PSI - Pressure Stall Information

Date: April, 2018
Author: Johannes Weiner <hannes@cmpxchg.org>

When CPU, memory or IO devices are contended, workloads experience latency spikes, throughput losses, and run the risk of OOM kills.

Without an accurate measure of such contention, users are forced to either play it safe and under-utilize their hardware resources, or roll the dice and frequently suffer the disruptions resulting from excessive overcommit.

The psi feature identifies and quantifies the disruptions caused by such resource crunches and the time impact it has on complex workloads or even entire systems.

Having an accurate measure of productivity losses caused by resource scarcity aids users in sizing workloads to hardware--or provisioning hardware according to workload demand.

As psi aggregates this information in realtime, systems can be managed dynamically using techniques such as load shedding, migrating jobs to other systems or data centers, or strategically pausing or killing low priority or restartable batch jobs.

This allows maximizing hardware utilization without sacrificing workload health or risking major disruptions such as OOM kills.

Pressure interface

Pressure information for each resource is exported through the respective file in /proc/pressure/ -- cpu, memory, and io.

The format for CPU is as such:

```
some avg10=0.00 avg60=0.00 avg300=0.00 total=0
```

and for memory and IO:

```
some avg10=0.00 avg60=0.00 avg300=0.00 total=0
full avg10=0.00 avg60=0.00 avg300=0.00 total=0
```

The "some" line indicates the share of time in which at least some tasks are stalled on a given resource.

The "full" line indicates the share of time in which all non-idle tasks are stalled on a given resource simultaneously. In this state actual CPU cycles are going to waste, and a workload that spends extended time in this state is considered to be thrashing. This has severe impact on performance, and it's useful to distinguish this situation from a state where some tasks are stalled but the CPU is still doing productive work. As such, time spent in this subset of the stall state is tracked separately and exported in the "full" averages.

The ratios (in %) are tracked as recent trends over ten, sixty, and three hundred second windows, which gives insight into short term events as well as medium and long term trends. The total absolute stall time (in us) is tracked and exported as well, to allow detection of latency spikes which wouldn't necessarily make a dent in the time averages, or to average trends over custom time frames.

Monitoring for pressure thresholds

Users can register triggers and use poll() to be woken up when resource pressure exceeds certain thresholds.

A trigger describes the maximum cumulative stall time over a specific time window, e.g. 100ms of total stall time within any 500ms window to generate a wakeup event.

To register a trigger user has to open psi interface file under /proc/pressure/ representing the resource to be monitored and write the desired threshold and time window. The open file descriptor should be used to wait for trigger events using select(), poll() or epoll(). The following format is used:

```
<some|full> <stall amount in us> <time window in us>
```

For example writing "some 150000 1000000" into /proc/pressure/memory would add 150ms threshold for partial memory stall measured within 1sec time window. Writing "full 50000 1000000" into /proc/pressure/io would add 50ms threshold for full io stall measured within 1sec time window.

Triggers can be set on more than one psi metric and more than one trigger for the same psi metric can be specified. However for each trigger a separate file descriptor is required to be able to poll it separately from others, therefore for each trigger a separate open() syscall should be made even when opening the same psi interface file. Write operations to a file descriptor with an already existing psi trigger will fail with EBUSY.

Monitors activate only when system enters stall state for the monitored psi metric and deactivates upon exit from the stall state. While system is in the stall state psi signal growth is monitored at a rate of 10 times per tracking window.

The kernel accepts window sizes ranging from 500ms to 10s, therefore min monitoring update interval is 50ms and max is 1s. Min limit is set to prevent overly frequent polling. Max limit is chosen as a high enough number after which monitors are most likely not needed and psi averages can be used instead.

When activated, psi monitor stays active for at least the duration of one tracking window to avoid repeated activations/deactivations when system is bouncing in and out of the stall state.

Notifications to the userspace are rate-limited to one per tracking window.

The trigger will de-register when the file descriptor used to define the trigger is closed.

Userspace monitor usage example

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <string.h>
#include <unistd.h>

/*
 * Monitor memory partial stall with 1s tracking window size
 * and 150ms threshold.
 */
int main() {
      const char trig[] = "some 150000 1000000";
      struct pollfd fds;
      int n;

      fds.fd = open("/proc/pressure/memory", O_RDWR | O_NONBLOCK);
      if (fds.fd < 0) {
              printf("/proc/pressure/memory open error: %s\n",
                      strerror(errno));
              return 1;
      }
      fds.events = POLLPRI;

      if (write(fds.fd, trig, strlen(trig) + 1) < 0) {
              printf("/proc/pressure/memory write error: %s\n",
                      strerror(errno));
              return 1;
      }

      printf("waiting for events...\n");
      while (1) {
              n = poll(&fds, 1, -1);
              if (n < 0) {
                      printf("poll error: %s\n", strerror(errno));
                      return 1;
              }
              if (fds.revents & POLLERR) {
                      printf("got POLLERR, event source is gone\n");
                      return 0;
              }
              if (fds.revents & POLLPRI) {
                      printf("event triggered!\n");
              } else {
                      printf("unknown event received: 0x%x\n", fds.revents);
                      return 1;
              }
      }

      return 0;
}
```

Cgroup2 interface

In a system with a CONFIG_CGROUP=y kernel and the cgroup2 filesystem mounted, pressure stall information is also tracked for tasks grouped into cgroups. Each subdirectory in the cgroupfs mountpoint contains cpu.pressure, memory.pressure, and io.pressure files; the format is the same as the /proc/pressure/ files.

Per-cgroup psi monitors can be specified and used the same way as system-wide ones.