PSI 简介

Presssure Stall Information 提供了一种评估系统资源压力的方法，系统有三个基础资源：CPU、Memory 和 IO，无论这些资源配置如何增加，似乎永远无法满足软件的需求。一旦产生资源竞争，就有可能带来延迟增大，使用户体验到卡顿。

如果没有一种相对准确的方法检测系统的资源压力程度，有两种后果，一种是资源使用者过度克制，没有充分使用系统资源；另一种是经常产生资源竞争，过度使用资源导致等待延迟过大。准确的检测方法可以帮忙资源使用者确定合适的工作量，同时也可以帮助系统定制高效的资源调度策略，最大化利用系统资源，最大化改善用户体验。

出现背景

在 PSI 出现之前，Linux也有一些资源压力的评估方法，最具代表性的是 load average 和 vmpressure。

这里主要介绍 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。

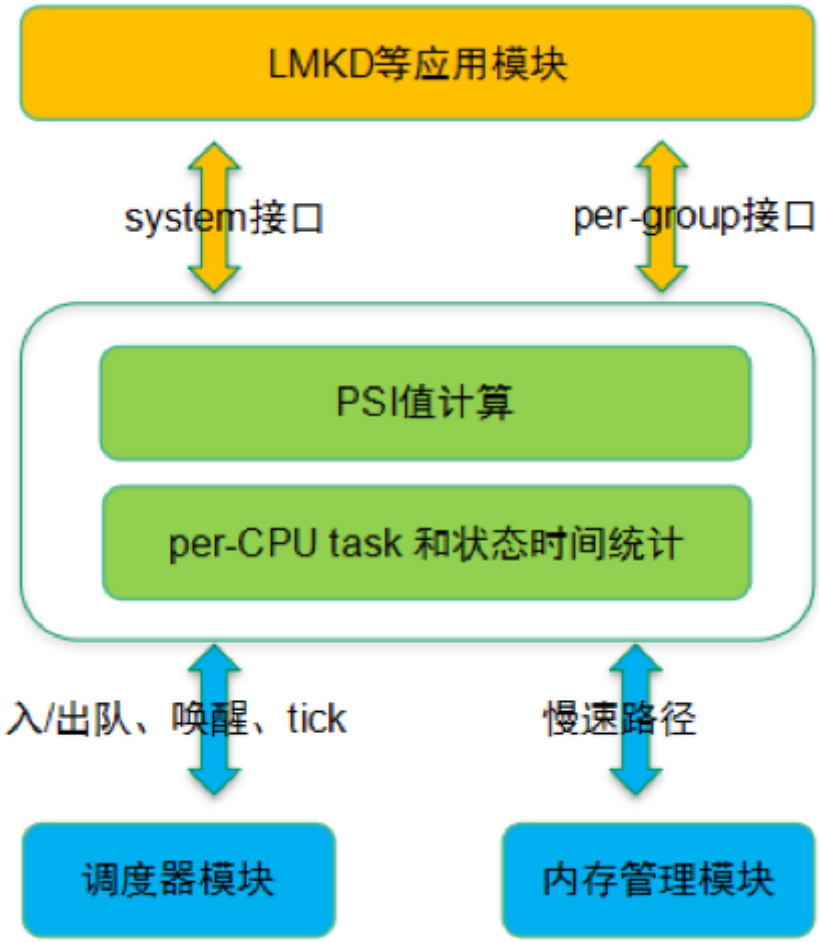
默认定义了三级压力：low、medium、critical。

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 软件架构



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）。PSI 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 用户接口定义

