



Linux Ubuntu 26.04 Manual Pages on command 'perf_event_open.2'

\$ man perf_event_open.2

perf_event_open(2) System Calls Manual perf_event_open(2)

NAME

perf_event_open - set up performance monitoring

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <linux/perf_event.h> /* Definition of PERF_* constants */
```

```
#include <linux/hw_breakpoint.h> /* Definition of HW_* constants */
```

```
#include <sys/syscall.h> /* Definition of SYS_* constants */
```

```
#include <unistd.h>
```

```
int syscall(SYS_perf_event_open, struct perf_event_attr *attr,  
           pid_t pid, int cpu, int group_fd, unsigned long flags);
```

Note: glibc provides no wrapper for perf_event_open(), necessitating the use of syscall(2).

DESCRIPTION

Given a list of parameters, perf_event_open() returns a file descriptor, for use in subsequent system calls (read(2), mmap(2), prctl(2), fcntl(2), etc.).

A call to perf_event_open() creates a file descriptor that allows measuring performance information. Each file descriptor corresponds to one event that is measured; these can be grouped together to measure multiple events simultaneously.

Events can be enabled and disabled in two ways: via ioctl(2) and via prctl(2). When an event is disabled it does not count or generate overflows but does continue to exist and maintain its count value.

Events come in two flavors: counting and sampled. A counting event is one that is used for

counting the aggregate number of events that occur. In general, counting event results are gathered with a `read(2)` call. A sampling event periodically writes measurements to a buffer that can then be accessed via `mmap(2)`.

Arguments

The `pid` and `cpu` arguments allow specifying which process and CPU to monitor:

`pid == 0` and `cpu == -1`

This measures the calling process/thread on any CPU.

`pid == 0` and `cpu >= 0`

This measures the calling process/thread only when running on the specified CPU.

`pid > 0` and `cpu == -1`

This measures the specified process/thread on any CPU.

`pid > 0` and `cpu >= 0`

This measures the specified process/thread only when running on the specified CPU.

`pid == -1` and `cpu >= 0`

This measures all processes/threads on the specified CPU. This requires `CAP_PERFMON` (since Linux 5.8) or `CAP_SYS_ADMIN` capability or a `/proc/sys/kernel/perf_event_paranoid` value of less than 1.

`pid == -1` and `cpu == -1`

This setting is invalid and will return an error.

When `pid` is greater than zero, permission to perform this system call is governed by `CAP_PERFMON` (since Linux 5.9) and a `ptrace` access mode `PTRACE_MODE_READ_REALCREDS` check on older Linux versions; see `ptrace(2)`.

The `group_fd` argument allows event groups to be created. An event group has one event which is the group leader. The leader is created first, with `group_fd = -1`. The rest of the group members are created with subsequent `perf_event_open()` calls with `group_fd` being set to the file descriptor of the group leader. (A single event on its own is created with `group_fd = -1` and is considered to be a group with only 1 member.) An event group is scheduled onto the CPU as a unit: it will be put onto the CPU only if all of the events in the group can be put onto the CPU. This means that the values of the member events can be meaningfully compared ?added, divided (to get ratios), and so on? with each other, since they have counted events for the same set of executed instructions.

The `flags` argument is formed by ORing together zero or more of the following values:

`PERF_FLAG_FD_CLOEXEC` (since Linux 3.14)

This flag enables the close-on-exec flag for the created event file descriptor, so that the file descriptor is automatically closed on `execve(2)`. Setting the close-on-exec flags at creation time, rather than later with `fcntl(2)`, avoids potential race conditions where the calling thread invokes `perf_event_open()` and `fcntl(2)` at the same time as another thread calls `fork(2)` then `execve(2)`.

PERF_FLAG_FD_NO_GROUP

This flag tells the event to ignore the `group_fd` parameter except for the purpose of setting up output redirection using the `PERF_FLAG_FD_OUTPUT` flag.

PERF_FLAG_FD_OUTPUT (broken since Linux 2.6.35)

This flag re-routes the event's sampled output to instead be included in the `mmap` buffer of the event specified by `group_fd`.

PERF_FLAG_PID_CGROUP (since Linux 2.6.39)

This flag activates per-container system-wide monitoring. A container is an abstraction that isolates a set of resources for finer-grained control (CPUs, memory, etc.).

In this mode, the event is measured only if the thread running on the monitored CPU belongs to the designated container (cgroup). The cgroup is identified by passing a file descriptor opened on its directory in the `cgroupfs` filesystem. For instance, if the cgroup to monitor is called `test`, then a file descriptor opened on `/dev/cgroup/test` (assuming `cgroupfs` is mounted on `/dev/cgroup`) must be passed as the `pid` parameter. cgroup monitoring is available only for system-wide events and may therefore require extra permissions.

The `perf_event_attr` structure provides detailed configuration information for the event being created.

```
struct perf_event_attr {
    __u32 type;           /* Type of event */
    __u32 size;           /* Size of attribute structure */
    __u64 config;         /* Type-specific configuration */
    union {
        __u64 sample_period; /* Period of sampling */
        __u64 sample_freq;   /* Frequency of sampling */
    };
    __u64 sample_type; /* Specifies values included in sample */
    __u64 read_format; /* Specifies values returned in read */
};
```

```

__u64 disabled      : 1, /* off by default */

inherit            : 1, /* children inherit it */

pinned            : 1, /* must always be on PMU */

exclusive         : 1, /* only group on PMU */

exclude_user      : 1, /* don't count user */

exclude_kernel    : 1, /* don't count kernel */

exclude_hv        : 1, /* don't count hypervisor */

exclude_idle      : 1, /* don't count when idle */

mmap              : 1, /* include mmap data */

comm              : 1, /* include comm data */

freq              : 1, /* use freq, not period */

inherit_stat      : 1, /* per task counts */

enable_on_exec    : 1, /* next exec enables */

task              : 1, /* trace fork/exit */

watermark         : 1, /* wakeup_watermark */

precise_ip        : 2, /* skid constraint */

mmap_data         : 1, /* non-exec mmap data */

sample_id_all     : 1, /* sample_type all events */

exclude_host      : 1, /* don't count in host */

exclude_guest     : 1, /* don't count in guest */

exclude_callchain_kernel : 1,

                    /* exclude kernel callchains */

exclude_callchain_user : 1,

                    /* exclude user callchains */

mmap2             : 1, /* include mmap with inode data */

comm_exec         : 1, /* flag comm events that are

                        due to exec */

use_clockid       : 1, /* use clockid for time fields */

context_switch    : 1, /* context switch data */

write_backward    : 1, /* Write ring buffer from end

                        to beginning */

namespaces        : 1, /* include namespaces data */

ksymbol           : 1, /* include ksymbol events */

```

```

bpf_event    : 1, /* include bpf events */
aux_output   : 1, /* generate AUX records
                  instead of events */
cgroup       : 1, /* include cgroup events */
text_poke    : 1, /* include text poke events */
build_id     : 1, /* use build id in mmap2 events */
inherit_thread : 1, /* children only inherit */
              /* if cloned with CLONE_THREAD */
remove_on_exec : 1, /* event is removed from task
                  on exec */
sigtrap      : 1, /* send synchronous SIGTRAP
                  on event */
__reserved_1 : 26;
union {
    __u32 wakeup_events; /* wakeup every n events */
    __u32 wakeup_watermark; /* bytes before wakeup */
};
__u32 bp_type; /* breakpoint type */
union {
    __u64 bp_addr; /* breakpoint address */
    __u64 kprobe_func; /* for perf_kprobe */
    __u64 uprobe_path; /* for perf_uprobe */
    __u64 config1; /* extension of config */
};
union {
    __u64 bp_len; /* breakpoint size */
    __u64 kprobe_addr; /* with kprobe_func == NULL */
    __u64 probe_offset; /* for perf_[k,u]probe */
    __u64 config2; /* extension of config1 */
};
__u64 branch_sample_type; /* enum perf_branch_sample_type */
__u64 sample_regs_user; /* user regs to dump on samples */
__u32 sample_stack_user; /* size of stack to dump on

```

