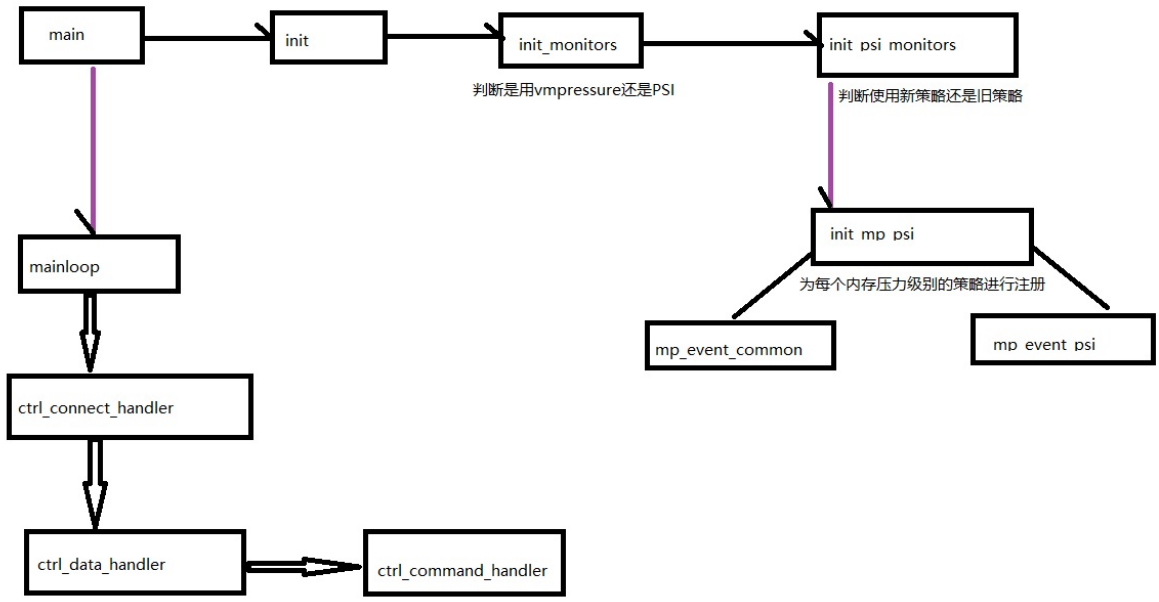
lmkd 简介

lmkd 全称是低内存终止守护程序，该进程可以监控运行中的 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。

内存压力：内存压力是系统内存不足的一种状态，他需要 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。

目前而言，在 Android 10 及以上版本中，支持新版本的 LMKD 模式，它使用的是内核压力失速信息（PSI）监视器来检测内存压力。旧版本的 Android 仍然使用 Vmpressure 信号，缺点就是在于太多误报，需要 lmkd 进行信息过滤，这样就会额外耗费计算资源。

lmkd 代码逻辑简述



main 函数

1. 在进程最初，需要获取所有 lmkd 的 prop，为 init 做准备。
2. 开始 init ，初始化完成后判断是否使用旧的内核 lmk 驱动程序
3. 给虚拟空间上锁，防止内存交换。
4. 设置调度策略，一般是先进先出
5. 进入循环等待 polling。

根据上述过程可以发现，main 函数的核心部分主要在 step 2和step 5。

init 函数

1. 创建 epoll，整个 lmkd 都是依赖 epoll 机制，这里创建了9个 event。
2. 初始化 socket lmkd，用来监听相应事件，并回调函数进行数据处

3. 根据prop ***ro.lmk.use_psi**** 确认是否使用PSI 还是vmpressure；
4. 根据prop **ro.lmk.use_new_strategy** 判断使用PSI 时的新策略还是旧策略；
5. 会执行 init_monitors 用来注册 psi 或者是旧版本的 common 的 adj 策略，并将其添加到 epoll 中。
6. 后期 epoll 触发的主要处理函数是 mp_event_psi 或 mp_event_common。