每类资源的压力信息都通过 proc 文件系统的独立文件来提供，路径为 /proc/pressure/ -- cpu, memory, and io，

其中 CPU 压力信息格式如下：

some avg10=2.98、avg60=2.81、avg300=1.41、total=268109926

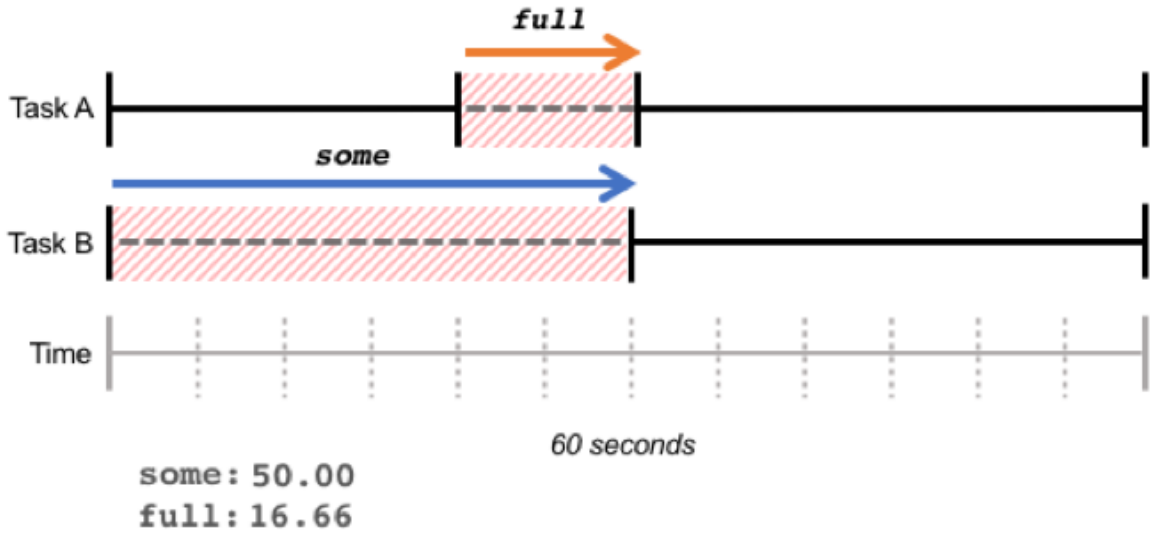
memory 和 io 格式如下：

some avg10=0.12、avg60=0.05、avg300=0.01、total=1856503

full avg10=0.12、avg60=0.05、avg300=0.01、total=1856503

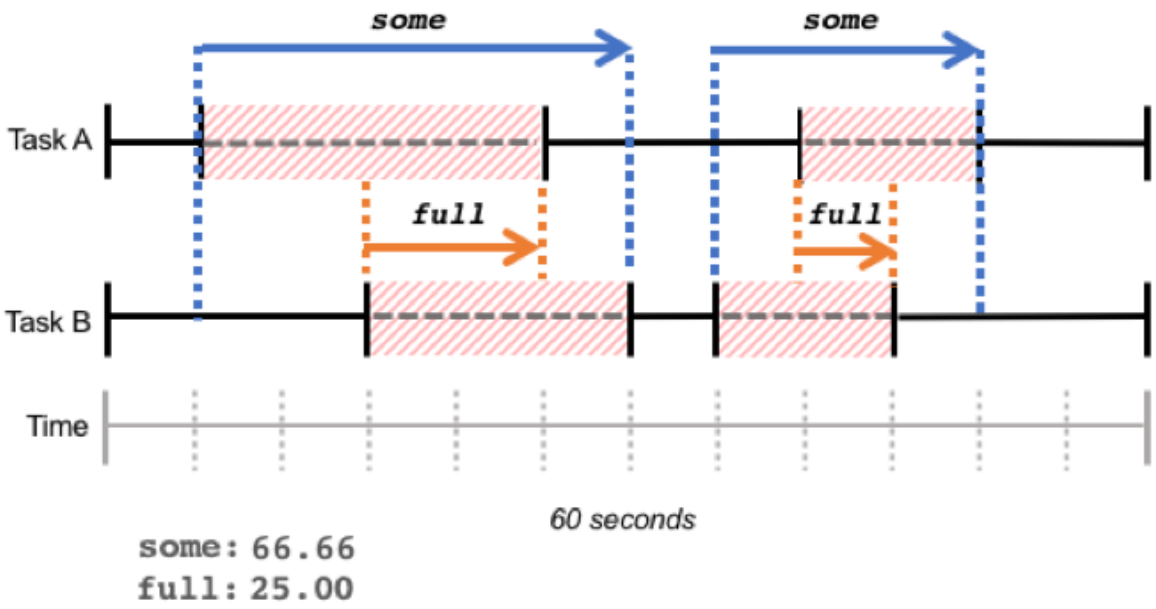
avg后面的数字分别代表10s、60s、300s 的时间周期内的阻塞时间百分比。total 是总累计时间，以毫秒为单位。

some 这一行，代表至少有一个任务在某个资源上阻塞的时间占比，full 这一行，代表所有非 idle 任务同时被阻塞的时间占比，这期间 CPU 被完全浪费，会带来严重的性能问题。我们以 IO 的 some 和 full 来举例说明，假设在60秒的时间段内，系统又两个 task，在60秒的周期内的运行情况如下图所示：



红色阴影部分表示任务由于等待 IO 资源而进入阻塞状态。Task A 和 Task B 同时阻塞的部分为 full，占比 16.66%；至少有一个任务阻塞（仅 Task B 阻塞的部分也计算入内）的部分为 some，占比 50%。

some 和 full 都是在某一时间段内阻塞时间占比的总和，阻塞时间不一定连续，如下图所示：



IO 和 memory 都有 some 和 full 两个维度，那是因为的确有可能系统中的所有任务都阻塞在 IO 或者 memory 资源，同时 CPU 进入 idle 状态。