```

        samples */
__s32 clockid;      /* clock to use for time fields */
__u64 sample_regs_intr; /* regs to dump on samples */
__u32 aux_watermark; /* aux bytes before wakeup */
__u16 sample_max_stack; /* max frames in callchain */
__u16 __reserved_2; /* align to u64 */
__u32 aux_sample_size; /* max aux sample size */
__u32 __reserved_3; /* align to u64 */
__u64 sig_data; /* user data for sigtrap */
};

```

The fields of the `perf_event_attr` structure are described in more detail below:

type This field specifies the overall event type. It has one of the following values:

PERF_TYPE_HARDWARE

This indicates one of the "generalized" hardware events provided by the kernel. See the `config` field definition for more details.

PERF_TYPE_SOFTWARE

This indicates one of the software-defined events provided by the kernel (even if no hardware support is available).

PERF_TYPE_TRACEPOINT

This indicates a tracepoint provided by the kernel tracepoint infrastructure.

PERF_TYPE_HW_CACHE

This indicates a hardware cache event. This has a special encoding, described in the `config` field definition.

PERF_TYPE_RAW

This indicates a "raw" implementation-specific event in the `config` field.

PERF_TYPE_BREAKPOINT (since Linux 2.6.33)

This indicates a hardware breakpoint as provided by the CPU. Breakpoints can be read/write accesses to an address as well as execution of an instruction address.

dynamic PMU

Since Linux 2.6.38, `perf_event_open()` can support multiple PMUs. To enable this, a value exported by the kernel can be used in the `type` field to indicate which PMU to use. The value to use can be found in the `sysfs` filesystem:

there is a subdirectory per PMU instance under `/sys/bus/event_source/devices`.

In each subdirectory there is a `type` file whose content is an integer that can be used in the `type` field. For instance, `/sys/bus/event_source/devices/cpu/type` contains the value for the core CPU PMU, which is usually 4.

`kprobe`

`uprobe` (both since Linux 4.17)

These two dynamic PMUs create a `kprobe/uprobe` and attach it to the file descriptor generated by `perf_event_open`. The `kprobe/uprobe` will be destroyed on the destruction of the file descriptor. See fields `kprobe_func`, `uprobe_path`, `kprobe_addr`, and `probe_offset` for more details.

`size` The size of the `perf_event_attr` structure for forward/backward compatibility. Set this using `sizeof(struct perf_event_attr)` to allow the kernel to see the struct size at the time of compilation.

The related define `PERF_ATTR_SIZE_VER0` is set to 64; this was the size of the first published struct. `PERF_ATTR_SIZE_VER1` is 72, corresponding to the addition of `breakpoints` in Linux 2.6.33. `PERF_ATTR_SIZE_VER2` is 80 corresponding to the addition of branch sampling in Linux 3.4. `PERF_ATTR_SIZE_VER3` is 96 corresponding to the addition of `sample_regs_user` and `sample_stack_user` in Linux 3.7. `PERF_ATTR_SIZE_VER4` is 104 corresponding to the addition of `sample_regs_intr` in Linux 3.19. `PERF_ATTR_SIZE_VER5` is 112 corresponding to the addition of `aux_watermark` in Linux 4.1.

`config` This specifies which event you want, in conjunction with the `type` field. The `config1` and `config2` fields are also taken into account in cases where 64 bits is not enough to fully specify the event. The encoding of these fields are event dependent.

There are various ways to set the `config` field that are dependent on the value of the previously described `type` field. What follows are various possible settings for `config` separated out by type.

If `type` is `PERF_TYPE_HARDWARE`, we are measuring one of the generalized hardware CPU events. Not all of these are available on all platforms. Set `config` to one of the following:

`PERF_COUNT_HW_CPU_CYCLES`

Total cycles. Be wary of what happens during CPU frequency scaling.

`PERF_COUNT_HW_INSTRUCTIONS`

Retired instructions. Be careful, these can be affected by various issues, most notably hardware interrupt counts.

PERF_COUNT_HW_CACHE_REFERENCES

Cache accesses. Usually this indicates Last Level Cache accesses but this may vary depending on your CPU. This may include prefetches and coherency messages; again, this depends on the design of your CPU.

PERF_COUNT_HW_CACHE_MISSES

Cache misses. Usually this indicates Last Level Cache misses; this is intended to be used in conjunction with the PERF_COUNT_HW_CACHE_REFERENCES event to calculate cache miss rates.

PERF_COUNT_HW_BRANCH_INSTRUCTIONS

Retired branch instructions. Prior to Linux 2.6.35, this used the wrong event on AMD processors.

PERF_COUNT_HW_BRANCH_MISSES

Mispredicted branch instructions.

PERF_COUNT_HW_BUS_CYCLES

Bus cycles, which can be different from total cycles.

PERF_COUNT_HW_STALLED_CYCLES_FRONTEND (since Linux 3.0)

Stalled cycles during issue.

PERF_COUNT_HW_STALLED_CYCLES_BACKEND (since Linux 3.0)

Stalled cycles during retirement.

PERF_COUNT_HW_REF_CPU_CYCLES (since Linux 3.3)

Total cycles; not affected by CPU frequency scaling.

If type is PERF_TYPE_SOFTWARE, we are measuring software events provided by the kernel. Set config to one of the following:

PERF_COUNT_SW_CPU_CLOCK

This reports the CPU clock, a high-resolution per-CPU timer.

PERF_COUNT_SW_TASK_CLOCK

This reports a clock count specific to the task that is running.

PERF_COUNT_SW_PAGE_FAULTS

This reports the number of page faults.

PERF_COUNT_SW_CONTEXT_SWITCHES

This counts context switches. Until Linux 2.6.34, these were all re-

ported as user-space events, after that they are reported as happening in the kernel.

PERF_COUNT_SW_CPU_MIGRATIONS

This reports the number of times the process has migrated to a new CPU.

PERF_COUNT_SW_PAGE_FAULTS_MIN

This counts the number of minor page faults. These did not require disk I/O to handle.

PERF_COUNT_SW_PAGE_FAULTS_MAJ

This counts the number of major page faults. These required disk I/O to handle.

PERF_COUNT_SW_ALIGNMENT_FAULTS (since Linux 2.6.33)

This counts the number of alignment faults. These happen when unaligned memory accesses happen; the kernel can handle these but it reduces performance. This happens only on some architectures (never on x86).

PERF_COUNT_SW_EMULATION_FAULTS (since Linux 2.6.33)

This counts the number of emulation faults. The kernel sometimes traps on unimplemented instructions and emulates them for user space. This can negatively impact performance.

PERF_COUNT_SW_DUMMY (since Linux 3.12)

This is a placeholder event that counts nothing. Informational sample record types such as mmap or comm must be associated with an active event. This dummy event allows gathering such records without requiring a counting event.

PERF_COUNT_SW_BPF_OUTPUT (since Linux 4.4)

This is used to generate raw sample data from BPF. BPF programs can write to this event using bpf_perf_event_output helper.

PERF_COUNT_SW_CGROUP_SWITCHES (since Linux 5.13)

This counts context switches to a task in a different cgroup. In other words, if the next task is in the same cgroup, it won't count the switch.

If type is PERF_TYPE_TRACEPOINT, then we are measuring kernel tracepoints. The value to use in config can be obtained from `under debugfs tracing/events/*/id` if ftrace is enabled in the kernel.

If type is PERF_TYPE_HW_CACHE, then we are measuring a hardware CPU cache event. To

calculate the appropriate config value, use the following equation:

```
config = (perf_hw_cache_id) |  
         (perf_hw_cache_op_id << 8) |  
         (perf_hw_cache_op_result_id << 16);
```

where perf_hw_cache_id is one of:

```
PERF_COUNT_HW_CACHE_L1D  
    for measuring Level 1 Data Cache  
PERF_COUNT_HW_CACHE_L1I  
    for measuring Level 1 Instruction Cache  
PERF_COUNT_HW_CACHE_LL  
    for measuring Last-Level Cache  
PERF_COUNT_HW_CACHE_DTLB  
    for measuring the Data TLB  
PERF_COUNT_HW_CACHE_ITLB  
    for measuring the Instruction TLB  
PERF_COUNT_HW_CACHE_BPU  
    for measuring the branch prediction unit  
PERF_COUNT_HW_CACHE_NODE (since Linux 3.1)  
    for measuring local memory accesses
```

and perf_hw_cache_op_id is one of:

```
PERF_COUNT_HW_CACHE_OP_READ  
    for read accesses  
PERF_COUNT_HW_CACHE_OP_WRITE  
    for write accesses  
PERF_COUNT_HW_CACHE_OP_PREFETCH  
    for prefetch accesses
```

and perf_hw_cache_op_result_id is one of:

```
PERF_COUNT_HW_CACHE_RESULT_ACCESS  
    to measure accesses  
PERF_COUNT_HW_CACHE_RESULT_MISS  
    to measure misses
```

If type is PERF_TYPE_RAW, then a custom "raw" config value is needed. Most CPUs sup?

port events that are not covered by the "generalized" events. These are implementa?

tion defined; see your CPU manual (for example, the Intel Volume 3B documentation, or the AMD BIOS and Kernel Developer Guide). The libpfm4 library can be used to translate from the name in the architectural manuals to the raw hex value
perf_event_open() expects in this field.

If type is PERF_TYPE_BREAKPOINT, then leave config set to zero. Its parameters are set in other places.

If type is kprobe or uprobe, set retprobe (bit 0 of config, see /sys/bus/event_source/devices/[k,u]probe/format/retprobe) for kretprobe/uretprobe.

See fields kprobe_func, uprobe_path, kprobe_addr, and probe_offset for more details.

kprobe_func

uprobe_path

kprobe_addr

probe_offset

These fields describe the kprobe/uprobe for dynamic PMUs kprobe and uprobe. For kprobe: use kprobe_func and probe_offset, or use kprobe_addr and leave kprobe_func as NULL. For uprobe: use uprobe_path and probe_offset.

sample_period

sample_freq

A "sampling" event is one that generates an overflow notification every N events, where N is given by sample_period. A sampling event has sample_period > 0. When an overflow occurs, requested data is recorded in the mmap buffer. The sample_type field controls what data is recorded on each overflow.

sample_freq can be used if you wish to use frequency rather than period. In this case, you set the freq flag. The kernel will adjust the sampling period to try and achieve the desired rate. The rate of adjustment is a timer tick.

sample_type

The various bits in this field specify which values to include in the sample. They will be recorded in a ring-buffer, which is available to user space using mmap(2). The order in which the values are saved in the sample are documented in the MMAP Layout subsection below; it is not the enum perf_event_sample_format order.

PERF_SAMPLE_IP

Records instruction pointer.

PERF_SAMPLE_TID

Records the process and thread IDs.

PERF_SAMPLE_TIME

Records a timestamp.

PERF_SAMPLE_ADDR

Records an address, if applicable.

PERF_SAMPLE_READ

Record counter values for all events in a group, not just the group leader.

PERF_SAMPLE_CALLCHAIN

Records the callchain (stack backtrace).

PERF_SAMPLE_ID

Records a unique ID for the opened event's group leader.

PERF_SAMPLE_CPU

Records CPU number.

PERF_SAMPLE_PERIOD

Records the current sampling period.

PERF_SAMPLE_STREAM_ID

Records a unique ID for the opened event. Unlike PERF_SAMPLE_ID the actual ID is returned, not the group leader. This ID is the same as the one returned by PERF_FORMAT_ID.

PERF_SAMPLE_RAW

Records additional data, if applicable. Usually returned by tracepoint events.

PERF_SAMPLE_BRANCH_STACK (since Linux 3.4)

This provides a record of recent branches, as provided by CPU branch sampling hardware (such as Intel Last Branch Record). Not all hardware supports this feature.

See the `branch_sample_type` field for how to filter which branches are reported.

PERF_SAMPLE_REGS_USER (since Linux 3.7)

Records the current user-level CPU register state (the values in the process before the kernel was called).

PERF_SAMPLE_STACK_USER (since Linux 3.7)

Records the user level stack, allowing stack unwinding.

PERF_SAMPLE_WEIGHT (since Linux 3.10)

Records a hardware provided weight value that expresses how costly the sampled event was. This allows the hardware to highlight expensive events in a profile file.

PERF_SAMPLE_DATA_SRC (since Linux 3.10)

Records the data source: where in the memory hierarchy the data associated with the sampled instruction came from. This is available only if the underlying hardware supports this feature.

PERF_SAMPLE_IDENTIFIER (since Linux 3.12)

Places the SAMPLE_ID value in a fixed position in the record, either at the beginning (for sample events) or at the end (if a non-sample event).

This was necessary because a sample stream may have records from various different event sources with different sample_type settings. Parsing the event stream properly was not possible because the format of the record was needed to find SAMPLE_ID, but the format could not be found without knowing what event the sample belonged to (causing a circular dependency).

The PERF_SAMPLE_IDENTIFIER setting makes the event stream always parsable by putting SAMPLE_ID in a fixed location, even though it means having duplicate SAMPLE_ID values in records.

PERF_SAMPLE_TRANSACTION (since Linux 3.13)

Records reasons for transactional memory abort events (for example, from Intel TSX transactional memory support).

The precise_ip setting must be greater than 0 and a transactional memory abort event must be measured or no values will be recorded. Also note that some perf_event measurements, such as sampled cycle counting, may cause extraneous aborts (by causing an interrupt during a transaction).

PERF_SAMPLE_REGS_INTR (since Linux 3.19)

Records a subset of the current CPU register state as specified by sample_regs_intr. Unlike PERF_SAMPLE_REGS_USER the register values will return kernel register state if the overflow happened while kernel code is running.

If the CPU supports hardware sampling of register state (i.e., PEBS on Intel x86) and precise_ip is set higher than zero then the register values returned are those captured by hardware at the time of the sampled instruction's retirement.

tirement.

PERF_SAMPLE_PHYS_ADDR (since Linux 4.13)

Records physical address of data like in PERF_SAMPLE_ADDR.

PERF_SAMPLE_CGROUP (since Linux 5.7)

Records (perf_event) cgroup ID of the process. This corresponds to the id field in the PERF_RECORD_CGROUP event.

PERF_SAMPLE_DATA_PAGE_SIZE (since Linux 5.11)

Records page size of data like in PERF_SAMPLE_ADDR.

PERF_SAMPLE_CODE_PAGE_SIZE (since Linux 5.11)

Records page size of ip like in PERF_SAMPLE_IP.

PERF_SAMPLE_WEIGHT_STRUCT (since Linux 5.12)

Records hardware provided weight values like in PERF_SAMPLE_WEIGHT, but it can represent multiple values in a struct. This shares the same space as PERF_SAMPLE_WEIGHT, so users can apply either of those, not both. It has the following format and the meaning of each field is dependent on the hardware implementation.

