Linux Accounting Documentation

The kernel development community

Jan 15, 2023

CONTENTS

CHAPTER

ONE

CONTROL GROUPSTATS

Control Groupstats is inspired by the discussion at https://lore.kernel.org/r/461CF883.2030308@sw.ru and implements per cgroup statistics as suggested by Andrew Morton in https://lore.kernel.org/r/20070411114927.1277d7c9.akpm@linux-foundation.org.

Per cgroup statistics infrastructure re-uses code from the taskstats interface. A new set of cgroup operations are registered with commands and attributes specific to cgroups. It should be very easy to extend per cgroup statistics, by adding members to the cgroupstats structure.

The current model for cgroupstats is a pull, a push model (to post statistics on interesting events), should be very easy to add. Currently user space requests for statistics by passing the cgroup path. Statistics about the state of all the tasks in the cgroup is returned to user space.

NOTE: We currently rely on delay accounting for extracting information about tasks blocked on I/O. If CONFIG_TASK_DELAY_ACCT is disabled, this information will not be available.

To extract cgroup statistics a utility very similar to getdelays.c has been developed, the sample output of the utility is shown below:

```
~/balbir/cgroupstats # ./getdelays  -C "/sys/fs/cgroup/a"
sleeping 1, blocked 0, running 1, stopped 0, uninterruptible 0
~/balbir/cgroupstats # ./getdelays  -C "/sys/fs/cgroup"
sleeping 155, blocked 0, running 1, stopped 0, uninterruptible 2
```

CHAPTER

TWO

DELAY ACCOUNTING

Tasks encounter delays in execution when they wait for some kernel resource to become available e.g. a runnable task may wait for a free CPU to run on.

The per-task delay accounting functionality measures the delays experienced by a task while

a) waiting for a CPU (while being runnable)

b) completion of synchronous block I/O initiated by the task

c) swapping in pages

d) memory reclaim

e) thrashing page cache

f) direct compact

g) write-protect copy

and makes these statistics available to userspace through the taskstats interface.

Such delays provide feedback for setting a task's cpu priority, io priority and rss limit values appropriately. Long delays for important tasks could be a trigger for raising its corresponding priority.

The functionality, through its use of the taskstats interface, also provides delay statistics aggregated for all tasks (or threads) belonging to a thread group (corresponding to a traditional Unix process). This is a commonly needed aggregation that is more efficiently done by the kernel.

Userspace utilities, particularly resource management applications, can also aggregate delay statistics into arbitrary groups. To enable this, delay statistics of a task are available both during its lifetime as well as on its exit, ensuring continuous and complete monitoring can be done.

2.1 Interface

Delay accounting uses the taskstats interface which is described in detail in a separate document in this directory. Taskstats returns a generic data structure to userspace corresponding to per-pid and per-tgid statistics. The delay accounting functionality populates specific fields of this structure. See

include/uapi/linux/taskstats.h

for a description of the fields pertaining to delay accounting. It will generally be in the form of counters returning the cumulative delay seen for cpu, sync block I/O, swapin, memory reclaim, thrash page cache, direct compact, write-protect copy etc.

Taking the difference of two successive readings of a given counter (say cpu_delay_total) for a task will give the delay experienced by the task waiting for the corresponding resource in that interval.

When a task exits, records containing the per-task statistics are sent to userspace without requiring a command. If it is the last exiting task of a thread group, the per-tgid statistics are also sent. More details are given in the taskstats interface description.

The getdelays.c userspace utility in tools/accounting directory allows simple commands to be run and the corresponding delay statistics to be displayed. It also serves as an example of using the taskstats interface.

2.2 Usage

Compile the kernel with:

```
CONFIG_TASK_DELAY_ACCT=y
CONFIG_TASKSTATS=y
```

Delay accounting is disabled by default at boot up. To enable, add:

```
delayacct
```

to the kernel boot options. The rest of the instructions below assume this has been done. Alternatively, use sysctl kernel.task_delayacct to switch the state at runtime. Note however that only tasks started after enabling it will have delayacct information.

After the system has booted up, use a utility similar to getdelays.c to access the delays seen by a given task or a task group (tgid). The utility also allows a given command to be executed and the corresponding delays to be seen.