但是 CPU 资源不可能出现这个情况：不可能全部的 runnable 的任务都等待 CPU 资源，至少有一个 runnable 任务会被调度器选中占有 CPU 资源，因此 CPU 资源没有 full 维度的 PSI 信息呈现。

通过这些阻塞占比数据，我们可以看到短期以及中长期一段时间内各种资源的压力情况，可以较精确的确定时延抖动原因，并指定对应的负载管理策略。

源码解析

初始化

step 1:

在 psi_proc_init 函数中完成 PSI 接口文件节点的创建。首先建立 proc/pressure 目录，然后3个 proc_create 函数创建了 io、memory、cpu 三个proc属性文件

```
static int __init psi_proc_init(void)
{
    if (psi_enable) {
        proc_mkdir("pressure", NULL);
        proc_create("pressure/io", 0, NULL, &psi_io_proc_ops);
        proc_create("pressure/memory", 0, NULL, &psi_memory_proc_ops);
        proc_create("pressure/cpu", 0, NULL, &psi_cpu_proc_ops);
    }
    return 0;
}
```

step 2:

在 psi_init 函数中初始化统计管理结构和更新任务的周期

```
void __init psi_init(void)
{
    if (!psi_enable) {
        static_branch_enable(&psi_disabled);
        return;
    }

    if (!cgroup_psi_enabled())
        static_branch_disable(&psi_cgroups_enabled);

    psi_period = jiffies_to_nsecs(PSI_FREQ);
    group_init(&psi_system);
}
```

我们把相关任务组成一个 group，然后针对这个任务组计算其 PSI 值。如果不支持 control group，那么实际上系统中只有一个 PSI group：

```
static DEFINE_PER_CPU(struct psi_group_cpu, system_group_pcpu);
struct psi_group psi_system = {
    .pcpu = &system_group_pcpu,
};
```

`struct psi_group` 用来定义 PSI 统计管理数据，其中包括各 CPU 状态、=周期性更新函数、更新时间戳、以及各 PSI 状态的时间记录。PSI 状态一共有六种：

```
static bool test_state(unsigned int *tasks, enum psi_states state)
{
    switch (state) {
    case PSI_IO_SOME:
        return tasks[NR_IOWAIT];
    case PSI_IO_FULL:
        return tasks[NR_IOWAIT] && !tasks[NR_RUNNING];
    case PSI_MEM_SOME:
```

```
        return tasks[NR_MEMSTALL];
    case PSI_MEM_FULL:
        return tasks[NR_MEMSTALL] && !tasks[NR_RUNNING];
    case PSI_CPU_SOME:
        return tasks[NR_RUNNING] > tasks[NR_ONCPU];
    case PSI_NONIDLE:
        return tasks[NR_IOWAIT] || tasks[NR_MEMSTALL] ||
            tasks[NR_RUNNING];
    default:
        return false;
    }
}
```

