# The perfmon2 interface specification

Stephane Eranian
Internet Systems and Storage Laboratory
HP Laboratories
HPL-2004-200(R.1)
February 7, 2005*

E-mail: eranian@hpl.hp.com

Performance Monitoring Unit, PMU, performance tools, hardware counters, IPF, IA-64 Linux, perfmon kernel interface

Monitoring program execution is becoming key to achieving world class performance. All modern processors implement a sophisticated set of hardware performance counters to collect a lot of micro-architectural events which are important clues for software optimizations. Yet there is no standardized interface to access those counters which makes developing portable performance tools challenging. We have designed a powerful monitoring interface to access the performance counters present on all modern processors. The interface is generic and does not compromise access to processor specific features. It enables a diversity of tools to be developed or ported. In particular tools can collect simple counts or profiles on a per-thread or system-wide basis. The interface introduces several innovations such as customizable kernel-level sampling buffer formats, time and overflow-based event set multiplexing. An implementation exists for Linux 2.6 kernel for the Itanium Processor Family (IPF). Several open-source and commercial tools based on interface are available from HP and others. This document presents version 0.6 of the full specification of the interface including the Itanium-specific features.

* Internal Accession Date Only
Approved for External Publication

# Contents

# Chapter 1

# Document revision history

## 1.1 Important Warning

This document does not describe the existing 2.6 Linux[1] kernel implementation of the interface for the Itanium Processor Family (Linux/ia64). Instead, this document provides a base for the principles behind the Linux/ia64 implementation.

## 1.2 Revision summary

| Revision | Date | Description |
|---|---|---|
| | 02/07/2005 | release version 0.6 (first public release) |

[1] Linux is a U.S. registered trademark of Linus Torvalds

# Chapter 2

# Introduction

Nowadays monitoring program execution is becoming key to achieving world class performance. Performance monitoring is the action of collecting performance-related information during the execution of a program. There are two levels of monitoring commonly used today:

- Program-level monitoring collects information such as basic block call counts. The information is collected through instrumentation of the program. This is achieved at compile time or dynamically by tools like HP Caliper [6], PIN [4], or Paradyn [16] for instance.
- Hardware level monitoring collects information at the micro-architectural level such as the number of caches misses. It requires hardware support in the processor. This is typically implemented by a Performance Monitoring Unit (PMU) which exports a set of programmable counters. Monitored program do not need to be recompiled nor modified to collect the performance information.

Performance monitoring has been used for a number of years by compilers to optimize code generation. On many platforms, compilers support a technique called Profile Based Optimization (PBO), where the program is first instrumented, then it is run, and finally the generated execution profile is fed back to the compiler to adjust certain optimizations. For instance, the profile can indicate the most commonly used path inside a function. With this information, it is possible to reorder the basic blocks of the path such that the path becomes straight, i.e., no more branches. Even though today, PBO is mostly done with program-level monitoring, it is moving towards using the hardware counters to gain access to lower-level information. For instance, by knowing where cache misses are, the compiler could insert prefetch instructions.

Similarly, managed runtime environments (MREs), such as Java, are generating code of the fly and they can certainly benefit from performance monitoring information to tweak the code they generate. Both monitoring levels could be used, but hardware level monitoring is more adequate to low level tuning of the code.

Finally, with the rise of multi-threaded processors, it becomes useful for operating system scheduler to understand the execution profile of a thread, especially the cache behavior, to adjust the scheduling decision to avoid cache trashing, for instance. Such profile could fairly easily be obtained using hardware level monitoring.

These few examples show that monitoring is becoming a very common practice that is applied at various levels of a computer system. In particular, hardware monitoring looks promising because it does not require modifications to the monitored code. This is an important point because it means

that source code access is not necessary, the monitored code is running at close to the original speed, very-low level code paths of the kernel can be monitored. Furthermore, hardware monitoring can help solve difficult problems which otherwise cannot be solved using standard instrumentation techniques such as cache, TLB related issues.

By definition, the PMU is a very specific piece of hardware which operates at the micro-architectural level. It can change a lot from one processor implementation to another and sometimes inside the same processor family. For instance, in the Itanium Processor Family [8] the number of events that can be measured goes from about 200 for Itanium® to about 500 for Itanium® 2. Events with the same name are not necessarily encoded the same way. The width of the counters goes from 32-bit to 47 bits between the two generations of processor. Until recently, PMUs have often been documented poorly. They were kept secret and were used internally by hardware vendors during processor bring up and for optimization of key proprietary applications.

Although the PMU can harvest very useful information, it is not always exploited to the fullest of its capabilities. Besides the documentation problem, this can be attributed to the lack of a standard kernel interface to access the PMU. It is very common that access to the PMU registers requires running at the most privileged level of execution, i.e., in the operating system kernel. Hence, it is not possible to develop only a user level library for the interface, some kernel level support is required.

All modern processors incorporate a PMU. Depending on the processor family the level of public information varies. But it is clear, that the functionalities of each PMU model can vary greatly. Many PMU models go beyond just providing simple counters, many can capture addresses, latencies and branches for instance. Similarly, monitoring tools have very different needs depending on what they measure. For instance, some tools collect simple counts while other collect profiles. Some tools operates on a per-thread basis while others measure on a system-wide basis, i.e., across all threads and possibly across multiple processors.

All this diversity is good because it means that a lot of interesting information can be collected. However it can also be overwhelming because it can be hard to develop tools that can be portable across multiple PMU models and operating systems. There is obviously a challenge at the hardware level but also at the software level because there is no standard kernel-level interface to access the PMU. This did not use to be a real problem because PMU were undocumented and operating systems were proprietary. Open-source operating systems, such as Linux which runs across all major hardware platforms, are changing the dynamics and making the lack of a standard monitoring interface a pressing issue that needs to be resolved now.

To understand the issue clearly, it is important to distinguish between hardware diversity and *software-imposed* diversity. Many hardware differences can be managed with some level of indirection or a virtualization layer. Obviously, if a tool is exploiting a unique feature of a PMU then it could be hard to port it to another PMU. Differences at the kernel interface level may also be difficult to deal with and could limit the ability to port tools. Existing interfaces do not always offer the same level of functionalities. For instance, some do not offer per-thread monitoring, others do not support collecting profiles in system-wide mode. Those artificial differences come on top of the hardware differences and may become show-stopper because they could impose deep structural changes to tools. Those differences are totally orthogonal to the PMU diversity and are most commonly explained by software design limitations which could be avoided by using a *standardized interface*. Such interface would guarantee that the same set of generic features would be implemented across all platforms. With such interface in place, tool developer could focus on the core added-value of their tool and not have to waste time working around artificial software differences.

The Linux operating system as it stands today offers access to the PMU hardware on several platforms such as IA-32, IA-64, PowerPC, and X86_64. The problem here is not the lack of interface but rather the multiplicity of interfaces. Just on the leading IA-32 platform, The VTUNE [11] performance analyzer

uses its own kernel interface which is implemented by an open-source device driver. Then, there is the OProfile [3] interface used by tools such as Prospect [13]. There is also the perfctr [12] interface used by all the tools based on the PAPI [14] toolkit. This approach of one tool one kernel interface is dangerous because there is clearly code duplication but more importantly there is no coordination between the various interfaces which can all exist more or less at the same time and share access to the same PMU resource. Such duplication of interfaces creates fragmentation and does not makes it very attractive for developers, especially for ISVs, to create or port professional-grade tools. Up until recently, none of these interfaces was even part of the standard kernel distributions and users had to download patches or kernel modules. In order for the development of tools to really become attractive and pervasive, it is important that the interface be standardized and implemented in the official Linux kernel. This would ensure that it is robust, maintained and shipped by all Linux distributors.

The goal of the work presented in this document is to address the so-called perfmon challenge: *how to design a generic performance monitoring interface that would provide access to all existing and future PMU implementations and that would also support a variety of monitoring tools?*

Despite the diversity, it is possible to design a generic interface by:

- focusing on the providing access to the resource and not on the programming of the resource
- exploiting certain key characteristics of all PMU models

Accessing and programming the PMU are too very distinct operations. The former focuses on access the PMU registers whereas the latter implies knowledge about events and what they measure. It is very unlikely that there will ever be a complete standardization on events, they are just too specific to each implementation. For instance, an IA64_INST_RETIRED event on Itanium® 2 does not necessarily measure the same exact thing as the INSTR_RETIRED event on a Pentium 4 because *retirement* may be defined differently. As such knowledge about events should never be part of a kernel interface and is best encapsulated into user level libraries, such as those provided by the PAPI toolkit or the Performance Counter Library (PCL) [5]. Both packages go further by providing their own set of *standard* events which are then mapped onto the actual events. For instance, there is a PAPI_L1_DCM to measure data cache misses in the first level cache. Tools, such as TAU [15] which use those logical events, can then be ported across PMU models. This approach works for most events yet it is subject to the limitation we just described.

The key characteristic that the interface must leverage is that all PMU models export a set of programmable registers which can be read and written with simple instructions. For instance, all PMU models inside the Itanium Processor Family (IPF) export a set of Performance Monitoring Configuration (PMC) registers and a set of Performance Monitoring Data (PMD) registers. Similarly, the Pentium 4 [10] PMU uses Machine Specific Registers (MSR).

The interface is primarily targeted for the Linux operating system because this is where the issue is really problematic but it can easily be implemented into other open-source or commercial operating systems.

After we begin with the goals of interface and a short section on terminology we present in details the interface, the PMU register mapping scheme, the support for sampling and event-sets, and security. Then a chapter describes the Itanium specific extensions and we conclude with future extensions.

## 2.1 Goals of the interface

The goal of the interface is to establish an industry standard to access the PMU resource, just like the POSIX pthread interface is a standard way of implementing threads of execution. Current trends

in processor and application developments show that the need for performance monitoring support is growing rapidly. The lack of a unified monitoring interface limits the scope of tools to only a one platforms. This, in turn, makes developing commercial tools not very attractive. Similar arguments do apply to academic research projects.

In order to be successful, the interface must meet certain requirements which are presented next.

The paramount requirement is that the interface and its implementation must be an integral part of the operating system kernel. In other words, it must be built-in and come by default with the kernel as shipped by OS vendors. It is not possible to have performance monitoring support be an optional module that one can download from the Internet or recompile separately. This is key to enabling performance tools to be developed on a larger scale. As recent history has shown, having a certain software package bundled with the operating system can be quite critical to its success. Similarly, this requirement also ensures that the implementation is correctly maintained and that it presents a robust level of security.

The interface must be portable across operating systems. Although the primary target is the Linux operating system, it is envisioned that other systems could also adopt it.

The interface must be portable across PMU models. No PMU specific features must be exposed explicitly. For instance, the number of PMU registers must not be part of the interface.

The interface must not prevent access to PMU specific features. It must not use the *least common denominator* approach to satisfy the portability requirement. In particular, the interface must not limit access only to the PMU registers which implement counters. Modern PMUs, oftentimes, implement registers which are used for other purposes, such as buffers for instance. Access to PMU-specific features must be done using the existing framework of the interface whenever possible, i.e., no *back door*.

The interface must be unified across implementations. The set of features not dictated by hardware must be identical across platforms. For instance, the ability to monitor per-thread and on a system-wide basis must always be implemented. This is important to ensure that software using one mode could be ported from one platform to another.

The interface must provide a way to collect simple counts, i.e., accumulate the number of occurrences of PMU events. This is the very basic kind of measurement a tool would want to make.

The interface must provide a way to collect profiles by sampling PMU registers. It must be possible to sample directly from the application. Sampling at the kernel level may be provided to minimize overhead.

The interface must be generic to allow a variety of performance tools to be implemented. Some tools are just collecting simple counts, others are collecting profiles. Both kinds of tool must access the PMU using the same interface.

The interface must allow a performance tool to measure only one thread of execution at a time. This mode of operation is called *per-thread*.

The interface must allow *per-thread* monitoring to work on complicated workloads which are using multiple processes or multiple threads. Monitoring multiple threads at the same time and independently of each other must be supported by the interface.

The interface must allow a performance tool to measure across all processes running on all or a subset of the processors of a system. This mode of operation is called *system-wide*.

The interface must be designed such that it minimizes the amount of PMU specific knowledge necessary inside the operating system kernel. As much as possible such knowledge must be pushed into user level libraries.

The interface must not impose the use of a particular set of user level libraries to program the PMU. Each tool may use its own support library in particular for event encodings and assignments to PMU registers.

The interface must be designed for efficient monitoring. The overhead, especially when sampling, must be kept to a bare minimum to avoid perturbing the execution of the monitored application or system as this could lead to misinterpretations of the collected data.

The interface must scale to large machine configurations. With current processor developments, machines with a single processor will soon disappear from the marketplace. Multi-core and multi-threaded processors will transform simple machines into large SMP machines. For instance, a 4-processor socket machine, which today provides 4-way would become a 16-way machines with dual-core dual-threaded processors. Large machine configurations will soon reach several thousands of processors. Monitoring across large number of processors must be possible with minimal overhead.

The interface must not impose special compilation flags to the operating systems kernel or applications to allow monitoring to function.

## 2.2 Design choices

In the next few sections, the key design choices based on the requirements for the interface are presented.

### 2.2.1 Logical PMU

The first issue is the diversity that exists between the various PMU models. From one architecture to another, the set of registers, how they are named, what they do and how they are related to each other can be very different. The only common characteristic that can be relied upon is that the hardware interface is always composed of registers. A register has a unique name and can be read or written. There are typically two kinds of registers in a PMU:

- control registers to describe what is to be measured, start and stop monitoring
- data registers where the results are stored

To hide the differences in naming and provide for better portability and simplicity, the interface exposes a *logical* view of the PMU to applications. This *logical* PMU is then mapped onto the actual PMU by each implementation. The *logical* PMU exports a uniform set of registers:

- Performance Monitoring Control, or *PMC*, registers
- Performance Monitoring Data, or *PMD*, registers

Each register is identified by a simple (index,value) pair. Each index identifies a unique hardware register. This scheme works well when the hardware PMU uses indexed registers, such as with the Itanium Processor Family PMU, but also when the registers have individual names such as with the Pentium 4 PMU [10].

The interface maintains the distinction between data and control registers because by nature, they do not play the same role.

The mappings to the actual PMU registers can be retrieved by a specific call through the interface. Knowledge of the mapping may be needed to use certain helper libraries which do not use the same mapping as the interface.

The number of PMC and PMD registers as well as what they measure is specific to each PMU model and is not part of the interface.

The interface exposes all PMC and PMD registers as 64-bit registers no matter what the underlying PMU implements. In particular all counters are exported as 64-bit wide. It is common to have hardware counters implemented with only about forty bits.

### 2.2.2 Basic operations on PMU registers

The interface exports basically two operations on PMU registers: read and write. It stays very close to the hardware interface to avoid using software abstractions that would not be portable.

The PMC registers can only be written whereas the PMD registers can be read and written. it is not possible to read PMC registers, because their are only programmed by applications.

Registers can be read or written either individually or in batches. Both modes are important to minimize the cost of accessing the PMU. Sets of registers are typically access during the setup phase of a measurement. Individual registers are accessed during the measurement to adjust monitoring. For instance, a PMD may be modified to adjust a sampling period.

By operating at the register level, the interface avoids having to manipulate the entire PMU machine state during each call. This is clearly more scalable as the size of the data structures passed between the application and the kernel are smaller. This is also more portable because the structure do not hard-code the actual size of the machine state.

The interface does not interpret the values of the PMC and PMD registers unless necessary for security reason. This is important for portability but also to avoid kernel bloat which would occur if the kernel had to know about each possible PMU event.

The other key operations are: start and stop monitoring. The interface exposes simple start and stop operations. There may exist several methods for starting and stopping. Some PMU models may have this operation at the PMC level, other may be using a common control bit, some may use both methods. While the interface does not prevent using the PMC level mode, it also provides a *centralized* and portable method.

### 2.2.3 System call interface

The interface uses the system call model instead of the usual device driver model. This choice is mandated by some of the requirements set forth in the previous section. In particular because the interface must be built-in and must allow *per-thread* monitoring. System calls are rarely an optional feature of a kernel. At best, they can be turned off at compile time of the kernel. As such a system call reinforces the notion that performance monitoring is a built-in feature of the kernel. There is no need to compile and insert a separate module. Support for *per-thread* monitoring is a show-stopper for the device driver model. Drivers are typically not allowed access to the context switch code of the kernel. Having hooks in the context switch code is necessary to save and restore the PMU state.

The system call interface is also very flexible in that it is not constrained to having a fixed number of parameters like *ioctl()* does.

The interface can be implemented via a single system call with multiple commands or a set of system calls.

### 2.2.4 System-wide support

The support for *system-wide* monitoring must be designed such that it scales to large NUMA-style machine configurations while, at the same time, minimizing the kernel code complexity.

The interface uses a *CPU-wide* model where monitoring across multiple CPU cores requires independent monitoring of each CPU core. The monitoring tool must spawn a worker thread or process on each CPU core it wants to monitor. The tool can then aggregate the results, if needed.

The design is quite scalable because all operations are local to each CPU core, i.e., strong affinity improves code and data locality and hence minimizes the overhead. This is especially important when sampling and is in line with hardware sampling support in PMU models such as the Pentium 4 [10] with its Precise Event Based Sampling (PEBS) support.

### 2.2.5 Sampling support

The interface supports two types of sampling: time-based and event-based sampling. In the first mode, the sampling period is determined by a timeout. In the second mode, the period is defined as a number of occurrences of a PMU event.

Time-based sampling is supported at the user level. Event-based sampling is supported at both user and kernel levels.

The interface provides an efficient message-based overflow notification mechanism which is used as the basis for sampling at the user level.

For efficiency reasons, the interface also provides support for a kernel level sampling buffer where samples are automatically stored into a kernel buffer. When the buffer fills up, a notification is sent to the monitoring tool.

Support for a kernel level sampling buffer is achieved without loss of flexibility using a unique mechanism that allows for customized formats to be plugged into the kernel at runtime. Each format is responsible for the information that gets recorded and how it is recorded in the kernel buffer.

### 2.2.6 Event sets and multiplexing

To work around certain frequent PMU limitations, such as limited number of counters or incompatible events, the interface provides the notion of an *event set*. Each event set encapsulates the entire PMU machine state. It is possible to define multiple independent sets. The interface can then multiplex the sets on the actual PMU, i.e., only one set is ever active at the same time. Set switching can happen on a timeout or after a certain number of PMU events have been observed.

## 2.3 Perfmon Terminology

Throughout this document, we use a set of terms to describe certain abstractions and logical names used by the interface. In this section, we define the commonly used terms.

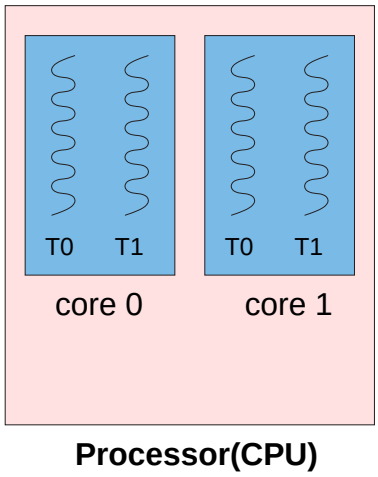


Figure 2.1: Processor anatomy.

### 2.3.1 Processor, CPU, and core

We refer to the *processor* or *CPU* as the physical package that is plugged into the processor socket on the computing board. A processor may implement multiple *processing cores* or *CPU cores*. Each core has its own set of physical resources such as registers, caches, execution pipeline, translation look-aside buffer.

In turn, each core can be multi-threaded. Current implementations, on different architectures, all expose each thread as a CPU core to the operating system. For instance, a dual-core multi-threaded processor appears as 4 single-threaded CPU cores. We show the example of a dual-core dual-threaded processor in figure 2.1.

### 2.3.2 The PMU

The term *PMU* refers to the Performance Monitoring Unit which is a piece of hardware present in most modern processors. The PMU typically exports a set of registers that can be programmed to collect certain hardware events.

There is one PMU per CPU core. For dual-core processors, there are two PMUs. Typically a multi-threaded core only implements one PMU but each thread has its own set of PMU registers that are multiplexed onto the PMU.

### 2.3.3 The PMU registers

The interface of a PMU consists, for the most part, of a set of control and data registers.

The control registers describe what to monitor. For that interface, they are called *Performance Monitoring Control* (PMC) registers. The data register is where the collected information is stored. For that interface, they are called *Performance Monitoring Data* (PMD) registers. A PMC register is typically associated with one PMD register or more. The pair forms what is called a *monitor*. A PMD register can be used as a counter. It is incremented at each occurrence of the event of interest programmed into the controlling PMC register. In that case the PMD register is called a *counting PMD* register and the

monitor as a whole is called a *counting monitor*. Depending on the PMU model, some PMD registers are may be used for other purposes such as buffers, for instance.

### 2.3.4 Threads, Processes, and tasks

Throughout the document we make reference to the terms *process*, *task*, and *threads*. The terminology is heavily leveraged from the Linux kernel.

A *thread* is referring to a *kernel thread*, i.e., a flow of control that is visible to the operating system. Depending on the operating system and environment, there may be *user threads* which are multiplexed by a library on top of a kernel thread. An application typically manipulates threads via the POSIX threads (pthreads) library. Depending on the environment, *pthreads* may be implemented using a one to one mapping with kernel threads or they may be multiplexed on top of a a kernel threads. This interface only deals the threads which are visible to the kernel.

A *process* is a software abstraction which encapsulates an address space, several other system resources, and at least one thread. Multiple threads can exist inside a single process.

A *task* is a term coming from the Linux world which represents a unique flow of control, in other words, a kernel thread. The Linux kernel sees tasks and not threads. The difference is that a *task* also has all the attributes of a process including an address space. In the case of a process with multiple flows of control, there are multiple tasks but they all share certain attributes such as the address space.

### 2.3.5 Perfmon context

The PMU state includes the values of the PMC and PMD registers and possibly other related registers. The PMU hardware state along with the associated software state is called a perfmon *context*. The user can create and manipulate a context through the interface. The internal structure of the context is dependent on the PMU and operating system and is never directly exposed to an application.

### 2.3.6 Perfmon session

A *perfmon session*, or *session* for short, describes the minimum sequence of actions necessary to collect a simple measurement. The set of actions is at least as follows:

1. create the perfmon context
2. program the PMU
3. start monitoring
4. run code to measure
5. stop monitoring
6. read results
7. destroy the perfmon context

Of course, there exists a multitude of variations of the sequence but the basic steps remain the same.

### 2.3.7 Reserved fields and bits

In this specification, some fields or bits inside fields are marked as *reserved*. This means that they are reserved for future use. Such fields or bits must be set to zero in order to guarantee that extensions will work correctly.

The interface does check that *reserved* bits inside bit-fields are indeed cleared, otherwise an error is returned.

# Chapter 3

# The interface

## 3.1 The perfmonctl() system call

The central piece of the interface is the perfmon system call which is defined as follows:

```
int perfmonctl(int fd, int cmd, void *arg, int narg);
```

The system call takes four arguments. The first argument is a file descriptor, *fd*, identifying the context on which to apply the command indicated by *cmd*. The file descriptor is obtained when a perfmon context is created. Some commands do not require a valid file descriptor, in which case, the first argument is ignored. A command may take some optional or mandatory parameters. In that case, the parameters are pointed to by *arg*. It is possible to apply the command to more than one parameter in one call. In that case, *arg* is actually a pointer to an array of parameters. The number of elements in the array is indicated by *narg*. This is an efficient way of minimizing the number of system calls which is especially useful when programming the PMU registers.

The kind of system call implemented by *perfmonctl()* is labeled a *multiplexing* call, where multiple commands are passed through a single entry point. Based on the command, the kernel then dispatches to the particular function implementing it, thereby *demultiplexing* the command. In the Linux world, the use of a system call per command is preferred. We certainly acknowledge this approach which has some advantages, such as a better type checking of the parameters for instance. We describe the alternative set of system calls in section 6.1. For historical reasons, the specification uses the single system call model but we do not envision any difficulty in switching to a multi system call model. This would not affect the semantics of the interface in any way.

The return value of the system call is `0` when successful. Otherwise the value is `-1` and the `errno` variable contains an error code describing the problem. Unlike the *open()* or *socket()* system calls, the file descriptor identifying the context is never passed as the return value of the *perfmonctl()* call, i.e., there is no overloading of the return value. The file descriptor is, instead, returned in the data structure passed with the command to create a new context.

Table 3.1 shows all the commands defined by the interface.

Each implementation must ensure that the *perfmonctl()* system call cannot be preempted, i.e., the call runs to completion or blocks in which case all perfmon-related locks must be released. This ensures that the calling thread cannot be rescheduled on a different CPU core than the one used to initiate the system call.

| Name | Description | Section |
|---|---|---|
| PFM_CREATE_CONTEXT | create a perfmon context | 3.1.1 |
| PFM_WRITE_PMCS | program PMC registers | 3.1.2 |
| PFM_WRITE_PMDS | program PMD registers | 3.1.3 |
| PFM_READ_PMDS | read PMD registers values | 3.1.4 |
| PFM_START | activate monitoring | 3.1.5 |
| PFM_STOP | stop monitoring | 3.1.6 |
| PFM_LOAD_CONTEXT | attach perfmon context | 3.1.7 |
| PFM_UNLOAD_CONTEXT | attach perfmon context | 3.1.8 |
| PFM_RESTART | resume monitoring after notification | 3.1.9 |
| PFM_CREATE_EVTSETS | create or modify event sets | 3.1.10 |
| PFM_DELETE_EVTSETS | delete event sets | 3.1.11 |
| PFM_GETINFO_EVTSETS | get information about event sets | 3.1.12 |
| PFM_GETINFO_PMCS | get information about PMC registers | 3.1.13 |
| PFM_GETINFO_PMDS | get information about PMD registers | 3.1.14 |
| PFM_SET_CONFIG | set global perfmon properties | 3.1.16 |
| PFM_GET_CONFIG | get global perfmon properties | 3.1.17 |

Table 3.1: List of commands.

