perf_event_open(2) — Linux manual page

NAME top

perf_event_open - set up performance monitoring

SYNOPSIS top

```
#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 top

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 */

                __reserved_1   : 30;

          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 length */
              __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 */

      };
```

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.

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 reported 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.

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 support events that are not covered by the "generalized" events. These are implementation 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.

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.

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**.

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.

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.

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.

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.

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 overwritable 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.

auxevent (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).

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 length 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.

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.

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 */
    } 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 */
};
```

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_enabled* 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*.

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_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_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.

RETURN VALUE top

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

ERRORS top

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*, *exclude_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 paranoid 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.

VERSION top

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

CONFORMING TO top

This **perf_event_open**() system call Linux-specific and should not be used in programs intended to be portable.

NOTES top

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_paranoid*.

CAP_PERFMON capability (since Linux 5.8) provides secure approach to performance monitoring and observability operations in a system according to the principal 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 top

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 "watermark" 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 top

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

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/ioctl.h>
#include <linux/perf_event.h>
#include <asm/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(__NR_perf_event_open, hw_event, pid, cpu,
                   group_fd, flags);
    return ret;
}

int
main(int argc, char *argv[])
{
    struct perf_event_attr pe;
    long long count;
    int fd;

    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) {
       fprintf(stderr, "Error opening leader %llx\n", pe.config);
       exit(EXIT_FAILURE);
    }

    ioctl(fd, PERF_EVENT_IOC_RESET, 0);
    ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);

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

    ioctl(fd, PERF_EVENT_IOC_DISABLE, 0);
    read(fd, &count, sizeof(count));

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

    close(fd);
}
```

SEE ALSO top

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

Documentation/admin-guide/perf-security.rst in the kernel source tree

COLOPHON top

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.