```
union perf_sample_weight {  
    u64 full;      /* PERF_SAMPLE_WEIGHT */  
    struct {       /* PERF_SAMPLE_WEIGHT_STRUCT */  
        u32 var1_dw;  
        u16 var2_w;  
        u16 var3_w;  
    };  
};
```

read_format

This field specifies the format of the data returned by read(2) on a perf_event_open() file descriptor.

PERF_FORMAT_TOTAL_TIME_ENABLED

Adds the 64-bit time_enabled field. This can be used to calculate estimated totals if the PMU is overcommitted and multiplexing is happening.

PERF_FORMAT_TOTAL_TIME_RUNNING

Adds the 64-bit time_running field. This can be used to calculate estimated totals if the PMU is overcommitted and multiplexing is happening.

PERF_FORMAT_ID

Adds a 64-bit unique value that corresponds to the event group.

PERF_FORMAT_GROUP

Allows all counter values in an event group to be read with one read.

PERF_FORMAT_LOST (since Linux 6.0)

Adds a 64-bit value that is the number of lost samples for this event. This would be only meaningful when `sample_period` or `sample_freq` is set.

disabled

The `disabled` bit specifies whether the counter starts out disabled or enabled. If disabled, the event can later be enabled by `ioctl(2)`, `prctl(2)`, or `enable_on_exec`. When creating an event group, typically the group leader is initialized with `disabled` set to 1 and any child events are initialized with `disabled` set to 0. Despite `disabled` being 0, the child events will not start until the group leader is enabled.

inherit

The `inherit` bit specifies that this counter should count events of child tasks as well as the task specified. This applies only to new children, not to any existing children at the time the counter is created (nor to any new children of existing children).

Inherit does not work for some combinations of `read_format` values, such as `PERF_FORMAT_GROUP`. Additionally, using it together with `cpu == -1` prevents the creation of the mmap ring-buffer used for logging asynchronous events in sampled mode.

pinned The `pinned` bit specifies that the counter should always be on the CPU if at all possible. It applies only to hardware counters and only to group leaders. If a pinned counter cannot be put onto the CPU (e.g., because there are not enough hardware counters or because of a conflict with some other event), then the counter goes into an 'error' state, where reads return end-of-file (i.e., `read(2)` returns 0) until the counter is subsequently enabled or disabled.

exclusive

The `exclusive` bit specifies that when this counter's group is on the CPU, it should be the only group using the CPU's counters. In the future this may allow monitoring programs to support PMU features that need to run alone so that they do not disrupt other hardware counters.

Note that many unexpected situations may prevent events with the `exclusive` bit set

from ever running. This includes any users running a system-wide measurement as well as any kernel use of the performance counters (including the commonly enabled NMI Watchdog Timer interface).

`exclude_user`

If this bit is set, the count excludes events that happen in user space.

`exclude_kernel`

If this bit is set, the count excludes events that happen in kernel space.

`exclude_hv`

If this bit is set, the count excludes events that happen in the hypervisor. This is mainly for PMUs that have built-in support for handling this (such as POWER). Extra support is needed for handling hypervisor measurements on most machines.

`exclude_idle`

If set, don't count when the CPU is running the idle task. While you can currently enable this for any event type, it is ignored for all but software events.

`mmap` The `mmap` bit enables generation of `PERF_RECORD_MMAP` samples for every `mmap(2)` call that has `PROT_EXEC` set. This allows tools to notice new executable code being mapped into a program (dynamic shared libraries for example) so that addresses can be mapped back to the original code.

`comm` The `comm` bit enables tracking of process command name as modified by the `execve(2)` and `prctl(PR_SET_NAME)` system calls as well as writing to `/proc/self/comm`. If the `comm_exec` flag is also successfully set (possible since Linux 3.16), then the `misc` flag `PERF_RECORD_MISC_COMM_EXEC` can be used to differentiate the `execve(2)` case from the others.

`freq` If this bit is set, then `sample_frequency` not `sample_period` is used when setting up the sampling interval.

`inherit_stat`

This bit enables saving of event counts on context switch for inherited tasks. This is meaningful only if the `inherit` field is set.

`enable_on_exec`

If this bit is set, a counter is automatically enabled after a call to `execve(2)`.

`task` If this bit is set, then `fork/exit` notifications are included in the ring buffer.

`watermark`

If set, have an overflow notification happen when we cross the `wakeup_watermark`

boundary. Otherwise, overflow notifications happen after wakeup_events samples.

precise_ip (since Linux 2.6.35)

This controls the amount of skid. Skid is how many instructions execute between an event of interest happening and the kernel being able to stop and record the event.

Smaller skid is better and allows more accurate reporting of which events correspond to which instructions, but hardware is often limited with how small this can be.

The possible values of this field are the following:

- 0 SAMPLE_IP can have arbitrary skid.
- 1 SAMPLE_IP must have constant skid.
- 2 SAMPLE_IP requested to have 0 skid.
- 3 SAMPLE_IP must have 0 skid. See also the description of PERF_RECORD_MISC_EXACT_IP.

mmap_data (since Linux 2.6.36)

This is the counterpart of the mmap field. This enables generation of PERF_RECORD_MMAP samples for mmap(2) calls that do not have PROT_EXEC set (for example data and SysV shared memory).

sample_id_all (since Linux 2.6.38)

If set, then TID, TIME, ID, STREAM_ID, and CPU can additionally be included in non-PERF_RECORD_SAMPLES if the corresponding sample_type is selected.

If PERF_SAMPLE_IDENTIFIER is specified, then an additional ID value is included as the last value to ease parsing the record stream. This may lead to the id value appearing twice.

The layout is described by this pseudo-structure:

```
struct sample_id {
    { u32 pid, tid; } /* if PERF_SAMPLE_TID set */
    { u64 time; } /* if PERF_SAMPLE_TIME set */
    { u64 id; } /* if PERF_SAMPLE_ID set */
    { u64 stream_id; } /* if PERF_SAMPLE_STREAM_ID set */
    { u32 cpu, res; } /* if PERF_SAMPLE_CPU set */
    { u64 id; } /* if PERF_SAMPLE_IDENTIFIER set */
};
```

exclude_host (since Linux 3.2)

When conducting measurements that include processes running VM instances (i.e., have

executed a KVM_RUN ioctl(2)), only measure events happening inside a guest instance.

This is only meaningful outside the guests; this setting does not change counts gathered

inside of a guest. Currently, this functionality is x86 only.

exclude_guest (since Linux 3.2)

When conducting measurements that include processes running VM instances (i.e., have

executed a KVM_RUN ioctl(2)), do not measure events happening inside guest instances.

This is only meaningful outside the guests; this setting does not change counts gathered

inside of a guest. Currently, this functionality is x86 only.

exclude_callchain_kernel (since Linux 3.7)

Do not include kernel callchains.

exclude_callchain_user (since Linux 3.7)

Do not include user callchains.

mmap2 (since Linux 3.16)

Generate an extended executable mmap record that contains enough additional information

to uniquely identify shared mappings. The mmap flag must also be set for this

to work.

comm_exec (since Linux 3.16)

This is purely a feature-detection flag, it does not change kernel behavior. If this

flag can successfully be set, then, when comm is enabled, the

PERF_RECORD_MISC_COMM_EXEC flag will be set in the misc field of a comm record header

if the rename event being reported was caused by a call to execve(2). This allows

tools to distinguish between the various types of process renaming.

use_clockid (since Linux 4.1)

This allows selecting which internal Linux clock to use when generating timestamps

via the clockid field. This can make it easier to correlate perf sample times with

timestamps generated by other tools.

context_switch (since Linux 4.3)

This enables the generation of PERF_RECORD_SWITCH records when a context switch occurs.

It also enables the generation of PERF_RECORD_SWITCH_CPU_WIDE records when

sampling in CPU-wide mode. This functionality is in addition to existing tracepoint

and software events for measuring context switches. The advantage of this method is

that it will give full information even with strict perf_event_paranoid settings.

write_backward (since Linux 4.6)

This causes the ring buffer to be written from the end to the beginning. This is to support reading from overwriteable ring buffer.

namespaces (since Linux 4.11)

This enables the generation of PERF_RECORD_NAMESPACES records when a task enters a new namespace. Each namespace has a combination of device and inode numbers.

ksymbol (since Linux 5.0)

This enables the generation of PERF_RECORD_KSYMBOL records when new kernel symbols are registered or unregistered. This is analyzing dynamic kernel functions like eBPF.

bpf_event (since Linux 5.0)

This enables the generation of PERF_RECORD_BPF_EVENT records when an eBPF program is loaded or unloaded.

aux_output (since Linux 5.4)

This allows normal (non-AUX) events to generate data for AUX events if the hardware supports it.

cgroup (since Linux 5.7)

This enables the generation of PERF_RECORD_CGROUP records when a new cgroup is created (and activated).

text_poke (since Linux 5.8)

This enables the generation of PERF_RECORD_TEXT_POKE records when there's a change to the kernel text (i.e., self-modifying code).

build_id (since Linux 5.12)

This changes the contents in the PERF_RECORD_MMAP2 to have a build-id instead of device and inode numbers.

inherit_thread (since Linux 5.13)

This disables the inheritance of the event to a child process. Only new threads in the same process (which is cloned with CLONE_THREAD) will inherit the event.

remove_on_exec (since Linux 5.13)

This closes the event when it starts a new process image by execve(2).

sigtrap (since Linux 5.13)

This enables synchronous signal delivery of SIGTRAP on event overflow.

wakeup_events

wakeup_watermark

This union sets how many samples (`wakeup_events`) or bytes (`wakeup_watermark`) happen before an overflow notification happens. Which one is used is selected by the watermark bit flag.

`wakeup_events` counts only `PERF_RECORD_SAMPLE` record types. To receive overflow notification for all `PERF_RECORD` types choose watermark and set `wakeup_watermark` to 1.

Prior to Linux 3.0, setting `wakeup_events` to 0 resulted in no overflow notifications; more recent kernels treat 0 the same as 1.

`bp_type` (since Linux 2.6.33)

This chooses the breakpoint type. It is one of:

`HW_BREAKPOINT_EMPTY`

No breakpoint.

`HW_BREAKPOINT_R`

Count when we read the memory location.

`HW_BREAKPOINT_W`

Count when we write the memory location.

`HW_BREAKPOINT_RW`

Count when we read or write the memory location.

`HW_BREAKPOINT_X`

Count when we execute code at the memory location.

The values can be combined via a bitwise or, but the combination of `HW_BREAKPOINT_R` or `HW_BREAKPOINT_W` with `HW_BREAKPOINT_X` is not allowed.

`bp_addr` (since Linux 2.6.33)

This is the address of the breakpoint. For execution breakpoints, this is the memory address of the instruction of interest; for read and write breakpoints, it is the memory address of the memory location of interest.

`config1` (since Linux 2.6.39)

`config1` is used for setting events that need an extra register or otherwise do not fit in the regular config field. Raw `OFFCORE_EVENTS` on Nehalem/Westmere/SandyBridge use this field on Linux 3.3 and later kernels.

`bp_len` (since Linux 2.6.33)

`bp_len` is the size of the breakpoint being measured if type is `PERF_TYPE_BREAKPOINT`.

Options are `HW_BREAKPOINT_LEN_1`, `HW_BREAKPOINT_LEN_2`, `HW_BREAKPOINT_LEN_4`, and `HW_BREAKPOINT_LEN_8`. For an execution breakpoint, set this to `sizeof(long)`.

config2 (since Linux 2.6.39)

config2 is a further extension of the config1 field.

branch_sample_type (since Linux 3.4)

If PERF_SAMPLE_BRANCH_STACK is enabled, then this specifies what branches to include in the branch record.

The first part of the value is the privilege level, which is a combination of one of the values listed below. If the user does not set privilege level explicitly, the kernel will use the event's privilege level. Event and branch privilege levels do not have to match.

PERF_SAMPLE_BRANCH_USER

Branch target is in user space.

PERF_SAMPLE_BRANCH_KERNEL

Branch target is in kernel space.

PERF_SAMPLE_BRANCH_HV

Branch target is in hypervisor.

PERF_SAMPLE_BRANCH_PLM_ALL

A convenience value that is the three preceding values ORed together.

In addition to the privilege value, at least one or more of the following bits must be set.

PERF_SAMPLE_BRANCH_ANY

Any branch type.

PERF_SAMPLE_BRANCH_ANY_CALL

Any call branch (includes direct calls, indirect calls, and far jumps).

PERF_SAMPLE_BRANCH_IND_CALL

Indirect calls.

PERF_SAMPLE_BRANCH_CALL (since Linux 4.4)

Direct calls.

PERF_SAMPLE_BRANCH_ANY_RETURN

Any return branch.

PERF_SAMPLE_BRANCH_IND_JUMP (since Linux 4.2)

Indirect jumps.

PERF_SAMPLE_BRANCH_COND (since Linux 3.16)

Conditional branches.

PERF_SAMPLE_BRANCH_ABORT_TX (since Linux 3.11)

Transactional memory aborts.

PERF_SAMPLE_BRANCH_IN_TX (since Linux 3.11)

Branch in transactional memory transaction.

PERF_SAMPLE_BRANCH_NO_TX (since Linux 3.11)

Branch not in transactional memory transaction. PERF_SAMPLE_BRANCH_CALL_STACK (since Linux 4.1) Branch is part of a hardware-generated call stack. This requires hardware support, currently only found on Intel x86 Haswell or newer.

sample_regs_user (since Linux 3.7)

This bit mask defines the set of user CPU registers to dump on samples. The layout of the register mask is architecture-specific and is described in the kernel header file arch/ARCH/include/uapi/asm/perf_regs.h.

sample_stack_user (since Linux 3.7)

This defines the size of the user stack to dump if PERF_SAMPLE_STACK_USER is specified.

clockid (since Linux 4.1)

If use_clockid is set, then this field selects which internal Linux timer to use for timestamps. The available timers are defined in linux/time.h, with CLOCK_MONOTONIC, CLOCK_MONOTONIC_RAW, CLOCK_REALTIME, CLOCK_BOOTTIME, and CLOCK_TAI currently supported.

aux_watermark (since Linux 4.1)

This specifies how much data is required to trigger a PERF_RECORD_AUX sample.

sample_max_stack (since Linux 4.8)

When sample_type includes PERF_SAMPLE_CALLCHAIN, this field specifies how many stack frames to report when generating the callchain.

aux_sample_size (since Linux 5.5)

When PERF_SAMPLE_AUX flag is set, specify the desired size of AUX data. Note that it can get smaller data than the specified size.

sig_data (since Linux 5.13)

This data will be copied to user's signal handler (through si_perf in the signinfo_t) to disambiguate which event triggered the signal.

Reading results

Once a perf_event_open() file descriptor has been opened, the values of the events can be

read from the file descriptor. The values that are there are specified by the read_format field in the attr structure at open time.

If you attempt to read into a buffer that is not big enough to hold the data, the error ENOSPC results.

Here is the layout of the data returned by a read:

? If PERF_FORMAT_GROUP was specified to allow reading all events in a group at once:

```
struct read_format {  
    u64 nr;          /* The number of events */  
  
    u64 time_enabled; /* if PERF_FORMAT_TOTAL_TIME_ENABLED */  
  
    u64 time_running; /* if PERF_FORMAT_TOTAL_TIME_RUNNING */  
  
    struct {  
        u64 value; /* The value of the event */  
  
        u64 id;    /* if PERF_FORMAT_ID */  
  
        u64 lost;  /* if PERF_FORMAT_LOST */  
    } values[nr];  
};
```

? If PERF_FORMAT_GROUP was not specified:

```
struct read_format {  
    u64 value; /* The value of the event */  
  
    u64 time_enabled; /* if PERF_FORMAT_TOTAL_TIME_ENABLED */  
  
    u64 time_running; /* if PERF_FORMAT_TOTAL_TIME_RUNNING */  
  
    u64 id; /* if PERF_FORMAT_ID */  
  
    u64 lost; /* if PERF_FORMAT_LOST */  
};
```

The values read are as follows:

nr The number of events in this file descriptor. Available only if PERF_FORMAT_GROUP was specified.

time_enabled

time_running

Total time the event was enabled and running. Normally these values are the same.

Multiplexing happens if the number of events is more than the number of available PMU counter slots. In that case the events run only part of the time and the time_en?

abled and time running values can be used to scale an estimated value for the count.

value An unsigned 64-bit value containing the counter result.

id A globally unique value for this particular event; only present if PERF_FORMAT_ID was specified in read_format.

lost The number of lost samples of this event; only present if PERF_FORMAT_LOST was specified in read_format.

MMAP layout

When using perf_event_open() in sampled mode, asynchronous events (like counter overflow or PROT_EXEC mmap tracking) are logged into a ring-buffer. This ring-buffer is created and accessed through mmap(2).

The mmap size should be $1+2^n$ pages, where the first page is a metadata page (struct perf_event_mmap_page) that contains various bits of information such as where the ring-buffer head is.

Before Linux 2.6.39, there is a bug that means you must allocate an mmap ring buffer when sampling even if you do not plan to access it.

The structure of the first metadata mmap page is as follows:

```
struct perf_event_mmap_page {
    __u32 version;      /* version number of this structure */
    __u32 compat_version; /* lowest version this is compat with */
    __u32 lock;         /* seqlock for synchronization */
    __u32 index;        /* hardware counter identifier */
    __s64 offset;       /* add to hardware counter value */
    __u64 time_enabled; /* time event active */
    __u64 time_running; /* time event on CPU */
    union {
        __u64 capabilities;
        struct {
            __u64 cap_usr_time / cap_usr_rdpmc / cap_bit0 : 1,
                cap_bit0_is_deprecated : 1,
                cap_user_rdpmc : 1,
                cap_user_time : 1,
                cap_user_time_zero : 1,
        };
    };
};
```

```

__u16 pmc_width;
__u16 time_shift;
__u32 time_mult;
__u64 time_offset;
__u64 __reserved[120]; /* Pad to 1 k */
__u64 data_head;      /* head in the data section */
__u64 data_tail;      /* user-space written tail */
__u64 data_offset;    /* where the buffer starts */
__u64 data_size;      /* data buffer size */
__u64 aux_head;
__u64 aux_tail;
__u64 aux_offset;
__u64 aux_size;
}

```

The following list describes the fields in the `perf_event_mmap_page` structure in `more de?`

tail:

version

Version number of this structure.

compat_version

The lowest version this is compatible with.

lock A seqlock for synchronization.

index A unique hardware counter identifier.

offset When using `rdpmc` for reads this offset value must be added to the one returned by

`rdpmc` to get the current total event count.

time_enabled

Time the event was active.

time_running

Time the event was running.

`cap_usr_time` / `cap_usr_rdpmc` / `cap_bit0` (since Linux 3.4)

There was a bug in the definition of `cap_usr_time` and `cap_usr_rdpmc` from Linux 3.4

until Linux 3.11. Both bits were defined to point to the same location, so it was

impossible to know if `cap_usr_time` or `cap_usr_rdpmc` were actually set.

Starting with Linux 3.12, these are renamed to `cap_bit0` and you should use the

cap_user_time and cap_user_rdpmc fields instead.

cap_bit0_is_deprecated (since Linux 3.12)

If set, this bit indicates that the kernel supports the properly separated cap_user_time and cap_user_rdpmc bits.

If not-set, it indicates an older kernel where cap_usr_time and cap_usr_rdpmc map to the same bit and thus both features should be used with caution.

cap_user_rdpmc (since Linux 3.12)

If the hardware supports user-space read of performance counters without syscall (this is the "rdpmc" instruction on x86), then the following code can be used to do a read:

```
u32 seq, time_mult, time_shift, idx, width;
u64 count, enabled, running;
u64 cyc, time_offset;
do {
    seq = pc->lock;
    barrier();
    enabled = pc->time_enabled;
    running = pc->time_running;
    if (pc->cap_usr_time && enabled != running) {
        cyc = rdtsc();
        time_offset = pc->time_offset;
        time_mult = pc->time_mult;
        time_shift = pc->time_shift;
    }
    idx = pc->index;
    count = pc->offset;
    if (pc->cap_usr_rdpmc && idx) {
        width = pc->pmc_width;
        count += rdpmc(idx - 1);
    }
    barrier();
} while (pc->lock != seq);
```

cap_user_time (since Linux 3.12)

This bit indicates the hardware has a constant, nonstop timestamp counter (TSC on x86).

cap_user_time_zero (since Linux 3.12)

Indicates the presence of time_zero which allows mapping timestamp values to the hardware clock.

pmc_width

If cap_usr_rdpmc, this field provides the bit-width of the value read using the rdpmc or equivalent instruction. This can be used to sign extend the result like:

```
pmc <= 64 - pmc_width;
pmc >= 64 - pmc_width; // signed shift right
count += pmc;
```

time_shift

time_mult

time_offset

If cap_usr_time, these fields can be used to compute the time delta since time_enabled (in nanoseconds) using rdtsc or similar.