General format of the getdelays command:

```
getdelays [-dilv] [-t tgid] [-p pid]
```

Get delays, since system boot, for pid 10:

```
# ./getdelays -d -p 10
(output similar to next case)
```

Get sum of delays, since system boot, for all pids with tgid 5:

```
# ./getdelays -d -t 5
print delayacct stats ON
TGID    5


CPU             count     real total  virtual total    delay total  delay␣
↪average
```

```
                    8         7000000         6872122         3382277          0.
↪423ms
IO                count    delay total  delay average
                    0               0              0ms
SWAP              count    delay total  delay average
                    0               0              0ms
RECLAIM           count    delay total  delay average
                    0               0              0ms
THRASHING         count    delay total  delay average
                    0               0              0ms
COMPACT           count    delay total  delay average
                    0               0              0ms
WPCOPY            count    delay total  delay average
                    0               0              0ms
```

Get IO accounting for pid 1, it works only with -p:

```
# ./getdelays -i -p 1
printing IO accounting
linuxrc: read=65536, write=0, cancelled_write=0
```

The above command can be used with -v to get more debug information.

CHAPTER

THREE

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.

3.1 Pressure interface

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

The format is as such:

```
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.

CPU full is undefined at the system level, but has been reported since 5.13, so it is set to zero for backward compatibility.

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.

3.2 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.

3.3 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;
}
```

3.4 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.

CHAPTER

FOUR

PER-TASK STATISTICS INTERFACE

Taskstats is a netlink-based interface for sending per-task and per-process statistics from the kernel to userspace.

Taskstats was designed for the following benefits:

- efficiently provide statistics during lifetime of a task and on its exit
- unified interface for multiple accounting subsystems
- extensibility for use by future accounting patches

4.1 Terminology

“pid”, “tid” and “task” are used interchangeably and refer to the standard Linux task defined by struct task_struct. per-pid stats are the same as per-task stats.

“tgid”, “process” and “thread group” are used interchangeably and refer to the tasks that share an mm_struct i.e. the traditional Unix process. Despite the use of tgid, there is no special treatment for the task that is thread group leader - a process is deemed alive as long as it has any task belonging to it.

4.2 Usage

To get statistics during a task’s lifetime, userspace opens a unicast netlink socket (NETLINK_GENERIC family) and sends commands specifying a pid or a tgid. The response contains statistics for a task (if pid is specified) or the sum of statistics for all tasks of the process (if tgid is specified).

To obtain statistics for tasks which are exiting, the userspace listener sends a register command and specifies a cpumask. Whenever a task exits on one of the cpus in the cpumask, its per-pid statistics are sent to the registered listener. Using cpumasks allows the data received by one listener to be limited and assists in flow control over the netlink interface and is explained in more detail below.

If the exiting task is the last thread exiting its thread group, an additional record containing the per-tgid stats is also sent to userspace. The latter contains the sum of per-pid stats for all threads in the thread group, both past and present.

getdelays.c is a simple utility demonstrating usage of the taskstats interface for reporting delay accounting statistics. Users can register cpumasks, send commands and process responses,

listen for per-tid/tgid exit data, write the data received to a file and do basic flow control by increasing receive buffer sizes.

4.3 Interface

The user-kernel interface is encapsulated in include/linux/taskstats.h

To avoid this documentation becoming obsolete as the interface evolves, only an outline of the current version is given. taskstats.h always overrides the description here.

struct taskstats is the common accounting structure for both per-pid and per-tgid data. It is versioned and can be extended by each accounting subsystem that is added to the kernel. The fields and their semantics are defined in the taskstats.h file.

The data exchanged between user and kernel space is a netlink message belonging to the NETLINK_GENERIC family and using the netlink attributes interface. The messages are in the format:

```
+----------+- - -+-------------+-------------------+
| nlmsghdr | Pad |  genlmsghdr | taskstats payload |
+----------+- - -+-------------+-------------------+
```

The taskstats payload is one of the following three kinds:

1. Commands: Sent from user to kernel. Commands to get data on a pid/tgid consist of one attribute, of type TASKSTATS_CMD_ATTR_PID/TGID, containing a u32 pid or tgid in the attribute payload. The pid/tgid denotes the task/process for which userspace wants statistics.

Commands to register/deregister interest in exit data from a set of cpus consist of one attribute, of type TASKSTATS_CMD_ATTR_REGISTER/DEREGISTER_CPUMASK and contain a cpumask in the attribute payload. The cpumask is specified as an ascii string of comma-separated cpu ranges e.g. to listen to exit data from cpus 1,2,3,5,7,8 the cpumask would be “1-3,5,7-8”. If userspace forgets to deregister interest in cpus before closing the listening socket, the kernel cleans up its interest set over time. However, for the sake of efficiency, an explicit deregistration is advisable.

2. Response for a command: sent from the kernel in response to a userspace command. The payload is a series of three attributes of type:

a) TASKSTATS_TYPE_AGGR_PID/TGID : attribute containing no payload but indicates a pid/tgid will be followed by some stats.

b) TASKSTATS_TYPE_PID/TGID: attribute whose payload is the pid/tgid whose stats are being returned.

c) TASKSTATS_TYPE_STATS: attribute with a struct taskstats as payload. The same structure is used for both per-pid and per-tgid stats.

