_refault - init_ws_refault) * 100 / (base_file_lru + 1);
```

vs.feild.workingset_refault 是当前的refault 值（kernel 5.9 之后改名了），init_ws_refault 是上一次的refault 值，base_file_lru 是file page（包括inactive 和active）。

有些时候计算后的oom_adj_min 却找不到大于该adj 的进程，此时需要重新找到一个虚拟的可以kill 的进程。

step5.每过一分钟计算一次水位

```
    if (watermarks.high_wmark == 0 || get_time_diff_ms(&wmark_update_tm,
&curr_tm) > 60000) {
        struct zoneinfo zi;

        if (zoneinfo_parse(&zi) < 0) {
            ALOGE("Failed to parse zoneinfo!");
            return;
        }

        calc_zone_watermarks(&zi, &watermarks);
        wmark_update_tm = curr_tm;
    }
```

通过读取 proc/zoneinfo 中的 min、low、high 水位和 protection 计算出这次的最终水位，并保存在静态结构体变量 watermarks 中，1分钟计算一次（最开始 high_wmark 为0）。

在获取到water mark 后，会确认当前触发 event 时处于什么水位：

```
wmark = get_lowest_watermark(&mi, &watermarks);

static enum zone_watermark get_lowest_watermark(union meminfo *mi,
                                                struct zone_watermarks
*watermarks)
{
    int64_t nr_free_pages = mi->field.nr_free_pages - mi->field.cma_free;

    if (nr_free_pages < watermarks->min_wmark) {
```

```
        return WMARK_MIN;
    }
    if (nr_free_pages < watermarks->low_wmark) {
        return WMARK_LOW;
    }
    if (nr_free_pages < watermarks->high_wmark) {
        return WMARK_HIGH;
    }
    return WMARK_NONE;
}
```

通过/proc/meminfo 中的nr_free_pages - cma_free 与水位进行比较。

step6.确定 kill reason 和 min_score_adj

```
  if (cycle_after_kill && wmark < WMARK_LOW) {
        /* 防止出现杀了进程但是还是不够内存的情况，大概就是回收速度比不上消耗速度
         * Prevent kills not freeing enough memory which might lead to OOM kill.
         * This might happen when a process is consuming memory faster than
reclaim can
         * free even after a kill. Mostly happens when running memory stress
tests.
         */
        kill_reason = PRESSURE_AFTER_KILL;
        strncpy(kill_desc, "min watermark is breached even after kill",
sizeof(kill_desc));
    } else if (level == VMPRESS_LEVEL_CRITICAL && events != 0) {
        /*
         * Device is too busy reclaiming memory which might lead to ANR.
         * Critical level is triggered when PSI complete stall (all tasks are
blocked because
         * of the memory congestion) breaches the configured threshold.
         */
        kill_reason = NOT_RESPONDING;
        strncpy(kill_desc, "device is not responding", sizeof(kill_desc));
    } else if (swap_is_low && thrashing > thrashing_limit_pct) {
        /* Page cache is thrashing while swap is low */
        kill_reason = LOW_SWAP_AND_THRASHING;
        snprintf(kill_desc, sizeof(kill_desc), "device is low on swap (%" PRId64
            "kB < %" PRId64 "kB) and thrashing (%" PRId64 "%%)",
            mi.field.free_swap * page_k, swap_low_threshold * page_k,
thrashing);
        /* Do not kill perceptible apps unless below min watermark or heavily
thrashing */
        if (wmark > WMARK_MIN && thrashing < thrashing_critical_pct) {
            min_score_adj = PERCEPTIBLE_APP_ADJ + 1;
        }
    } else if (swap_is_low && wmark < WMARK_HIGH) {
        /* Both free memory and swap are low */
        kill_reason = LOW_MEM_AND_SWAP;
        snprintf(kill_desc, sizeof(kill_desc), "%s watermark is breached and
swap is low (%"
            PRId64 "kB < %" PRId64 "kB)", wmark > WMARK_LOW ? "min" : "low",
            mi.field.free_swap * page_k, swap_low_threshold * page_k);
        /* Do not kill perceptible apps unless below min watermark or heavily
thrashing */
        if (wmark > WMARK_MIN && thrashing < thrashing_critical_pct) {
```