```
u64 quot, rem;
u64 delta;
quot = cyc >> time_shift;
rem = cyc & (((u64)1 << time_shift) - 1);
delta = time_offset + quot * time_mult +
        ((rem * time_mult) >> time_shift);
```

Where time_offset, time_mult, time_shift, and cyc are read in the seqcount loop described above. This delta can then be added to enabled and possible running (if idx), improving the scaling:

```
enabled += delta;
if (idx)
    running += delta;
quot = count / running;
rem = count % running;
count = quot * enabled + (rem * enabled) / running;
```

time_zero (since Linux 3.12)

If cap_usr_time_zero is set, then the hardware clock (the TSC timestamp counter on

x86) can be calculated from the `time_zero`, `time_mult`, and `time_shift` values:

```
time = timestamp - time_zero;
quot = time / time_mult;
rem  = time % time_mult;
cyc  = (quot << time_shift) + (rem << time_shift) / time_mult;
```

And vice versa:

```
quot = cyc >> time_shift;
rem  = cyc & (((u64)1 << time_shift) - 1);
timestamp = time_zero + quot * time_mult +
            ((rem * time_mult) >> time_shift);
```

`data_head`

This points to the head of the data section. The value continuously increases, it does not wrap. The value needs to be manually wrapped by the size of the mmap buffer before accessing the samples.

On SMP-capable platforms, after reading the `data_head` value, user space should issue an `rmb()`.

`data_tail`

When the mapping is `PROT_WRITE`, the `data_tail` value should be written by user space to reflect the last read data. In this case, the kernel will not overwrite unread data.

`data_offset` (since Linux 4.1)

Contains the offset of the location in the mmap buffer where perf sample data begins.

`data_size` (since Linux 4.1)

Contains the size of the perf sample region within the mmap buffer.

`aux_head`

`aux_tail`

`aux_offset`

`aux_size` (since Linux 4.1)

The AUX region allows `mmap(2)`-ing a separate sample buffer for high-bandwidth data streams (separate from the main perf sample buffer). An example of a high-bandwidth stream is instruction tracing support, as is found in newer Intel processors.

To set up an AUX area, first `aux_offset` needs to be set with an offset greater than `data_offset+data_size` and `aux_size` needs to be set to the desired buffer size. The

desired offset and size must be page aligned, and the size must be a power of two.

These values are then passed to `mmap` in order to map the AUX buffer. Pages in the AUX buffer are included as part of the `RLIMIT_MEMLOCK` resource limit (see `setrlimit(2)`), and also as part of the `perf_event_mlock_kb` allowance.

By default, the AUX buffer will be truncated if it will not fit in the available space in the ring buffer. If the AUX buffer is mapped as a read only buffer, then it will operate in ring buffer mode where old data will be overwritten by new. In overwrite mode, it might not be possible to infer where the new data began, and it is the consumer's job to disable measurement while reading to avoid possible data races. The `aux_head` and `aux_tail` ring buffer pointers have the same behavior and ordering rules as the previously described `data_head` and `data_tail`.

The following 2ⁿ ring-buffer pages have the layout described below.

If `perf_event_attr.sample_id_all` is set, then all event types will have the `sample_type` selected fields related to where/when (identity) an event took place (TID, TIME, ID, CPU, STREAM_ID) described in `PERF_RECORD_SAMPLE` below, it will be stashed just after the `perf_event_header` and the fields already present for the existing fields, that is, at the end of the payload. This allows a newer `perf.data` file to be supported by older `perf` tools, with the new optional fields being ignored.

The `mmap` values start with a header:

```
struct perf_event_header {
    __u32 type;
    __u16 misc;
    __u16 size;
};
```

Below, we describe the `perf_event_header` fields in more detail. For ease of reading, the fields with shorter descriptions are presented first.

size This indicates the size of the record.

misc The `misc` field contains additional information about the sample.

The CPU mode can be determined from this value by masking with `PERF_RECORD_MISC_CPUMODE_MASK` and looking for one of the following (note these are not bit masks, only one can be set at a time):

`PERF_RECORD_MISC_CPUMODE_UNKNOWN`

Unknown CPU mode.

PERF_RECORD_MISC_KERNEL

Sample happened in the kernel.

PERF_RECORD_MISC_USER

Sample happened in user code.

PERF_RECORD_MISC_HYPERVISOR

Sample happened in the hypervisor.

PERF_RECORD_MISC_GUEST_KERNEL (since Linux 2.6.35)

Sample happened in the guest kernel.

PERF_RECORD_MISC_GUEST_USER (since Linux 2.6.35)

Sample happened in guest user code.

Since the following three statuses are generated by different record types, they alias to the same bit:

PERF_RECORD_MISC_MMAP_DATA (since Linux 3.10)

This is set when the mapping is not executable; otherwise the mapping is executable.

PERF_RECORD_MISC_COMM_EXEC (since Linux 3.16)

This is set for a PERF_RECORD_COMM record on kernels more recent than Linux 3.16 if a process name change was caused by an `execve(2)` system call.

PERF_RECORD_MISC_SWITCH_OUT (since Linux 4.3)

When a PERF_RECORD_SWITCH or PERF_RECORD_SWITCH_CPU_WIDE record is generated, this bit indicates that the context switch is away from the current process (instead of into the current process).

In addition, the following bits can be set:

PERF_RECORD_MISC_EXACT_IP

This indicates that the content of PERF_SAMPLE_IP points to the actual instruction that triggered the event. See also `perf_event_attr.precise_ip`.

PERF_RECORD_MISC_SWITCH_OUT_PREEMPT (since Linux 4.17)

When a PERF_RECORD_SWITCH or PERF_RECORD_SWITCH_CPU_WIDE record is generated, this indicates the context switch was a preemption.

PERF_RECORD_MISC_MMAP_BUILD_ID (since Linux 5.12)

This indicates that the content of PERF_SAMPLE_MMAP2 contains build-ID data instead of device major and minor numbers as well as the inode number.

PERF_RECORD_MISC_EXT_RESERVED (since Linux 2.6.35)

This indicates there is extended data available (currently not used).

PERF_RECORD_MISC_PROC_MAP_PARSE_TIMEOUT

This bit is not set by the kernel. It is reserved for the user-space perf utility to indicate that /proc/pid/maps parsing was taking too long and was stopped, and thus the mmap records may be truncated.

type The type value is one of the below. The values in the corresponding record (that follows the header) depend on the type selected as shown.

PERF_RECORD_MMAP

The MMAP events record the PROT_EXEC mappings so that we can correlate user-space IPs to code. They have the following structure:

```
struct {  
    struct perf_event_header header;  
    u32  pid, tid;  
    u64  addr;  
    u64  len;  
    u64  pgoff;  
    char filename[];  
};
```

pid is the process ID.

tid is the thread ID.

addr is the address of the allocated memory. **len** is the size of the allocated memory. **pgoff** is the page offset of the allocated memory. **filename** is a string describing the backing of the allocated memory.

PERF_RECORD_LOST

This record indicates when events are lost.

```
struct {  
    struct perf_event_header header;  
    u64  id;  
    u64  lost;  
    struct sample_id sample_id;  
};
```

id is the unique event ID for the samples that were lost.

lost is the number of events that were lost.

PERF_RECORD_COMM

This record indicates a change in the process name.

```
struct {  
    struct perf_event_header header;  
    u32  pid;  
    u32  tid;  
    char comm[];  
    struct sample_id sample_id;  
};
```

pid is the process ID.

tid is the thread ID.

comm is a string containing the new name of the process.

PERF_RECORD_EXIT

This record indicates a process exit event.

```
struct {  
    struct perf_event_header header;  
    u32  pid, ppid;  
    u32  tid, ptid;  
    u64  time;  
    struct sample_id sample_id;  
};
```

PERF_RECORD_THROTTLE

PERF_RECORD_UNTHROTTLE

This record indicates a throttle/unthrottle event.

```
struct {  
    struct perf_event_header header;  
    u64  time;  
    u64  id;  
    u64  stream_id;  
    struct sample_id sample_id;  
};
```

PERF_RECORD_FORK

This record indicates a fork event.

```

struct {
    struct perf_event_header header;

    u32  pid, ppid;

    u32  tid, ptid;

    u64  time;

    struct sample_id sample_id;

};

```

PERF_RECORD_READ

This record indicates a read event.

```

struct {
    struct perf_event_header header;

    u32  pid, tid;

    struct read_format values;

    struct sample_id sample_id;

};

```

PERF_RECORD_SAMPLE

This record indicates a sample.

```

struct {
    struct perf_event_header header;

    u64  sample_id; /* if PERF_SAMPLE_IDENTIFIER */

    u64  ip;        /* if PERF_SAMPLE_IP */

    u32  pid, tid;  /* if PERF_SAMPLE_TID */

    u64  time;      /* if PERF_SAMPLE_TIME */

    u64  addr;      /* if PERF_SAMPLE_ADDR */

    u64  id;        /* if PERF_SAMPLE_ID */

    u64  stream_id; /* if PERF_SAMPLE_STREAM_ID */

    u32  cpu, res;  /* if PERF_SAMPLE_CPU */

    u64  period;    /* if PERF_SAMPLE_PERIOD */

    struct read_format v;

        /* if PERF_SAMPLE_READ */

    u64  nr;        /* if PERF_SAMPLE_CALLCHAIN */

    u64  ips[nr];   /* if PERF_SAMPLE_CALLCHAIN */

    u32  size;      /* if PERF_SAMPLE_RAW */

```

```

char  data[size]; /* if PERF_SAMPLE_RAW */

u64   bnr;        /* if PERF_SAMPLE_BRANCH_STACK */

struct perf_branch_entry lbr[bnr];

        /* if PERF_SAMPLE_BRANCH_STACK */

u64   abi;        /* if PERF_SAMPLE_REGS_USER */

u64   regs[weight(mask)];

        /* if PERF_SAMPLE_REGS_USER */

u64   size;       /* if PERF_SAMPLE_STACK_USER */

char  data[size]; /* if PERF_SAMPLE_STACK_USER */

u64   dyn_size;   /* if PERF_SAMPLE_STACK_USER &&

        size != 0 */

union perf_sample_weight weight;

        /* if PERF_SAMPLE_WEIGHT */

        /* || PERF_SAMPLE_WEIGHT_STRUCT */

u64   data_src;   /* if PERF_SAMPLE_DATA_SRC */

u64   transaction; /* if PERF_SAMPLE_TRANSACTION */

u64   abi;        /* if PERF_SAMPLE_REGS_INTR */

u64   regs[weight(mask)];

        /* if PERF_SAMPLE_REGS_INTR */

u64   phys_addr;  /* if PERF_SAMPLE_PHYS_ADDR */

u64   cgroup;     /* if PERF_SAMPLE_CGROUP */

u64   data_page_size;

        /* if PERF_SAMPLE_DATA_PAGE_SIZE */

u64   code_page_size;