可以看到在 init 函数中主要是根据设定好的参数进行策略的选择，接下来看看不同策略的函数的主要过程和区别。

init_psi_monitor 函数

1. 最开始使用 use_new_strategy 用以确认是使用 PSI 策略还是 Vmpressure
2. 如果是 PSI 策略旧需要重设阈值，即 psi_thresholds 数组。
3. 新策略最后需要为每个压力级别分别注册新策略（psi 只用 some 和 full，与压力等级的 medium 和 critical 对应），在这之中会调用 init_mp_psi 函数。

init_mp_psi 函数

1. 传入 psi_thresholds 的 level 值给 fd，后期如果超过阈值就会触发 epoll。
2. 判断是新策略还是旧策略，分别导向 mp_event_psi 和 mp_event_common
3. 通过register_psi_mtongonitor 将节点/proc/pressure/memory 添加到epoll 中监听

至此，有关初始化的所有工作都做完啦，接下来就是具体的监听过程

mainloop 解读

在 main函数中，首先进行的是 init ，随后整个函数的关键在于 mainloop。

该函数主要是使用 epoll 机制，通过 epoll_wait 阻塞等待触发，当等待被唤醒后主要做两件事，确认是否有 connect 断开，执行 handler，一切正常一般回调 ctrl_connect_handler。

ctrl_connect_handler 函数

lmkd 进程的客户端是 ActivityManager，通过 socket 和 lmkd 进行通信，当有客户连接的时候，就会回调 ctrl_connect_handler 函数。

1. ctrl_sock 上调用 accept 接收客户端的链接
2. 接着将客户连接对应的处理函数设为 ctrl_data_handler。

ctrl_data_handler 和 ctrl_command_handler

在 ctrl_data_handler 的代码中，客户端建立连接后，通过 socket 给 lmkd 发送命令使用该代码进行处理，并且当时链接添加到 epoll 时是用 EPOLLIN 添加的，因此接下来会调用 ctrl_command_handler，处理命令。

各种命令如下：

```
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 */
};
```

1. **PROCPRIO**

- 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；

2. **procremove**

   当进程不再启动时，就会调用这个命令，判断 proc 是否存在，如果存在旧通过 pid_remove 进行移除。

3. **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

4. **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

mp_event_common 和 mp_event_psi

下面是这两个方法的主流程

mp_event_common

1. **初始化，并且加载 meminfo 和 zoneinfo 信息**
2. **找到此时的 min_score_adj**

> 如果是用minfree_level，则通过判断当前的 other_free 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。