### 3.1.1 The PFM_CREATE_CONTEXT command

#### Description

The PFM_CREATE_CONTEXT command creates a new perfmon context. This is the first command that must be invoked before any useful monitoring work can be accomplished. Without a context, it is not possible to program the PMU. With this command, certain characteristics of the context are defined such as the type of the context: per-thread or system-wide. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(0, PFM_CREATE_CONTEXT, ctx, 1);
```

The `cmd` argument is set to PFM_CREATE_CONTEXT. The file descriptor argument is ignored for this call, we show it as 0. The third argument must be a pointer to a structure of type `pfarg_ctx_t`. The command only supports the creation of one context at a time, hence the last argument must have the value `1`.

The `pfarg_ctx_t` structure is defined as follows:

```
typedef unsigned char pfm_uuid_t[16];
typedef struct {
        uint32_t        ctx_flags;
        int             ctx_fd;
        pfm_uuid_t      ctx_smpl_buf_id;
        size_t          ctx_smpl_buf_size;
} pfarg_ctx_t;
```

The fields of the pfarg_ctx_t structure are as follows:

- ctx_flags : describes the properties of the context. The flags are divided in two categories: generic and platform specific. The latter category describes certain features which are specific either to the host PMU or the host operating system. The bits which are not defined are *reserved* and must be cleared. It is only possible to set bits that are defined. You need to refer to the PMU model and/or operating system specific sections for more details. The following set of generic flags are defined:
  - PFM_FL_SYSTEM_WIDE: indicates that the context is for a system-wide session. By default, a context is created for a per-thread session.
  - PFM_FL_NOTIFY_BLOCK: indicates that the thread being monitored should be blocked during an overflow notification. This flag is only valid for a non self-monitoring per-thread session. The default behavior is to let the monitored thread run while the overflow notification is processed.
  - PFM_FL_OVFL_NO_MSG: indicates that the application is not interested in receiving overflow notification messages. By default, one message is generated for every notification. See section 3.3 for more details on this flag.
- ctx_fd: this field is ignored on input. Upon successful return, this field contains the file descriptor which uniquely identifies the context within the calling process. This descriptor is needed by most commands. The file descriptor remains valid until the context is destroyed via *close()*.
- ctx_smpl_buf_id : contains the unique identifier (UUID) of the sampling format to use for the context. When a sampling buffer format is not needed, this field must be initialized to all zeroes. See section 3.4 for more details.
- ctx_smpl_buf_size: this field is ignored on input. Upon successful return and when the selected sampling buffer format exists and uses the buffer re-mapping service, this field contains the actual size in bytes of the buffer. Otherwise this field contains 0. See section 3.4 for more details on the support for sampling.

When the command is successful, the pfarg_ctx_t structure is modified therefore if the same configuration is to be shared between multiple contexts, separate copies must be passed for the creation of each context to avoid conflicts should the calls occurs simultaneously.

The file descriptor returned in ctx_fd is a regular file descriptor. It can be used with several system calls such as, *read()*, *select()*, *poll()*, *fcntl()*, *close()*. We describe for what purpose an application might want to use those system calls in section 3.3.

Theoretically, the number of contexts that can be created within a process is limited by the amount of memory available in the system. Each implementation may impose other resources limitations such as the maximum number of open files.

Upon return from the call, the context is ready to accept further commands. It is not attached to any particular thread or CPU core. The context is said to be *unloaded*. It is necessary to invoke the PFM_LOAD_CONTEXT to bind the context to a thread or CPU core. Until that happens, the actual PMU is never accessed, only the software maintained PMU state can be modified.

Not all combinations of the ctx_flags are valid for any context. This is especially true with regards to the PFM_FL_NOTIFY_BLOCK flag. Table 3.2 shows the valid combinations. It is important to note that for some combinations the validity of the flags can only be assessed when the context is actually attached to a thread of CPU core using the PFM_LOAD_CONTEXT command. Platform specific flags may have additional restrictions.

| | system-wide | per-thread, not self | per-thread, self |
|---|---|---|---|
| PFM_FL_NOTIFY_BLOCK | no | yes | no |
| w/o PFM_FL_NOTIFY_BLOCK | yes | yes | yes |
| PFM_FL_OVFL_NO_MSG | yes | yes | yes |
| w/o PFM_FL_OVFL_NO_MSG | yes | yes | yes |

Table 3.2: possible context flags combinations.

### System-wide context

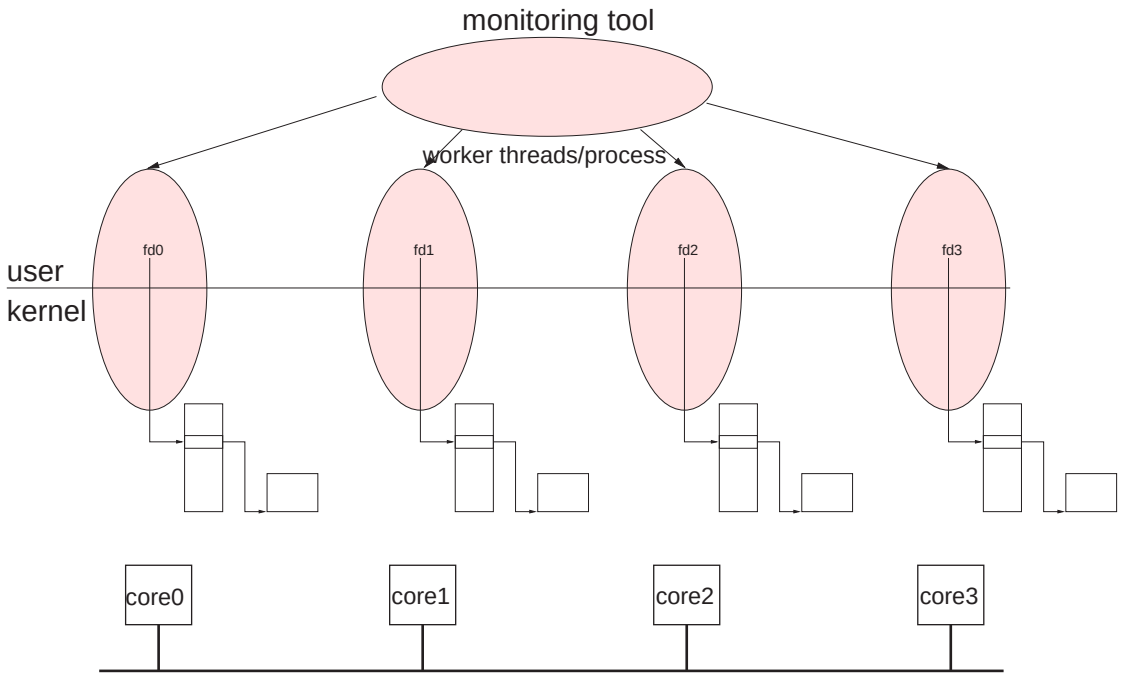


Figure 3.1: how to measure across multiple CPU cores.

A system-wide context can only be used to monitor one CPU core at a time. To construct monitoring tools that measure across all CPU cores of a multi-processor (or multi-threaded processor) machine, it is necessary to create multiple contexts and use multiple threads or processes, each pinned to run on a distinct CPU core, to achieve the desired coverage. This is illustrated in figure 3.1 for a 4-way (4 cores) machine. The monitoring tool must spawn additional threads or processes. Each one must be pinned to a specific CPU core. See the description of PFM_LOAD_CONTEXT in section 3.1.7 for more details.

### Controlling threads and processes

Any thread with access to the file descriptor identifying a context is called a *controlling thread*, meaning that it can access the context with any of the defined commands. The interface relies, for the most part, upon the file descriptor security semantics for access control. It should be noted that the threads do not all necessarily have to belong to the same process, in case of a *fork()*.

Given that a file descriptor is accessible to all threads inside a process, all threads inside a multi-threaded process are *controlling threads*. Similarly the process is called the *controlling process*.

For a given context, a thread ceases to be a *controlling thread* once the file descriptor is closed. The same behavior applies at the process level.

### Behavior on *fork()*

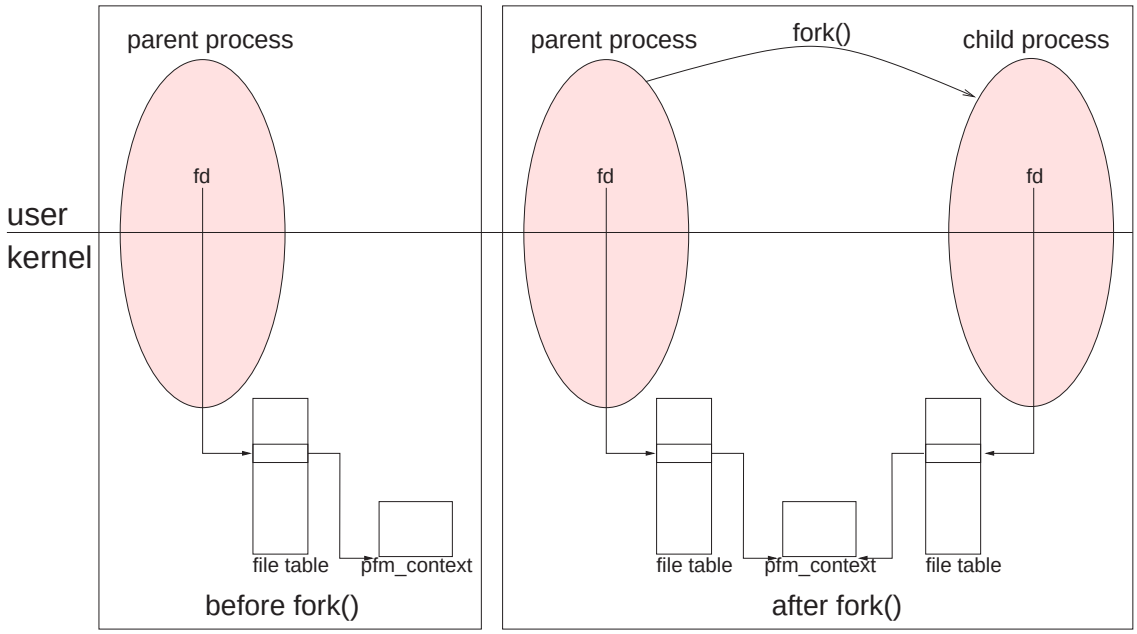


Figure 3.2: controlling threads and *fork()*

When a controlling thread invokes the *fork()* system call, the regular POSIX semantics for file descriptor sharing apply. This is shown in figure 3.2. The file descriptor is cloned in the child process. Both descriptors lead to the same context. In other words, all the threads of the parent and children processes have access to the context. This behavior can be altered by explicitly closing the file descriptor in each newly created child process.

### Behavior on *pthread_create()*

By definition, a file descriptor is accessible to all threads of process. This is independent of the threads being implemented purely at the user level or at the kernel level. Therefore the context is automatically accessible to all threads.

### Behavior on *exec()*

When a controlling thread invokes the *exec()* system call, the regular file descriptor semantics apply. Unless the descriptor was explicitly setup to be closed on exec via *fcntl()* and the F_SETFD command, it is valid in the newly loaded code and therefore the context remains accessible.

### Behavior when sampling

When a valid custom sampling buffer format (see section 3.5.1), is passed and that format is using the perfmon buffer allocated and re-mapping services, the actual size of the buffer is returned in `ctx_smpl_buf_size`. In all other case, the value of this field is 0.

Based on the format, an application may be able to specify a sampling buffer size via a format specific structure passed during this command. The requested and actual size may differ due to some platform constraints, such as alignment. The buffer is not automatically accessible from the user level address as a result of the command. An explicit call to *mmap()* is necessary. See section 3.4 for more details.

### Interactions with event sets

When a context is created the initial event set, *set0*, is systematically created and ready for use. Event sets are described in section 3.7.

### Security considerations

Access to the context is granted to all threads with access to the file descriptor. The operating system semantics for sharing a file descriptor govern the access to the context.

The creation of a context may fail because the caller may not have the right credentials. In particular, the system administrator may have setup some restrictions on the creation of a context based on a group of users. See chapter 4 for more details.

Similarly, the command may fail if there is a problem allocating the memory resources associated with a particular sampling format. For instance, the system administrator may have setup some memory size restrictions. See section 4.4.3 for more details.

### Return values

- 0: the command was successful. The `ctx_fd` and potentially `ctx_smpl_buf_size` fields are updated.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel.
  - EINVAL : invalid arguments.
  - ENOMEM : no enough memory to allocate the context or the sampling buffer.
  - E2BIG : size of sampling format specific argument is too big.
  - EFAULT : an invalid address is passed, most likely *ctx* is invalid.
  - EPERM : the application is not allowed to create a perfmon context.

### 3.1.2 The PFM_WRITE_PMCS command

#### Description

The PFM_WRITE_PMCS command is used to program the PMC registers for a particular context. More than one register can be programmed in one call. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(fd, PFM_WRITE_PMCS, pmcs, n);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. The third argument, *pmcs*, is a pointer to an array of structures of type `pfarg_pmc_t`. The fourth argument, *n*, indicates the number of elements in the array. It must be greater than 0.

The `pfarg_pmc_t` structure is defined as follows:

```
typedef struct {
        uint16_t    reg_num;
        uint16_t    reg_set;
        uint32_t    reg_flags;
        uint64_t    reg_value;
        uint64_t    reg_reset_pmds[PFM_MAX_PMD_BITVECTOR];
        uint64_t    reg_smpl_pmds[PFM_MAX_PMD_BITVECTOR];
        uint64_t    reg_smpl_eventid;
} pfarg_pmc_t;
```

The fields are used as follows for this command:

- `reg_num` : the PMC register index.
- `reg_set` : the event set for the PMC register.
- `reg_value` : the value to load into the PMC register. The value is highly specific to the host PMU. The interface normally treats the value as opaque however implementations may check the validity of the value, see section 3.1.13 for more details.
- `reg_flags` : a set of flags to enable certain features. The bits which are not defined are *reserved* and must be cleared. It is only possible to set bits that are defined. Upon return, the field may be updated to reflect possible error conditions. The defined flags are as follows:
  - PFM_REGFL_OVFL_NOTIFY : when the counting PMD associated with the PMC register overflows, a overflow notification message is appended to the message queue of the context. Depending on the PMU, not all PMC registers are necessarily associated with a counting PMD. For those, this flag is ignored. The default behavior is that no notification is sent on counter overflow, i.e., the counter simply wraps around.
  - PFM_REGFL_RANDOM : when the associated counting PMD register overflows, the reset value must be chosen randomly. By default, this flag is off. Depending on the PMU, not all PMC registers are necessarily associated with counting PMD registers. For those, this flag is ignored.
- `reg_smpl_pmds` : this bitvector contains the list of PMD registers to be recorded in a sample when the associated counting PMD overflows. Each bit represents a PMD register. The PMD registers referenced in the bitvector are interpreted as being from the same set as the controlling

PMC. If no PMD is of interest, the bitvector must be all zeroes. It is possible to specify PMD registers that are not used as counters. This field is ignored for all PMC registers not controlling a counting PMD register. Specifying PMD registers that are unimplemented generates an error. It is up to the buffer sampling format selected by the application to use this field to record the requested information. The default sampling format uses this field.

- `reg_reset_pmds` : this bitvector contains the list of PMD registers to reset when the associated counting PMD register overflows. Each bit represents a PMD register. The PMD registers referenced in the bitvector are interpreted as being from the same set as the PMC register. If no PMD is to be reset, the bitvector must be all zeroes. It is possible to specify PMD registers that are not used as counters. Specifying PMD registers that are unimplemented generates an error.
- `reg_smpl_eventid` : this value is passed to the sampling buffer format when the associated counting PMD register overflows. The value is not interpreted by the interface and is passed directly to the sampling format. This mechanism can be used to pass information to the sampling format. This field is ignored for all PMC registers not controlling a counting PMD register. It is up to the format to use this information. The default sampling format does not use of this field.

At any time, the interface maintains a software state for the PMU that includes the current value for both the PMC and PMD registers. When a context is created, each PMC register is initialized with a default value. This value is specific to each PMU but must be chosen such that nothing is measured. Hence, applications only need to setup the PMC registers they need. Perfmon guarantees that all non explicitly used PMC registers do not measure anything. It is possible to retrieve the default value for a PMC register using the PFM_GETINFO_PMCS command.

The interface does not impose any ordering between modifying a PMC register and its associated PMD register(s). It is up to the application to decide the best order. It is not necessary to program all PMC registers, just to get a know stable base. The interface guarantees that the unused PMC registers will be programmed with their respective default values. Those values must be chosen by each implementation such that nothing is captured by the PMC and its associated PMD registers.

It is always possible to modify a PMC register when the context is *unloaded*. However, when the context is *loaded* the following must be possible:

- for a system-wide session, any thread with access to the file descriptor and running on the monitored CPU must be able to execute this command
- for a per-thread session which is self-monitoring, i.e., the context is attached to the controlling thread, this command must always be possible

For all other cases, each implementation may decide to restrict access depending on the state of the thread. In particular, there may be some restrictions on multi-processor machines to ensure that the thread is actually stopped.

**Bitvector sizes**

The bitvectors represent all PMD registers accessible to an application. For this command the `pfarg_pmc_t` structure includes two bitvectors: `reg_smpl_pmds` and `reg_reset_pmds`.

Bitvectors are always represented as an array of 64-bit elements. The size of the array depends on the host PMU and implementation and is determined by the following formula:

```
#define PFM_MAX_PMD_BITVECTOR (((PFM_MAX_PMDS+64-1)/64)
```

The constant PFM_MAX_PMDS represents the maximum number of PMD registers that are accessible to an application. Holes in the PMD name space are supported and the value of this constant takes it into account, i.e., the actual number of PMD registers may be less than PFM_MAX_PMDS.

### How to use the reg_smpl_pmds bitvector

The reg_smpl_pmds bitvector is used when sampling. It allows an application to indicate which PMD registers must be recorded in each sample. It is always up to the sampling format to honor this information. As many bits as there are implemented PMD registers can be set in the bitvector.

The use of the bitvector is best explained using a simple example. Let us suppose that PMC4/PMD4, a counting monitor, is used as the sampling period and that PMC8/PMD8 and PMC9/PMC9 are programmed to count two other events of interest. When PMD4 overflows, the application would like to record in the sample, the values for PMD8, PMD9. In that case, it would program PMC4 such that reg_smpl_pmds would be equal to 0x300, i.e., bit 8 and bit 9 are set.

### How to use the reg_reset_pmds bitvector

The reg_reset_pmds bitvector is used when sampling. It allows an application to indicate which PMD registers must be reset at the end of a sampling period. The reset operation is performed after the sample is recorded or upon restart, i.e., PFM_RESTART, when monitoring is masked. This feature is independent of the sampling format used.

This field is interesting to compute differences in the value of some sampled counters between two samples. It avoids having to post-process the sampling buffer to compute the differences.

Let us take a simple example to illustrate how to use this feature. Let us suppose that an application would like to take a sample every one million cycles. In each sample, it would also like to record the number of instructions that have retired since the last sample was taken. For this measurement, the application would program, PMC4/PMD4 to be the sampling period counting cycles and it would program PMC5/PMD5 to count the number of instructions retired, for instance. When programming PMC4, the application would indicate that PMD5 must be included in each sample by setting reg_smpl_pmds to 0x20. It would also set reg_reset_pmds to 0x20. With this setup each sample will contain the delta of instructions retired between each sample.

### Interactions with event sets

It is not possible to specify an event set that does not exist. An event set needs to be explicitly created, except for *set0* which is created by default.

### Security considerations

For security reasons or because of resource limitations, each implementation may limit the number of elements that can be passed in the pfarg_pmc_t array.

**Return values**

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EINVAL : invalid arguments (see below for discussion)
  - EBADF : invalid file descriptor
  - EFAULT : an invalid address is passed. Most likely the `pfarg_pmc_t` array is invalid or the number of elements in the array is too big.

There are several reasons why the call may return EINVAL. It may be due to an invalid register index or because the PMC value is invalid or the event set does not exist. As alluded to earlier, some implementations may check the validity of the value. In the case where multiple PMC registers are programmed in one call, it may be difficult to figure out which element in the array caused the problem. However the interface includes a simple mechanism to help identify the invalid element by using the `reg_flags` field. When EINVAL is returned, the caller may want to scan the vector of `pfarg_pmc_t` elements using the following macro and definitions:

- `PFM_REG_HAS_ERROR(flags)` : the *flags* parameter must be the value of the `reg_flags` field. The macro returns non-zero if *flags* contains an error code. Otherwise, the value 0 is returned.
- PFM_REG_RETFL_EINVAL: if this flag is set in the `reg_flags` then the value for the corresponding PMC is invalid
- PFM_REG_RETFL_NOTAVAIL: if this flag is set in the `reg_flags` then the index of the PMC register is invalid
- PFM_REG_RETFL_NOSET: if this flag is set in the `reg_flags` field then the requested set does not exist

The command aborts at the first error therefore no further elements are processed beyond the invalid element. Elements placed before the invalid element are guaranteed processed by the interface, i.e., there is no need to resubmit them. If no error is reported by the macro, then the reason for EINVAL is different.

### 3.1.3 The PFM_WRITE_PMDS command

**Description**

The PFM_WRITE_PMDS command is used to program the PMD registers for a particular context. More than one register can be programmed in one call. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(fd, PFM_WRITE_PMDS, pmds, n);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. The third argument, *pmds*, is a pointer to an array of structures of type `pfarg_pmd_t`. The fourth argument, *n*, indicates the number of elements in the array. It must be greater than 0.

The pfarg_pmd_t structure is defined as follows:

```
typedef struct {
        uint16_t    reg_num;
        uint16_t    reg_set;
        uint32_t    reg_flags;
        uint64_t    reg_value;
        uint64_t    reg_long_reset;
        uint64_t    reg_short_reset;
        uint64_t    reg_last_reset_val;
        uint64_t    reg_ovflsw_thres;
        uint64_t    reg_random_mask;
        uint32_t    reg_random_seed;
} pfarg_pmd_t;
```

The fields are used as follows for this command:

- reg_num : the PMD register index.
- reg_set : the event set for the PMD register.
- reg_value : the value to load into the PMD register. For all counting PMD registers, the value is always 64-bit no matter what the underlying PMU supports. For other types of PMD registers, it depends on the underlying PMU and some of the bits in the value may be ignored. Certain implementations may check the validity of the value, see section 3.1.14 for more details. If the context is detached, then the value represents the initial value to be loaded into the actual PMD register when the context is attached. If the context is attached, the value is used to update the actual PMD register.
- reg_flags : there are no input flags defined for this command. All but the return flag bits are *reserved*.
- reg_long_reset: this field contains the 64-bit value to reload into the PMD register during the PFM_RESTART command following an overflow notification. Depending on the sampling format, the register may have been reset prior to restart, in which case, this value is not used. When the designated PMD register is not a counter, it does not overflow, therefore the value of this field is normally ignored. However if the PMD register is specified as part of the reg_reset_pmds bitvector of a PMC register, it will be reset.
- reg_short_reset: this field contains the 64-bit value to reload into the PMD register when it overflows and there is no user level notification requested or necessary. The actual behavior may be different when the sampling format is not the default format. When the designated PMD register is not a counter, it does not overflow, therefore the value of this field is normally ignored. However if the PMD register is specified as part of the reg_reset_pmds bitvector of a PMC register, it will be reset.
- reg_last_reset_val : this field is ignored for this command.
- reg_ovflsw_thres: the number of overflows before switching to the next set. Switching only effectively occurs when the PFM_SETFL_OVFL_SWITCH flag is selected for the event set, otherwise this field is ignored. This field is also ignored for all non counting PMD registers.
- reg_random_mask: this field contains the 64-bit mask used to compute the pseudo-random value generated on reset. This field is used only with a counting PMD register for which the controlling

PMC register enabled randomization by setting the PFM_REGFL_RANDOM flag. Otherwise, this field is ignored. The mask determines the range of allowed variation for the new pseudo-random value. See section 3.4.2 for more details.

- `reg_random_seed`: this field contains the seed value to initialize the built-in pseudo-random number generator. This field is used only with a counting PMD register for which the controlling PMC register enabled randomization by setting the PFM_REGFL_RANDOM flag. Otherwise, this field is ignored. There is no restriction on the 32-bit value of this field. See section 3.4.2 for more details on randomization.

At any time, perfmon maintains a software state for the PMU that includes the current value for both the PMC and PMD registers. When a context is created, each PMD register is initialized with a default value. This value is specific to each PMU however for all counting PMD registers, it is guaranteed to be 0. Hence, applications do not need to initialize all the PMD registers.

The interface does not impose any ordering between modifying a PMC register and its associated PMD register. It is up to the application to decide the best order.

It is always possible to modify a PMD register when the context is detached. However, when the context is attached the following must be possible:

- for a system-wide session, any thread with access to the file descriptor and running on the monitored CPU must be able to execute this command
- for a per-thread session which is self-monitoring, i.e., the context is attached to the controlling thread, this command must always be possible

For all other cases, each implementation may decide to restrict access depending on the state of the thread. In particular, there may be some restrictions on multi-processor machines to ensure that the thread is actually stopped.

#### 64-bit emulation

All counting PMD register values are exposed as being 64-bit wide. Each PMU does not necessarily implement full 64 bit counters. In that case, an implementation must emulate in software. The emulation is based on the counter overflow interrupt mechanism which all modern PMU provide. Always exposing a 64-bit value for a counter provides stability for applications because there is no need to know about each PMU specific width. In figure 3.3 we show an hypothetical PMU which implements actual PMD registers, shown as `hwPMD`, which are only 32-bit counters. At the top, an application provides the value 0 for the register, the upper 32 bits are stored in the *software* PMD register which is 64 bits wide. The lower 32 bits are stored in the actual hardware register, `hwPMD`, and the bottom 32 bits of the *software* PMD are set to 0. The bottom of the figure shows what happens when the hardware PMD overflows, i.e., when value goes from $2^{32} - 1$ to $2^{32}$ in which case it wraps around to $0$. The PMU generates an interrupt, the interrupt handler emulates the 64-bit counter by adding to the software counter the value $2^{32}$ changing the value to 0x100000000. Of course, this mechanism requires that the PMU be configured such that a counter overflow generates an interrupt.

For non counting PMD registers, up to 64-bit values are supported but no emulation is done. Those registers are typically used as buffers, in which case the hardware PMD probably implements as many bits as is necessary for the value to be meaningful. The bits in `reg_value`, `reg_long_reset`, `reg_short_reset` which are not relevant to the hardware PMD register are *ignored*.

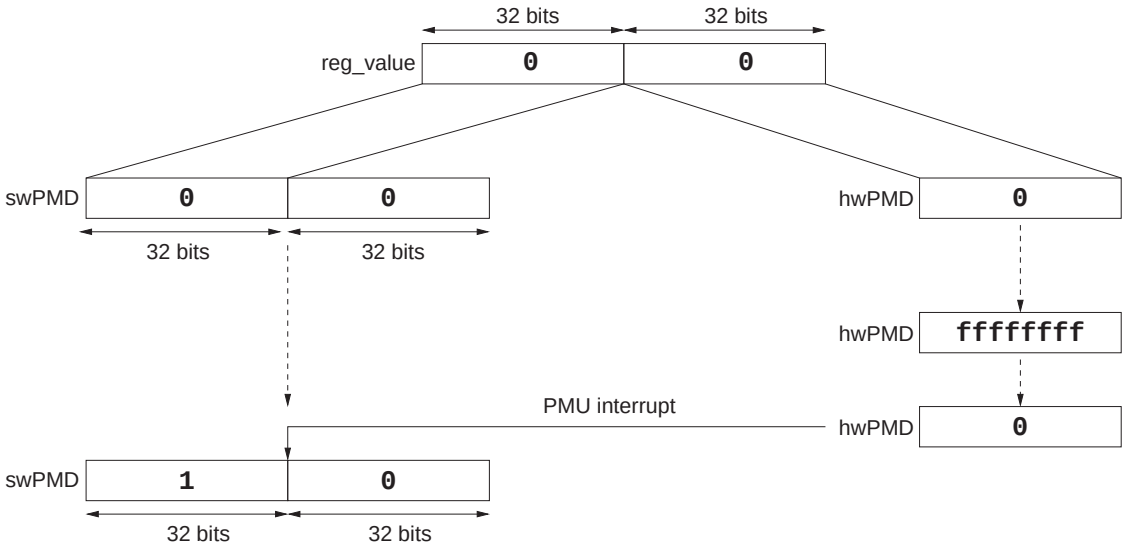


Figure 3.3: 64-bit emulation for counting PMD registers

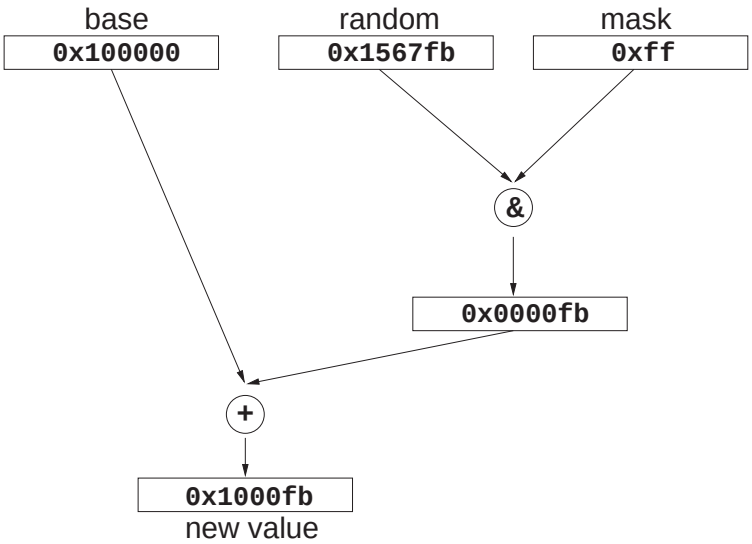


Figure 3.4: randomization of the value of a PMD register

### Randomization

When randomization is selected, the new value for a counting PMD is built by combining the new random value, the mask and the base value, which is either the value in the `reg_short_reset` or `reg_long_reset` field depending when the reset happens. The combination is as follows:

$$\text{new\_pmd\_value} = \text{base} + (\text{new\_random\_value} \ \& \ \text{mask})$$

In figure 3.4, we illustrate the formula with a simple example. The base value `0x100000` is the value provided in the `reg_value` field. The mask is set such that it selects only the bottom 8 bits, allowing values from `0` to `0xff`. The random number generator produces `0x1567fb` which is then masked and added to the base value, yielding `0x1000fb` as the new random value. As can be seen in this example, if the mask is set up such that it masks all the bits from $0$ to the $n^{th}$ bit, then a range of $2^n$ values is possible.

### Interactions with event sets

It is not possible to specify an event set that does not exist. An event set needs to be explicitly created, except for *set0* which is created by default.

### Security considerations

For security reasons or because of resource limitations, each implementation may limit the number of elements that can be passed in the `pfarg_pmd_t` array.

### Return values

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EINVAL : invalid arguments (see below for discussion)
  - EBADF : invalid file descriptor
  - EFAULT : an invalid address is passed. Most likely the `pfarg_pmd_t` array is invalid or the number of elements in the array is too big.

There are several reasons why the call may return EINVAL. It may be due an invalid index or value or set. As alluded to earlier, some implementations may check the validity of the value. In the case where multiple PMD registers are programmed in one call, it may be difficult to figure out which element in the array caused the problem. However the interface includes a simple mechanism to help identify the invalid element by using the `reg_flags` field. When EINVAL is returned, the caller may want to scan the vector of `pfarg_pmd_t` elements using the following macro and definitions:

- `PFM_REG_HAS_ERROR(flags)` : the *flags* parameter must be the value of the `reg_flags` field. The macro returns non-zero if *flags* contains an error code. Otherwise, the value 0 is returned.

- PFM_REG_RETFL_EINVAL: if this flag is set in the `reg_flags` then the value for the corresponding PMD is invalid
- PFM_REG_RETFL_NOTAVAIL: if this flag is set in the `reg_flags` then the index of the PMD register is invalid
- PFM_REG_RETFL_NOSET: if this flag is set in the `reg_flags` field then the requested set does not exist

The command aborts at the first error therefore no further elements are processed beyond the invalid element. Elements placed before that the invalid element are guaranteed processed by the interface, i.e., there is no need to resubmit them. If no error is reported by the macro, then the reason for EINVAL is different.

### 3.1.4 The PFM_READ_PMDS command

**Description**

The PFM_READ_PMDS command is used to read the value of PMD registers for a particular context. More than one register can be read in one call. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(fd, PFM_READ_PMDS, pmds, n);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. The third argument, *pmds*, is a pointer to an array of structures of type `pfarg_pmd_t`. The fourth argument, *n*, indicates the number of elements in the array. It must be greater than 0.

The `pfarg_pmd_t` structure is the same as the one used by `PFM_WRITE_PMDS`. It is defined as follows:

```
typedef struct {
        uint16_t    reg_num;
        uint16_t    reg_set;
        uint32_t    reg_flags;
        uint64_t    reg_value;
        uint64_t    reg_long_reset;
        uint64_t    reg_short_reset;
        uint64_t    reg_last_reset_val;
        uint64_t    reg_ovflsw_thres;
        uint64_t    reg_random_mask;
        uint32_t    reg_random_seed;
} pfarg_pmd_t;
```

The fields are used as follows for this command:

- `reg_num` : the PMD register index.
- `reg_set` : the event set for the PMD register.
- `reg_value` : this field is ignored on input. Upon successful return, the field contains the current value of the PMD register.

- `reg_flags` : there are no input flags defined for this command. All but the return flag bits are *reserved*.
- `reg_long_reset`: this field is not used by this command.
- `reg_short_reset`: this field is not used by this command.
- `reg_last_reset_val`: this field is not used on input. Upon successful return, it contains the last value that was loaded in the PMD register. This is useful for a counting PMD register when randomization is selected on the controlling PMC register. In this case, the field contains the last pseudo random value that was loaded into the register otherwise it contains either the value of `reg_short_reset` or `reg_long_reset`.
- `reg_ovflsw_thres`: this field is not used on input. Upon successful return and when the event set for the PMD register has the PFM_SETFL_OVFL_SWITCH flag enabled, the field contains the number of overflows left before switching to the next set. When the PFM_SETFL_OVFL_SWITCH flag is not enabled or if the PMD is not a counting PMD, then this field contains 0.
- `reg_random_mask` : this field is not used by this command.
- `reg_random_seed` : this field is not used by this command.

At any time, perfmon maintains a software state for the PMU that includes the current value for both the PMC and PMD registers. When a context is created, each PMD register is initialized with a default value. This value is specific to each PMU however for all counting PMD registers, it is guaranteed to be 0. Hence, reading a PMD register just after the context is created returns its default value.

For all counting PMD registers, the 64-bit value of the counter is returned. When the host PMU does not implement 64-bit counters, the implementation must emulate in software. We have described the mechanism in figure 3.3. For reading, the 64-bit value is constructed by combining the hardware counter and its software emulation when the context is attached. When the context is detached the 64-bit value is contained in the software counter.

For all non counting PMD registers, no emulation is done. When the actual PMD register is less than 64-bit wide, the remaining upper bits are zero-extended. The extension applies to `reg_value` and `reg_last_reset_val`.

For a detached context, the command returns the last value of the PMD, i.e., the value the register had when the context was last detached.

It is always possible to read a PMD register when the context is detached. However, when the context is attached the following must be possible:

- for a system-wide session, any thread with access to the file descriptor and running on the monitored CPU must be able to execute this command
- for a per-thread session which is self-monitoring, i.e., the context is attached to the controlling thread, this command must always be possible
- for any type of session and when monitoring is masked following an overflow notification, this command must be possible. Note that this behavior may be affected by the use of a custom sampling format. However, it is guaranteed to work with the default format.

For all other cases, each implementation may decide to restrict access depending on the state of the thread.

### Interactions with event sets

It is not possible to specify an event set that does not exist. An event set needs to be explicitly created, except for *set0* which is created by default.

### Security considerations

For security reasons or because of resource limitations, each implementation may limit the number of elements that can be passed in the `pfarg_pmd_t` array.

### Return values

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EBADF : invalid file descriptor
  - EINVAL : invalid arguments (see below for discussion)
  - EFAULT : an invalid address is passed. Most likely the `pfarg_pmd_t` array is invalid or the number of elements in the array is too big.
  - EBUSY : the command cannot be executed at this time.

There are several reasons why the call may return EINVAL. It may be due to an invalid index or invalid set. In the case where multiple PMD registers are read, it may be difficult to figure out which element in the array caused the problem. However the interface includes a simple mechanism to help identify the invalid element by using the `reg_flags` field. When EINVAL is returned, the caller may want to scan the vector of `pfarg_pmd_t` elements using the following macro and definitions:

- `PFM_REG_HAS_ERROR(flags)` : the *flags* parameter must be the value of the `reg_flags` field. The macro returns non-zero if *flags* contains an error code. Otherwise, the value 0 is returned.
- PFM_REG_RETFL_NOTAVAIL: if this flag is set in the `reg_flags` field then the index of the PMD register is invalid
- PFM_REG_RETFL_NOSET: if this flag is set in the `reg_flags` field then the requested set does not exist

The command aborts at the first error therefore no further elements are processed beyond the invalid entry. Elements placed before the invalid entry are guaranteed processed by the interface, i.e., there is no need to resubmit them. If no error is reported by the macro, then the reason for EINVAL is different.

### 3.1.5 The PFM_START command

#### Description

The PFM_START command is used to activate monitoring for a context. Activation is the operation which puts the PMU into a state where the PMD registers are actually collecting qualified events. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(fd, PFM_START, arg, 1);
```

or

```
perfmonctl(fd, PFM_START, NULL, 0);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. The command takes an optional argument `arg` of type `pfarg_start_t`. When the argument is not used the value must be NULL. When the third argument is used, the fourth argument must be set to 1, i.e., this command does not take vectors. Otherwise the fourth argument must be set to 0.

The `pfarg_start_t` data structure is defined as follows:

```
typedef struct {
        uint16_t        start_set;
} pfarg_start_t;
```

The fields are defined as follows for this command:

- `start_set` : the set to activate first

This command is only valid for a context that is attached to a thread or a CPU core. Upon return, the interface guarantees that monitoring is active. The interface does not guarantee that the activation is necessarily atomic. Some PMU may not have a mechanism to activate monitoring with a single atomic instruction. For a non self-monitoring per-thread session, monitoring is guaranteed activated but the effect are not visible until that thread runs again and this may occur after returning from the command.

Depending on the implementation, there may be ways to activate monitoring without invoking this command. In that case, this command is still usable and must not conflict.

For all non self-monitoring sessions, some implementations may impose restrictions as to when this command can be issued. In particular, there may be some restrictions on multi-processor machines to ensure that the thread is actually stopped.

When sampling, it should be noted that this command does not actually trigger a call in the sampling format managing the sampling buffer. There is no format call-back for this command. In the case of the default sampling format, the buffer is not reset as it would with a PFM_RESTART command.

**Interactions with event sets**

The activation applies to all defined event sets. There can only be one set on the PMU at any time. That set is called the *active set*. When multiple sets are defined, an application can indicate which set must be activated first by specifying its number with PFM_LOAD_CONTEXT or by passing a `pfarg_start_t` argument to this command. Specifying a set with this command overrides the first set designated with the PFM_LOAD_CONTEXT.

When an application does not specify the first active set, then it will be determined as follows:

- the set specified during the last PFM_LOAD_CONTEXT
- whatever was the last active set from the previous activation

On implementations where it is possible to activate monitoring without calling PFM_START, the first set follows the possibilities listed above.

Until monitoring is activated no set switching occurs. The PFM_LOAD_CONTEXT does not, by itself, enable time-based set switching. That restriction also applies to implementations which can effectively activate monitoring without calling PFM_START.

As a consequence of this command, both time-based and overflow-based set switching are activated.

**Return values**

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EBADF : invalid file descriptor
  - EBUSY : the command cannot be executed at this time.
  - EFAULT : an invalid address is passed. Most likely the `pfarg_start_t` pointer is invalid.

### 3.1.6 The PFM_STOP command

**Description**

The PFM_STOP command is used to stop monitoring for a context. Stopping is the operation which puts the PMU into a state where the PMD registers are not collecting events. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(fd, PFM_STOP, NULL, 0);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. The command takes no argument therefore the third and fourth arguments are shown as NULL and 0 respectively.

This command is only valid for a context that is attached to a thread or a CPU core. Upon return, the interface guarantees that monitoring is inactive. The interface does not guarantee that the deactivation is necessarily atomic. Some PMU may not have a mechanism to deactivate monitoring with a

single atomic instruction. For a non self-monitoring per-thread session, monitoring is guaranteed deactivated but the effect are not visible until that thread runs again and this may occur after returning from the command.

The effects of monitoring, especially when sampling, may extend beyond the point in time where the PFM_STOP command returns. This behavior depends upon the implementation and the PMU model. In other words, it may be possible to collect one extra sample beyond the point in time when the PFM_STOP command returns.

Depending on the implementation, there may be ways to stop monitoring without calling into the kernel via the `perfmonctl` system call. In that case, this command is still usable and must not conflict.

For all non self-monitoring threads, some implementations may impose restrictions as to when this command can be issued. In particular, there may be some restrictions on multi-processor machines to ensure that the thread is actually stopped.

#### Event set interactions

The deactivation applies to all defined event sets. Stopping monitoring does stop time-based and overflow-based set switching, i.e., no more set switching occurs. This also applies on implementations where stopping monitoring can be done without calling PFM_STOP.

#### Return values

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EBADF : invalid file descriptor
  - EBUSY : the command cannot be executed at this time.

### 3.1.7 The PFM_LOAD_CONTEXT command

#### Description

The PFM_LOAD_CONTEXT command is used to attach a context to a thread. This command effectively loads the PMU software state onto the actual PMU. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(fd, PFM_LOAD_CONTEXT, load_args, 1);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. The command takes one argument of type `pfarg_load_t` pointed to by *load_args*. The fourth argument must be `1`. The `pfarg_load_t` structure is defined as follows:

```
typedef struct {
     pid_t              load_pid;
     uint16_t           load_set;
} pfarg_load_t;
```

The fields of the structure are defined as follows:

- `load_pid` : the identification of the thread to which the context must be attached.
- `load_set` : indicates the event set to load first.

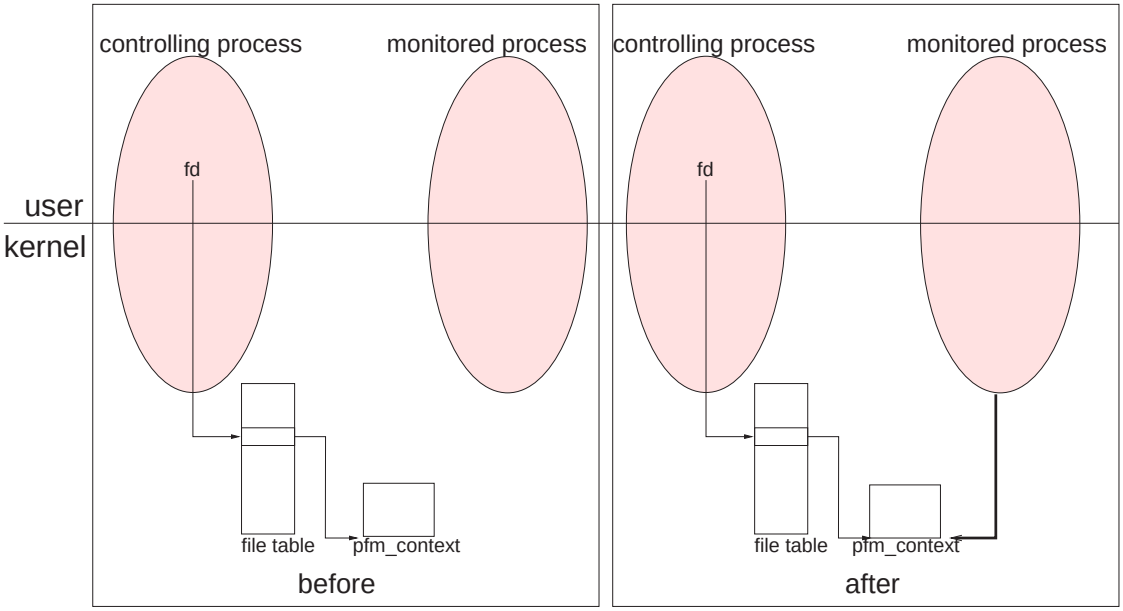


Figure 3.5: the effect of the PFM_LOAD_CONTEXT command.

The value of `load_pid` is the identification of the kernel thread. On Linux, this is the value returned by *gettid()* and not *getpid()* on NPTL-enabled systems. This value may be different from that returned by *pthread_create()*.

A context can only be attached to one thread at a time. A thread can have, at most, one context attached to it. The thread to which the context is attached is called the *monitored thread*. While the context is attached, it is under the control of the threads of the controlling process, called *controlling threads*. For self-monitoring threads, the controlling thread is the *monitored thread*. Upon return, the interface guarantees that the context is effectively attached to the designated thread. The effects of the command are depicted in figure 3.5 where we only consider the case of a single-threaded process. The link between the *monitored* thread on the right-hand side and the context is established by the command.

The caller must have permissions to access the *monitored* thread. It is up to each implementation to define to set of permissions and capabilities necessary to allow the operation.

As part of the attachment, the notification message queue is reset. All pending messages are lost.

The command loads the current software state of the designated event set onto the actual PMC and PMD registers. For the PMC registers, the last values programmed via the PFM_WRITE_PMCS command are used. For PMD registers, their current values, as would be returned by a PFM_READ_PMDS

command, are used. If a PMC or PMD register has never been explicitly programmed, then its default value is used.

The interface guarantees that the PMU is not active, i.e., no events are effectively collected. A subsequent call to PFM_START or equivalent is required to activate monitoring.

For a non self-monitoring per-thread context, some implementations may impose restrictions as to when this command can be issued. In particular, there may be some restrictions on multi-processor machines to ensure that the thread is actually stopped.

#### Per-thread context

Each implementation must ensure that more than one per-thread session may exist at the same time in the entire system. This implies that the PMU state must be saved and restored on context switches.

#### System-wide context

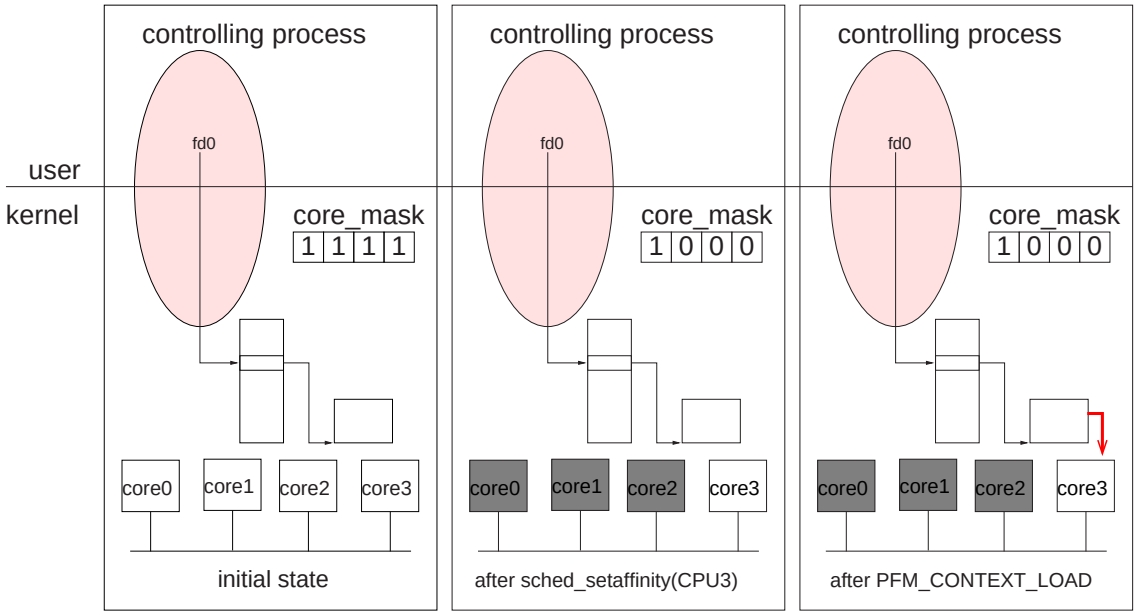


Figure 3.6: PFM_LOAD_CONTEXT for a system-wide context.

For a system-wide context, the command is used to bind a context to a specific CPU core. By design, a system-wide context only captures events on one CPU core. Simultaneous full coverage of a multi-processor machine is achieved using multiple contexts.

For a system-wide context, the value of `load_pid` must be that of the calling thread. In other words, self-monitoring is enforced. Any other combination is rejected by this command for this type of context. However, this restriction only applies to this command. Later on, access to the context, for other commands, is granted to threads with access to the file descriptor and which run on the monitored CPU core.

The determination of the CPU core the context is bound to is done during the execution of this command. The interface uses the CPU core on which the calling thread is executing. Implementations

must ensure that preemption is disabled during the call. The CPU core cannot be changed unless the context is detached via PFM_CONTEXT_UNLOAD and then re-attached to another CPU core. To avoid problems, it is highly recommended to pin the controlling thread to the monitored CPU core prior to calling this command. On Linux, this can easily be done using the *sched_setaffinity()* system call. A typical sequence of code would be:

```
unsigned long mask;
pfarg_load_t load;
mask = 1UL << core_to_monitor;
ret = sched_setaffinity(getpid(), &mask, sizeof(mask));
if (ret) exit(1);
/* now running on the right core */
load.load_pid = gettid();
perfmonctl(fd, PFM_LOAD_CONTEXT, &load, 1);
```

Affinity is adjusted at the thread level not at the process level, hence the *gettid()* instead of the *getpid()* for NPTL-enabled Linux systems. The perfmon interface does not implicitly perform the affinity operation. There can only be one system-wide context bound to a CPU core at a time.

An example is shown in figure 3.6 in a 4-way (4 CPU cores) machine. To simplify the figure, we consider a single-threaded process but the same behavior would apply to a multi-threaded process. On the left-hand side, the controlling thread can run on any of the 4 cores as indicated by the `core_mask` which represents the set of allowed cores for the thread. In the middle of the figure, we see what happens after the thread changed its CPU core affinity to run only on core3. The shaded processor boxes indicate forbidden cores. Then, the PFM_LOAD_CONTEXT command is issued. The effect of the command is shown on the right-hand side, where the context is now bound to core3.

### Per-thread and System-wide contexts co-existence

There can be more than one per-thread contexts in the system. The interface allows one system-wide context per CPU core. A context is said to *exist* on a CPU core when it is *loaded*. When it is *unloaded*, it does not have access the PMU, therefore there is no resource sharing issue.

If a system-wide and a per-thread contexts were to exist at the same time, they would have to share the PMU resource when the thread runs on the CPU core which is monitored. In theory, there is nothing to prevent such thing from happening. In some cases, it would even make sense. For instance, imagine a configuration with long running system-wide monitoring session, such as what happens with DCPI [2], running as a background process and an application developer trying to collect performance data on a program. It would certainly be nice to allow the two measurements to run at the same time. Yet there are some difficulties in sharing the PMU. The two measurements may need the same PMU registers and they may not start and stop at the same time.

The interface manages the system-wide and per-thread contexts totally independently from each other. Therefore there no inherent barrier in allowing co-existence. There are too many PMU and operating system constraints to assume sharing could be accomplished for every system. Thus, it is up to each implementation to figure out if co-existence is possible and would be useful. However such determination must be done only when the context is attached not when it is created.

When co-existence is not possible, the interface currently enforces *mutual exclusion*. In that case, a per-thread and system-wide contexts cannot exist at the same time. In the future, the interface will provide a PMU preemption mechanism to allow better sharing of the PMU, see section 6.5.

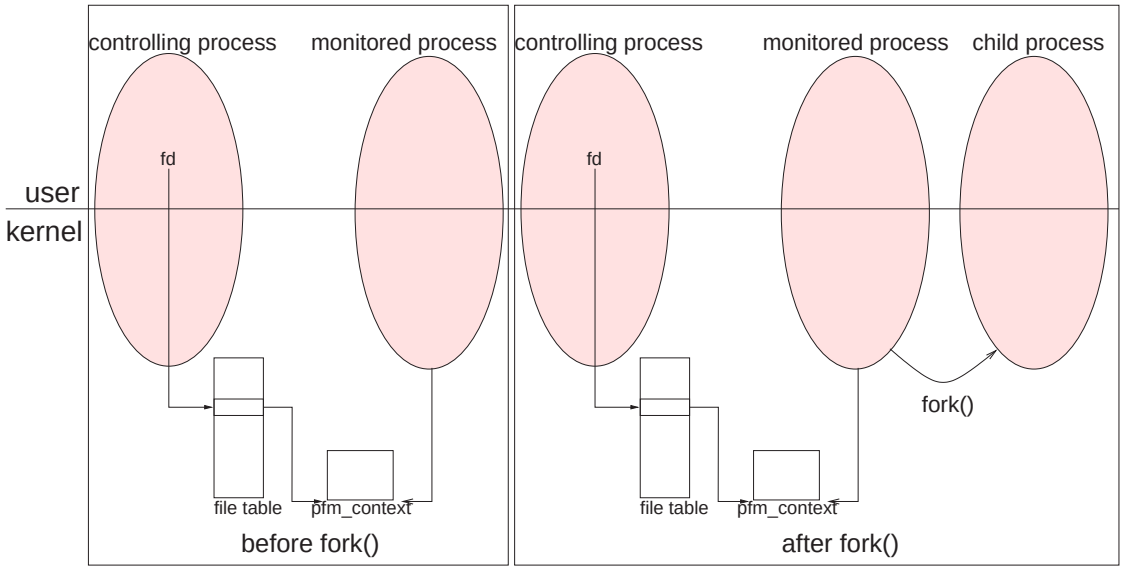


Figure 3.7: monitored process and *fork()*

### Behavior on *fork()*

When the monitored thread invokes the *fork()* system call, the context is not inherited by the initial thread of the child process. The same is true during a *vfork()*. The behavior is depicted in figure 3.7. If an application needs to monitor across *fork()*, it needs to detect the creation of the new process and create a new context for each new thread associated with it. On Linux, the detection can be achieved with the *ptrace()* system call. See section 3.1.1 for the file descriptor inheritance rules on *fork()*.

### Behavior on *pthread_create()*

When a monitored thread creates new threads with *pthread_create()*, the behavior depends on the type of implementation of the thread package.

When threads are implemented as a user level library, the kernel has no visibility on the thread, therefore the context is shared for all threads.

When threads are implemented using kernel threads, they can independently be identified by the kernel and, hence, by the perfmon interface. A context can only be attached to a single thread at a time. Hence, it is never inherited in the newly created thread. If a monitoring tool needs to monitor a newly created thread, it needs to use another operating system interface to detect the creation of the thread. Then it needs to create and attach the context to the new thread. On Linux, the detection can be achieved with the *ptrace()* system call.

### Behavior on *exec()*

When a monitored thread invokes the *exec()* system call, the context remains attached. If monitoring was active, it remains active.

**Interaction with event sets**

The set specified in `load_set` is loaded onto the PMU first. It can be any set from the list of defined sets. This command verifies that the list of events and especially the link from one set to another are sane, otherwise an error is returned. After this command returns, the list of event sets cannot be modified until the context is detached.

When the set specified in the `load_set` field has the PFM_SETFL_OVFL_SWITCH flag set, all counting monitors designated as triggers have their switch overflow counter reset to the value specified in the `reg_ovflsw_thres` field of their PMD register. Similarly, if the set has the PFM_SETFL_TIME_SWITCH flags, set the timeout is reset to the value specified in the `set_timeout` field when the set was created or last modified.

**Return values**

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EBADF : invalid file descriptor
  - EPERM : the caller does not have permission to operate on the designated thread
  - EINVAL : the designated event set does not exist
  - EFAULT : an invalid address is passed. Most likely the `pfarg_load_t` argument is invalid or the number of element is different from `1`
  - EBUSY : the command cannot be executed at this time. Some of the possible reasons are:
    * the designated thread already has a context attached to it
    * the current CPU already has a context bound to it

### 3.1.8 The PFM_UNLOAD_CONTEXT command

**Description**

The PFM_UNLOAD_CONTEXT command is used to detach a context from a thread. The state of the actual PMU is saved into the software state. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(fd, PFM_UNLOAD_CONTEXT, NULL, 0);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. The command takes no argument therefore the third and fourth arguments are shown as NULL and 0 respectively.

The context must already be attached for this command to succeed. Upon successful, return the context is detached and the PMU state is saved. Any subsequent calls to PFM_READ_PMDS returns the values that were in the PMU at the time the context was detached.

As part of the call, the context is stopped as if a PFM_STOP command had been issued. This avoids bad surprises should the context be re-attached later.

The notification message queue is not drained during this command, therefore pending messages can still be extracted.

Once detached, a context can be re-attached to any thread or CPU core. The attach-detach cycle can be repeated as many times as is necessary for the measurement.

For all non self-monitoring threads, some implementations may impose restrictions as to when this command can be issued. In particular, there may be some restrictions on multi-processor machines to ensure that the thread is actually stopped.

**System-wide context**

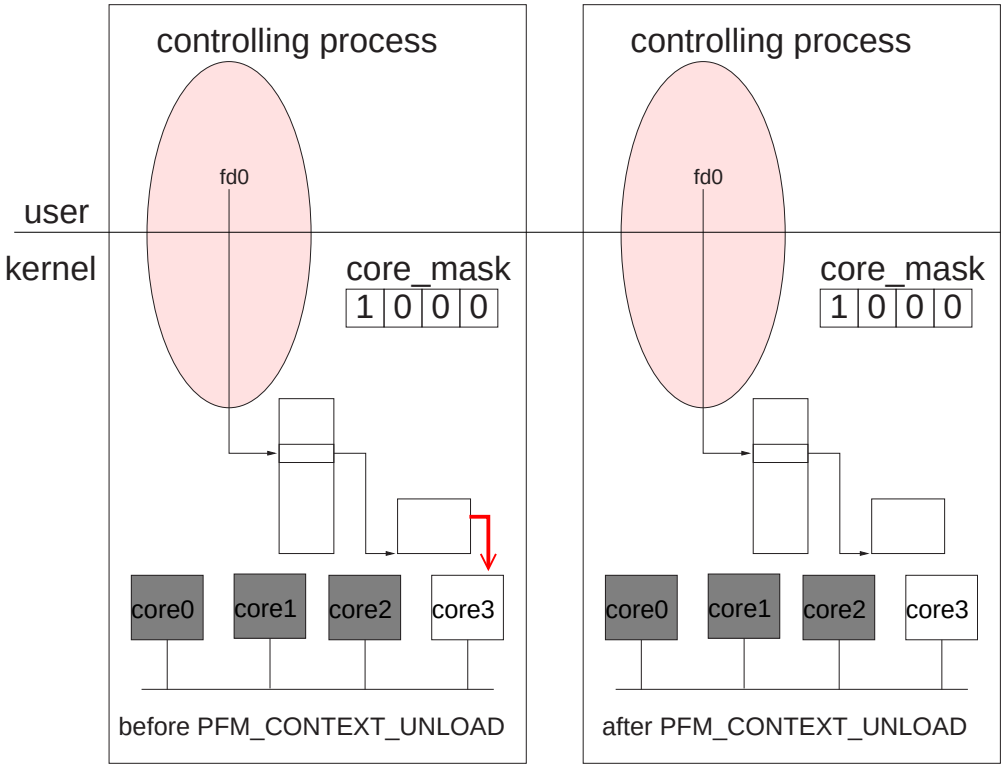


Figure 3.8: effect of the PFM_UNLOAD_CONTEXT command.

Upon successful return, a system-wide context is not bound to a CPU core anymore. As such another system-wide context can now be bound to the same CPU core. To detach a system-wide context, the calling thread must be running on the CPU core the context is bound to. This can easily be achieved using the *sched_setaffinity()* system call or equivalent. The perfmon interface never changes the CPU core affinity of the calling thread.

Once detached a system-wide context can be re-attached to any CPU core. In that case, the CPU core affinity of the controlling thread must be adjusted accordingly.

**Event set interactions**

On unload, monitoring is implicitly stopped, therefore set switching is stopped as well. The interface does not guarantee which will be the last active set a the time of unload.

**Return values**

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EBADF : invalid file descriptor
  - EBUSY : the command cannot be executed at this time.

### 3.1.9 The PFM_RESTART command

**Description**

The PFM_RESTART command is used to inform the kernel that the caller is done processing an overflow notification. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(fd, PFM_RESTART, NULL, 0);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. The command takes no argument therefore the third and fourth arguments are shown as NULL and 0 respectively.

The context must be attached for this command to succeed. The command is used to finish the overflow notification *procedure*. As described in section 3.1.2, it is possible to setup a PMC register to send a notification message when the associated counting PMD register overflows.

This basic mechanism can be used to implement user-level sampling. It is also used in conjunction with kernel level sampling formats. In that case, the overflow notification is delivered when a certain condition is reached. That condition is determined by the sampling format. For instance, with the default sampling format the condition is that the buffer has become full.

Whether or not a sampling buffer is associated to the context, this command signals that the controlling thread is done processing the notification.

The interface mandates that there is one call to PFM_RESTART per overflow notification message received. More than one message can be extracted in a single *read()* call, see section 3.3 for more details.

**Behavior without a sampling buffer format**

When no sampling buffer is used and there is an overflow, monitoring is *masked*. That means that no events are actually collected in any PMD register. The term *masked* is used instead of *stopped* to avoid confusion with the PFM_STOP command. It is up to each implementation to figure out how to implement the *masked* state. However, each implementation must ensure that a PFM_START command is not required to reactivate monitoring after a PFM_RESTART. Between the moment the overflow occurred and the time the restart is issued the following conditions exist:

- no qualified events are actually collected
- if the PFM_FL_NOTIFY_BLOCK flag is set for the context and the monitored thread is not self-monitoring, then it is blocked.

The consequences of invoking the command are:

- each counting PMD register which overflowed is reloaded with a new value. This value, called *long reset value*, is the content of the `reg_long_reset` field provided during the PFM_WRITE_PMDS command. If randomization is requested a pseudo-random value is used. A non-overflowed PMD register keeps its value unless it is designated in the bitvector `reg_reset_pmds` of an overflowed counting PMD register. in which case, it is reset with its *long reset value*.
- if the monitored thread had been blocked with the PFM_FL_NOTIFY_BLOCK flag, it is awaken. The interface does not guarantee that the monitored thread is actually running when the call returns.
- monitoring resumes. Qualified events are again collected once the monitored thread is scheduled on a processor.

In the case of a non self-monitoring, per-thread, session, reloading of the overflowed PMD registers and resuming of monitoring may not be completed by the time the command returns. For all others types of session, the interface does guarantee both operations are indeed completed.

#### Behavior with a sampling buffer format

The use of this command with sampling buffer format can vary a lot depending on what the format does. The following control points exists for a format:

- when to notify
- the need to mask monitoring on counter overflow
- blocking of the monitored thread
- the need for using PFM_RESTART
- which PMD registers must be reset before the notification is sent
- the need to reset overflowed PMD registers

Unlike the PFM_START, this command triggers a call-back into the sampling format. Refer to section 3.5.15 for details about the call-back. The effects of the call-backs for the default sampling format are presented in section 3.6. For other formats, please refer to the appropriate documentation.

In the case of a non self-monitoring, per-thread, session, and when the format requests reloading of the overflowed PMD registers and resuming of monitoring, the interface does not guarantee that those two operations are completed by the time the command returns. For all others types of session, the interface does guarantee both operations are indeed completed.

#### Interactions with event sets

When the active set at the time of the notification is using time-based set switching, then switching is stopped until the PFM_RESTART command is issued. That implies that the *restarted* set is the same as the set that was active at the time of the notification.

When the active set at the time of the notification is using overflow-based set switching, switching occurs if at least one of the overflowed PMD registers has reached its allowed number of overflows as indicated by the `reg_ovflsw_thres` field.

**Return values**

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EBADF : invalid file descriptor
  - EBUSY : the command cannot be executed at this time.

### 3.1.10 The PFM_CREATE_EVTSETS command

**Description**

The PFM_CREATE_EVTSETS command is used to create or modify an event set. More than one event set can be created or modified in one call. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(fd, PFM_CREATE_EVTSETS, setdesc, n);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. The third argument, *setdesc*, is a pointer to an array of structures of type `pfarg_setdesc_t`. The fourth argument, *n*, indicates the number of elements in the array. It must be greater than 0. The `pfarg_setdesc_t` structure is defined as follows:

```
typedef struct {
        uint16_t          set_id;
        uint16_t          set_id_next;
        uint32_t          set_flags;
        struct timespec   set_timeout;
} pfarg_setdesc_t;
```

The fields are defined as follows for this command:

- `set_id` : a unique number to identify the event set.
- `set_id_next`: the set to which to switch to, also called the *explicit next*. This field is ignored unless the PFM_SETFL_EXPL_NEXT flag is set.
- `set_flags` : a set of flags describing the properties of the event set. The flags are divided in two categories: generic and platform specific. The latter category describes certain features which are specific to either the host PMU or the host operating system. The bits which are not defined are *reserved* and must be cleared. It is only possible to set bits that are defined. You need to refer to the PMU model and/or operating system specific sections for more details. The following set of generic flags are defined:
  - PFM_SETFL_EXCL_IDLE : the idle thread is not monitored. This flag is only valid for system-wide sessions. It is ignored for per-thread sessions.
  - PFM_SETFL_OVFL_SWITCH: the set uses overflow-based switching.
  - PFM_SETFL_TIME_SWITCH: the set uses time-based switching.
  - PFM_SETFL_EXPL_NEXT: use the `set_id_next` as the next set to switch to from this one.

- `set_timeout` : on input this field contains the requested timeout before switching to the next set. Upon return, this field contains the effective timeout which is guaranteed to be greater or equal to the requested timeout. This field is only used when the PFM_SETFL_TIME_SWITCH flag is set. Otherwise, it is ignored.

When a context is created, a default event set, *set0* is systematically created. All other sets must be explicitly created by the application.

If the set already exists, its properties are modified with the values provided in the `pfarg_setdesc_t` structure. In all other cases, a new set is created. The initial set, *set0*, always exists, hence it can only be modified by this command.

It is only possible to create event sets when the context is detached.

Event Sets can only be created or modified when the context is detached. Theoretically, the maximum number of sets is 65536, including *set0*, however each implementation may impose other resources limitations such as the amount of memory used by a thread.

Set identifications do not need to be contiguous, holes are supported. Sets are ordered in increasing value of their `set_id` field. That order determines the default switching order. When it is time to switch from a set, the next higher order set is used in a round-robin fashion. In other words, once the last set in the list has been reached, the next set is the first in the list, i.e., *set0*. It is possible to modify this behavior using the PFM_SETFL_EXPL_NEXT flag. When this flag is set, the next set is indicated by the `set_id_next` and not by the next set in the ordered list. This field must contain the identification of a valid set once the context is attached. Hence it is possible to specify a set which has not yet been created. The checking is deferred until the context is attached with the PFM_LOAD_CONTEXT command.

Each event set can be in either no-switch, time-based or overflow-based switching mode. The initial set, *set0* is created such that there is no switching from the set.

The switching mode is determined on a per-set basis. For a given context some sets may be using time-based switching while others are using overflow-based switching.

Time-based switching is expected to be driven off the timer tick, i.e., the lowest level timer, of the operating system hence reaching a good granularity. Depending on the operating system and the underlying hardware, it may not be possible to achieve the desired timeout granularity. In this case, the *effective* timeout may be different from the *requested* timeout. The *effective* timeout is always rounded up to the next multiple of the timer granularity. It is up to application to either accept the *effective* timeout or adjust the *requested* timeout. In per-thread mode, the timeout only is active only when the monitored thread is active, i.e., the timeout is not measuring wall-clock time. More information about time-based switching is provided in section 3.7.6.

#### Interactions with custom sampling format

Some sampling formats may not support multiple event sets. This is not the case for the default sampling format. In that case, they must set the PFM_FMTFL_NOSET flag in the `pfm_buffer_fmt_t` structure. When this flag is set and the format is being used by the context, this command fails with EINVAL when the `set_id` is different from 0.

#### Security considerations

For security reasons or because of resource limitations, each implementation may limit the number of elements that can be passed in the `pfarg_setdesc_t` array.

**Return values**

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EINVAL : invalid arguments (see below for discussion)
  - EFAULT : an invalid address is passed. Most likely the `pfarg_setdesc_t` array is invalid or the number of elements is too big.

The EINVAL error code is returned when one of the `pfarg_setdesc_t` argument contains invalid information. In the case where multiple sets are created in one call, it may be difficult to figure out which element in the array caused the problem. However the interface includes a simple mechanism to help identify the invalid element by using the `set_flags` field. When EINVAL is returned, the caller may want to scan the array of `pfarg_setdesc_t` elements using the following macro and definitions:

- `PFM_SET_HAS_ERROR(flags)` : the *flags* parameter must be the value of the `set_flags` field. The macro returns non-zero if *flags* contains an error code. Otherwise, the value 0 is returned.
- PFM_SET_RETFL_EINVAL: if this flag is set in the `set_flags` field, it may be because the set is already defined, or the `set_flags` are invalid.

The command aborts at the first error therefore no further elements are processed beyond the invalid element. Elements placed before the invalid element are guaranteed processed by the interface, i.e., there is no need to resubmit them. If no error is reported by the macro, then the reason for EINVAL is different.

### 3.1.11 The PFM_DELETE_EVTSETS command

**Description**

The PFM_DELETE_EVTSETS command is used to delete an existing event set. More than one event set can be deleted in one call. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(fd, PFM_DELETE_EVTSETS, setdesc, n);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. The third argument, *setdesc*, is a pointer to an array of structures of type `pfarg_setdesc_t`. The fourth argument, *n*, indicates the number of elements in the array. It must be greater than 0.

The `pfarg_setdesc_t` data structure, as used with the PFM_CREATE_EVTSETS command, is defined as follows:

```
typedef struct {
        uint16_t          set_id;
        uint16_t          set_id_next;
        uint32_t          set_flags;
        struct timespec   set_timeout;
} pfarg_setdesc_t;
```

The fields are defined as follows for this command:

- `set_id` : a unique number to identify the event set to delete.
- `set_id_next` : this field is ignored for this command.
- `set_flags` : there are no input flags defined for this command. All but the return flag bits are *reserved*.

  return this field may contain additional error information.
- `set_timeout` :this field is ignored for this command.

The command can only be issued when the context is detached. It is not possible to delete *set0* which is automatically created by the PFM_CREATE_CONTEXT command. That implies that there is always at least one event set defined for a context.

Removing event sets may break the chain used for switching. However, the sanity of the chain is only verified when the context is attached with PFM_UNLOAD_CONTEXT.

**Security considerations**

For security reasons or because of resource limitations, each implementation may limit the number of elements that can be passed in the `pfarg_setdesc_t` array.

**Return values**

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EINVAL : invalid arguments (see below for discussion)
  - EFAULT : an invalid address is passed. Most likely the `pfarg_setdesc_t` array is invalid or the number of elements is too big.

The EINVAL error code is returned when one of the `pfarg_setdesc_t` argument contains invalid information. In the case where multiple sets are deleted in one call, it may be difficult to figure out which element in the array caused the problem. However the interface includes a simple mechanism to help identify the invalid element by using the `set_flags` field. When EINVAL is returned, the caller may want to scan the array of `pfarg_setdesc_t` elements using the following macro and definitions:

- `PFM_SET_HAS_ERROR(flags)` : the *flags* parameter must be the value of the `set_flags` field. The macro returns non-zero if *flags* contains an error code. Otherwise the value 0 is returned.
- PFM_SET_RETFL_NOSET: if this flag is set in the `set_flags` field, the specified set does not exist.

The command aborts at the first error therefore no further elements are processed beyond the invalid element. Elements placed before the invalid element are guaranteed processed by the interface, i.e., there is no need to resubmit them. If no error is reported by the macro, then the reason for EINVAL is different.

### 3.1.12 The PFM_GETINFO_EVTSETS command

#### Description

The PFM_GETINFO_EVTSETS command is used to retrieve information about event sets. Information on more than one event set can be retrieved in one call. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(fd, PFM_GETINFO_EVTSETS, setinfo, n);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. The third argument, *setinfo*, is a pointer to an array of structures of type `pfarg_setinfo_t`. The fourth argument, *n*, indicates the number of elements in the array. It must be greater than 0.

The `pfarg_setinfo_t` data structure is defined as follows:

```
typedef struct {
        uint16_t        set_id;
        uint16_t        set_id_next;
        uint32_t        set_flags;
        uint64_t        set_switch_pmds[PFM_MAX_PMD_BITVECTOR];
        uint64_t        set_runs;
        struct timespec set_timeout;
        struct timespec set_act_duration;
} pfarg_setinfo_t;
```

The fields are defined as follows for this command:

- `set_id` : a unique number to identify the event set.
- `set_flags` : on input, there are no flags defined for this command. All but the return flag bits are *reserved*. Upon successful return, this field contains the flag values that were passed when the set was created or last modified. Details about the flags are given in section 3.1.10. In case of error, extra information is also contained in this field.
- `set_switch_pmds`: this field is ignored on input. Upon successful return and if the set is using overflow-based switching, this bitvector contains the last overflowed PMD registers which caused a switch from this set. When time-based switching is used, this vector is all zeroes.
- `set_runs` : on input, this field is ignored. Upon successful return, this field contains the number of times the set was the active set. This is a cumulative count since the context was created. It is not reset when the context is attached via PFM_LOAD_CONTEXT.
- `set_timeout` : on input, this field is ignored. Upon successful return, this field contains the remainder of the effective timeout. If the set is using overflow-based switching, this field is all zeroes.
- `set_act_duration`: on input, this field is ignored. Upon successful return, this field contains the cumulative time duration the set was the *active* set.

The bitvector `set_switch_pmds` is systematically reset when the context is attached to a thread to avoid reporting stale information. Hence issuing the command just after a PFM_LOAD_CONTEXT command yields an all-zero bitvector.

The set_runs fields may be used to scale the accumulated values of the PMD registers. This field can return a value of 0 if the set was never designated as the first set with PFM_LOAD_CONTEXT or PFM_START and it was never switched to. In the case of PFM_LOAD_CONTEXT, the number of runs is incremented even though monitoring may never be activated via PFM_START. In the case, the set is the active set and the next set happens to be itself, the number of runs is also incremented. This situation arises when the set is the only set defined and switching is selected or when the set has an *explicit next* pointing to itself.

The set_active_duration returns the period of time the set was the active set for the context. Only the periods of time for which monitoring is active are accounted for. That means the periods between a PFM_START and PFM_STOP or PFM_UNLOAD_CONTEXT. The periods of time monitoring is masked following an overflow are not included. The reported time is cumulative since the set was created. The only way to reset, is to delete the set and recreate it. It is the responsibility of the application to maintain a past value to derive a time delta.

The command can be issued at any time including when the context is attached to a thread. However, for all non-self monitoring per-thread context, the interface does not guarantee that set_act_duration is accurate unless the context is detached.

**Security considerations**

For security reasons or because of resource limitations, each implementation may limit the number of elements that can be passed in the pfarg_setinfo_t array.

**Return values**

- 0: the command was successful.
- -1: there was an error. The value of errno can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EINVAL : invalid arguments (see below for discussion)
  - EFAULT : an invalid address is passed. Most likely the pfarg_setdesc_t array is invalid or the number of elements is too big.

The EINVAL error code is returned when one of the pfarg_setdesc_t argument contains invalid information. In the case where the information for multiple sets is retrieved in one call, it may be difficult to figure out which element in the array caused the problem. However the interface includes a simple mechanism to help identify the invalid element by using the set_flags field. When EINVAL is returned, the caller may want to scan the array of pfarg_setdesc_t elements using the following macro and definitions:

- PFM_SET_HAS_ERROR(flags) : the *flags* parameter must be the value of the set_flags field. The macro returns non-zero if *flags* contains an error code. Otherwise, the value 0 is returned.
- PFM_SET_RETFL_NOSET: if this flag is set in the set_flags field, the specified set does not exist.

The command aborts at the first error therefore no further elements are processed beyond the invalid element. Elements placed before the invalid element are guaranteed processed by the interface, i.e., there is no need to resubmit them. If no error is reported by the macro, then the reason for EINVAL is different.

### 3.1.13 The PFM_GETINFO_PMCS command

#### Description

The PFM_GETINFO_PMCS command can be used at any time to retrieve the information about implemented PMC registers. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(0, PFM_GETINFO_PMCS, pmcs, n);
```

There is no need to have a context to access this command, therefore the first argument is ignored. The third argument, *pmcs*, is a pointer to an array of structures of type `pfarg_pmcinfo_t`. The fourth argument, *n*, indicates the number of elements in the array. It must be greater than 0.

The `pfarg_pmcinfo_t` structure is defined as follows:

```
typedef struct {
        uint16_t    reg_num;
        uint16_t    reg_type;
        uint32_t    reg_flags;
        uint64_t    reg_def_value;
        uint64_t    reg_rsvd_mask;
        size_t      reg_index;
} pfarg_pmcinfo_t;
```

The `pfarg_pmcinfo_t` structure is defined as follows:

- `reg_num` : the PMC register index.
- `reg_def_value` : ignored on input. Upon successful return, this field contains the default value of the PMC register. The value is highly specific to the host PMU. When the width of the actual register is less than 64 bits, the value is zero-extended.
- `reg_flags` : there are no input flags defined for this command. All but the return flag bits are *reserved*.
- `reg_rsvd_mask` : ignored on input. Upon successful return, this field contains a bitmask indicating the *reserved* bits in the actual register. When a bit is set in the mask, the corresponding bit in the PMC register is *reserved*. When the width of the actual register is less than 64 bits, the value is ero-extended.
- `reg_type`: ignored on input. Upon successful return, this field indicates the *type* of the actual hardware register. On all PMU models, each PMC registers maps to a particular hardware register. The following *types* are defined:
  - PFM_REG_TYPE_PMC : the register maps to a Peformance Monitoring Configuration (PMC) register on the actual PMU. This is the case for Itanium® 2, for instance.
  - PFM_REG_TYPE_MSR: the register maps to a Machine Specific Register (MSR) on the actual PMU. This is the case for a Pentium 4, for instance.
  - PFM_REG_TYPE_SPR: the register maps to a Special Purpose Register (SPR) on the actual PMU. This is the case for a PowerPC, for instance.
  - PFM_REG_TYPE_IBR: the register maps to an Instruction Breakpoint Register (IBR) on the actual PMU. This is the case for a Itanium® and Itanium® 2, for instance.

  - PFM_REG_TYPE_DBR: the register maps to a Data Breakpoint Register (DBR) on the actual PMU. This is the case for a Itanium® and Itanium® 2, for instance.
  - PFM_REG_TYPE_PMD: the register maps to a Performance Monitoring Data (PMD) register on the actual PMU. This constant is mostly used for PMD registers.

- `reg_index` : ignored on input. Upon successful return, this field indicates the index or address of the actual hardware register corresponding to the PMC. When the actual register is completely described by its name, this field contains the special value PFM_REG_IDX_NONE.

It is important to note that this is not because an application can retrieve the default value of a PMC register that this register can actually be programmed. Depending upon the PMU model, some PMC registers may be reserved for kernel use only.

The `reg_rsvd_mask` field is used to retrieve which bits in the PMC register are *reserved*. Reserved bits must retain their default value as returned by `reg_def_value`. An implementation must preserve all reserved bits as follows:

$$\texttt{final\_value} = (\texttt{value} \ \& \ \sim \texttt{reg\_rsvd\_mask}) \mid (\texttt{reg\_dfl\_value} \ \& \ \texttt{reg\_rsvd\_mask})$$

The masking operation may be done silently during the PFM_WRITE_PMCS command, i.e., no error may be returned if the value passed by the application has modified reserved fields.

The command can be used to figure out which actual hardware register corresponds to a PMC register. The mapping is dictated by the implementation and is unique for each PMC register. Refer to section 3.2 for more details on register mappings.

The `reg_index` data type is chosen to accommodate platforms where the index is in fact some address or offset in a large space.

The interface guarantees that the register type and index always corresponds to the actual hardware registers and not some PMU-specific logical name. Using the type and index, it must be possible to craft the assembly instruction(s) to read or write the register.

#### Extensibility of types and indexes

As new PMU models are developed with new register names, the list of types can easily be extended without backward compatibility problems.

When the PMU uses non-indexed registers, dedicated types must be created.

#### Security considerations

For security reasons or because of resource limitations, each implementation may limit the number of elements that can be passed in the `pfarg_pmcinfo_t` array.

#### Return values

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:

- ENOSYS : the perfmon subsystem is not compiled into the kernel
- EINVAL : invalid arguments (see below for discussion)
- EFAULT : an invalid address is passed. Most likely the `pfarg_pmcinfo_t` array is invalid or the number of elements is too big.

The EINVAL error code is returned when one of the requested PMC register is unimplemented by the host PMU. In the case where multiple PMC registers are requested in one call, it may be difficult to figure out which element in the array caused the problem. However the interface includes a simple mechanism to help identify the invalid element by using the `reg_flags` field. When EINVAL is returned, the caller may want to scan the array of `pfarg_pmcinfo_t` elements using the following macro and definitions:

- `PFM_REG_HAS_ERROR(flags)` : the *flags* parameter must be the value of the `reg_flags` field. The macro returns non-zero if *flags* contain an error code. Otherwise, the value 0 is returned.
- PFM_REG_RETFL_EINVAL: if this flag is set in the `reg_flags` then the value for the corresponding PMC is invalid

The command aborts at the first error therefore no further elements are processed beyond the invalid element. Elements placed before the invalid element are guaranteed processed by the interface, i.e., there is no need to resubmit them. If no error is reported by the macro, then the reason for EINVAL is different.

### 3.1.14 The PFM_GETINFO_PMDS command

**Description**

The PFM_GETINFO_PMDS command can be used at any time to retrieve the information about implemented PMD registers. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(0, PFM_GETINFO_PMDS, pmds, n);
```

There is no need to have a context to access this command, therefore the first argument is ignored. The third argument, *pmds*, is a pointer to an array of structures of type `pfarg_pmdinfo_t`. The fourth argument, *n*, indicates the number of elements in the array. It must be greater than 0.

The `pfarg_pmdinfo_t` structure is defined as follows:

```
typedef struct {
        uint16_t    reg_num;
        uint16_t    reg_type;
        uint32_t    reg_flags;
        uint64_t    reg_def_value;
        uint64_t    reg_rsvd_mask;
        size_t      reg_index;
} pfarg_pmdinfo_t;
```

The `pfarg_pmdinfo_t` structure is defined as follows:

- `reg_num` : the PMD register index.

- `reg_def_value` : ignored on input. Upon successful return, this field contains the default value of the PMD register. The value is generally 0 but some depends on the host PMU. When the width of the actual register is less than 64 bits, the value is zero-extended.
- `reg_flags` : there are no input flags defined for this command. All but the return flag bits are *reserved*.
- `reg_rsvd_mask` : ignored on input. Upon successful return, this field contains a bitmask indicating the *reserved* bits in the actual register. When a bit is set in the mask, the corresponding bit in the PMD register is *reserved*. When the width of the actual register is less than 64 bits, the value is zero-extended.
- `reg_type`: ignored on input. Upon successful return, this field indicates the *type* of the actual hardware register. On all PMU models, each PMD registers maps to a particular hardware register. The supported *types* are listed in section 3.1.13.
- `reg_index` : ignored on input. Upon successful return, this field indicate the index of the actual hardware register corresponding to the PMD. When the actual register is completely described by its name, this field contains the special value PFM_REG_IDX_NONE.

It is important to note that this is not because an application can retrieve the default value of a PMD register that this register can actually be programmed. Depending on the PMU model, some PMD registers may be reserved for kernel use only.

The `reg_rsvd_mask` field is used to retrieve which bits in the PMC register are *reserved*. Reserved bits must retain the value they have in the default value returned in `reg_def_value`. An implementation must preserve all reserved bits as follows:

$$\texttt{final\_value} = (\texttt{value} \ \& \ \sim \texttt{reg\_rsvd\_mask}) \mid (\texttt{reg\_dfl\_value} \ \& \ \texttt{reg\_rsvd\_mask})$$

The masking operation may be done silently during the PFM_WRITE_PMDS command, i.e., no error may be returned if the value passed by the application has modified reserved fields.

The command can be used to figure out which actual hardware register corresponds to a PMD register. The mapping is dictated by the implementation and is unique for each PMD register. Refer to section 3.2 for more details on register mappings.

The `reg_index` data type is chosen to accommodate platforms where the index is in fact some address or offset in a large space.

The interface guarantees that the register type and index always corresponds to the actual hardware registers and not some PMU-specific logical name. Using the type and index, it must be possible to craft the assembly instruction(s) to read or write the register.

#### Extensibility of types and indexes

As new PMU models are developed with new register names, the list of types can easily be extended without backward compatibility problems.

When the PMU uses non-indexed registers, dedicated types must be created.

#### Security considerations

For security reasons or because of resource limitations, each implementation may limit the number of elements that can be passed in the `pfarg_pmdinfo_t` array.

**Return values**

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EINVAL : invalid arguments (see below for discussion)
  - EFAULT : an invalid address is passed. Most likely the `pfarg_pmdinfo_t` array is invalid or the number of elements is too big.

The EINVAL error code is returned when one of the requested PMD register is unimplemented by the host PMU. In the case where multiple PMD registers are requested in one call, it may be difficult to figure out which element in the array caused the problem. However the interface includes a simple mechanism to help identify the invalid element by using the `reg_flags` field. When EINVAL is returned, the caller may want to scan the array of `pfarg_pmdinfo_t` elements using the following macro and definitions:

- `PFM_REG_HAS_ERROR(flags)` : the *flags* parameter must be the value of the `reg_flags` field. The macro returns non-zero if *flags* contain an error code. Otherwise, the value 0 is returned.
- PFM_REG_RETFL_EINVAL: if this flag is set in the `reg_flags` then the value for the corresponding PMD is invalid

The command aborts at the first error therefore no further elements are processed beyond the invalid element. Elements placed before the invalid element are guaranteed processed by the interface, i.e., there is no need to resubmit them. If no error is reported by the macro, then the reason for EINVAL is different.

### 3.1.15 Destroying a context with *close()*

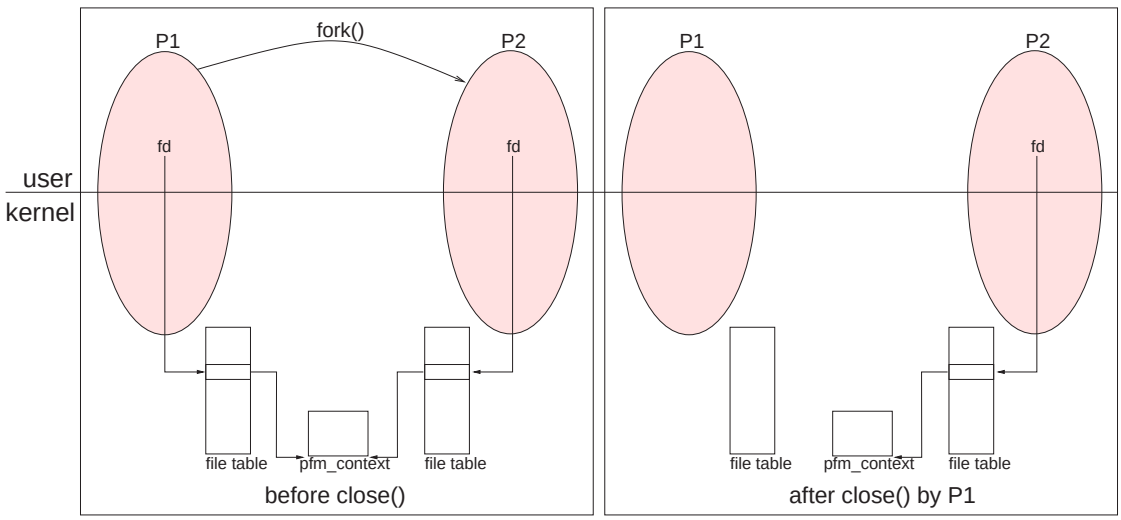


Figure 3.9: effect of *close()* on a shared detached context.

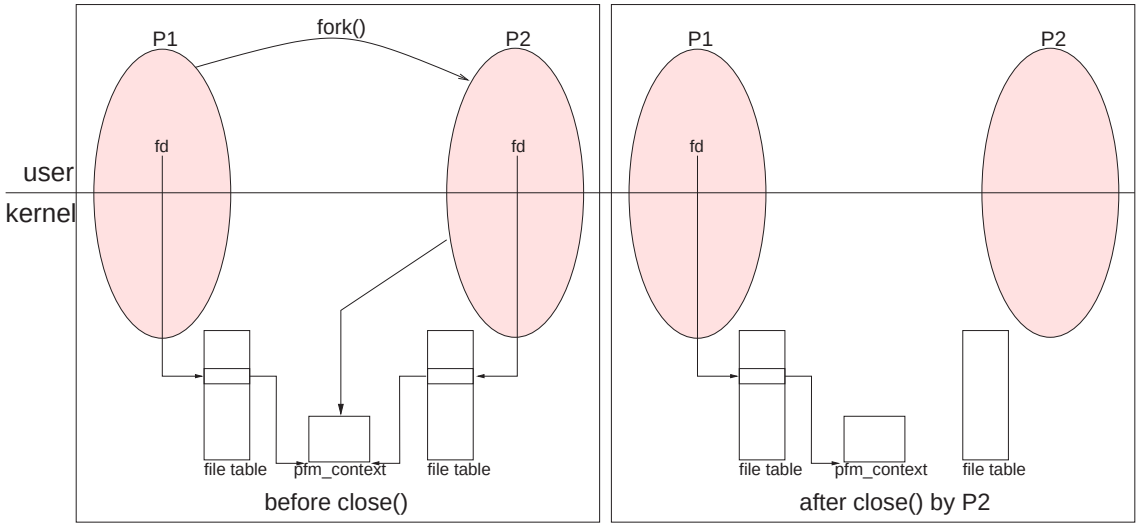


Figure 3.10: effect of *close()* on a shared attached context.

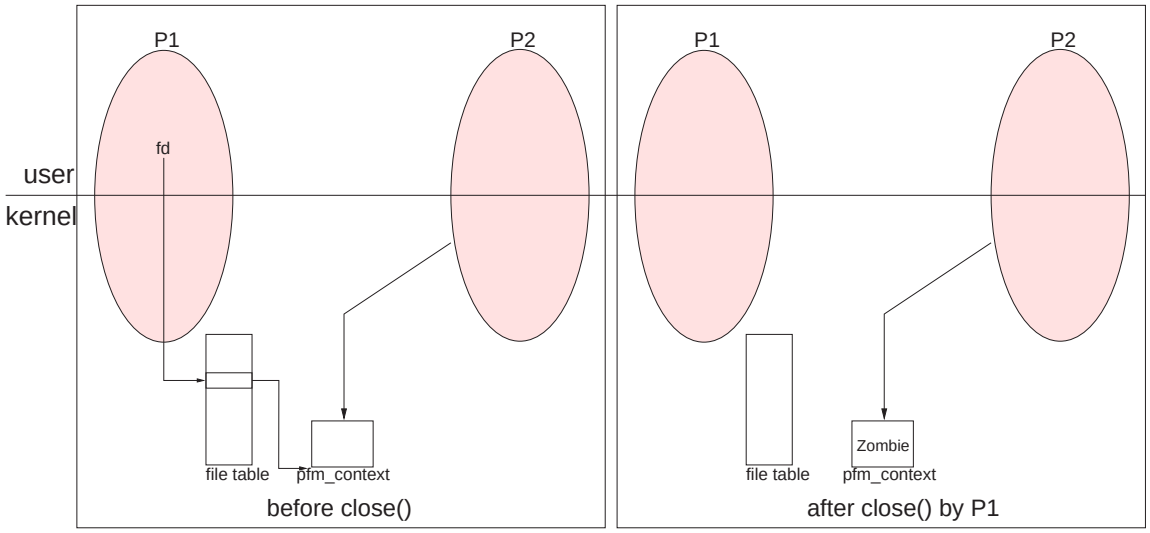


Figure 3.11: effect of *close()* on an attached context.

There is no specific *perfmonctl()* command to destroy a context. Instead the interface leverages the fact that a context is uniquely identified by a file descriptor. To destroy a context, the following system call is used:

```
close(fd);
```

The command applies to the context identified by *fd*. The descriptor must identify a valid context. Because the kernel automatically closes all open files on *exit()*, be it a voluntary exit or because the program was killed, a context is guaranteed to be destroyed at some point in time.

To destroy a context, two basic operations are necessary. First the context must be detached if attached to a thread or CPU core, and second it must be effectively destroyed, i.e., the resources must be freed. Depending on the number of remaining controlling processes and the state of the context, both operations may not necessarily be performed during *close()*. However, the interface guarantees that a controlling process loses access to the context after the call and that when eventually no more controlling processes exist and the context is detached, the context will be physically destroyed.

It is important to understand the difference between being a *controlling thread* and having a context attached to a thread or CPU core. A thread may have a context attached to it, yet this does not give it access to the context, i.e., it is not a *controlling thread*. Only threads with access to the file descriptor of the context are *controlling threads*. This aspect of a context is relevant to *close()*, because only a *controlling thread* can issue a *close()*.

Except for self-monitoring sessions, this means that if a context is attached to another thread it must somehow be detached during the call or as a consequence of it. It is up to the implementation to decide what to do in this situation. The interface guarantees that eventually the context will be destroyed.

#### Example with a shared context

Figure 3.9 shows a configuration where two processes, P1 and P2, both single-threaded processes, share access to the same detached context. When a controlling thread in P1 closes the file descriptor, the context is not actually destroyed because P2 still has its file descriptor open, i.e., remains a controlling process.

Figure 3.10 shows a similar situation, with two processes, P1 and P2, both single-threaded processes, sharing access to the same context. However, P2 also has the context attached, i.e., is self-monitoring. When that the single thread in P2 invokes *close()*, the context is unloaded and the file descriptor is closed. P2 efectively loses access to the context, however, the context remains, process P1 remains. Eventually when P1 terminates the context is actually destroyed.

When the last controlling thread invokes *close()* and the context is attached to another thread, the interface does not guarantee that the context will be unloaded and destroyed upon return. However the interface guarantees that eventually both operations will happen. As such it is possible to have a situation where a context is not controlled by any threads but it is still attached. In this case, the context is said to be *zombie*. Implementation must ensure that as soon as possible the context will be destroyed by the monitored thread.

#### Example with an attached context

Figure 3.11 shows a configuration with two processes, P1 and P2, both single-threaded. P1 is the controlling process and the context is attached to P2. When a thread in P1 invokes *close()*, P1 loses access

to the context. Yet the interface does not guarantee that the context is necessarily detached and destroyed as is shown on the left-hand side of the figure. However, there is the guarantee that eventually the context will disappear. This configuration leaves the possibility of having *zombie* contexts in the system. They remain *zombie* until the monitored thread detects the condition and cleans up or possibly next time an application tries to attach a context to P2. Implementation may chose to effectively force the destruction of the context during the call, instead of dealing with *zombies*.

#### Effects on the sampling buffer

Refer to section 3.4 for a description of what happens to the sampling buffer on *close()*.

#### Return values

- 0: the command was successful.
- -1: there was an error. The error code is in `errno`. Refer to the manual of *close()* for a description of the possible `errno` values.

### 3.1.16 The PFM_SET_CONFIG command

The PFM_SET_CONFIG command can be used at any time by a system administrator to setup global properties of the perfmon interface. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(0, PFM_SET_CONFIG, config, 1);
```

There is no need to have a context to access this command, therefore the first argument is ignored. The third argument, *config*, is a pointer to a structure of type `pfarg_config_t`. Only one such structure can be passed at a time, therefore the fourth argument must be `1`.

The `pfarg_config_t` structure is defined as follows:

```
typedef struct {
        uint32_t      cf_version;
        uint32_t      cf_flags;
        uint64_t      cf_impl_pmcs[PFM_MAX_PMC_BITVECTOR];
        uint64_t      cf_impl_pmds[PFM_MAX_PMD_BITVECTOR];
        gid_t         cf_sys_gid;
        gid_t         cf_thread_gid;
        size_t        cf_arg_size_max;
        size_t        cf_smpl_buf_size_max;
        uint16_t      cf_counter_width;
} pfarg_config_t;
```

The fields are defined as follows:

- `cf_version` : this field is ignored for this command.
- `cf_flags` : a set of flags to enable or disable certain perfmon features. The flags are divided into two groups: command and PMU-specific flags. At this point, no common flags are defined.

- `cf_impl_pmcs` : this field is ignored for this command.
- `cf_impl_pmds` : this field is ignored for this command.
- `cf_sys_gid` : the identification of the group of users allowed to create system-wide contexts. By default, any user from any group can create such a context. To restore the default value, the special value PFM_GROUP_ANY must be passed.
- `cf_thread_gid` : the identification of the group of users allowed to create per-thread contexts. By default, any user from any group can create such a context. To restore the default value, the special value PFM_GROUP_ANY must be passed.
- `cf_arg_size_max` : the maximum size in bytes for the vector arguments of the *perfmonctl()* system call. By default, the limit is set to the default page size of the operating system. The limit cannot be smaller than the smallest page size.
- `cf_smpl_buf_size_max` : the maximum size in bytes that can be allocated for all the sampling buffers existing in the system at any one time. The limit cannot be smaller than the smallest page size.
- `cf_counter_width` : this field is ignored for this command.

This command can be issued at any time. However, only the most privileged user is allowed to make the call. On a UNIX system, that would normally correspond to the `root` user. The effects of the command are not retroactive.

It is the responsibility of the system administrator to decide which users are part of the groups. This command does not check the validity of the group identifications. The group checking are performed when the PFM_CREATE_CONTEXT command is issued and are based on the group identification of the calling thread. By default, there is no group restrictions.

The maximum size for the vector arguments protects against potential abuses which could lead to large consumption of system memory. By default, the limit is set to the default size of a page. The limit cannot be smaller than the smallest page size to ensure that, at least, a vector with only one element can be passed for any command. On operating systems supporting variable page sizes, the limit must be set to the smallest page size.

The maximum size for sampling buffer protects against potential abuses which could lead to large consumption of system memory. Depending on the sampling format used the buffer may not be allocated through the custom sampling format interface, in which case this limit is not relevant. By default, the interface does not impose a limit because it is very hard to evaluate as it applies to all sampling buffers in the system at any one time. The value depends on factors which can only be determined at runtime such as the amount of memory installed in the system, for instance. It is advised that each implementation adjust the limit at boot time.

### Linux implementation

On a Linux system, the interface to access perfmon configuration could also be implemented using a `sysctl` or `sysfs` interface. But the command-style interface must be preserved to ensure portability across operating systems.

**Return values**

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EFAULT : an invalid address is passed, most likely `pfarg_config_t` is invalid or the number of elements is different from `1`.
  - EPERM : the caller does not have the permission to issue the command.

### 3.1.17 The PFM_GET_CONFIG command

The PFM_GET_CONFIG command can be used to retrieve the current settings for the global properties of the perfmon interface. For this command, the invocation of *perfmonctl()* is as follows:

```
perfmonctl(0, PFM_GET_CONFIG, config, 1);
```

There is no need to have a context to access this command, therefore the first argument is ignored. The third argument, *config*, is a pointer to a structure of type `pfarg_config_t`. Only one such structure can be passed at a time, therefore the fourth argument must be `1`.

The `pfarg_config_t` is the same as the one used for the PFM_SET_CONFIG command. It is defined as follows:

```
typedef struct {
        uint32_t      cf_version;
        uint32_t      cf_flags;
        uint64_t      cf_impl_pmcs[PFM_MAX_PMC_BITVECTOR];
        uint64_t      cf_impl_pmds[PFM_MAX_PMD_BITVECTOR];
        gid_t         cf_sys_gid;
        gid_t         cf_thread_gid;
        size_t        cf_arg_size_max;
        size_t        cf_smpl_buf_size_max;
        uint16_t      cf_counter_width;
} pfarg_config_t;
```

The fields are used as follows:

- `cf_version` : contains the version number for the implementation. The version is decomposed into a major and minor numbers. Each number occupies one half of the 32-bit field, i.e., 16-bit each. The following macros are defined to access the two numbers:
  - `PFM_VERSION_MAJOR(m)` : extract the major number from the version field value in *m*
  - `PFM_VERSION_MINOR(m)` : extract the minor number from the version field value in *m*
- `cf_flags` : ignored on input. Upon successful return, this field contains a set of flags describing certain perfmon features. The flags are divided into two groups: command and PMU-specific flags. At this point, no common flags are defined.
- `cf_impl_pmcs` : ignored on input. Upon successful return, this field contains a bitvector where each bit set indicates an implemented PMC register.

- `cf_impl_pmds` : ignored on input. Upon successful return, this field contains a bitvector where each bit set indicates an implemented PMD register.
- `cf_sys_gid` : ignored on input. Upon successful return, this field contains the identification of the group of users allowed to create system-wide contexts. The value is PFM_GROUP_ANY when any group of user is allowed to create this type of context.
- `cf_thread_gid` : ignored on input. Upon successful return, this field contains the identification of the group of users allowed to create per-thread contexts. The value is PFM_GROUP_ANY when any group is allowed to create this type of context.
- `cf_arg_size_max` : ignored on input. Upon successful return, this field contains the maximum size in bytes for the vector arguments of the *perfmonctl()* system call.
- `cf_smpl_buf_size_max` : ignored on input. Upon successful return, this field contains the maximum size in bytes that can be allocated by all sampling buffers existing at any one time.
- `cf_counter_width` : ignored on input. Upon successful return, this field contains the bit width of the actual hardware counters.

This command can be issued at any time. Unlike, PFM_SET_CONFIG any user can invoke the command to query the current configuration.

#### Linux implementation

On a Linux system, the interface to access perfmon configuration could also be implemented using a `sysctl` or `sysfs` interface. But the command-style interface must be preserved to ensure portability across operating systems.

#### Bitvector sizes

The PFM_MAX_PMC_BITVECTOR constant is defined like its equivalent for the PMD registers as described in section 3.1.2:

```
#define PFM_MAX_PMC_BITVECTOR (((PFM_MAX_PMCS+64-1)/64)
```

The constant PFM_MAX_PMCS represents the maximum number of PMC registers that are accessible to an application. Holes in the PMC name space are supported and the value of this constant takes it into account, i.e., the actual number of PMC registers may be less than PFM_MAX_PMCS.

#### Return values

- 0: the command was successful.
- -1: there was an error. The value of `errno` can be any one of:
  - ENOSYS : the perfmon subsystem is not compiled into the kernel
  - EFAULT : an invalid address is passed, most likely `pfarg_config_t` is invalid or the number of elements is different from `1`.

## 3.2 PMU register mappings

The perfmon interfaces exposes only a logical view of the PMU registers to applications. Internally, the logical PMC and PMD registers are mapped onto the actual PMU registers or potentially other software resources. This provides a uniform interface across all platforms which simplifies the development of application and increase the potential for code reuse.

### 3.2.1 Logical versus actual PMU registers

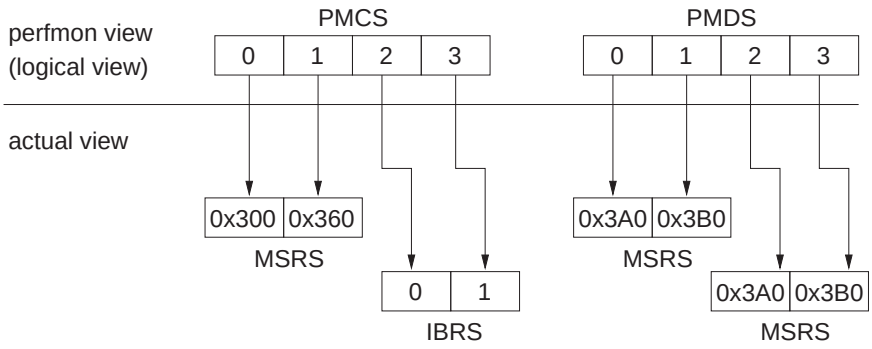


Figure 3.12: logical vs. hardware view of the PMU registers

The logical vs. hardware view is shown in figure 3.12. Any perfmon-based application must pass PMU setup in terms of PMC and PMD registers. The figure shows an hypothetical PMU with 4 logical PMC and 4 logical PMD registers. the bottom of the figure shows the actual registers and how they are mapped onto by the logical registers. Both PMC0 and PMC1 map to MSR registers with a given address or index. But both PMC2 and PMC3 map to a distinct group of registers, here some debug registers IBR0 and IBR1. The figure shows all PMD registers mapping onto MSR registers. The logical PMC and PMD register name space is not necessarily contiguous, holes are supported. Sometimes they may make it easier for the mapping. For this PMU, a PFM_WRITE_PMCS command on PMC0 eventually modifies the MSR at address `0x300`. Using a PFM_READ_PMDS command for PMD2, reads the value of the MSR at address `0x3A0`.

When the actual PMU is using a PMC and PMD based naming scheme, then the mapping may be as simple as a one to one mapping. This is the case for the Itanium Processor Family PMU, for instance.

### 3.2.2 Extending to virtual PMD registers

Using the mapping scheme, it is possible to create *virtual PMD* registers which would map onto operating system or other processor resources. For instance, one could envision that a *virtual* PMD register could indicate the amount of free physical memory in the system. By associating a PMD name to such a resource, it becomes possible to include it into profiles using the existing perfmon infrastructure, such as PFM_WRITE_PMCS and the `reg_smpl_pmds` field.

The perfmon interface does not define a default set of *virtual* PMD registers because this set may vary depending on the operating system, the type of processor and the host PMU. Enforcing a specific index in the PMD name space across all platforms may make it difficult to come up with a good mapping for actual PMD registers. As such, the definition of miscellaneous *virtual* PMD register is left to each implementation.

Certain *virtual* PMD registers may not be written.

### 3.2.3 Access to mappings

The perfmon implementation is the *authority* which dictates the mapping from logical to actual register. Each read of write operation needs to eventually touch the actual register and hence need to know the actual register name and possibly index. As such the mapping has to exist inside the kernel.

For applications, it may be necessary to know the mapping used by the implementation. This is useful if the application relies upon a helper library to figure out the event to PMU configuration register assignment and the library uses its own names for the PMU registers.

The interface provides commands to extract the mappings for both PMC and PMD registers and they are:

- PFM_GETINFO_PMCS: see section 3.1.13
- PFM_GETINFO_PMDS: see section 3.1.14

Using the mappings from figure 3.12, a call to PFM_GET_PMCS_INFO for PMC1 would return that `reg_type` is equal to PFM_REG_TYPE_MSR and that the `reg_index` is equal to `0x360`.

### 3.2.4 Mapping to the logical view

Any perfmon application must use the logical view to program the PMU via PFM_WRITE_PMCS and PFM_WRITE_PMDS. The interface does not know anything about PMU events, their encodings and how they are assigned to various PMU configuration registers. This work is left to user level code and is typically provided by a *helper* library which determines the values and assignment of PMC registers given a set of events to measure. The interface does not provide such library.

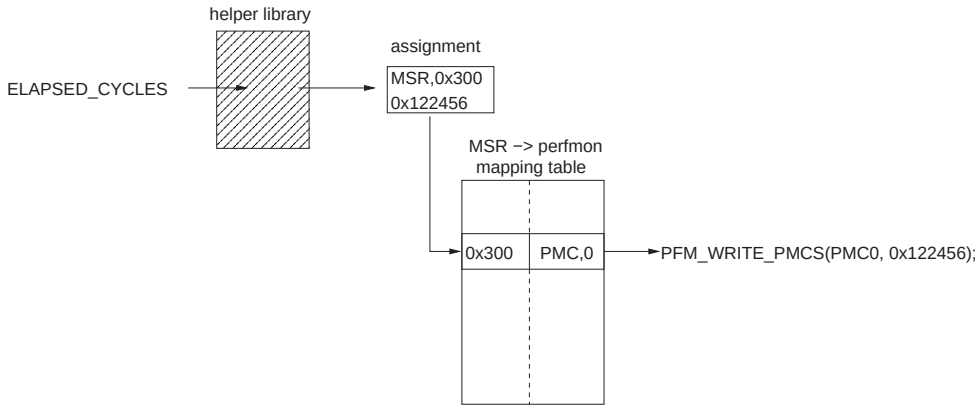


Figure 3.13: example of mapping back to perfmon logical view

Two possible situations exist:

1. the library is directly using the logical names of the perfmon interface. In other words, the library knows about perfmon.

2. the library uses its own naming scheme which is different from that of the perfmon interface.

In the first case, the value assignment can directly be passed to the PFM_WRITE_PMCS command. In the second case however, some name translation is needed. That translation is likely table-driven. The information for the perfmon mapping can easily be extracted using the two commands mentioned in the previous section. The mappings returned by the commands is guaranteed to correspond to the actual registers. We show an example of such translation in figure 3.13. The application wants to measure the ELAPSED_CYCLES event. The helper library returns the valid assignment which uses the MSR at address `0x300`. A translation table built from the information returned by PFM_GETINFO_PMCS is then used to figure the PCM register that corresponds to the MSR, in this case PMC0. Then PMC0 is programmed using the PFM_WRITE_PMCS command. Of course, the table may also be hardcoded in the application instead of being dynamically generated.

## 3.3 Event notifications

An application can receive a notification when any of the following events occur:

- a counter overflows
- a monitored thread terminates

In this section we describe the event notification interface and the termination message. The counter overflow notification is described in section 3.4 as it is associated with sampling.

### 3.3.1 The message queue

A notification is sent using a message. Each message is appended to the message queue of the context. There is only one message queue per context. The queue is accessed using the file descriptor identifying the context.

The messages are managed in a first-in, first-out (FIFO) manner therefore a new message is always appended at the end of the queue. The queue is managed as a stream of messages.

When using a sampling buffer format, it is the responsibility of the format the indicate the size of the message queue it needs to operate correctly. Without a sampling buffer format, the message queue is still necessary and it is guaranteed to accommodate at least one message at a time.

The size of the actual queue is never exposed to user level applications.

### 3.3.2 The message structure

A message consists of a contiguous set of bytes in the message queue.

Different type of messages are defined, as such each message must be uniquely identified by its *type*. Therefore all messages always begin with a type field followed by an optional *payload*. The interface defines a message structure as a union of all messages:

```
typedef union {
         uint32_t       msg_type;
         pfm_ovfl_msg_t pfm_ovfl_msg;
} pfm_msg_t;
```

The following types of message are defined:

- PFM_MSG_OVFL: overflow notification message. It is associated with the `pfm_ovfl_msg_t` structure.
- PFM_MSG_END : termination message. It is not associated with any particular message type.

A message must, first, be identified by its type using the `msg_type` field in the `pfm_msg_t` union. Then, based on the type, an application can further decode the message using the structure that corresponds to the message type, if any. Some messages do not have dedicated structure because all the information they carry is provided by the type field.

### 3.3.3 Extracting messages

All messages are extracted via the *read()* system call using the file descriptor identifying the context. Message must be extracted one at a time, i.e., one `pfm_msg_t` structure for each call. A typical sequence would be as follows:

```
pfmon_msg_t msg;
...
ret = read(fd, &msg, sizeof(msg));
if (ret != sizeof(msg)) {
    perror("cannot extract message");
}
switch(msg.msg_type) {
    case PFM_OVFL_MSG:
        ...
        break;
    case PFM_END_MSG: // does not have a body
        ...
        break;
    default:
        printf("unknown message type %d\n", msg.msg_type);
}
```

In the `default` case, an application would simply consume the message just to discard it.

Messages are extracted from the head of the queue. Once a message is read from the queue, it is removed from the queue. It is not possible to skip bytes in the queue, i.e., *lseek()* is not supported.

No partial message is ever appended to the queue, as such messages are always guaranteed to be complete.

The behavior of *read()* is such that only full messages can be extracted. in other words, the size of the read must always be a multiple of the message size as returned by the `sizeof(pfm_msg_t)` function. Based on the size of the *read()*, the following behavior is observed:

- if `size` is not a multiple of the message size : the read fails and return EINVAL.
- if `size` is a multiple of the message size : at least one full message is returned when available.

When the number of bytes in the queue is smaller than the size of the read request, the caller blocks unless non-blocking mode is enabled via the appropriate *fcntl()* call on the file descriptor.

The return value of *read()* is the number of bytes read or `-1` with `errno` indicating the reason for the error. Refer to the man page of the *read()* system call for more details on the possible return values.

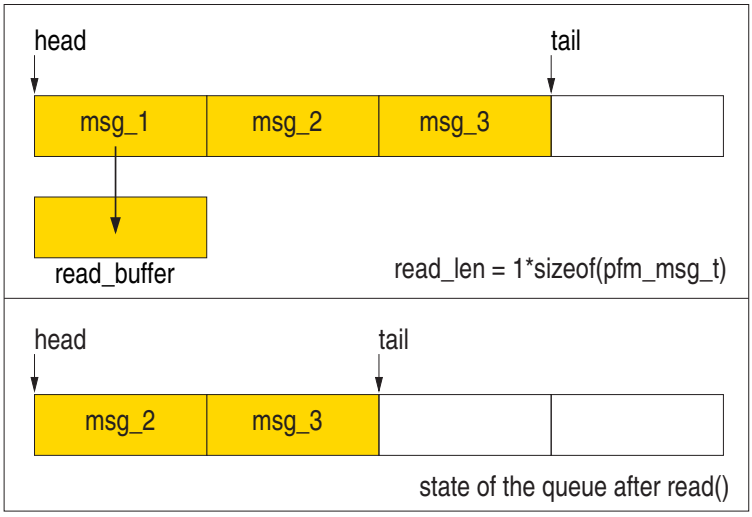


Figure 3.14: Extracting a single notification message

Figure 3.14 shows an example with a *read()* for one message. By the time of the *read()*, the queue contains 3 messages. The read request extracts the first message and copies it into the read buffer. After the call, only two messages are left in the queue.

### 3.3.4 Size of the message queue

The size of the queue is an internal parameter that is not needed by user level applications. They simply see a continuous stream of messages.

### 3.3.5 Message queue reset

The message queue is reset, each time the context is attached,i.e., all *unconsumed* messages are deleted. This is necessary to avoid stale overflows or termination messages especially in the case where the context is re-attached to a different thread.

### 3.3.6 Termination notifications

It is possible to attach a context to a thread of a process that is not a direct child of the controlling process. In that scenario, the termination of the monitored process does not translate into a SIGCHLD sent to the controlling process. Yet getting such notification is important to complete the monitoring

work. Although it is always possible to implement a polling mechanism or attach to the thread using a *ptrace()* call or equivalent, it is not necessarily very efficient to do so. For this reason, the interface provides a termination notification.