        /* if PERF_SAMPLE_CODE_PAGE_SIZE */

u64   size;       /* if PERF_SAMPLE_AUX */

char  data[size]; /* if PERF_SAMPLE_AUX */

};

```

sample_id

If PERF_SAMPLE_IDENTIFIER is enabled, a 64-bit unique ID is included. This is a duplication of the PERF_SAMPLE_ID id value, but included at the beginning of the sample so parsers can easily obtain the value.

ip If PERF_SAMPLE_IP is enabled, then a 64-bit instruction pointer value is in?

cluded.

pid

tid If PERF_SAMPLE_TID is enabled, then a 32-bit process ID and 32-bit thread ID are included.

time

If PERF_SAMPLE_TIME is enabled, then a 64-bit timestamp is included. This is obtained via local_clock() which is a hardware timestamp if available and the jiffies value if not.

addr

If PERF_SAMPLE_ADDR is enabled, then a 64-bit address is included. This is usually the address of a tracepoint, breakpoint, or software event; otherwise the value is 0.

id If PERF_SAMPLE_ID is enabled, a 64-bit unique ID is included. If the event is a member of an event group, the group leader ID is returned. This ID is the same as the one returned by PERF_FORMAT_ID.

stream_id

If PERF_SAMPLE_STREAM_ID is enabled, a 64-bit unique ID is included. Unlike PERF_SAMPLE_ID the actual ID is returned, not the group leader. This ID is the same as the one returned by PERF_FORMAT_ID.

cpu

res If PERF_SAMPLE_CPU is enabled, this is a 32-bit value indicating which CPU was being used, in addition to a reserved (unused) 32-bit value.

period

If PERF_SAMPLE_PERIOD is enabled, a 64-bit value indicating the current sampling period is written.

v If PERF_SAMPLE_READ is enabled, a structure of type read_format is included which has values for all events in the event group. The values included depend on the read_format value used at perf_event_open() time.

nr

ips[nr]

If PERF_SAMPLE_CALLCHAIN is enabled, then a 64-bit number is included which indicates how many following 64-bit instruction pointers will follow. This is the current callchain.

size

data[size]

If `PERF_SAMPLE_RAW` is enabled, then a 32-bit value indicating `size` is included followed by an array of 8-bit values of size `size`. The values are padded with 0 to have 64-bit alignment.

This RAW record data is opaque with respect to the ABI. The ABI doesn't make any promises with respect to the stability of its content, it may vary depending on event, hardware, and kernel version.

bnr

lbr[bnr]

If `PERF_SAMPLE_BRANCH_STACK` is enabled, then a 64-bit value indicating the number of records is included, followed by `bnr perf_branch_entry` structures which each include the fields:

from This indicates the source instruction (may not be a branch).

to The branch target.

mispred

The branch target was mispredicted.

predicted

The branch target was predicted.

in_tx (since Linux 3.11)

The branch was in a transactional memory transaction.

abort (since Linux 3.11)

The branch was in an aborted transactional memory transaction.

cycles (since Linux 4.3)

This reports the number of cycles elapsed since the previous branch stack update.

The entries are from most to least recent, so the first entry has the most recent branch.

Support for `mispred`, `predicted`, and `cycles` is optional; if not supported, those values will be 0.

The type of branches recorded is specified by the `branch_sample_type` field.

abi

regs[weight(mask)]

If PERF_SAMPLE_REGS_USER is enabled, then the user CPU registers are recorded.

The abi field is one of PERF_SAMPLE_REGS_ABI_NONE, PERF_SAMPLE_REGS_ABI_32, or PERF_SAMPLE_REGS_ABI_64.

The regs field is an array of the CPU registers that were specified by the sample_regs_user attr field. The number of values is the number of bits set in the sample_regs_user bit mask.

size

data[size]

dyn_size

If PERF_SAMPLE_STACK_USER is enabled, then the user stack is recorded. This can be used to generate stack backtraces. size is the size requested by the user in sample_stack_user or else the maximum record size. data is the stack data (a raw dump of the memory pointed to by the stack pointer at the time of sampling). dyn_size is the amount of data actually dumped (can be less than size). Note that dyn_size is omitted if size is 0.

weight

If PERF_SAMPLE_WEIGHT or PERF_SAMPLE_WEIGHT_STRUCT is enabled, then a 64-bit value provided by the hardware is recorded that indicates how costly the event was. This allows expensive events to stand out more clearly in profile files.

data_src

If PERF_SAMPLE_DATA_SRC is enabled, then a 64-bit value is recorded that is made up of the following fields:

mem_op

Type of opcode, a bitwise combination of:

PERF_MEM_OP_NA	Not available
PERF_MEM_OP_LOAD	Load instruction
PERF_MEM_OP_STORE	Store instruction
PERF_MEM_OP_PFETCH	Prefetch
PERF_MEM_OP_EXEC	Executable code

mem_lvl

Memory hierarchy level hit or miss, a bitwise combination of the follow?

ing, shifted left by PERF_MEM_LVL_SHIFT:

PERF_MEM_LVL_NA	Not available
PERF_MEM_LVL_HIT	Hit
PERF_MEM_LVL_MISS	Miss
PERF_MEM_LVL_L1	Level 1 cache
PERF_MEM_LVL_LFB	Line fill buffer
PERF_MEM_LVL_L2	Level 2 cache
PERF_MEM_LVL_L3	Level 3 cache
PERF_MEM_LVL_LOC_RAM	Local DRAM
PERF_MEM_LVL_REM_RAM1	Remote DRAM 1 hop
PERF_MEM_LVL_REM_RAM2	Remote DRAM 2 hops
PERF_MEM_LVL_REM_CCE1	Remote cache 1 hop
PERF_MEM_LVL_REM_CCE2	Remote cache 2 hops
PERF_MEM_LVL_IO	I/O memory
PERF_MEM_LVL_UNC	Uncached memory

mem_snoop

Snoop mode, a bitwise combination of the following, shifted left by

PERF_MEM_SNOOP_SHIFT:

PERF_MEM_SNOOP_NA	Not available
PERF_MEM_SNOOP_NONE	No snoop
PERF_MEM_SNOOP_HIT	Snoop hit
PERF_MEM_SNOOP_MISS	Snoop miss
PERF_MEM_SNOOP_HITM	Snoop hit modified

mem_lock

Lock instruction, a bitwise combination of the following, shifted left by

PERF_MEM_LOCK_SHIFT:

PERF_MEM_LOCK_NA	Not available
PERF_MEM_LOCK_LOCKED	Locked transaction

mem_dtlb

TLB access hit or miss, a bitwise combination of the following, shifted left by PERF_MEM_TLB_SHIFT:

PERF_MEM_TLB_NA	Not available
PERF_MEM_TLB_HIT	Hit

PERF_MEM_TLB_MISS	Miss
PERF_MEM_TLB_L1	Level 1 TLB
PERF_MEM_TLB_L2	Level 2 TLB
PERF_MEM_TLB_WK	Hardware walker
PERF_MEM_TLB_OS	OS fault handler

transaction

If the PERF_SAMPLE_TRANSACTION flag is set, then a 64-bit field is recorded describing the sources of any transactional memory aborts.

The field is a bitwise combination of the following values:

PERF_TXN_ELISION

Abort from an elision type transaction (Intel-CPU-specific).

PERF_TXN_TRANSACTION

Abort from a generic transaction.

PERF_TXN_SYNC

Synchronous abort (related to the reported instruction).

PERF_TXN_ASYNC

Asynchronous abort (not related to the reported instruction).

PERF_TXN_RETRY

Retryable abort (retrying the transaction may have succeeded).

PERF_TXN_CONFLICT

Abort due to memory conflicts with other threads.

PERF_TXN_CAPACITY_WRITE

Abort due to write capacity overflow.

PERF_TXN_CAPACITY_READ

Abort due to read capacity overflow.

In addition, a user-specified abort code can be obtained from the high 32 bits of the field by shifting right by PERF_TXN_ABORT_SHIFT and masking with the value PERF_TXN_ABORT_MASK.

abi

regs[weight(mask)]

If PERF_SAMPLE_REGS_INTR is enabled, then the user CPU registers are recorded.

The abi field is one of PERF_SAMPLE_REGS_ABI_NONE, PERF_SAMPLE_REGS_ABI_32, *Page 39/57*

or PERF_SAMPLE_REGS_ABI_64.

The `regs` field is an array of the CPU registers that were specified by the `sample_regs_intr` attr field. The number of values is the number of bits set in the `sample_regs_intr` bit mask.

`phys_addr`

If the PERF_SAMPLE_PHYS_ADDR flag is set, then the 64-bit physical address is recorded.

`cgroup`

If the PERF_SAMPLE_CGROUP flag is set, then the 64-bit cgroup ID (for the `perf_event` subsystem) is recorded. To get the pathname of the cgroup, the ID should match to one in a PERF_RECORD_CGROUP.

`data_page_size`

If the PERF_SAMPLE_DATA_PAGE_SIZE flag is set, then the 64-bit page size value of the data address is recorded.

`code_page_size`

If the PERF_SAMPLE_CODE_PAGE_SIZE flag is set, then the 64-bit page size value of the ip address is recorded.

`size`

`data[size]`

If PERF_SAMPLE_AUX is enabled, a snapshot of the aux buffer is recorded.

PERF_RECORD_MMAP2

This record includes extended information on `mmap(2)` calls returning executable mappings. The format is similar to that of the PERF_RECORD_MMAP record, but includes extra values that allow uniquely identifying shared mappings. Depending on the PERF_RECORD_MISC_MMAP_BUILD_ID bit in the header, the extra values have different layout and meanings.

```
struct {  
    struct perf_event_header header;  
    u32  pid;  
    u32  tid;  
    u64  addr;  
    u64  len;  
    u64  pgoff;
```

```

union {
    struct {
        u32  maj;
        u32  min;
        u64  ino;
        u64  ino_generation;
    };
    struct { /* if PERF_RECORD_MISC_MMAP_BUILD_ID */
        u8  build_id_size;
        u8  __reserved_1;
        u16 __reserved_2;
        u8  build_id[20];
    };
};

u32  prot;
u32  flags;
char filename[];
struct sample_id sample_id;
};

```

pid is the process ID.

tid is the thread ID.

addr is the address of the allocated memory.

len is the size of the allocated memory.

pgoff is the page offset of the allocated memory.

maj is the major ID of the underlying device.

min is the minor ID of the underlying device.

ino is the inode number.

ino_generation

is the inode generation.

build_id_size

is the actual size of build_id field (up to 20).

build_id

is a raw data to identify a binary.

prot is the protection information.

flags is the flags information.

filename

is a string describing the backing of the allocated memory.

PERF_RECORD_AUX (since Linux 4.1)

This record reports that new data is available in the separate AUX buffer region.

```
struct {  
    struct perf_event_header header;  
    u64 aux_offset;  
    u64 aux_size;  
    u64 flags;  
    struct sample_id sample_id;  
};
```

aux_offset

offset in the AUX mmap region where the new data begins.

aux_size

size of the data made available.

flags describes the AUX update.

PERF_AUX_FLAG_TRUNCATED

if set, then the data returned was truncated to fit the available
buffer size.

PERF_AUX_FLAG_OVERWRITE

if set, then the data returned has overwritten previous data.

PERF_RECORD_ITRACE_START (since Linux 4.1)

This record indicates which process has initiated an instruction trace event, al?

lowing tools to properly correlate the instruction addresses in the AUX buffer
with the proper executable.

```
struct {  
    struct perf_event_header header;  
    u32 pid;  
    u32 tid;  
};
```

pid process ID of the thread starting an instruction trace.

tid thread ID of the thread starting an instruction trace.

PERF_RECORD_LOST_SAMPLES (since Linux 4.2)

When using hardware sampling (such as Intel PEBS) this record indicates some number of samples that may have been lost.

```
struct {  
    struct perf_event_header header;  
    u64 lost;  
    struct sample_id sample_id;  
};
```

lost the number of potentially lost samples.

PERF_RECORD_SWITCH (since Linux 4.3)

This record indicates a context switch has happened. The PERF_RECORD_MISC_SWITCH_OUT bit in the misc field indicates whether it was a context switch into or away from the current process.