> 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
```

> 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

3. **find and kill process**

   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。

mp_event_psi

1. **与 mp_event_common 一样，先初始化并加载一些前置信息**
2. **确定 swap 是否足够，通过判断 swap_free_low_percentage**
3. **确定 reclaim 状态**

   > 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 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）

4. **thrashing 计算**

   > 该步骤总的来说就是重置 thrashing 值，主要就是计算工作集 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）

5. **计算水位，每分钟计算一次**
6. **确定 kill reason 和 min_score_adj**
   - PRESSURE_AFTER_KILL

     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。

   - NOT_RESPONDING

     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）。

- > LOW_SWAP_AND_THRASHING
  >
  > swap_is_low 是 swap 空间已经超过底线了，即 swap_free_low_percentage
- > LOW_MEM_AND_SWAP
  >
  > 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）。
- > LOW_MEM_AND_THRASHING
  >
  > 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。
- > DIRECT_RECL_AND_THRASHING
  >
  > 当抖动大于limit 值，kswap 进入reclaim状态时，就会kill apps

min_score_adj = PERCEPTIBLE_APP_ADJ + 1

Vmpressure

Vmpressure 的计算在每次系统尝试做 do_try_to_free_pages 回收内存时进行。其计算方式非常简单：

$$(1 - reclaimed/scanned) * 100$$

也就是回收失败的内存页越多，内存压力就越大。

同时 Vmpressure 提供了通知机制，用户态或内核态程序都可以注册事件通知，应对不同等级的压力。

默认定义了三级压力：low、medium、critical。

1. low 代表正常回收
2. medium 代表中等压力，可能存在页面交换或者回写，默认值是65%
3. critical 代表内存压力很大，即将 OOM，建议应用即可采取行动，默认值是90%

Vmpressure 也有一些缺陷：

- 结果仅体现内存回收压力，不能反映系统在申请内存上的资源等待时间
- 计算周期比较粗
- 粗略的几个等级通知，无法精细化管理。

PSI

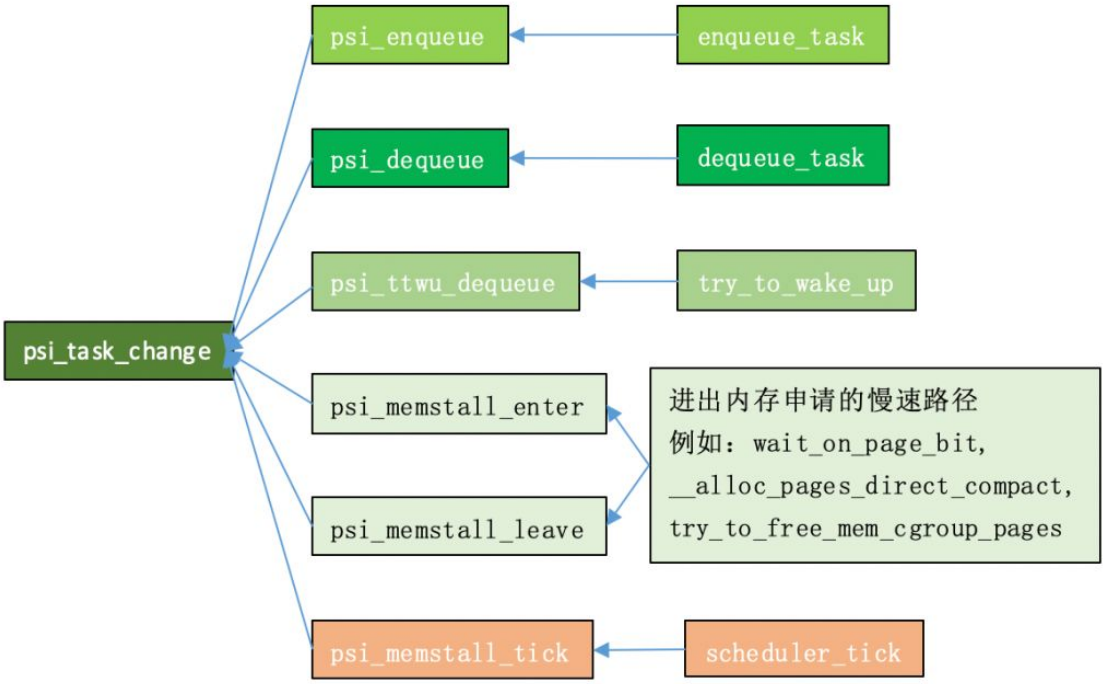
对上，PSI 模块通过文件系统结点向用户空间开放两种形态的接口。一种是系统级别的接口，即输出整个系统级别的资源压力信息。另外一种就是结合 control group，进行更精细化的分组。

对下，**PSI 模块通过在内存管理模块以及调度器模块中插桩，我们可以跟踪每一个任务由于 memory、IO 以及 CPU 资源而进入等待状态的信息。**例如系统中处于 IOwait 状态的 task 数目、由于等待 memory 资源而处于阻塞状态的任务数目。

基于 task 维度的信息，PSI 模块会将其汇聚成 PSI group 上的 per cpu 维度的时间信息。例如该 CPU 上部分任务由于等待 IO 操作而阻塞的时间长度（CPU 并没有浪费，还有其他任务在执行）。P**SI group 还会设定一个固定的周期去计算该采样周期内核的当前 PSI 值（基于该 group 的 per CPU 时间统计信息）。**

为了避免 PSI 值抖动，实际上上层应用通过系统调用获取某个 PSI group 的压力值的时候会上报近期一段时间值的滑动平均值。

psi 的关键点在于状态埋点



状态的标记主要是通过函数 psi_task_change，这个函数在任务每次进出调度队列时。都会被调用，从而准确标注任务状态。

周期性统计

第一步 get_recent_times，对每个 cpu 更新各状态的时间并统计各状态系统总时间；

第二步 calc_avgs，更新每个状态的 10s、60s、300s 三个间隔的时间占比。

> **有了 PSI 对系统资源压力的准确评估，可以做很多有意义的功能来最大化系统资源的利用，其核心思想是给 /proc/pressure/memory 的 SOME 和 FULL 设定阈值，当延时超过阈值时，触发 lmkd 进程选择进程杀死。同时，还可以结合 meminfo 的剩余内存大小来判断需要清理的程度和所选进程的优先级。**