3. New message sent by kernel whenever a task exits. The payload consists of a series of attributes of the following type:

a) TASKSTATS_TYPE_AGGR_PID: indicates next two attributes will be pid+stats

b) TASKSTATS_TYPE_PID: contains exiting task’s pid

c) TASKSTATS_TYPE_STATS: contains the exiting task’s per-pid stats

d) TASKSTATS_TYPE_AGGR_TGID: indicates next two attributes will be tgid+stats

e) TASKSTATS_TYPE_TGID: contains tgid of process to which task belongs

f) TASKSTATS_TYPE_STATS: contains the per-tgid stats for exiting task's process

4.4 per-tgid stats

Taskstats provides per-process stats, in addition to per-task stats, since resource management is often done at a process granularity and aggregating task stats in userspace alone is inefficient and potentially inaccurate (due to lack of atomicity).

However, maintaining per-process, in addition to per-task stats, within the kernel has space and time overheads. To address this, the taskstats code accumulates each exiting task's statistics into a process-wide data structure. When the last task of a process exits, the process level data accumulated also gets sent to userspace (along with the per-task data).

When a user queries to get per-tgid data, the sum of all other live threads in the group is added up and added to the accumulated total for previously exited threads of the same thread group.

4.5 Extending taskstats

There are two ways to extend the taskstats interface to export more per-task/process stats as patches to collect them get added to the kernel in future:

1. Adding more fields to the end of the existing struct taskstats. Backward compatibility is ensured by the version number within the structure. Userspace will use only the fields of the struct that correspond to the version its using.

2. Defining separate statistic structs and using the netlink attributes interface to return them. Since userspace processes each netlink attribute independently, it can always ignore attributes whose type it does not understand (because it is using an older version of the interface).

Choosing between 1. and 2. is a matter of trading off flexibility and overhead. If only a few fields need to be added, then 1. is the preferable path since the kernel and userspace don't need to incur the overhead of processing new netlink attributes. But if the new fields expand the existing struct too much, requiring disparate userspace accounting utilities to unnecessarily receive large structures whose fields are of no interest, then extending the attributes structure would be worthwhile.

4.6 Flow control for taskstats

When the rate of task exits becomes large, a listener may not be able to keep up with the kernel's rate of sending per-tid/tgid exit data leading to data loss. This possibility gets compounded when the taskstats structure gets extended and the number of cpus grows large.

To avoid losing statistics, userspace should do one or more of the following:

- increase the receive buffer sizes for the netlink sockets opened by listeners to receive exit data.

- create more listeners and reduce the number of cpus being listened to by each listener. In the extreme case, there could be one listener for each cpu. Users may also consider setting the cpu affinity of the listener to the subset of cpus to which it listens, especially if they are listening to just one cpu.

Despite these measures, if the userspace receives ENOBUFS error messages indicated overflow of receive buffers, it should take measures to handle the loss of data.

CHAPTER

FIVE

THE STRUCT TASKSTATS

This document contains an explanation of the struct taskstats fields.

There are three different groups of fields in the struct taskstats:

1) **Common and basic accounting fields** If CONFIG_TASKSTATS is set, the taskstats interface is enabled and the common fields and basic accounting fields are collected for delivery at do_exit() of a task.

2) **Delay accounting fields** These fields are placed between:

```
/* Delay accounting fields start */
```

and:

```
/* Delay accounting fields end */
```

Their values are collected if CONFIG_TASK_DELAY_ACCT is set.

3) **Extended accounting fields** These fields are placed between:

```
/* Extended accounting fields start */
```

and:

```
/* Extended accounting fields end */
```

Their values are collected if CONFIG_TASK_XACCT is set.

4) Per-task and per-thread context switch count statistics

5) Time accounting for SMT machines

6) Extended delay accounting fields for memory reclaim

Future extension should add fields to the end of the taskstats struct, and should not change the relative position of each field within the struct.

```
struct taskstats {
```