```
struct {  
    struct perf_event_header header;  
    struct sample_id sample_id;  
};
```

PERF_RECORD_SWITCH_CPU_WIDE (since Linux 4.3)

As with PERF_RECORD_SWITCH this record indicates a context switch has happened, but it only occurs when sampling in CPU-wide mode and provides additional information on the process being switched to/from. The PERF_RECORD_MISC_SWITCH_OUT bit in the misc field indicates whether it was a context switch into or away from the current process.

```
struct {  
    struct perf_event_header header;  
    u32 next_prev_pid;  
    u32 next_prev_tid;  
    struct sample_id sample_id;  
};
```

next_prev_pid

The process ID of the previous (if switching in) or next (if switching out) process on the CPU.

next_prev_tid

The thread ID of the previous (if switching in) or next (if switching out) thread on the CPU.

PERF_RECORD_NAMESPACES (since Linux 4.11)

This record includes various namespace information of a process.

```
struct {  
    struct perf_event_header header;  
    u32  pid;  
    u32  tid;  
    u64  nr_namespaces;  
    struct { u64 dev, inode } [nr_namespaces];  
    struct sample_id sample_id;  
};
```

pid is the process ID

tid is the thread ID

nr_namespace

is the number of namespaces in this record

Each namespace has dev and inode fields and is recorded in the fixed position

like below:

NET_NS_INDEX=0

Network namespace

UTS_NS_INDEX=1

UTS namespace

IPC_NS_INDEX=2

IPC namespace

PID_NS_INDEX=3

PID namespace

USER_NS_INDEX=4

User namespace

MNT_NS_INDEX=5

Mount namespace

CGROUP_NS_INDEX=6

Cgroup namespace

PERF_RECORD_KSYMBOL (since Linux 5.0)

This record indicates kernel symbol register/unregister events.

```
struct {  
    struct perf_event_header header;  
    u64  addr;  
    u32  len;  
    u16  ksym_type;  
    u16  flags;  
    char name[];  
    struct sample_id sample_id;  
};
```

addr is the address of the kernel symbol.

len is the size of the kernel symbol.

ksym_type

is the type of the kernel symbol. Currently the following types are available:

PERF_RECORD_KSYMBOL_TYPE_BPF

The kernel symbol is a BPF function.

flags If the PERF_RECORD_KSYMBOL_FLAGS_UNREGISTER is set, then this event is for unregistering the kernel symbol.

PERF_RECORD_BPF_EVENT (since Linux 5.0)

This record indicates BPF program is loaded or unloaded.

```
struct {  
    struct perf_event_header header;  
    u16 type;  
    u16 flags;  
    u32 id;  
    u8 tag[BPF_TAG_SIZE];  
    struct sample_id sample_id;  
};
```

type is one of the following values:

PERF_BPF_EVENT_PROG_LOAD

A BPF program is loaded

PERF_BPF_EVENT_PROG_UNLOAD

A BPF program is unloaded

`id` is the ID of the BPF program.

`tag` is the tag of the BPF program. Currently, `BPF_TAG_SIZE` is defined as 8.

PERF_RECORD_CGROUP (since Linux 5.7)

This record indicates a new cgroup is created and activated.

```
struct {  
    struct perf_event_header header;  
    u64 id;  
    char path[];  
    struct sample_id sample_id;  
};
```

`id` is the cgroup identifier. This can be also retrieved by `name_to_handle_at(2)` on the cgroup path (as a file handle).

`dle_at(2)` on the cgroup path (as a file handle).

`path` is the path of the cgroup from the root.

PERF_RECORD_TEXT_POKE (since Linux 5.8)

This record indicates a change in the kernel text. This includes addition and removal of the text and the corresponding size is zero in this case.

```
struct {  
    struct perf_event_header header;  
    u64 addr;  
    u16 old_len;  
    u16 new_len;  
    u8 bytes[];  
    struct sample_id sample_id;  
};
```

`addr` is the address of the change

`old_len`

is the old size

`new_len`

is the new size

`bytes` contains old bytes immediately followed by new bytes.

Events can be set to notify when a threshold is crossed, indicating an overflow. Overflow conditions can be captured by monitoring the event file descriptor with `poll(2)`, `select(2)`, or `epoll(7)`. Alternatively, the overflow events can be captured via a signal handler, by enabling I/O signaling on the file descriptor; see the discussion of the `F_SETOWN` and `F_SETSIG` operations in `fcntl(2)`.

Overflows are generated only by sampling events (`sample_period` must have a nonzero value). There are two ways to generate overflow notifications.

The first is to set a `wakeup_events` or `wakeup_watermark` value that will trigger if a certain number of samples or bytes have been written to the mmap ring buffer. In this case, `POLL_IN` is indicated.

The other way is by use of the `PERF_EVENT_IOC_REFRESH` ioctl. This ioctl adds to a counter that decrements each time the event overflows. When nonzero, `POLL_IN` is indicated, but once the counter reaches 0 `POLL_HUP` is indicated and the underlying event is disabled.

Refreshing an event group leader refreshes all siblings and refreshing with a parameter of 0 currently enables infinite refreshes; these behaviors are unsupported and should not be relied on.

Starting with Linux 3.18, `POLL_HUP` is indicated if the event being monitored is attached to a different process and that process exits.

rdpmc instruction

Starting with Linux 3.4 on x86, you can use the `rdpmc` instruction to get low-latency reads without having to enter the kernel. Note that using `rdpmc` is not necessarily faster than other methods for reading event values.

Support for this can be detected with the `cap_usr_rdpmc` field in the mmap page; documentation on how to calculate event values can be found in that section.

Originally, when `rdpmc` support was enabled, any process (not just ones with an active perf event) could use the `rdpmc` instruction to access the counters. Starting with Linux 4.0, `rdpmc` support is only allowed if an event is currently enabled in a process's context. To restore the old behavior, write the value 2 to `/sys/devices/cpu/rdpmc`.

perf_event ioctl calls

Various ioctls act on `perf_event_open()` file descriptors:

`PERF_EVENT_IOC_ENABLE`

This enables the individual event or event group specified by the file descriptor argument.

If the `PERF_IOC_FLAG_GROUP` bit is set in the `ioctl` argument, then all events in a group are enabled, even if the event specified is not the group leader (but see BUGS).

`PERF_EVENT_IOC_DISABLE`

This disables the individual counter or event group specified by the file descriptor argument.

Enabling or disabling the leader of a group enables or disables the entire group; that is, while the group leader is disabled, none of the counters in the group will count. Enabling or disabling a member of a group other than the leader affects only that counter; disabling a non-leader stops that counter from counting but doesn't affect any other counter.

If the `PERF_IOC_FLAG_GROUP` bit is set in the `ioctl` argument, then all events in a group are disabled, even if the event specified is not the group leader (but see BUGS).

`PERF_EVENT_IOC_REFRESH`

Non-inherited overflow counters can use this to enable a counter for a number of overflows specified by the argument, after which it is disabled. Subsequent calls of this `ioctl` add the argument value to the current count. An overflow notification with `POLL_IN` set will happen on each overflow until the count reaches 0; when that happens a notification with `POLL_HUP` set is sent and the event is disabled. Using an argument of 0 is considered undefined behavior.

`PERF_EVENT_IOC_RESET`

Reset the event count specified by the file descriptor argument to zero. This resets only the counts; there is no way to reset the multiplexing `time_enabled` or `time_running` values.

If the `PERF_IOC_FLAG_GROUP` bit is set in the `ioctl` argument, then all events in a group are reset, even if the event specified is not the group leader (but see BUGS).

`PERF_EVENT_IOC_PERIOD`

This updates the overflow period for the event.

Since Linux 3.7 (on ARM) and Linux 3.14 (all other architectures), the new period takes effect immediately. On older kernels, the new period did not take effect until after the next overflow.

The argument is a pointer to a 64-bit value containing the desired new period.

Prior to Linux 2.6.36, this `ioctl` always failed due to a bug in the kernel.

`PERF_EVENT_IOC_SET_OUTPUT`

This tells the kernel to report event notifications to the specified file descriptor rather than the default one. The file descriptors must all be on the same CPU.

The argument specifies the desired file descriptor, or -1 if output should be ignored.

`PERF_EVENT_IOC_SET_FILTER` (since Linux 2.6.33)

This adds an ftrace filter to this event.

The argument is a pointer to the desired ftrace filter.

`PERF_EVENT_IOC_ID` (since Linux 3.12)

This returns the event ID value for the given event file descriptor.

The argument is a pointer to a 64-bit unsigned integer to hold the result.

`PERF_EVENT_IOC_SET_BPF` (since Linux 4.1)

This allows attaching a Berkeley Packet Filter (BPF) program to an existing kprobe tracepoint event. You need `CAP_PERFMON` (since Linux 5.8) or `CAP_SYS_ADMIN` privileges to use this `ioctl`.

The argument is a BPF program file descriptor that was created by a previous `bpf(2)` system call.

`PERF_EVENT_IOC_PAUSE_OUTPUT` (since Linux 4.7)

This allows pausing and resuming the event's ring-buffer. A paused ring-buffer does not prevent generation of samples, but simply discards them. The discarded samples are considered lost, and cause a `PERF_RECORD_LOST` sample to be generated when possible. An overflow signal may still be triggered by the discarded sample even though the ring-buffer remains empty.

The argument is an unsigned 32-bit integer. A nonzero value pauses the ring-buffer, while a zero value resumes the ring-buffer.

`PERF_EVENT_MODIFY_ATTRIBUTES` (since Linux 4.17)

This allows modifying an existing event without the overhead of closing and reopening a new event. Currently this is supported only for breakpoint events.

The argument is a pointer to a `perf_event_attr` structure containing the updated event settings.

`PERF_EVENT_IOC_QUERY_BPF` (since Linux 4.16)

This allows querying which Berkeley Packet Filter (BPF) programs are attached to an

existing kprobe tracepoint. You can only attach one BPF program per event, but you can have multiple events attached to a tracepoint. Querying this value on one tracepoint event returns the ID of all BPF programs in all events attached to the tracepoint. You need CAP_PERFMON (since Linux 5.8) or CAP_SYS_ADMIN privileges to use this ioctl.

The argument is a pointer to a structure