It is not possible to suppress the termination message.

### Termination guarantee

The notification indicates that the monitored thread exited. This could be because the entire process exited or simply because that thread terminated. The termination could be explicit such as through an *exit()* or implicit because of an unrecoverable error. The message applies only non self-monitoring threads. It is never generated for a system-wide context.

By the time, the message is received, the interface guarantees that the PMU state of the monitored thread has been entirely saved and the context is detached.

### Termination message

There is no specific termination message body because all the information is carried by the type and also by the file descriptor. The thread can easily be identified using the file descriptor because a context can only be attached to one thread at a time.

### Termination notification example

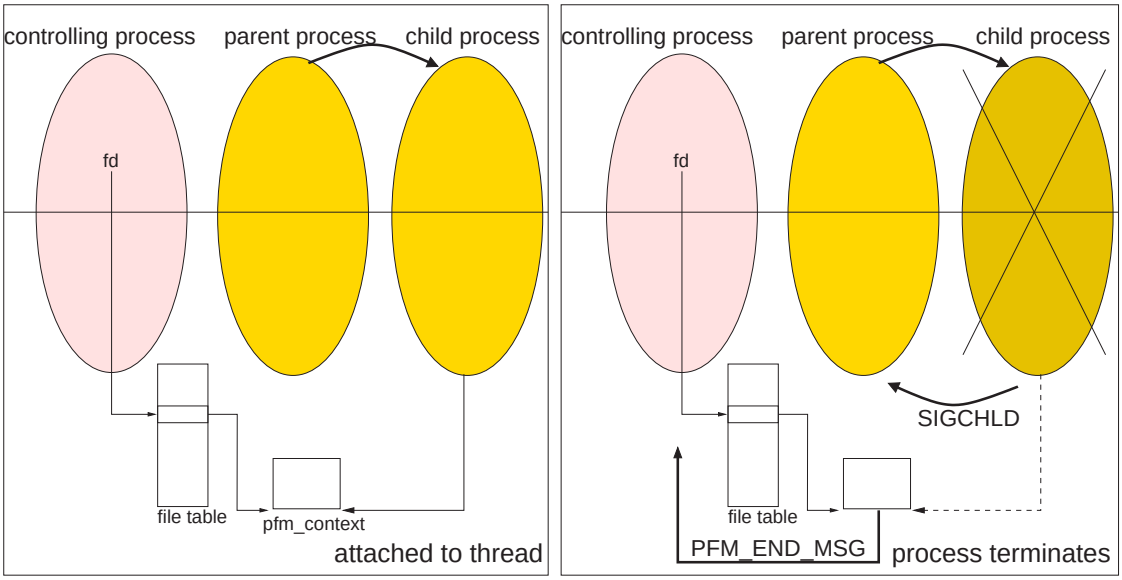


Figure 3.15: The PFM_END_MSG message.

We are considering a situation where all processes are single-threaded to simplify our description. Figure 3.15 shows a situation where a tool attaches a context to a thread in a process which is not related to the controlling process.

On the left of the figure the process in the middle is not related to the controlling process but it is the parent of the process to monitor.

On the right side of the figure, when the monitored process exits, it sends a SIGCHLD to its parent process and becomes zombie. The signal goes to the parent and not the controlling process. As part of the exit procedure of the thread, the state of the PMU is saved in the context. The context is then detached from the thread and a PFM_END_MSG is appended to the message queue. The controlling process can detect the termination by checking the queue with a *read()* system call.

#### Termination message during *fork()* and *exec()*

The message is only sent when the monitored thread does not have access to the file descriptor identifying the context, otherwise, the kernel believes it is a controlling thread that is self-monitoring.

File descriptors are cloned during *fork()* (see figure 3.2) and, by default, they are also inherited across *exec()*. When a monitored thread does a *fork()* followed by *exec()*, the thread in the newly created process has access to the file descriptor but it is not monitored. In this case, no termination message is sent because the interface assumes the new thread is a controlling thread. It is possible to alter this behavior by either explicitly closing the descriptor after *fork()* or by setting up the *close-on-exec* flag on the descriptor (FD_CLOEXEC) via *fcntl()* system call.

### 3.3.7 Asynchronous notifications

The *read()* system call is the only way to extract messages from the queue. The regular semantics of *read()* apply with the restrictions we have described in section 3.3.3. Non-blocking reads are also supported.

Under certain circumstances, it may be beneficial to receive notifications in an asynchronous manner, especially for self-monitoring threads. Here again, the regular mechanisms used on file descriptors are supported. In particular, it is possible to request that a signal be sent when a message is appended to the message queue. The default signal is SIGIO, however it may be possible to change it. This may be needed when the controlling thread already uses SIGIO for another purpose. On Linux, for instance, it is possible to change the asynchronous notification signal with the F_SETSIG command of the *fcntl()* system call. In any case, the setup follows the regular procedure to request asynchronous notification on a file descriptor. On Linux, for instance, the caller must use the *fcntl()* system call to:

- request that the file be put in asynchronous notification mode using the O_ASYNC flag.
- request ownership of the descriptor using the F_SETOWN command.

By default, the signal does not carry enough information to figure out which descriptor triggered it. In the case where a process controls multiple contexts at the same time, some polling may be required once the signal is received. Some operating systems may provide ways to include the file descriptor in the `siginfo` structure passed to the signal handler. For instance, on Linux, it is possible to get the file descriptor as the side effect of the F_SETSIG command of the *fcntl()* system call.

Each implementation must guarantee that the notification message is always appended to the message queue before the signal is sent. This ensures that the notified application can always read the message right after receiving the signal.

#### Interaction with the PFM_FL_OVFL_NO_MSG

When the PFM_FL_OVFL_NO_MSG is set for the context, no overflow message is generated. However asynchronous notifications with a signal is maintained. The idea is that the signal by itself carries enough information for the application to figure out what to do.

Even though no message is received, a call to the PFM_RESTART command is necessary to indicate that the processing of the notification is complete.

### 3.3.8 Waiting on multiple contexts

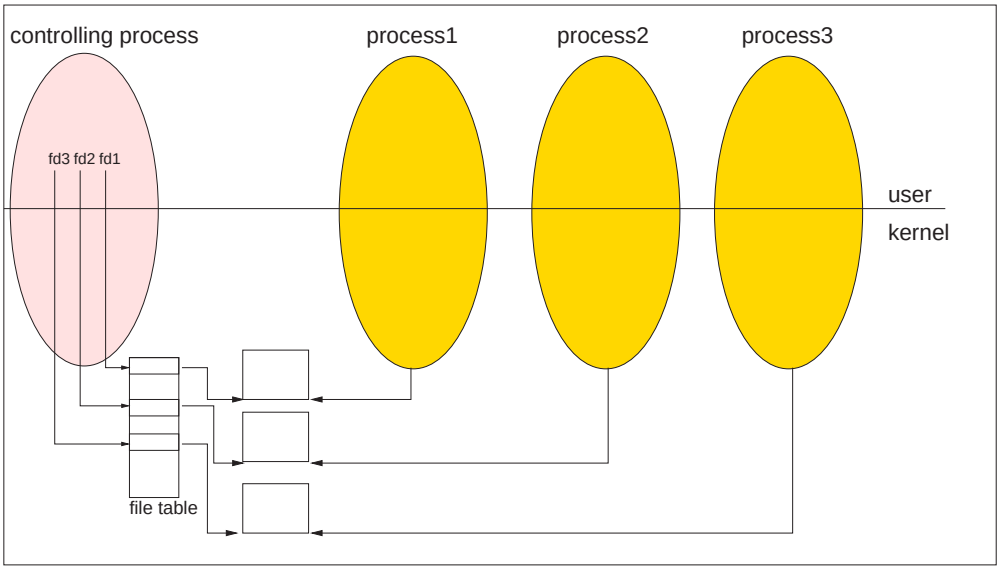


Figure 3.16: monitoring multiple threads.

By construction, it is possible for a single thread to control multiple contexts at the same time. An example of such setup is shown in figure 3.16. In that case the thread may have to wait for notifications coming from multiple contexts at the same time, i.e., multiple file descriptors.

The interface fully supports the *select()* and *poll()* interfaces to wait on multiple file descriptors at the same time.

For *select()*, the file descriptor for the context must appear in the read set. For *poll()*, the requested event must be POLLIN.

### 3.3.9 Extensibility of the notification interface

The ability to extend the notification to new types of messages is very important as new needs may surface in the future. It is important to ensure that the new messages can be added without breaking the existing applications at both source and binary levels. Obviously, the space for message type if very large, up to 4 billion. As such, messages without a body part are very easy to add. When a body is needed, it must fit within the size of the existing union structure to avoid breaking compatibility. The size of the union is determined by the size of its largest member. The largest member of the `pfm_msg_t`

union is the `pfm_ovfl_msg_t` structure. This structure is fairly large even when excluding the header. On Linux/ia64, it is currently set to 48 bytes.

The design choice of using a union implies some limitations on the extensibility of the interface. In particular, all messages must fit within a predetermined size. On the other hand, it makes it very simple and efficient to extract messages. A single call to *read()* if sufficient. The other alternative would be to view the queue as a continuous stream of bytes which could be extracted in any chunk size, i.e., like for a TCP socket. But in this case, it would be necessary to split the extraction into two calls to *read()*: one to read the type and then one to read the body. A length field in the header would also be necessary to allow skipping of unknown messages. The union approach could appear as wasting space in the queue because each message is as big as it largest member. But the performance impact is negligible because the largest member, i.e., the overflow notification message, is expected to be, by far, the most frequently generated message.

## 3.4 Support for sampling

Sampling is a technique commonly used by monitoring tools to collect profiles. There exist different types of sampling but they are all based on the same principle. At some interval record information about the state of the execution of a program. The information is stored in memory into a data structure called *sample*. Samples are collected into a *sampling buffer*. What is recorded can be diverse, such as where the program is (IP), how many caches misses, how much free memory and so on. When the interval is expressed as a unit of time, the tool is said to be using *Time-Based Sampling* (TBS). But it can also be expressed as a number of occurrences of an event, in which case the tool is said to use *Event-Based Sampling* (EBS). A good example of a tool using sampling is `gprof`. This tool produces an execution profile of a program as well as the call graph. The PMU is not used by this tool. The profile is collected using time-based sampling (TBS). Each sample contains the instruction pointer (IP) which determines where the program is at a specific time. The final profile shows where the time is spent using a per-function breakdown.The approach is based on the fact that if a program spends a lot of time in a particular function, this is either because the function is called a lot or because the function is slow to execute.

Time-based sampling using the PMU requires the following mechanisms:

- the ability to arm a timeout
- the ability to be notified when the timeout expires
- the ability to read certain counters at the end of the interval
- optionally, the ability to stop monitoring when the timeout expires. This could be useful to avoid measuring the monitoring tool itself. In that case the ability to restart monitoring is required.

The perfmon interface supports time-based sampling without any specific commands. By combining the basic *perfmonctl()* commands with a simple timeout system call such as the *nanosleep()* or *setitimer()*, it is possible to build a time-based sampling tool.

In contrast, event-based sampling using the PMU requires the following mechanisms:

- the ability to express a sampling period as a number of occurrences of an event
- the ability to load a counter with a sampling period
- the ability for the PMU to detect when a counter reaches a specific value
- the ability to receive a notification when a counter overflows
- the ability to read certain counters at the end of the interval
- optionally, the ability to stop monitoring on overflow. This could be useful to avoid measuring the overhead of the monitoring tool. In that case the ability to restart monitoring is required.

Most modern PMU can be configured to generate an interrupt when a counter overflows. This mechanism is key to implementing event-based sampling. Perfmon supports event-based sampling by providing notification on overflow, multiple sampling periods, kernel level sampling buffer, randomization of sampling periods, stop/restart on overflow. It is important to note that perfmon allows event-based sampling to be implemented entirely at the user level or partially in the kernel for better performance.

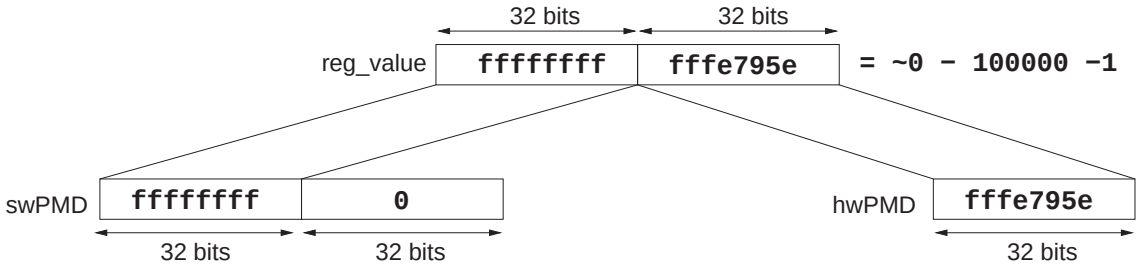


Figure 3.17: 64-bit counter emulation and sampling period.

### 3.4.1 Setting sampling periods

For event-based sampling, a sampling period is expressed as a number of occurrences of an event. For instance, a program could record a sample every time 100,000 instructions have been retired. Programming a PMD register with the value 0 and waiting until the counter reaches 100,000 does not work because PMUs typically do not have a mechanism to detect that a counter has reached a user-specified threshold. However, most modern PMUs do have a mechanism to detect when a counter overflows, i.e., when it wraps around to $0$. A 32-bit counter overflows when its value goes from $2^{32}-1$ back to $0$. In other words, there is an implicit threshold set to the $2^{\texttt{width}}$ where `width` represents the number of bits implemented by the hardware counter. When the counter reaches the threshold, an interrupt can be generated. The kernel catches the interrupt and eventually notifies the monitoring tool. Using this mechanism, a sampling period is expressed as an offset from the maximum value of the counter.

The interface exports all counters as 64-bit counters no matter what the underlying PMU implements. When the hardware does not implement 64-bit counters, the perfmon implementation must emulate, see section 3.1.3. From the point of view of a monitoring tool, counters are always 64 bits. Therefore a sampling period $p$ is always expressed as

$$\text{pmd\_value} = 2^{64} - p =\sim 0 - p - 1 = -p$$

For our example, it means that the value of the counter must be $-100000$ which expressed in hexadecimal is `0xfffffffffffe795e`. In figure 3.17, we show how the sampling period is used during a call to PFM_WRITE_PMDS to setup the counter when the host PMU only implements 32-bit counters.

Although periods are expressed as the number of occurrences of an event, it is possible to emulate time-based sampling using an event with a high correlation to time such as the number of elapsed cycles which many PMUs offer. In fact, such an event usually provides a much finer granularity for the time interval than a regular kernel timeout function.

The interface provides *three* sampling periods per counter. All three periods must be programmed using the PFM_WRITE_PMDS command. They are defined as follows:

- the *current period* is specified in the `reg_value` field. It represents the current sampling period. When the context is detached, this is actually the initial sampling period to be used.
- the *short period* is specified in the `reg_short_reset` field. It represents the sampling period to reload into the PMD register after an overflow which does not trigger a user-level notification.
- the *long period* is specified in the `reg_long_reset` field. It represents the sampling period to reload into the PMD register during a PFM_RESTART command.

It is important to realize that the interface does not have implicit knowledge of what a sampling period is. It treats all PMD register values as opaque, subject to some PMU specific limitations. As such, there is no limit on the number of sampling periods supported by the interface, it is only limited by the number of counting PMD registers which can generate an interrupt on overflow. This allows multiple distinct sampling measurements to be run in parallel, assuming there are enough counters.

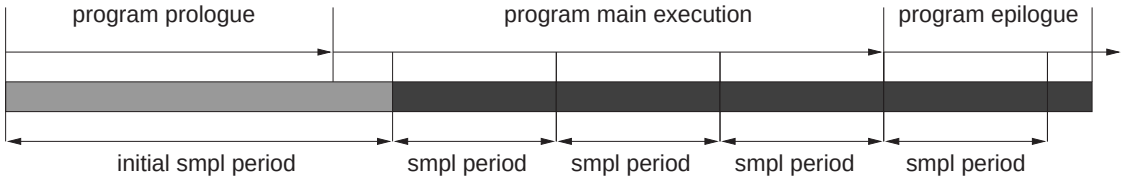


Figure 3.18: initial sampling period.

When a context is detached, the *current period* actually represents the value to load into the PMD register when the context is eventually attached, i.e., it represents the first sampling period. This value is used only in that case. If the value is later modified while the context is attached, then the current sampling period is changed. This mechanism can be useful if samples should not be captured when the context is attached. The rationale behind this is that typically a tool launches the program it wants to monitor but it is not really interested in sampling the initialization part of the execution. Therefore it is often interesting to specify a large initial period to skip over the initialization and then fall back to a smaller value during the bulk of the execution. This is shown in figure 3.18, where the initial period is skipping over all the prologue, then a different sampling period is used throughout the rest of the execution. This can be the long or short reset periods depending on how sampling is managed. A sampling period is always expressed as a number of occurrences of an event, so it is expected that monitoring tools will use an estimate for the initial sampling period, unless a *perfect* value can be determined.

The short period is only used in conjunction with kernel level sampling buffers and is subject to the behavior of the sampling format (see section 3.5). In general the short period is used after an overflow which does not require a user level notification, i.e., the kernel treated the overflow and execution resumes right away.

The long period is used to reset the overflowed PMD registers during the PFM_RESTART command. Some variations may exist depending on the sampling format behavior. In any case, the controlling thread has been notified of the overflow. Such notification is expected to require more time than simply processing the overflow in the kernel and resuming execution. Moreover, because the controlling thread is involved, some perturbations at the system and micro-architecture levels are expected. For instance, it is likely that the Translation Look-aside Buffer (TLB) and caches will be polluted with data and code coming from the controlling thread. The interface allows the monitored thread to keep on running during the notification. The motivation behind this is to keep the caches and TLB somewhat warm. Of course, on single processor systems, the perturbation is likely to be greater than on SMP systems where both threads do not necessarily run on the same processor. In any case, once monitoring resumes, it is likely that the monitored thread will not immediately return to its normal execution pattern. Caches and TLB will need to be reloaded. We call this execution period after monitoring is resumed the *recovery period*. During this period, and depending on the types of events being measured, it may be important not to capture samples because they would not necessarily reflect the true behavior of the thread. By using the long and short reset periods smartly, it is possible to account for the recovery period by simply choosing a long period that is larger than the short period. The adjustment is highly dependent on the processor architecture, the operating system, the event used for the sampling period and the overall system (SMP vs. UP). There is no magic settings and experiments

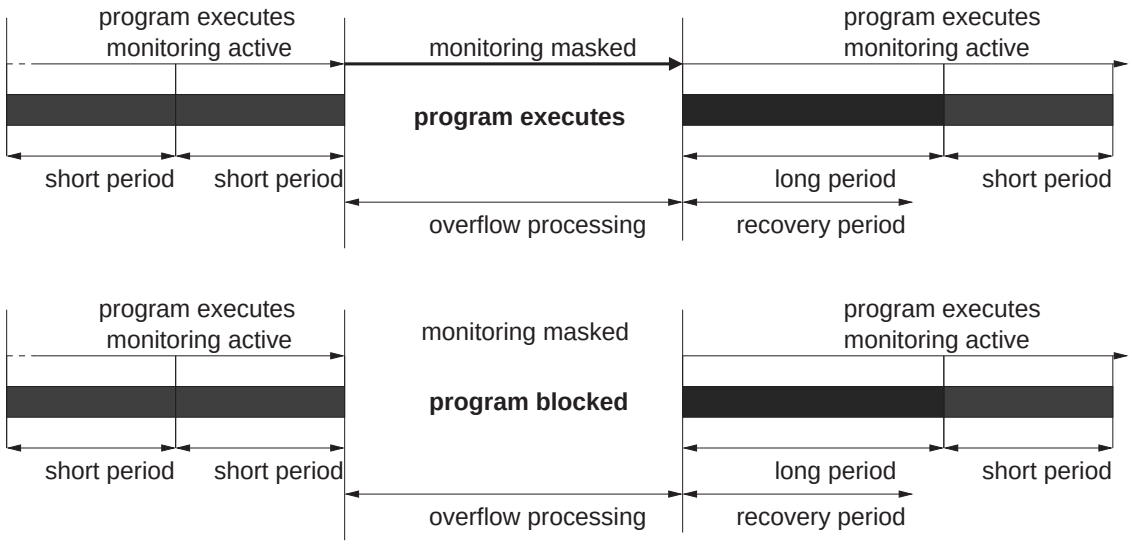


Figure 3.19: long vs. short sampling period.

are required, however a mechanism is offered. In cases where the perturbations can be ignored, a tool can revert to a single sampling period by setting the short and long periods to the same value.

The interface does not impose any minimum nor maximum value for any sampling periods. It is always up to the user to determine the right combination.

Figure 3.19 shows how the long and short period are used with the default sampling format. As long as the sampling buffer is not full, the short period is used to reset the overflowed PMD register. When the buffer becomes full, monitoring is masked and a notification is sent to the controlling thread. In the example shown at the top of the figure, the monitored thread runs while the notification is being processed, whereas in the bottom example, it is blocked. In either case, when monitoring is resumed, via PFM_RESTART, the long period is used once to skip over the recovery period, then the short period is used.

### 3.4.2 Randomization of sampling periods

The interface allows randomization for each sampling period. Why is that important? Event-based sampling is a statistical approach based on the idea that by taking only a few samples, one can determine the overall behavior of an application. Sampling *is not* tracing: not all instructions are captured. To be meaningful, sampling must be done such that:

- enough samples are collected to be representative of the execution (quantity factor)
- samples are taken at different moments/locations in the execution (quality factor)

For both factors, there is no magic answer. It all depends on the type of measurement and the workload. In general, programs tend to spend most of their time executing their main-loop. Samples are collected only at certain points during the execution, is it important to ensure that those points are representative of the execution. Using a fixed sampling period may easily lead to biased results. We give on example in figure 3.20, where the program enters a loop. The sampling event is E. The top of the figure shows what happens without randomization. We have exaggerated the sampling period by

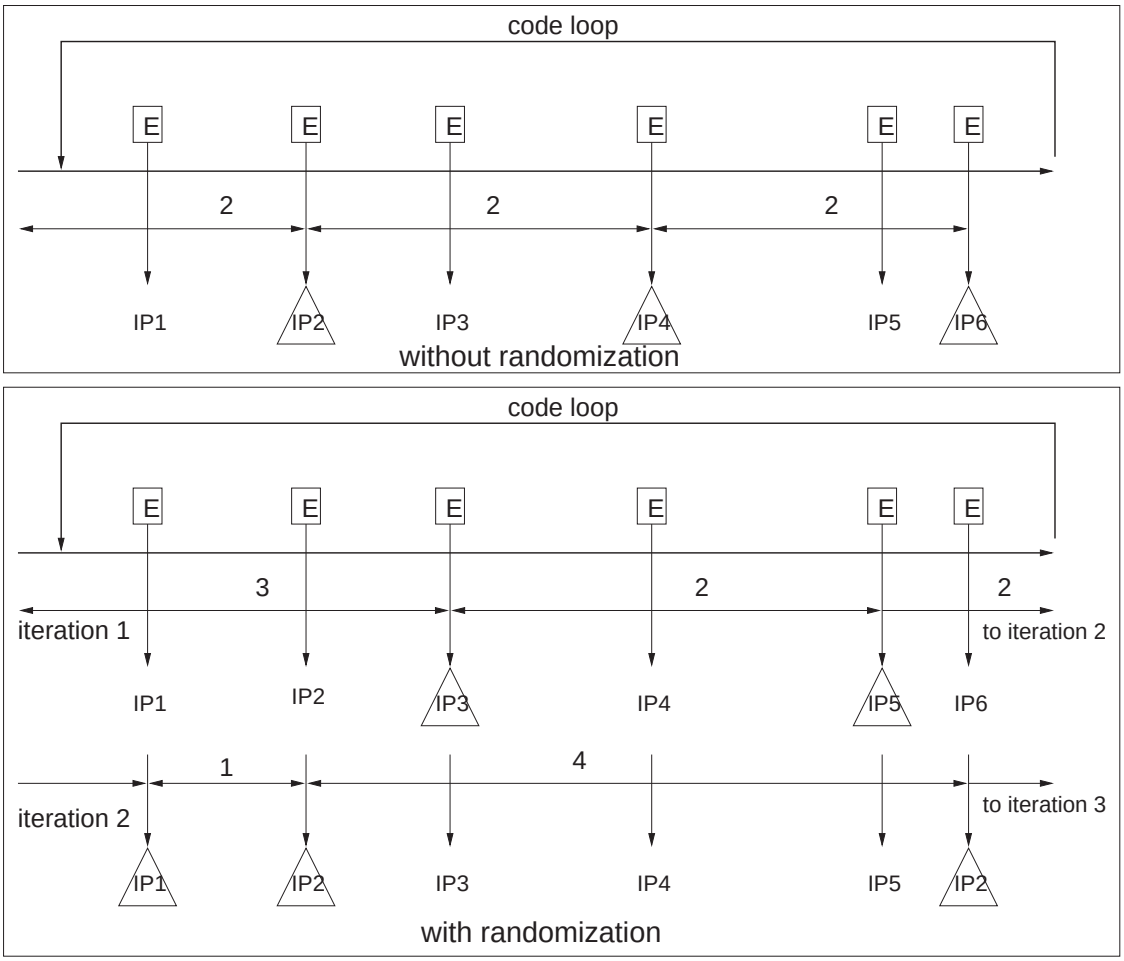


Figure 3.20: biased results without randomization.

setting it to 2: every two E events, we take a sample. Each sample includes the instruction pointer IPx. The loop actually contains six occurrences of event E. Supposing we enter the loop with no observed events. we collect samples at IP2, IP4, and IP6 at the first iteration. Then execution loops back and we capture exactly the same sequence of samples. The bias is easily visible here because none of the E events occurring at IP1, IP3, nor IP5 show up. The reason is that the sampling period is in lockstep with the number of events in the loop. The period is 2 and we have 6 occurrences in the loop and they perfectly align. Although this is an artificial example, this kind of behavior can be observed very frequently, especially when the sampling event is one that happens very frequently in programs such as a branch-related event. To avoid the problem, the sampling period must change to ensure that all events get a chance to be captured. The bottom of the figure shows what happens for the first two iterations when we randomize the period and allow variations from 1 to 4. The first iteration captures IP3 and IP5. But the second captures IP1, IP2, and IP6. In two iterations we get a more accurate view of what is going in the loop than what will be captured without randomization.

### Characteristics of the pseudo-random number generator

Randomization is obtained by using a pseudo-random number generator. Such generator is widely available on all platforms. In fact, the standard C library provides the *rand()* and *random()* interfaces. The generator does not need to be fancy, it does not even need to be generating 64-bit pseudo-random values, 32-bit values are enough because you never want to vary the period by very much. A simple generator such as the one describe in [1] is sufficient.

The series of numbers must be completely determined by the seed value. The interface guarantees that if the same seed is re-used, then the same series of numbers is produced, assuming the implementation of the generator has not changed. This is important to be able to reproduce the same *pseudo-random* sampling intervals, should this be necessary.

### How to setup randomization

Randomization can be achieved without kernel support when sampling is done at the user level. The tool gets a notification that a counter has overflowed. Next, It generates a pseudo-random number using an interface such as the *random()* function defined in the standard C library. Then, it invokes PFM_WRITE_PMDS on the overflowed PMD register and passes the new value in reg_long_reset. Finally it invokes the PFM_RESTART command which resets the overflowed PMD using the long reset value.

For efficiency reason as we shall see shortly, the interface supports sampling at the kernel level. As such, it includes support for randomization. Each implementation must include a kernel-level pseudo-random number generator. Randomization can be setup per counting PMD register. Randomization is ignored on non counting PMD registers. Randomization is activated when the PFM_REGFL_RANDOM flag is used on the controlling PMC register. A per PMD register seed value can be passed as well. The seed, in reg_random_seed field, fully determines the series of pseudo-random numbers. A bitmask, in reg_random_mask field, is also be passed to limit the range of variation on the pseudo-random value.

### How does randomization work?

Randomization is applied each time a PMD register is reset. This can happen as a consequence of a PFM_RESTART, i.e., during long reset. When a kernel level sampling buffer is used, and depending on the sampling format, a PMD register can be reset immediately after overflowing, i.e., during a short reset.