前五种状态的定义在本文上一节已经介绍，PSI_NONIDLE是指 CPU 非空闲状态，最终的时间占比是以 CPU 的非空闲时间来计算的。

状态埋点

整个 PSI 技术的核心难点其实在于如何准确捕捉到任务状态的变化，并统计状态持续时间。我们首先看看 task 维度的埋点信息。

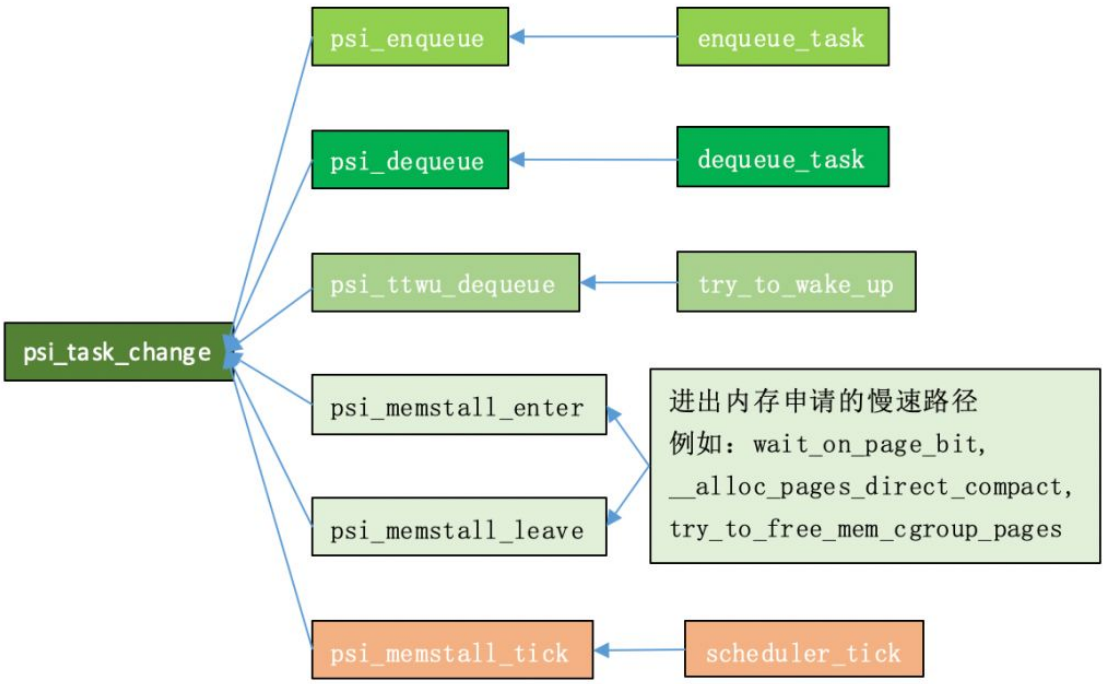
psi 在 task_struct 结构中加入了一个新成员：PSI_flags，用于标注任务所处状态，状态定义有以下几种：

```
#define TSK_IOWAIT(1 << NR_IOWAIT)

#define TSK_MEMSTALL(1 << NR_MEMSTALL)

#define TSK_RUNNING(1 << NR_RUNNING)
```

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。



其中 psi_memstall_tick 并没有任务状态的转换，只是在每个调度 tick 及时更新各状态的积累时间。

周期性统计

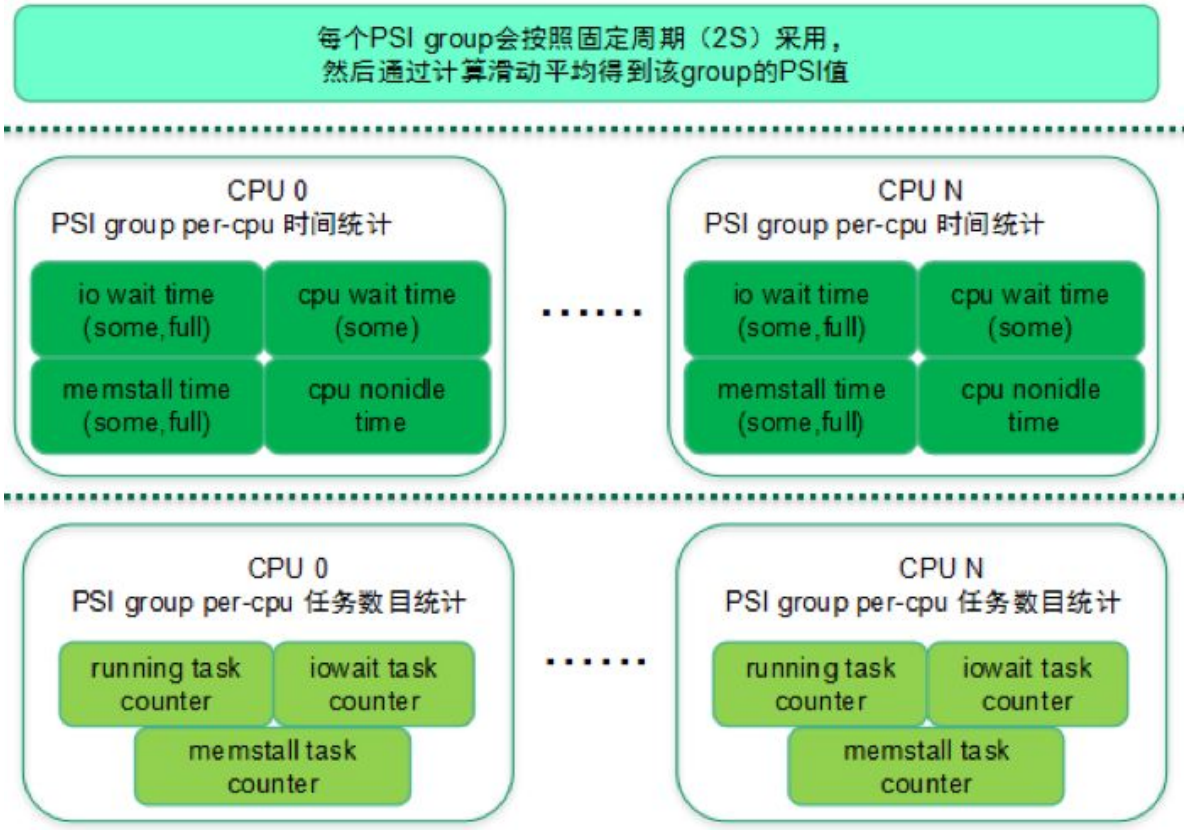
周期性的更新任务 psi_update_work 函数非常简单，更新统计数据，然后设定下一次任务唤醒的时间。周期间隔为 PSI_FREQ，2s。