```
struct perf_event_query_bpf {  
    __u32  ids_len;  
    __u32  prog_cnt;  
    __u32  ids[0];  
};
```

The `ids_len` field indicates the number of ids that can fit in the provided ids array.

The `prog_cnt` value is filled in by the kernel with the number of attached BPF programs. The `ids` array is filled with the ID of each attached BPF program. If there are more programs than will fit in the array, then the kernel will return `ENOSPC` and `ids_len` will indicate the number of program IDs that were successfully copied.

Using prctl(2)

A process can enable or disable all currently open event groups using the `prctl(2)` `PR_TASK_PERF_EVENTS_ENABLE` and `PR_TASK_PERF_EVENTS_DISABLE` operations. This applies only to events created locally by the calling process. This does not apply to events created by other processes attached to the calling process or inherited events from a parent process. Only group leaders are enabled and disabled, not any other members of the groups.

perf_event related configuration files

Files in `/proc/sys/kernel/`

`/proc/sys/kernel/perf_event_paranoid`

The `perf_event_paranoid` file can be set to restrict access to the performance counters.

- 2 allow only user-space measurements (default since Linux 4.6).
- 1 allow both kernel and user measurements (default before Linux 4.6).
- 0 allow access to CPU-specific data but not raw tracepoint samples.
- 1 no restrictions.

The existence of the `perf_event_paranoid` file is the official method for determining if a kernel supports `perf_event_open()`.

`/proc/sys/kernel/perf_event_max_sample_rate`

This sets the maximum sample rate. Setting this too high can allow users to sample at a rate that impacts overall machine performance and potentially lock up the machine. The default value is 100000 (samples per second).

`/proc/sys/kernel/perf_event_max_stack`

This file sets the maximum depth of stack frame entries reported when generating a call trace.

`/proc/sys/kernel/perf_event_mlock_kb`

Maximum number of pages an unprivileged user can mlock(2). The default is 516 (kB).

Files in `/sys/bus/event_source/devices/`

Since Linux 2.6.34, the kernel supports having multiple PMUs available for monitoring. Information on how to program these PMUs can be found under `/sys/bus/event_source/devices/`. Each subdirectory corresponds to a different PMU.

`/sys/bus/event_source/devices/*/type` (since Linux 2.6.38)

This contains an integer that can be used in the type field of `perf_event_attr` to indicate that you wish to use this PMU.

`/sys/bus/event_source/devices/cpu/rdpmc` (since Linux 3.4)

If this file is 1, then direct user-space access to the performance counter registers is allowed via the `rdpmc` instruction. This can be disabled by echoing 0 to the file.

As of Linux 4.0 the behavior has changed, so that 1 now means only allow access to processes with active perf events, with 2 indicating the old allow-anyone-access behavior.

`/sys/bus/event_source/devices/*/format/` (since Linux 3.4)

This subdirectory contains information on the architecture-specific subfields available for programming the various config fields in the `perf_event_attr` struct.

The content of each file is the name of the config field, followed by a colon, followed by a series of integer bit ranges separated by commas. For example, the file `event` may contain the value `config1:1,6-10,44` which indicates that event is an attribute that occupies bits 1, 6-10, and 44 of `perf_event_attr::config1`.

`/sys/bus/event_source/devices/*/events/` (since Linux 3.4)

This subdirectory contains files with predefined events. The contents are strings describing the event settings expressed in terms of the fields found in the previously mentioned `./format/` directory. These are not necessarily complete lists of all events supported by a PMU, but usually a subset of events deemed useful or interesting.

The content of each file is a list of attribute names separated by commas. Each entry has an optional value (either hex or decimal). If no value is specified, then it is assumed to be a single-bit field with a value of 1. An example entry may look like this: `event=0x2,inv,ldlat=3`.

`/sys/bus/event_source/devices/*/uevent`

This file is the standard kernel device interface for injecting hotplug events.

`/sys/bus/event_source/devices/*/cpumask` (since Linux 3.7)

The `cpumask` file contains a comma-separated list of integers that indicate a representative CPU number for each socket (package) on the motherboard. This is needed when setting up uncore or northbridge events, as those PMUs present socket-wide events.

RETURN VALUE

On success, `perf_event_open()` returns the new file descriptor. On error, -1 is returned and `errno` is set to indicate the error.

ERRORS

The errors returned by `perf_event_open()` can be inconsistent, and may vary across processor architectures and performance monitoring units.

E2BIG Returned if the `perf_event_attr` size value is too small (smaller than `PERF_ATTR_SIZE_VER0`), too big (larger than the page size), or larger than the kernel supports and the extra bytes are not zero. When **E2BIG** is returned, the `perf_event_attr` size field is overwritten by the kernel to be the size of the structure it was expecting.

EACCES Returned when the requested event requires `CAP_PERFMON` (since Linux 5.8) or `CAP_SYS_ADMIN` permissions (or a more permissive `perf_event` paranoid setting). Some common cases where an unprivileged process may encounter this error: attaching to a process owned by a different user; monitoring all processes on a given CPU (i.e., specifying the `pid` argument as -1); and not setting `exclude_kernel` when the `paranoid` setting requires it.

EBADF Returned if the `group_fd` file descriptor is not valid, or, if `PERF_FLAG_PID_CGROUP` is set, the cgroup file descriptor in `pid` is not valid.

EBUSY (since Linux 4.1)

Returned if another event already has exclusive access to the PMU.

EFAULT Returned if the `attr` pointer points at an invalid memory address.

EINTR Returned when trying to mix perf and ftrace handling for a uprobe.

EINVAL Returned if the specified event is invalid. There are many possible reasons for this. A not-exhaustive list: `sample_freq` is higher than the maximum setting; the cpu to monitor does not exist; `read_format` is out of range; `sample_type` is out of range; the flags value is out of range; exclusive or pinned set and the event is not a group leader; the event config values are out of range or set reserved bits; the generic event selected is not supported; or there is not enough room to add the selected event.

EMFILE Each opened event uses one file descriptor. If a large number of events are opened, the per-process limit on the number of open file descriptors will be reached, and no more events can be created.

ENODEV Returned when the event involves a feature not supported by the current CPU.

ENOENT Returned if the type setting is not valid. This error is also returned for some unsupported generic events.

ENOSPC Prior to Linux 3.3, if there was not enough room for the event, `ENOSPC` was returned. In Linux 3.3, this was changed to `EINVAL`. `ENOSPC` is still returned if you try to add more breakpoint events than supported by the hardware.

ENOSYS Returned if `PERF_SAMPLE_STACK_USER` is set in `sample_type` and it is not supported by hardware.

EOPNOTSUPP

Returned if an event requiring a specific hardware feature is requested but there is no hardware support. This includes requesting low-skid events if not supported, branch tracing if it is not available, sampling if no PMU interrupt is available, and branch stacks for software events.

EOVERFLOW (since Linux 4.8)

Returned if `PERF_SAMPLE_CALLCHAIN` is requested and `sample_max_stack` is larger than the maximum specified in `/proc/sys/kernel/perf_event_max_stack`.

EPERM Returned on many (but not all) architectures when an unsupported `exclude_hv`, `ex?`

clude_idle, exclude_user, or exclude_kernel setting is specified.

It can also happen, as with EACCES, when the requested event requires CAP_PERFMON (since Linux 5.8) or CAP_SYS_ADMIN permissions (or a more permissive perf_event parano? noid setting). This includes setting a breakpoint on a kernel address, and (since Linux 3.13) setting a kernel function-trace tracepoint.

ESRCH Returned if attempting to attach to a process that does not exist.

STANDARDS

Linux.

HISTORY

perf_event_open() was introduced in Linux 2.6.31 but was called perf_counter_open(). It was renamed in Linux 2.6.32.

NOTES

The official way of knowing if perf_event_open() support is enabled is checking for the existence of the file /proc/sys/kernel/perf_event_parano? id.

CAP_PERFMON capability (since Linux 5.8) provides secure approach to performance monitoring and observability operations in a system according to the principle of least privilege (POSIX IEEE 1003.1e). Accessing system performance monitoring and observability operations using CAP_PERFMON rather than the much more powerful CAP_SYS_ADMIN excludes chances to misuse credentials and makes operations more secure. CAP_SYS_ADMIN usage for secure system performance monitoring and observability is discouraged in favor of the CAP_PERFMON capability.

BUGS

The F_SETOWN_EX option to fcntl(2) is needed to properly get overflow signals in threads. This was introduced in Linux 2.6.32.

Prior to Linux 2.6.33 (at least for x86), the kernel did not check if events could be scheduled together until read time. The same happens on all known kernels if the NMI watchdog is enabled. This means to see if a given set of events works you have to perf_event_open(), start, then read before you know for sure you can get valid measurements.

Prior to Linux 2.6.34, event constraints were not enforced by the kernel. In that case, some events would silently return "0" if the kernel scheduled them in an improper counter slot.

Prior to Linux 2.6.34, there was a bug when multiplexing where the wrong results could be returned.

Kernels from Linux 2.6.35 to Linux 2.6.39 can quickly crash the kernel if "inherit" is enabled and many threads are started.

Prior to Linux 2.6.35, PERF_FORMAT_GROUP did not work with attached processes.

There is a bug in the kernel code between Linux 2.6.36 and Linux 3.0 that ignores the "wake_up_event" field and acts as if a wakeup_event was chosen if the union has a nonzero value in it.

From Linux 2.6.31 to Linux 3.4, the PERF_IOC_FLAG_GROUP ioctl argument was broken and would repeatedly operate on the event specified rather than iterating across all sibling events in a group.

From Linux 3.4 to Linux 3.11, the mmap cap_usr_rdpmc and cap_usr_time bits mapped to the same location. Code should migrate to the new cap_user_rdpmc and cap_user_time fields instead.

Always double-check your results! Various generalized events have had wrong values. For example, retired branches measured the wrong thing on AMD machines until Linux 2.6.35.

EXAMPLES

The following is a short example that measures the total instruction count of a call to printf(3).

```
#include <err.h>
```

```
#include <linux/perf_event.h>
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <sys/ioctl.h>
```

```
#include <sys/syscall.h>
```

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
static long
```

```
perf_event_open(struct perf_event_attr *hw_event, pid_t pid,  
                int cpu, int group_fd, unsigned long flags)
```

```
{
```

```
    int ret;
```

```
    ret = syscall(SYS_perf_event_open, hw_event, pid, cpu,  
                group_fd, flags);
```

```

    return ret;
}

int
main(void)
{
    int            fd;

    long long      count;

    struct perf_event_attr pe;

    memset(&pe, 0, sizeof(pe));

    pe.type = PERF_TYPE_HARDWARE;

    pe.size = sizeof(pe);

    pe.config = PERF_COUNT_HW_INSTRUCTIONS;

    pe.disabled = 1;

    pe.exclude_kernel = 1;

    pe.exclude_hv = 1;

    fd = perf_event_open(&pe, 0, -1, -1, 0);

    if (fd == -1)

        err(EXIT_FAILURE, "Error opening leader %llx\n", pe.config);

    if (ioctl(fd, PERF_EVENT_IOC_RESET, 0) == -1)

        err(EXIT_FAILURE, "PERF_EVENT_IOC_RESET");

    if (ioctl(fd, PERF_EVENT_IOC_ENABLE, 0) == -1)

        err(EXIT_FAILURE, "PERF_EVENT_IOC_ENABLE");

    printf("Measuring instruction count for this printf\n");

    if (ioctl(fd, PERF_EVENT_IOC_DISABLE, 0) == -1)

        err(EXIT_FAILURE, "PERF_EVENT_IOC_DISABLE");

    if (read(fd, &count, sizeof(count)) != sizeof(count))

        err(EXIT_FAILURE, "read");

    printf("Used %lld instructions\n", count);

    if (close(fd) == -1)

        err(EXIT_FAILURE, "close");

}

```

SEE ALSO

perf(1), fcntl(2), mmap(2), open(2), prctl(2), read(2)