The seed and mask apply to both the short and long reset. The new value is determined as follows:

$$\text{new\_value} = \text{base} + (\text{random\_value} \ \& \ \text{mask})$$

Where base is the value of either `reg_short_reset` or `reg_long_reset` depending on the type of reset. We give an example in figure 3.4. The new value, called reset value, is saved and can be retrieved using the PFM_READ_PMDS command. The value appears in the `reg_last_reset` field. Depending on when the read command is issued it may be different from the current value of the PMD register.

### 3.4.3 Counter Overflow notifications

When a counter overflows, it is possible to receive a notification. The notification is never automatic and must be requested by specifying the PFM_REGFL_OVFL_NOTIFY flag on the controlling PMC register. When the flag is not set, the counter simply wraps around on overflow. Therefore it will not overflow again until it reaches $2^{64}$, which is a very very long time!

There is one notification per overflow event. In case of simultaneous overflows by multiple counters, only one notification is generated.

The overflow notification uses the message interface described in section 3.3.

#### The overflow message

The overflow message is defined as follows:

```
typedef struct {
        uint32_t    msg_type;
        uint16_t    msg_active_set;
        uint64_t    msg_ovfl_pmds[PFM_MAX_PMD_BITVECTOR];
} pfm_ovfl_msg_t;
```

The `pfm_ovfl_msg_t` structure fields are used as follows:

- `msg_type` : set to PFM_MSG_OVFL
- `msg_active_set` : the identifier for the event set that was active at the time of the overflow. Given that only one set can be active at a time.
- `msg_ovfl_pmds` : a bitvector indicating which PMD registers overflowed.

The bitvector contains all the information needed to figure out which counters overflowed. More than one bit can be set in case of simultaneous overflows by multiple PMD registers.

#### Effect of the notification

When no sampling format is used, monitoring is *masked* after an overflow. This means that no qualified event is collected. Monitoring remains *masked* even after the overflow message has been read. To resume monitoring, it is necessary to invoke the PFM_RESTART command. When a sampling format is used, the overflow notification behavior may be different. Refer to the specific documentation for sampling format specific behavior on notifications. For the default sampling format, refer to section 3.6.5.

**Current issue**

*The interface assumes that the tool is well behaved and does a at least one read() for each* PFM_RESTART. *Otherwise the queue fills up. One solution would be to tag the restart to associate it when an overflow message. The restart always refers to the topmost overflow message in the queue.*

**Suppressing the overflow notification message**

For some cases of self-monitoring contexts, it may be enough to receive an asynchronous notification via a signal and the overflow message is useless. This is the case when only one counter can overflow, for instance. Because messages are not discarded automatically on PFM_RESTART, they can fill up the queue and cause problem when not properly drained. To minimize the overhead involved with reading the message just to discard it, the interface provides the PFM_OVFL_NO_MSG flag which can be set as a context flag. When this flag is set, no overflow notification message is sent. To be notified the controlling thread must request asynchronous notification via a SIGIO signal. The thread, then, gets the signal but no message is placed in the message queue, therefore no *read()* system call is needed.

**Restarting monitoring after a notification**

When no sampling format is used, monitoring is *masked* on overflow notification. For a non self-monitoring context with the PFM_FL_NOTIFY_BLOCK flag set, the monitored thread is blocked as a result of the overflow. To resume monitoring, it is necessary to call the PFM_RESTART command. When a monitored thread was blocked, it is unblocked by the call. When a sampling format is used, the behavior depends on the format.

This behavior applies to both per-thread and system-wide contexts. For the latter type of context, the PFM_FL_NOTIFY_BLOCK flag is not supported therefore no thread is ever blocked as a consequence of a notification.

## 3.5 Support for kernel level sampling formats

Using the overflow notification mechanism, it is possible to implement sampling completely at the application level. On notification, a tool would simply read some PMD registers, save their values into a memory buffer and restart monitoring. This works well but there can be some important overhead especially when the sampling periods are small, i.e. with a high frequency. For the most part, the overhead is generated by the notification to the tool which involves at least two context switches for a non self-monitoring thread. Depending on the type of events measured, the overhead may cause serious problems on the final results.

In order to reduce the overhead, the interface supports kernel level sampling buffers. Instead of calling the application for each overflow, the kernel records the information into a kernel level buffer and only calls the tool when the buffer is full. In other words, the buffer acts as an *overflow cache* by recording samples on behalf of the tool. When the buffer is full, the overflow notification is sent to the tool which then processes the buffer. Eventually the tool invokes the PFM_RESTART to resume monitoring and reset the buffer. The idea is to *amortize* the cost of calling the tool by notifying only when many samples are available. To export the buffer in a efficient manner, the interface allows the application to re-map it into its the user-level address. This way, accessing the samples from the tool is very efficient and does not involve large data copying between the kernel and the tool.

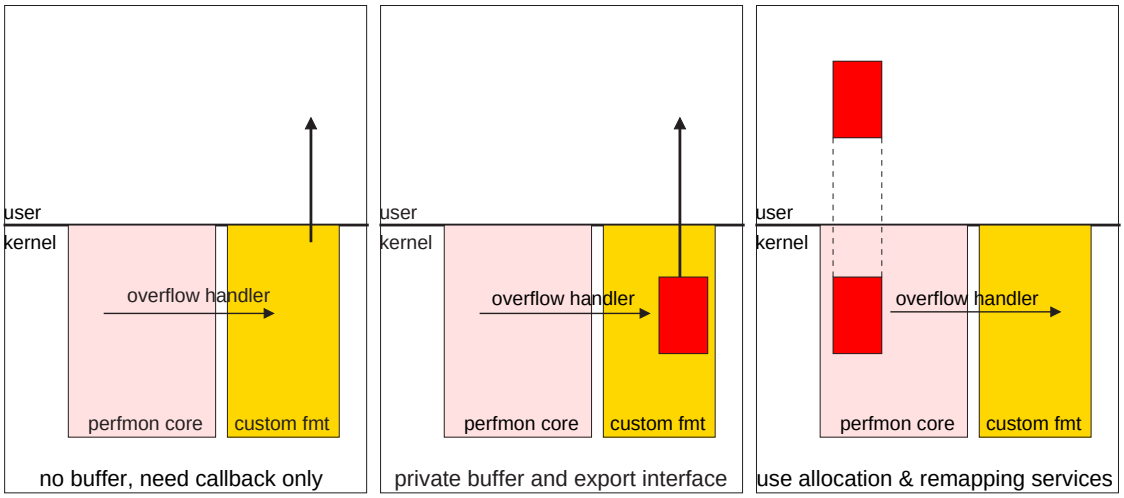


Figure 3.21: possible scenarios for a sampling format.

The difficulty with the kernel level sampling buffer is that the kernel records information on behalf of the tool. But there can be a variety of tools, not all tools need the same information and certainly not always in the same format. For instance, some tools want to keep the sequential ordering of the collected samples, others want to aggregate identical samples. Some tools want to record information that is not coming from the PMU itself, such as the amount of free memory or the number of active processes, the current thread identifier and so on.

In the introduction, we insisted on the fact that the interface must be flexible to support a variety of monitoring tools, yet it must be efficient. Using a kernel level sampling buffer is a way of making the interface efficient, the problem is to leave enough flexibility for tools to describe what they want to record in each sample. As we have seen in the description of the commands, it is possible to specify the set of PMD registers to record when a counter overflows. There is also a way to associate an overflow with a unique event identifier (see `reg_smpl_eventid`). Those are just basic mechanisms to pass information to the function in charge of implementing the policy, i.e. what to record and how. It is very unlikely that we can come up with a *universal* interface that would suit all needs without being prohibitively complicated. Moreover we want to ensure that existing monitoring tools can be ported without too much effort.

The solution we chose is to decouple the policy from the basic mechanisms by using *custom sampling buffer formats*. A format is responsible for recording the information needed on counter overflow. It controls what is recorded and how it is recorded. A format may be implemented using a dynamically loadable kernel module (DKLM). Such facility is available on most modern operating systems. The format, then, hooks up with the core perfmon subsystem. At context creation, a tool indicates which format it needs. The tool has implicit knowledge of what the particular format does.

Each format contains a set of call-backs which are invoked by the perfmon core on specific events. In particular, there is a mandatory call-back handler for counter overflow. The handler can record whatever information it needs into whatever format it wants. Conversely there is also a kernel-level perfmon interface which formats can call when they need to access perfmon core information, such as the current 64-bit value of a PMD register.

The buffer allocation and re-mapping services are not part of formats, they are part of the perfmon core. A format may choose to use the services. A format may choose to export the information it records using another dedicated interface, such as a device driver-style interface.

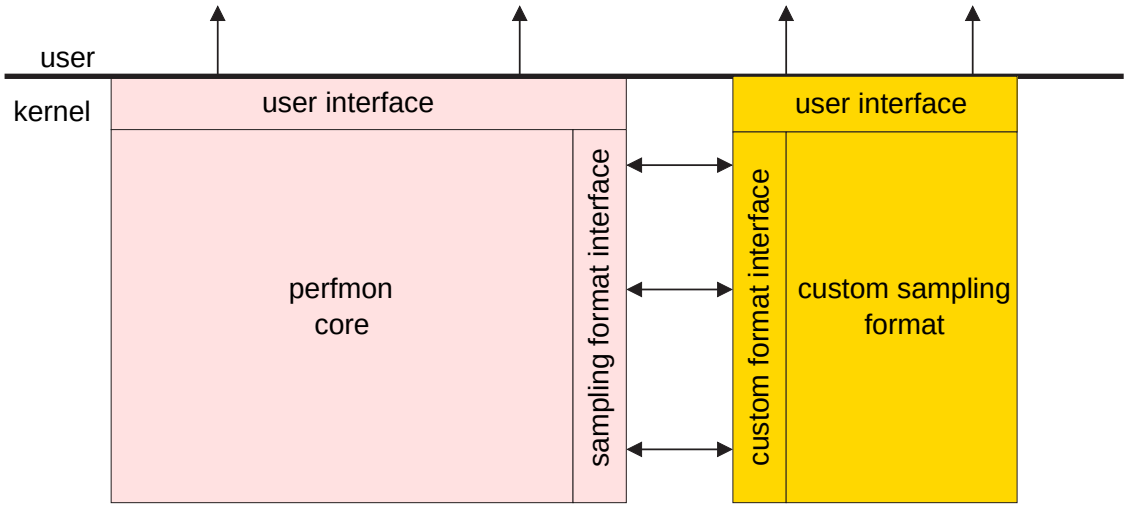


Figure 3.22: custom sampling format interfaces.

There are several scenarios possible and they are depicted in figure 3.21. On the left, this is a case where the format does not use a buffer. It may simply be counting certain things. The information is exported via a private interface which may be following a device driver model, for instance. The scenario in the middle depicts a format which allocates its own buffer and exports it via a private interface. This could be used to port existing sampling drivers such as OProfile for instance. On the right, we have a format which uses the buffer allocation and re-mapping services of the perfmon core.

The interface includes a default sampling format to provide a basic sampling support. The format can also be used as an example to develop other formats. The default format is guaranteed to be present on all implementations. We present its characteristics in section 3.6.

### 3.5.1 Custom sampling format interfaces

There are three distinct perfmon interfaces to deal with custom sampling formats:

- the *user level interface* describes how a user-level tool indicates the format it needs, how to pass parameters to a format
- the *kernel level interface* describes how a format hooks up with the perfmon core
- the *format-specific user interface* describes how the format interacts with regards to overflow notifications, restarts, how the format exports the collected information

The three interfaces are depicted in figure 3.22. In the following sections, we describe each interface.

### 3.5.2 Identification of sampling formats

A tool needs to identify each format in a unique fashion, yet we want to encourage new formats to be written without having to go through a centralized registration service which would allocate some unique identifier to each one. Instead, we use 128-bit Universal Unique Identifier (UUID) for each format. This naming scheme is similar to the GUID identifiers used by the Extensible Firmware

Interface (EFI). A UUID can easily be generated using a tool such as `uuidgen` which does not require any centralized authority. This tool generates a UUID from a shell as follows:

```
$ uuidgen
24aab4dd-2144-4c1f-80c0-62c2e17d3744
```

To avoid byte-ordering problems, the interface uses the following type definition for a UUID:

```
typedef unsigned char pfm_uuid_t[16];
```

The UUID generated by `uuidgen` can easily be mapped onto this data type. For our example, we would build the UUID as follows:

```
#define TEST_UUID { 0x24, 0xaa, 0xb4, 0xdd,
                    0x21, 0x44, 0x4c, 0x1f,
                    0x90, 0xc0, 0x62, 0xc2,
                    0xe1, 0x7d, 0x37, 0x44 }
```

To use a format, a tool must provide the UUID of the format when the context is created. The UUID must be stored in the `ctx_smpl_uuid` field. It is expected that such UUID is available in the header file describing the format which is necessarily included by the tool.

Each implementation must provide a way for a user level application to figure out the list of available formats. The access should not require any special privileges, i.e., should not require to be `root`. For instance, on Linux, this could be implemented via a `/proc` interface or `/sys` interface. For each format, the UUID and logical name must be listed.

### 3.5.3 Passing arguments to a sampling format

There are two ways to pass parameters to a format:

- parameters affecting all contexts can be passed when the module is inserted into the kernel ,i.e., command line arguments. On Linux, this can be done with the `insmod` command.
- parameters affecting a particular context must be passed when the context is created.

The global parameters are unlikely to be widely used and the way values are passed is highly dependent on the implementation. We do not recommend using this type of parameters except for debug purposes.

The context-specific argument can be used to set parameters specific to a measurement. For instance, if the format uses the perfmon buffer allocation service, it may let the user specify the size of the buffer. The size would then be a parameter available to the tool when the context is created.

Because parameters can vary widely in numbers and nature, it is not possible for the interface to provide generic parameters. Module specific parameters can be passed on a per context basis with the PFM_CONTEXT_CREATE command. The parameters are expected to be encapsulated into a data structure which is appended to the `pfarg_ctx_t` structure. There is no need for a specific pointer to the parameters in that structure. If a format uses parameters then they must be provided during the call. For instance, if a format uses the following parameters:

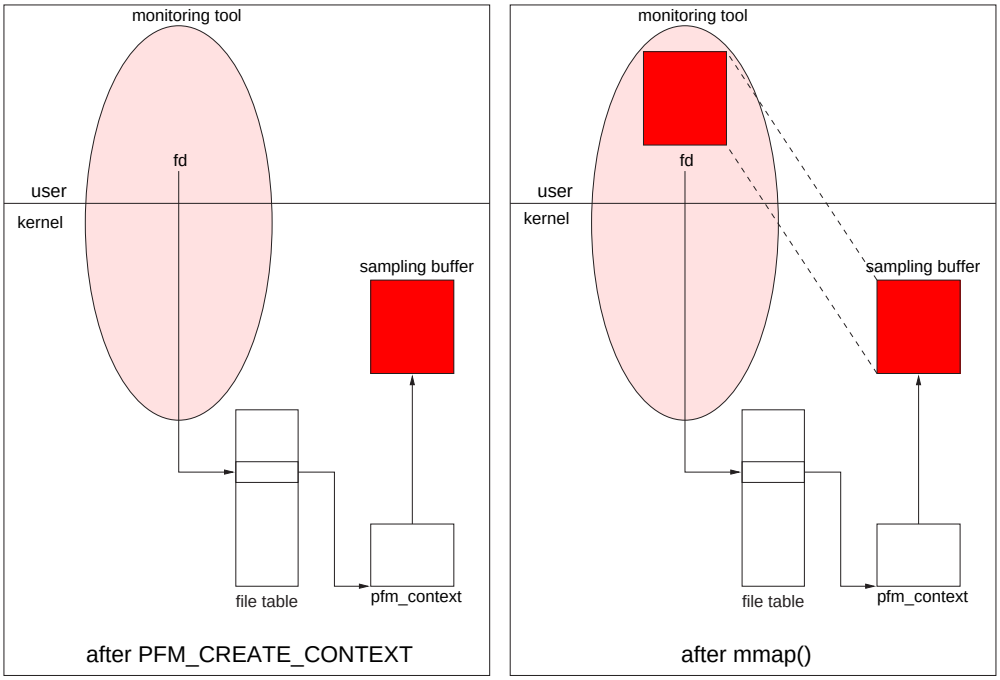


Figure 3.23: mapping the sampling buffer with *mmap()*.

```
typedef struct {
        size_t buffer_size;
} format_param_t;
```

Then the following data structure must be passed to PFM_CONTEXT_CREATE:

```
typedef struct {
        pfarg_ctx_t     ctx;
        format_param_t  param;
} my_context_arg_t;
```

Each implementation must ensure proper alignment for the `pfarg_ctx_t` structure such that the first byte of the format specific structure immediately follows, i.e., no padding is necessary. As such, it is advised that the structure be defined such that its size be at least a multiple of 8 bytes. While this scheme is not necessarily ideal, we believe it makes it easier on the implementation compared to having an opaque pointer in the `pfarg_ctx_t` structure.

The size of the format-specific structure is checked against the argument size limit set which is indicated in the `cf_arg_size_max` of the PFM_SET_CONFIG command. See section 3.1.16 for more details.

### 3.5.4 Accessing the sampling buffer

When the buffer allocation and re mapping services are used by the format, the sampling buffer is not directly accessible upon return from the PFM_CONTEXT_CREATE command. An explicit call to *mmap()* is necessary to re-map the buffer into the user level address space of the controlling process. Without the *mmap()* call the buffer exists and samples are stored into it but they remain totally unaccessible

from the user level. As seen in section 3.3 the *read()* system call is used to extract event notifications and cannot be used to access the sampling buffer.

### Re-mapping the sampling buffer with *mmap()*

An application can access the sampling buffer by explicitly mapping it into its user level address space using the *mmap()* system call. A typical code sequence to re-map the buffer would be as follows:

```
pfarg_ctx_t ctx;
int fd, ret;
size_t size;
...
ret = perfmonctl(0, PFM_CREATE_CONTEXT, &ctx, 1);
if (ret) exit(1);
fd = ctx.ctx_fd;
size = ctx.ct_smpl_buf_size;
vaddr = mmap(NULL, size, PROT_READ, MAP_SHARED, fd, 0);
if (vaddr == (void *)-1) exit(1);
```

Because of the nature of the sampling buffer, special constraints are imposed on the *mmap()* call. the following constraints exist:

- the buffer is always exported as read-only. As such only PROT_READ is valid. Any write access must be flagged by the virtual memory subsystem and may result in the termination of the controlling process.
- the buffer has a fixed size, as such the size of the mapping is predetermined and must be as reported in the `ctx_smpl_buf_size` field returned by the PFM_CREATE_CONTEXT command.
- the buffer can only be mapped once inside a process.
- there cannot be a non zero offset.
- not all flags and protections combinations are supported. In particular the following are not supported:
  - MAP_WRITE, MAP_PRIVATE, MAP_EXECUTABLE, MAP_LOCKED, MAP_GROWSDOWN
  - PROT_EXEC, PROT_WRITE, PROT_NONE

The effect of the *mmap()* call in shown in figure 3.23.

When the allocation and re-mapping services are not used, either there is no access to the buffer which is stupid or most likely the access is specific to the format. Please refer to the proper format document for further details.

In the case of the default sampling format, see section 3.6, the allocation and re-mapping services are used.

### the sampling buffer

It is possible to use *munmap()* to remove the buffer virtual mapping in the user level address space of a controlling process. The call unmaps all or parts of the buffer but the physical buffer remains intact, i.e., the physical mapping is never affected by this call. This is shown in figure 3.24. The physical buffer, without a virtual mapping, survives as long as the context exists.

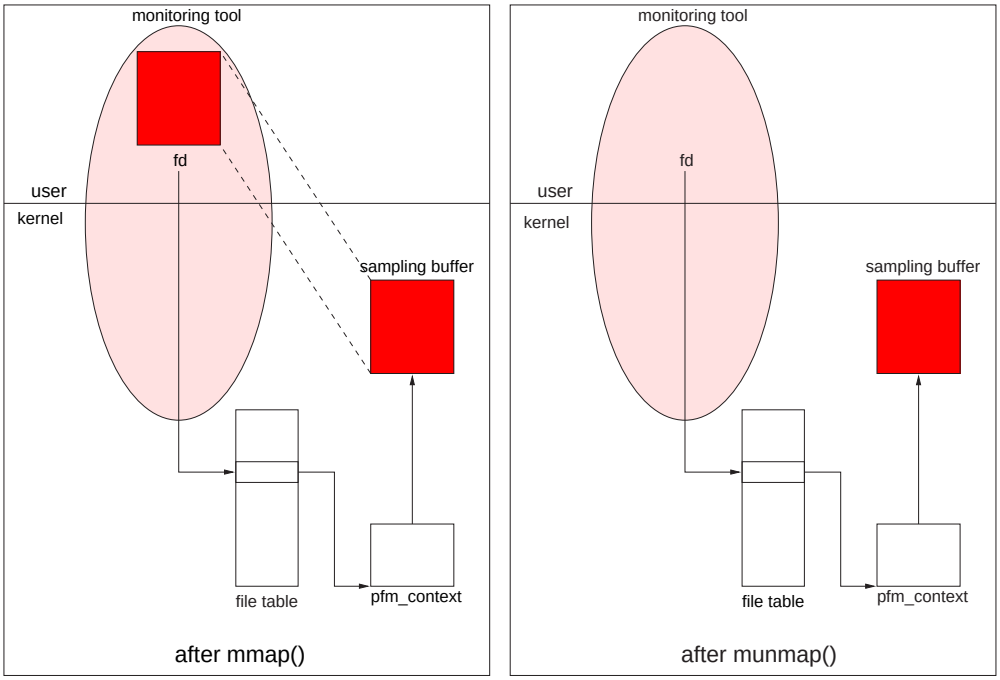


Figure 3.24: effect of *munmap()* on the sampling buffer.

### Behavior on *fork()*

The re-mapping is inherited in child processes via *fork()*. Figure 3.25 shows what happens after a *fork()* when the parent has a buffer re-mapped. The buffer is exported read-only, therefore this is no risk of copy-on-write. The physical buffer is fully shared between the two processes. This can be used by the parent process to delegate the processing of the buffer to the child process, for instance.

### Behavior on *pthread_create()*

By definition, all threads inside a process shared the same address space, therefore they can all access the buffer at the same address. The physical buffer is obviously identical.

### Behavior on *exec()*

By definition of *exec()*, the virtual address space is completely reloaded, therefore the virtual mapping of the buffer is removed but the physical buffer remains but is unaccessible. The context and associated resources are destroyed on *exec()* if the file descriptor is setup to close on *exec()*, via the *fcntl()* system call.

### Behavior on *close()*

When a controlling thread invokes the *close()* system call, the mapping of the buffer is removed from the address space of the controlling process, no matter how many threads exist. This implies that

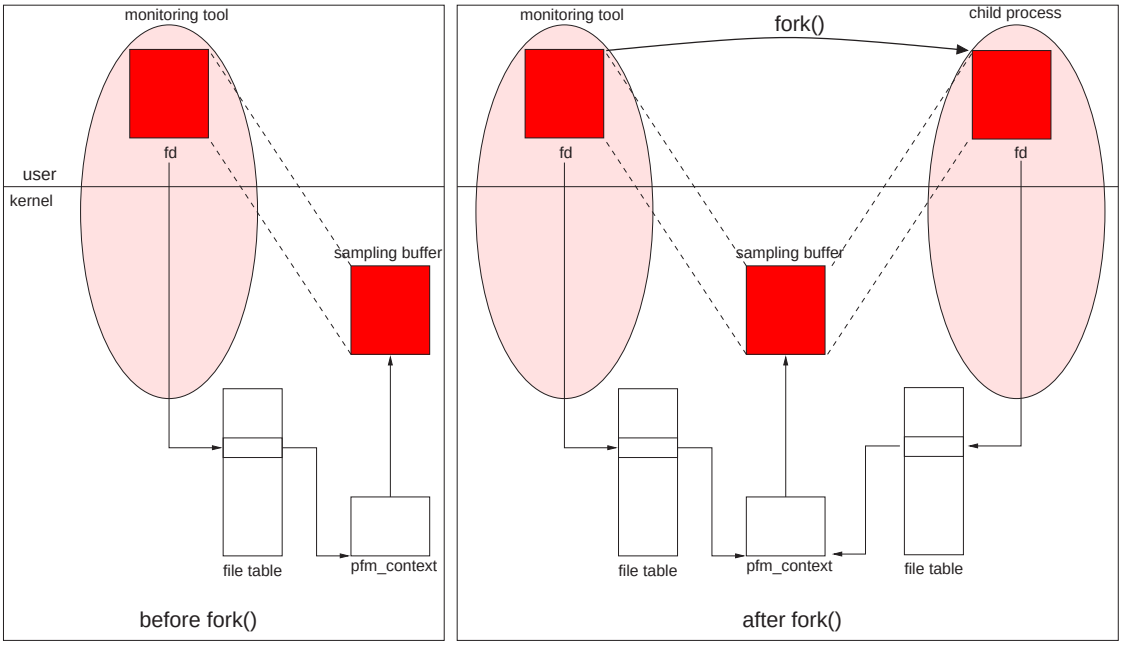


Figure 3.25: buffer re-mapping and *fork()*.

some coordination between threads is necessary in order to avoid race conditions when more than one thread is accessing the buffer.

The physical buffer remains until the last controlling thread closes the file descriptor. Note that controlling threads may be scattered between several processes in case of *fork()*.

### 3.5.5 Buffer initialization

Independently of the re-mapping, the buffer is guaranteed to be initialized upon return from the PFM_CONTEXT_CREATE command. If the format does not have an explicit initialization call-back, the buffer is guaranteed to be all zeroes.

### 3.5.6 Buffer content

The content of the buffer is never interpreted by the core perfmon subsystem. The first byte of the buffer pointed to by the address returned by *mmap()* contains format specific information, usually a buffer header.

Depending on the format, the content of the buffer may be constantly changing. The interface does not provide any guarantee with regards to getting a consistent view of the buffer. It is necessary to refer to the format specific documentation to understand when it safe to look at the content of the buffer.

Each implementation must ensure that the memory region allocated for the buffer is aligned to, at least, an 16-byte boundary. Mostly likely, it will be aligned on the smallest page size that the system supports.

### 3.5.7 Kernel level interface overview

This interface is divided into two parts:

- perfmon core interface : what the perfmon core provides to the formats
- format kernel interface: what the formats must provide to the perfmon core, i.e., a list of call-backs

When a format is added, via a kernel module or equivalent, it must first register with the perfmon core before it can be used by any context. The interface provides two registration-related functions:

```
int pfm_register_buffer_fmt(pfm_buffer_fmt_t *fmt);
int pfm_unregister_buffer_fmt(pfm_uuid_t uuid);
```

During registration the format must provide a data structure describing its entry points and its UUID.

This data structure is defined as follows:

```
typedef struct {
        char              *fmt_name;
        pfm_uuid_t        fmt_uuid;
        size_t            fmt_arg_size;
        size_t            fmt_msgq_depth;
        uint32_t          fmt_flags;

        fmt_validate_t    fmt_validate;
        fmt_getsize_t     fmt_getsize;
        fmt_init_t        fmt_init;
        fmt_handler_t     fmt_handler;
        fmt_restart_t     fmt_restart;
        fmt_exit_t        fmt_exit;
} pfm_buffer_fmt_t;
```

All but the first five fields of the structure are pointers to call-back functions for the perfmon core. Before we detail each one of them, we give an overview of the entire structure:

- `fmt_name` : the name of the format. This field is required.
- `fmt_uuid` : the UUID for the format. This field is required.
- `fmt_arg_size` : the size in bytes of the format specific parameters passed during the creation of the context. When no parameters are needed, this field must be zero.
- `fmt_msgq_depth` : the maximum number of entries that the format needs in the notification message queue. The minimal depth is `1`. If the value in this field is less than `1`, it will be forced to `1`. The depth is used to determine the size of the queue given the message size.
- `fmt_flags` : a set of flags to describe the capabilities of the format. The bits which are not defined are *reserved* and must be cleared. It is only possible to set bits that are defined. The following flags are defined:
  - PFM_FMTFL_NOSET : the format does not support multiple event sets. Any attempt to create an event set, different from *set0* returns an error.