```
            min_score_adj = PERCEPTIBLE_APP_ADJ + 1;
        }
    } else if (wmark < WMARK_HIGH && thrashing > thrashing_limit) {
        /* Page cache is thrashing while memory is low */
        kill_reason = LOW_MEM_AND_THRASHING;
        snprintf(kill_desc, sizeof(kill_desc), "%s watermark is breached and
thrashing (%"
            PRId64 "%%)", wmark > WMARK_LOW ? "min" : "low", thrashing);
        cut_thrashing_limit = true;
        /* Do not kill perceptible apps unless thrashing at critical levels */
        if (thrashing < thrashing_critical_pct) {
            min_score_adj = PERCEPTIBLE_APP_ADJ + 1;
        }
    } else if (reclaim == DIRECT_RECLAIM && thrashing > thrashing_limit) {
        /* Page cache is thrashing while in direct reclaim (mostly happens on
lowram devices) */
        kill_reason = DIRECT_RECL_AND_THRASHING;
        snprintf(kill_desc, sizeof(kill_desc), "device is in direct reclaim and
thrashing (%"
            PRId64 "%%)", thrashing);
        cut_thrashing_limit = true;
        /* Do not kill perceptible apps unless thrashing at critical levels */
        if (thrashing < thrashing_critical_pct) {
            min_score_adj = PERCEPTIBLE_APP_ADJ + 1;
        }
    }
```

kill reason 大致分为：

- PRESSURE_AFTER_KILL
- NOT_RESPONDING
- LOW_SWAP_AND_THRASHING
- LOW_MEM_AND_SWAP
- LOW_MEM_AND_THRASHING
- DIRECT_RECL_AND_THRASHING

（1）状态 PRESSURE_AFTER_KILL && wmark < WMARK_LOW

此状态条件是：cycle_after_kill 为 true 表明此时还处在 killing 状态，并且水位已经低于 low 水位。此状态通常发生在 memory 压力测试中。

（2）状态 NOT_RESPONDING

此状态条件是：level == VMPRESS_LEVEL_CRITICAL && events !=0

此时内存 pressure 已经超出了 PSI complete stall，即 full 状态设定的阈值。此时设备处于拼命 reclaim memory，这有可能导致 ANR 的产生（Application Not Response）。

（3）状态 LOW_SWAP_AND_THRASHING

此状态条件是：swap_is_low && thrashing > thrashing_limit_pct

- swap_is_low 是 swap 空间已经超过底线了，这个底线是详细看**step 2**。
- thrashing 是 workingset refault 值基于 file-backed 页面缓存的抖动百分比，详情看 **step 4**。

- thrashing_limit_pct 来自 prop **ro.lmk.thrashing_limit**，对于 low ram 该值为30，否则为100。

（4）状态 LOW_MEM_AND_SWAP

此状态的条件是：swap_is_low && wmark < WMARK_HIGH

此时 swap 低于设限的阈值，free pages 处于水位 LOW 之下（也有可能处于 MIN 之下了）。

但如果水位还没有低于 MIN，并且 thrashing 没有高于 critical_pct 时，就不去终止那些可感知到的应用。

（5）状态 LOW_MEM_and_THRASHING

此状态条件是： wmark < WMARK_HIGH &&thrashing > thrashing_limit

水位出LOW 之下(有可能处于MIN)，并且抖动值已经超过 thrashing_limit。标记此时处于低水位并抖动状态。

如果抖动的值没有超过了**ro.lmk.thrashing_limit_critical** 设定的(默认为**ro.lmk.thrashing_limit** 2倍)，则不去kill perceptible 之下的进程。

（6）DIRECT_RECL_AND_THRASHING

此状态条件是：reclaim == DIRECT_RECLAIM && thrashing > thrashing_limit

当抖动大于limit 值，kswap 进入reclaim状态时，就会kill apps。

默认 kill apps 的 min_score_adj 是从0开始，有些条件不是很过分时 min_score_adj 会选择从 PERCEPTIBLE_APP_ADJ + 开始。最终根据该 min_score_adj 传入的 find_and_kill_process 找到合适的进程进行 kill。

step 7. kill 进程

```
    if (kill_reason != NONE) {
        int pages_freed = find_and_kill_process(min_score_adj, kill_reason,
kill_desc, &mi, &wi, &curr_tm);
        if (pages_freed > 0) {
            killing = true;
            if (cut_thrashing_limit) {
                /*
                 * Cut thrasing limit by thrashing_limit_decay_pct percentage of
the                      * current
                 * thrashing limit until the system stops thrashing.
                 */
                thrashing_limit = (thrashing_limit * (100 -
thrashing_limit_decay_pct)) / 100;
            }
        }
    }
```

详细看 mp_event_common 中的相应部分。

注意这里thrashing_limit 有可能衰减，因为之前的thrasing 值已经超过了thrashing_limit。这种情况一般出现在短时间连续抖动的情况。