1) Common and basic accounting fields:

```
/* The version number of this struct. This field is always set to
 * TAKSTATS_VERSION, which is defined in <linux/taskstats.h>.
 * Each time the struct is changed, the value should be incremented.
```

```
 */
__u16   version;

/* The exit code of a task. */
__u32   ac_exitcode;            /* Exit status */

/* The accounting flags of a task as defined in <linux/acct.h>
 * Defined values are AFORK, ASU, ACOMPAT, ACORE, and AXSIG.
 */
__u8    ac_flag;                /* Record flags */

/* The value of task_nice() of a task. */
__u8    ac_nice;                /* task_nice */

/* The name of the command that started this task. */
char    ac_comm[TS_COMM_LEN];   /* Command name */

/* The scheduling discipline as set in task->policy field. */
__u8    ac_sched;               /* Scheduling discipline */

__u8    ac_pad[3];
__u32   ac_uid;                 /* User ID */
__u32   ac_gid;                 /* Group ID */
__u32   ac_pid;                 /* Process ID */
__u32   ac_ppid;                /* Parent process ID */

/* The time when a task begins, in [secs] since 1970. */
__u32   ac_btime;               /* Begin time [sec since 1970] */

/* The elapsed time of a task, in [usec]. */
__u64   ac_etime;               /* Elapsed time [usec] */

/* The user CPU time of a task, in [usec]. */
__u64   ac_utime;               /* User CPU time [usec] */

/* The system CPU time of a task, in [usec]. */
__u64   ac_stime;               /* System CPU time [usec] */

/* The minor page fault count of a task, as set in task->min_flt. */
__u64   ac_minflt;              /* Minor Page Fault Count */

/* The major page fault count of a task, as set in task->maj_flt. */
__u64   ac_majflt;              /* Major Page Fault Count */
```

2) Delay accounting fields:

```
/* Delay accounting fields start
 *
 * All values, until the comment "Delay accounting fields end" are
 * available only if delay accounting is enabled, even though the last
 * few fields are not delays
```

```
 *
 * xxx_count is the number of delay values recorded
 * xxx_delay_total is the corresponding cumulative delay in nanoseconds
 *
 * xxx_delay_total wraps around to zero on overflow
 * xxx_count incremented regardless of overflow
 */

/* Delay waiting for cpu, while runnable
 * count, delay_total NOT updated atomically
 */
__u64   cpu_count;
__u64   cpu_delay_total;

/* Following four fields atomically updated using task->delays->lock */

/* Delay waiting for synchronous block I/O to complete
 * does not account for delays in I/O submission
 */
__u64   blkio_count;
__u64   blkio_delay_total;

/* Delay waiting for page fault I/O (swap in only) */
__u64   swapin_count;
__u64   swapin_delay_total;

/* cpu "wall-clock" running time
 * On some architectures, value will adjust for cpu time stolen
 * from the kernel in involuntary waits due to virtualization.
 * Value is cumulative, in nanoseconds, without a corresponding count
 * and wraps around to zero silently on overflow
 */
__u64   cpu_run_real_total;

/* cpu "virtual" running time
 * Uses time intervals seen by the kernel i.e. no adjustment
 * for kernel's involuntary waits due to virtualization.
 * Value is cumulative, in nanoseconds, without a corresponding count
 * and wraps around to zero silently on overflow
 */
__u64   cpu_run_virtual_total;
/* Delay accounting fields end */
/* version 1 ends here */
```

3) Extended accounting fields:

```
/* Extended accounting fields start */

/* Accumulated RSS usage in duration of a task, in MBytes-usecs.
 * The current rss usage is added to this counter every time
 * a tick is charged to a task's system time. So, at the end we
```

```
 * will have memory usage multiplied by system time. Thus an
 * average usage per system time unit can be calculated.
 */
__u64   corememe;               /* accumulated RSS usage in MB-usec */

/* Accumulated virtual memory usage in duration of a task.
 * Same as acct_rss_mem1 above except that we keep track of VM usage.
 */
__u64   virtmem;                /* accumulated VM usage in MB-usec */

/* High watermark of RSS usage in duration of a task, in KBytes. */
__u64   hiwater_rss;            /* High-watermark of RSS usage */

/* High watermark of VM  usage in duration of a task, in KBytes. */
__u64   hiwater_vm;             /* High-water virtual memory usage */

/* The following four fields are I/O statistics of a task. */
__u64   read_char;              /* bytes read */
__u64   write_char;             /* bytes written */
__u64   read_syscalls;          /* read syscalls */
__u64   write_syscalls;         /* write syscalls */

/* Extended accounting fields end */
```

4) Per-task and per-thread statistics:

```
__u64   nvcsw;                  /* Context voluntary switch counter */
__u64   nivcsw;                 /* Context involuntary switch counter */
```

5) Time accounting for SMT machines:

```
__u64   ac_utimescaled;         /* utime scaled on frequency etc */
__u64   ac_stimescaled;         /* stime scaled on frequency etc */
__u64   cpu_scaled_run_real_total; /* scaled cpu_run_real_total */
```

6) Extended delay accounting fields for memory reclaim:

```
/* Delay waiting for memory reclaim */
__u64   freepages_count;
__u64   freepages_delay_total;
```

```
}
```