更新统计数据的函数 update_stats，主要有两步：

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。

计算一个 PSI group 的 PS I值的过程示意图如下所示：



从底层看，一个 psi group 的 psi值是基于任务数目统计的，当一个任务状态发生变化的时候，首先需要遍历该任务所属的 PSI group，更新 PSI group 的task counter。

一旦 task counter 发生了变化，那么我们需要进一步更新对应 CPU 上的时间统计信息。例如 iowait task count 从0变成1，那么 SOME 维度的 io wait time 需要更新。具体的 per-CPU PSI 状态时间统计信息如下：

PSI 状态	描述
PSI_IO_SOME	该 cpu 上的 task 中至少有一个 task 处于 iowait 状态
PSI_IO_FULL	该 cpu 上的 task 中至少有一个 task 处于 iowait 状态，并且该 CPU 没有可执行的程序，进入了 idle 状态
PSI_MEM_SOME	该 cpu 上的 task 中至少有一个 task 处于 memory stall 状态
PSI_MEM_FULL	该 cpu 上的 task 中至少有一个 task 处于 memory stall 状态，并且该 CPU 没有可执行的程序，进入了 idle 状态
PSI_CPU_SOME	该 cpu 的 runqueue 中至少有一个 task 在等待调度
PSI_NONIDLE	并不是说 cpu 进入了 idle 就是真的 idle 了，实际上有些 task 在等待 io，要不是 io 资源卡住了任务执行，cpu 是不会 idle 的。因此我们可以定义 cpu idle 如下： （1）该 cpu 上等待 io 的任务为 0 （2）该 cpu 上等待 memory 的任务为 0 （3）该 cpu runqueue 上没有任务等待，当前 CPU 上也没有任务执行。 满足了上面三个条件就说明 CPU Idle 了。PSI_NONIDLE 统计的就是 cpu 没有任务执行，并且也没有任务因为资源而阻塞的时间。

完成了上面 6 种状态的时间统计之后，在系统的每个 cpu 上就建立了 6 条 time line，而上层的 PSI group 会以固定周期来采样 time line 的数组。采样点之间相减就可以得到该周期内各种状态的时间长度值。通过下面的公式我们可以计算单个 CPU 上的 PSI 值：

%SOME = time(SOME) / period

%FULL = time(FULL) / period

在多 CPU 场景下，我们要综合考虑 CPU 个数和 non idle task 的个数，计算公式如下：

tNONIDLE = sum(tNONIDLE[i])

tSOME = sum(tSOME[i] * tNONIDLE[i]) / tNONIDLE

tFULL = sum(tFULL[i] * tNONIDLE[i]) / tNONIDLE

%SOME = tSOME / period

%FULL = tFULL / period

tNONIDLE[i]、tSOME[i] 和 tFULL[i] 已经在 per-CPU 状态统计中获取了，通过上面的公式即可以计算该 psi group 在当前周期内的 PSI 值。

在计算三种间隔的时间占比时，有人可能会有疑问，周期是 2s，如何做到每次都更新三种数据呢？这个问题其实在上面讲到的老技术 load average 计算时已经解决，采用公式：a1 = a0 * e + a * (1 - e)；

于是得到：newload = load * exp + active * (FIXED_1 - exp)

其中 active 是当前更新周期的 load average，load 是上个周期得到的 load average，exp 的定义如下：

#define EXP_10s 1677 /* 1/exp(2s/10s) as fixed-point */

#define EXP_60s 1981 /* 1/exp(2s/60s) */

#define EXP_300s 2034 /* 1/exp(2s/300s) */

PSI 的应用

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。比如 facebook 开发的 cgroup2 和 oomd。oomd 是一个用户态的 out of memory 监控管理服务。

Android 早期在 kernel 新增了一个功能叫 lmk(low memory killer)，在有了 PSI 之后，android 将默认的 LMK 替换成了用户态的 LMKD。其代码存放于 android/system/core/lmkd/。

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 daemon 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。