- `fmt_validate` : this function is called during the PFM_CONTEXT_CREATE command to validate the format-specific parameters and check the characteristics of the context.
- `fmt_getsize` : this function is called during the creation of the context to retrieve the size of the buffer.
- `fmt_init` : this function is called during the creation of the context to initialize the buffer.
- `fmt_handler`: this function is called whenever a counter overflows. This function is required.
- `fmt_restart` : this function is called during the PFM_RESTART command when monitoring was masked following an overflow notification.
- `fmt_exit` : this function is called when the physical buffer is actually freed.

### 3.5.8 The `pfm_register_buffer_fmt()` function

The function is invoked to register a new format. The definition of the function is as follows:

```
int pfm_register_buffer_fmt(pfm_buffer_fmt_t *fmt);
```

The description of the format is passed in the pointer argument `fmt`, a pointer to a structure of type `pfm_buffer_fmt_t`. A format can only be registered once. Upon successful return from this function, the format is usable by applications. Any format-specific global initialization must be done prior to calling this function.

The function checks that the required fields are provided.

When the registration is successful, the function returns 0, otherwise a valid `errno` value is returned. The possible values are:

- EINVAL : some of the required fields in the `pfm_buffer_fmt_t` structure are not provided
- EBUSY : a format with the same UUID is already registered.

### 3.5.9 The `pfm_unregister_buffer_fmt()` function

The function is invoked to remove a format. The definition of the function is as follows:

```
int pfm_unregister_buffer_fmt(pfm_uuid_t uuid);
```

The format is identified by its UUID in the `uuid` argument.

When the removal is successful, the function returns 0, otherwise a valid `errno` value is returned. The possible values are:

- EINVAL : the format is not registered

### 3.5.10 The fmt_validate() function

This function is invoked during the PFM_CREATE_CONTEXT command to validate the format specific parameters and verify that the context is compatible with the format. The function is defined as follows:

```
typedef int (*fmt_validate_t)( uint32_t ctx_flags,
                               uint16_t num_pmds,
                               void *arg);
```

The parameters are defined as follows:

- ctx_flags : the flags as passed in the ctx_flags during the PFM_CREATE_CONTEXT command. The flags may be needed to verify if they are compatible with format. For instance, some formats may only work with a system-wide context.
- num_pmds: the number of PMD registers implemented by the host PMU. This could be useful to determine the minimal buffer size, for instance.
- arg : a pointer to the format-specific parameters passed along with the pfarg_ctx_t structure.

This function is optional. If the format has no specific parameters or if no validation is necessary then fmt_validate field must be NULL.

The function must return 0 when successful, otherwise a valid errno value is expected.

### 3.5.11 The fmt_getsize() function

This function is invoked during the PFM_CREATE_CONTEXT command to retrieve the size of the buffer necessary for the context. The function is defined as follows:

```
typedef int (*fmt_getsize_t)( uint32_t ctx_flags,
                              void *arg,
                              size_t *size);
```

The parameters are defined as follows:

- ctx_flags : the flags for the context as passed in the ctx_flags field of the pfarg_ctx_t structure. They may be needed to compute the size.
- arg : a pointer to the format-specific parameters passed along with the pfarg_ctx_t structure. The structure may contain parameters necessary to compute the size.
- size : a pointer to where the size in bytes must be stored

This function is optional. If the format does not use the perfmon buffer allocation service, then fmt_getsize field must be NULL. If the returned size is 0, this is equivalent to not having using the buffer allocation and re-mapping services.

The size is checked against the sampling buffer global size limit as set in the cf_smpl_buf_size_max field passed to the PFM_SET_CONFIG command. If the size is too big, the PFM_CREATE_CONTEXT command fails.

The function must return 0 when successful, otherwise a valid errno value is expected.

### 3.5.12 The fmt_init() function

This function is invoked during the PFM_CREATE_CONTEXT command to initialize the buffer. The function is defined as follows:

```
typedef int (*fmt_init_t)( uint32_t ctx_flags,
                           void *arg,
                           void *buf);
```

The parameters are defined as follows:

- ctx_flags : the value of the flags as passed in the ctx_flags during the PFM_CREATE_CONTEXT command. They may be needed to initialize the buffer.
- arg : a pointer to the format-specific parameters passed along with the pfarg_ctx_t structure. They may be needed to initialize the buffer.
- buf : a pointer to the memory region allocated for the buffer.

This function is optional. If the format does not need per-context initialization, then fmt_init field must be NULL.

If the format does not use the buffer allocation service, it can still have an initialization call-back. In that case the buf argument is NULL.

If the format uses the buffer allocation service, but does not provide the *fmt_init()* call-back then the interface guarantees that the buffer will be set to all zeroes.

The function must return 0 when successful, otherwise a valid errno value is expected.

### 3.5.13 The fmt_exit() function

This function is invoked when the context is destroyed, i.e., when the last controlling thread close the file descriptor. The function is defined as follows:

```
typedef int (*fmt_exit_t)(void *buf);
```

The parameters are defined as follows:

- buf : a pointer to the memory region allocated for the buffer

This function is called just before the physical memory used by the buffer is freed. This is the last chance to cleanup.The function must free whatever resources it had allocated in association with the context.

This function is optional. If the format does not need a per-context exit call-back, then the fmt_exit field must be NULL. If the format does not use the buffer allocation service, it can still have an exit call-back. In that case the buf argument is NULL.

The function must return 0 when successful, otherwise a valid errno value is expected.

### 3.5.14 The `fmt_handler()` function

This function is invoked when a counter overflows. It is defined as follows:

```
typedef int (*fmt_handler_t)( void *buf,
                              pfm_ovfl_arg_t *arg,
                              uint64_t tstamp,
                              void *data);
```

The parameters are defined as follows:

- `buf` : a pointer to the memory region allocated for the buffer
- `arg` : a pointer to a structure of type `pfm_ovfl_arg_t` which describes the overflow and provides *control knobs* to the handler.
- `tstamp` : a constantly increasing time stamp guaranteed unique per processor
- `data` : a pointer to an implementation-specific data structure which may be needed by a handler. For instance, this could point to the interrupted machine state and/or the thread to which the overflow is attributed.

This function is required for all formats. Failure to provide a call-back results in an error during registration of the format.

The function does not bypass the normal PMU interrupt handler nor the 64-bit counter emulation layer. The sequence of calls is as follows:

1. some counters overflow
2. the PMU interrupt is posted
3. the PMU interrupt handler is invoked
4. the interrupt handler determines which counters overflowed and updates their software counterpart accordingly
5. if a 64-bit overflow is declared and a sampling format is used, the format's handler is invoked.

In case of simultaneous overflows by multiple counters, the function is called for each overflowed counter. The handler is expected to process one counter overflow at a time. The overflows are treated in increasing index of PMD register. Hence, if PMD4 and PMD5 overflow at the same time, the first call to the handler is for PMD4 and the second for PMD5. The time stamp is identical across all calls generated by the same overflow event, therefore it is possible to detect that calls are related.

A description of the overflow is passed in `arg` which is a pointer to a structure of type `pfm_ovfl_arg_t`. That structure is defined as follows:

```
typedef struct {
        uint16_t        ovfl_pmd;
        uint16_t        active_set;
        pfm_ovfl_ctrl_t ovfl_ctrl;
        uint64_t        pmd_last_reset;
        uint64_t        smpl_pmds_values[PFM_MAX_PMDS];
```

```
        uint64_t        pmd_value;
        uint64_t        pmd_eventid;
        uint64_t        smpl_pmds[PFM_MAX_PMD_BITVECTOR];
        uint16_t        ovfl_notify;
} pfm_ovfl_arg_t;
```

The fields are defined as follows for this command:

- `ovfl_pmd` : the index of the PMD register which overflowed
- `ovfl_notify` : is set to `1` when the PMC register controlling the overflowed PMD register has the PFM_REGFL_OVFL_NOTIFY flag set. This indicates that overflow notification was requested on this monitor. Otherwise, this field is set to 0.
- `active_set` : the active set at the moment of the overflow. This is the set to which the overflowed PMD register belongs.
- `pmd_last_reset` : the last 64-bit value loaded into the overflowed counter. When randomization is used, this field provides the last pseudo-random value that was used for the counter. Otherwise, this field contains either the `reg_short_reset` or `reg_long_reset` depending on how the previous overflow was processed.
- `smpl_pmds` : the bitvector of PMD registers of interest. This is the list of PMD registers the user specified in `reg_smpl_pmds` when programming the controlling PMC register. This bitvector indicates the list of PMD registers to include in a sample. It is up to the format to use this bitvector.
- `smpl_pmd_values` : the values of the PMD registers of interest. The values are stored contiguously in increasing PMD index. Only the PMD registers which have their bit set in `smpl_pmds` are included. This table contains meaningful values only when at least one bit is set in `smpl_pmds`. It is up to the format to use the values.
- `pmd_value` : the current 64-bit value of the overflowed PMD. Depending on the PMU, this value may not necessarily be 0.
- `pmd_eventid` : the opaque event identifier associated with the PMD register as passed in the `reg_smpl_eventid` field. It is up to the format to interpret the value of this field.
- `ovfl_ctrl` : a set of bits to control the behavior when returning from the handler.

Each format can use the information passed in the `pfm_ovfl_arg_t` structure as needed. There is absolutely no checking done by the perfmon core. It is possible that for some formats not all the information is necessary. In this case, it is expected that the tool set up the configuration of the PMC and PMD registers such that the overhead is minimized. For instance, if no PMD registers are of interest, then the `reg_smpl_pmds` field of the PMC register for the PFM_WRITE_PMCS command must be all zeroes to avoid storing any value in the `smpl_pmds_values` field.

The function can influence the behavior of the perfmon core upon return by using the control bits in the `ovfl_ctrl` structure. The structure is defined as follows:

```
typedef  struct {
        unsigned int  notify_user:1;
        unsigned int  reset_ovfl_pmds:1;
        unsigned int  block_thread:1;
        unsigned int  mask_monitoring:1;
} pfm_ovfl_ctrl_t;
```

The fields are defined as follows:

- notify_user : when set to 1, the perfmon core appends an overflow notification message to the message queue of the context upon return from the handler. The notification happens only if the controlling PMC register has the PFM_REGFL_OVFL_NOTIFY flag set. The default value is 0.
- reset_ovfl_pmds : when set to 1, the overflowed PMD register is reset. The PMD registers that were specified in the reg_reset_pmds bitvector of the controlling PMC register are also reset. For each PMD register to reset, the value specified in reg_short_reset during the PFM_WRITE_PMDS command is used. A new pseudo-random value is generated for each PMD register which requested this option. The reset happens right upon return from the handler. The default value for this field is 0.
- block_thread : when set to 1, the monitored thread is stopped. The bit is honored only for a non self-monitoring per-thread context with the PFM_FL_OVFL_BLOCK flag set, otherwise this field is ignored. The default value for this field is 0.
- mask_monitoring : when set to 1, monitoring is masked upon return from the handler. This means that no qualified event is collected until a PFM_RESTART command is executed. When set, this field cancels any reset_ovfl_pmds.

By setting any of the bits in the pfm_ovfl_ctrl_t structure, a format can decide whether PMD registers are reset, a notification is generated, or monitoring is masked. Using these control points, it is possible for the format to implement various policies. For instance, it is possible to construct a format that would use double-buffering, or one that would aggregate samples.

When monitoring is masked, no reset takes place even when the format sets the reset_ovfl_pmds field to 1. Masking has priority over reset. This is necessary to ensure that the controlling process sees a consistent view of all PMD registers should it try to read out the value of the registers with PFM_READ_PMDS. In that case, all resets will take place during the call to PFM_RESTART.

The controls are specified per overflowed counter yet when multiple counters overflow at the same time, the handler is called multiple times. For each call a distinct set of control bits is provided. In this case, the interface behaves as follows:

- notify_user : if the bit is set once and for a PMD register for which the controlling PMC register has the PFM_REGFL_OVFL_NOTIFY flag set, then a notification is sent.
- block_thread : if the bit is set once and the context is per-thread, non self-monitoring and has the PFM_FL_OVFL_BLOCK then the monitored thread is blocked.
- mask_monitoring : if the bit is set once, monitoring is masked.
- reset_ovfl_pmds : the bit is honored on a per overflowed PMD register basis. The overflowed PMD registers, for which this field is set, are not included in the set of PMD registers to reset during a PFM_RESTART command, they are instead reset right upon return from the handler using their short reset value.

When the block_thread bit is set and the context supports this feature, the interface guarantees that the monitored thread cannot go back to user level execution unless a PFM_RESTART command is issued before the thread reaches the location where it would have actually blocked.

When monitoring must be masked, it is masked upon return from the handler. This is true regardless of the thread going to block or not.

The function must return 0 when successful, otherwise a valid errno value is expected. The calls to the handler stop when the first error is reported. In case of simultaneous overflows by multiple counters, this function may be called less than the number of overflowed counters.

### 3.5.15 The `pfm_restart()` function

This function is invoked during the PFM_RESTART command. The function is defined as follows:

```
typedef int (*fmt_restart_t)( void *buf,
                              int is_active,
                              pfm_ovfl_ctrl_t *ctrl);
```

The arguments are defined as follows:

- `buf` : a pointer to the memory region allocated for the buffer.
- `is_active`: a boolean value. When set to 0, it means that monitoring is masked. When set to 1, it means that monitoring is active.
- `ctrl`: a pointer to a `pfm_ovfl_ctrl_t` structure which contains some control bits used by the perfmon core upon return from the call-back.

This function is invoked when monitoring is masked or when it is active, i.e., not masked. It is possible that a format never masks monitoring even though they are overflows. In that case, the restart call-back could be called when monitoring is still active. The value of the `is_active` can be used to tweak the behavior of the restart call-back based of the state of the context.

The control bits in the `pfm_ovfl_ctrl_t` structure have the following role for this function:

- `notify_user` : this field is ignored for this function
- `block_thread` : this field is ignored for this function
- `mask_monitoring` : when set to 1, monitoring is masked. The default value for this field is 0.
- `reset_ovfl_pmds` : when set to 1, all the PMD registers which last overflowed and which were not immediately reset upon return from `fmt_handler`, are reset. For each overflowed PMD register, the set of registers in the `reg_reset_pmds` bitvector for the controlling PMC registers are also reset. The value specified in `reg_long_reset` during the PFM_WRITE_PMD command is used. When randomization is used, new pseudo-random values are generated for the PMD registers. The reset happens right upon return from the call-back. The default value for this field is 0.

The function must return 0 when successful, otherwise a valid `errno` value is expected.

## 3.6 The default sampling format

The interface comes with a standard built-in sampling format on all implementations. The format is generic enough to work for both system-wide and per-thread context and it can handle all sorts of PMU configurations. The format fully supports multiple event sets per context.

The format uses the buffer allocation and re-mapping services. The samples are recorded sequentially in the buffer until it becomes full. The memory allocated for the buffer is managed logically as one buffer, i.e., there is no support for double-buffering. The buffer starts with a fixed size buffer header. Then, each sample consists of a fixed sized header and a variable size *body*. The layout is depicted in figure 3.26.

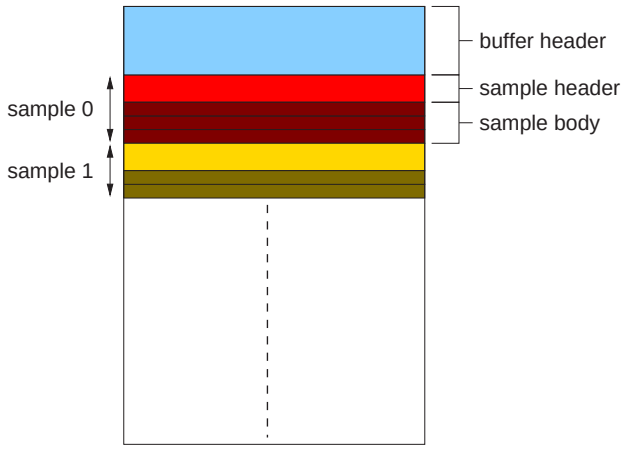


Figure 3.26: default format buffer layout.

### 3.6.1 Identification of the default format

The default format uses the following UUID on all implementations:

```
#define PFM_DEFAULT_SMPL_UUID {
            0x4d, 0x72, 0xbe, 0xc0,
            0x06, 0x64, 0x41, 0x43,
            0x82, 0xb4, 0xd3, 0xfd,
            0x27, 0x24, 0x3c, 0x97 }
```

### 3.6.2 Format specific parameters

The format requires parameters during the PFM_CREATE_CONTEXT call. The following data structure must be appended to the `pfarg_ctx_t` structure:

```
typedef struct {
         size_t         buf_size;
         uint32_t       buf_flags;
} pfm_default_smpl_arg_t;
```

The fields are defined as follows:

- `buf_size` : the size of the buffer in bytes.
- `buf_flags` : a set of flags to control the behavior of the format. The bits which are not defined are *reserved* and must be cleared. It is only possible to set bits that are defined. The following flags are defined:
  - PFM_DFL_SMPL_NO_HDR : samples do not have headers, only the body part is recorded. This is useful when the measurement is such that only one sampling period is used. In that case the size of the sample is fixed. With this flag a lot more samples can be packed into the buffer.

– PFM_DFL_SMPL_IIP_HDR: only record the instruction pointer in the header. The body part is not affected. This flag is useful when the measurement only uses the instruction pointer in which case the body is usually empty.

The buffer size represents the actual memory allocated for the buffer that includes the buffer header and the samples. Each format is responsible for checking that this size is large enough to hold any meaningful information.

When an invalid value is passed, the PFM_CREATE_CONTEXT command fails.

### 3.6.3 The buffer header

At the beginning of the buffer area, there is a buffer header which describes the status of the buffer. The buffer header is initialized when the context is created. The buffer header structure is defined as follows:

```
typedef struct {
        size_t          hdr_count;
        size_t          hdr_cur_offs;
        uint64_t        hdr_overflows;
        size_t          hdr_buf_size;
        uint32_t        hdr_version;
        uint32_t        hdr_flags;
} pfm_default_smpl_hdr_t;
```

The fields are defined as follows:

- hdr_count : the current number of samples in the buffer
- hdr_cur_offs : the current position in the buffer expressed as an offset from the beginning of the buffer (including buffer header)
- hdr_overflows : the number of times the buffer was full
- hdr_buf_size : the size in bytes of the buffer
- hdr_version : the version number of the buffer format
- hdr_flags : the flags passed to the format when the context is created

The hdr_count field reports the number of samples in the buffer since it was last reset. If monitoring is not masked, then this field can change at any time. It is only safe to look at it following an overflow notification. At that point, monitoring is masked and no more samples are recorded until a PFM_RESTART command is issued. In case the buffer is not full but monitoring has ended, this field can be used to determine how many samples are left in the buffer. This field is reset to zero during a PFM_RESTART command.

The hdr_cur_offs is used internally by the format. In order to locate samples, the hdr_count field should be used instead. The offset points to the next byte to write to, i.e., the first byte after the last sample. In case no sample is present, it points to the first byte after the buffer header.

The value of hdr_buf_size is the actual size of the memory chunk that was allocated. It is identical to the value that was passed in buf_size field in the format specific parameters.

The `hdr_overflows` field is a counter which represents the number of times the buffer was full. It is not reset as part of the PFM_RESTART. It is incremented by `1` each time the last sample that fits is recorded. This field is useful to verify if the samples in the buffer are new. It is monotonically increasing.

The buffer header is immediately followed by the first sample. Each implementation must ensure that the size of the buffer header is such that it aligns on, at least, an 8-byte boundary.

### 3.6.4 Structure of a sample

Each sample contains a fixed-size header followed by a variable size *body* which contains PMD register values. The sample header provides information about when, where the sample was recorded and what caused it to be recorded. The sample header structure is defined as follows:

```
typedef struct {
        pid_t           pid;
        pid_t           tid;
        uint16_t        ovfl_pmd;
        uint16_t        set;
        uint16_t        cpu;
        uint64_t        last_reset_val;
        uint64_t        tstamp;
        uintptr_t       ip;
} pfm_default_smpl_entry_t;
```

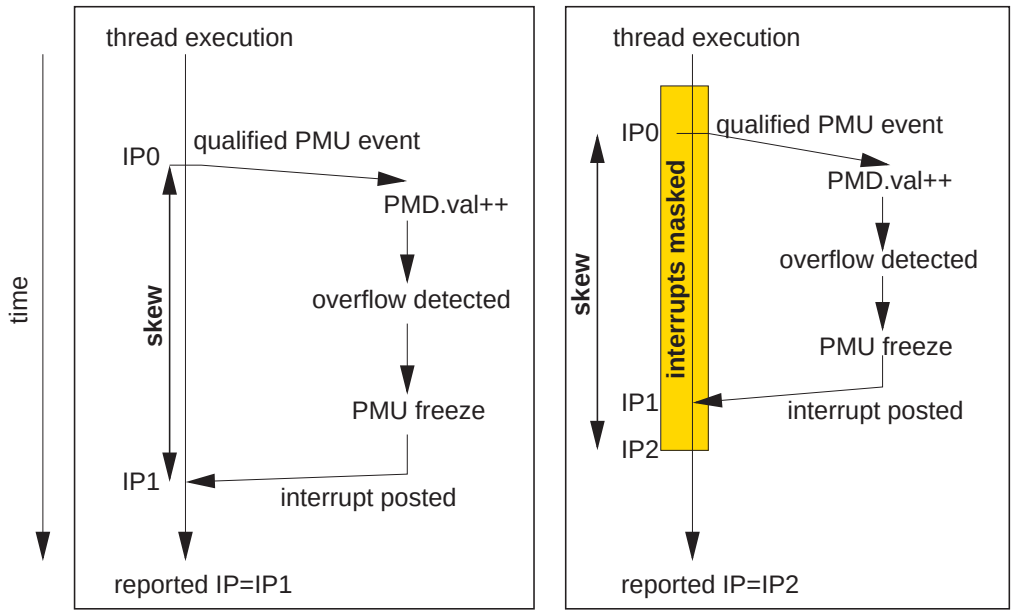


Figure 3.27: possible skew on the reported instruction pointer.

The fields are defined as follows:

- `pid`: the unique identification of the process that was running at the time of the counter overflow

- `tid`: the unique identification of the kernel thread that was running at the time of the counter overflow.
- `ovfl_pmd`: the index of the PMD register that triggered the recording of the sample
- `set` : the active set at the time of the overflow
- `last_reset_val`: the last 64-bit value that was loaded in the PMD register that overflowed
- `ip` : the instruction pointer of the current thread when it was interrupted because of the counter overflow
- `tstamp` : a unique time stamp for the processor on which the PMU overflow interrupt was triggered
- `cpu` : the CPU on which the PMU overflow interrupt was triggered.

The `pid` and `tid` fields are mostly interesting for system-wide contexts because monitoring is active across all threads executing on a specific processor. In per-thread mode, the value of this field is the identification of the thread the context was attached to at the time of the overflow.

The `ip` field should always be treated with caution. It denotes the location where the current thread was executing when the PMU overflow interrupted it. On most architectures, this is always after the location where the counter actually overflowed. This is explained by the fact that the PMU *logically* runs in parallel to the execution. In certain cases, the skew can be significant, especially when the overflow occurred in a section of code where interrupts were masked. Two examples are shown in figure 3.27. On the left-hand side, that is usual case where there is a skew between the instruction pointer at which the event occurred and the pointer reported at the time of the interrupt. On right-hand side, we show a case where the PMU interrupt is posted while interrupts are masked, the interrupt becomes visible only after interrupts are unmasked. This is when the instruction pointer is recorded by the hardware. The skew is not specific to this sampling format, it does affect any measurement that is based on the value of the instruction pointer.

The `last_reset_val` reflects the last value that was loaded into the PMD register that overflowed. When randomization is used, this corresponds to the pseudo-random value that was last loaded into the register.

The `cpu` field is mostly used for a per-thread context. It reflects the processor on which the thread was running when it got interrupted because of the overflow. For system-wide contexts, the value reflects the processor to which the context is bound. It remains constant unless the context is detached and then re-attached to another processor.

The format does not make use of the `reg_smpl_eventid` field of the `pfarg_pmd_t` data structure.

The *body* of a sample immediately follows the header. Each implementation must ensure that the size of the sample header is such that it aligns on, at least, an 8-byte boundary. The *body* contains the values for the PMD registers of interest that are associated with the overflowed counter. The values are stored contiguously in increasing PMD register index. The PMD registers of interest are those specified in the `reg_smpl_pmds` field of the controlling PMC register. An example is shown in figure 3.28 for the pair PMC4/PMD4. When PMD4 overflows, PMD5 and PMD7 must stored in the sample, this is indicated by the value `0xa0` in the `reg_smpl_pmd` bitvector. On overflow, the handler stores the values of the two PMD registers in increasing PMD index value. This corresponds to PMD5 then PMD7 in the figure.

In case where multiple counters are setup as sampling periods, it is possible to specify different PMD registers of interest for each of them. Hence, samples can have different sizes. To parse the samples,

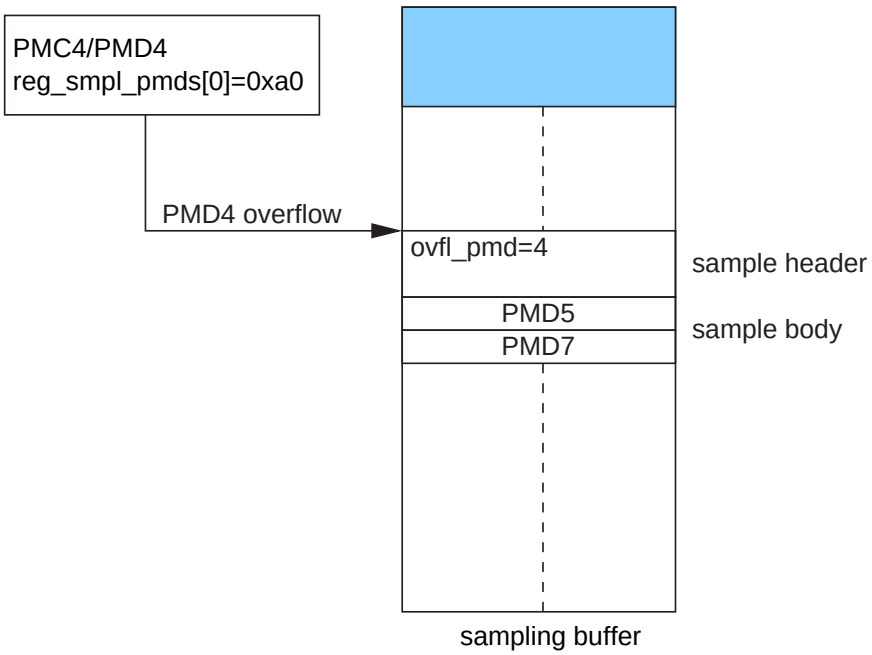


Figure 3.28: sample body layout.

it is necessary to refer to the `ovfl_pmd` field to determine what triggered the recording of a sample. Based on this information, it is possible to figure out the size of a sample. Of course, this assumes that the tool analyzing the samples knows how the measurement was setup, but this is usually the case.

In case of multiple simultaneous overflows, one sample is recorded per overflowed PMD register. Therefore the `ovfl_pmd` is different for each one of them. The samples are guaranteed to be consecutive in the buffer, unless the buffer becomes full before the last sample can be written. The samples all contain the same time stamp, making it possible to identify simultaneous overflows. Similarly, the `set` is guaranteed to be identical for all samples. An example is shown in figure 3.29, where both PMD4 and PMD5 overflow at the same time. First the sample for PMD4 is written, it is immediately followed by the sample for PMD5. Note that the `ovfl_pmd` fields are different but the time-stamps are identical.

There is no padding added at the end of a sample. It is always automatically aligned to, at least, an 8-byte boundary by construction.

It is totally conceivable to have sampling periods with no associated PMD registers of interest. This is the case if the information needed by the tool is all in the sample header, such as the instruction pointer. In that case, the *body* is empty and sample headers follow each other.

There is no partial sample. There is either enough space to record a complete sample or the sample is not written. We detail this behavior in the next section.

### 3.6.5 Overflow and restart behaviors

During an overflow, the format's handler routine is called, it records the samples into the buffer. If the buffer is not full, then the overflowed counters are all reset using their respective *short* periods and monitoring resumes. If any of the overflowed PMD register has some bits set in the `reg_reset_pmds`

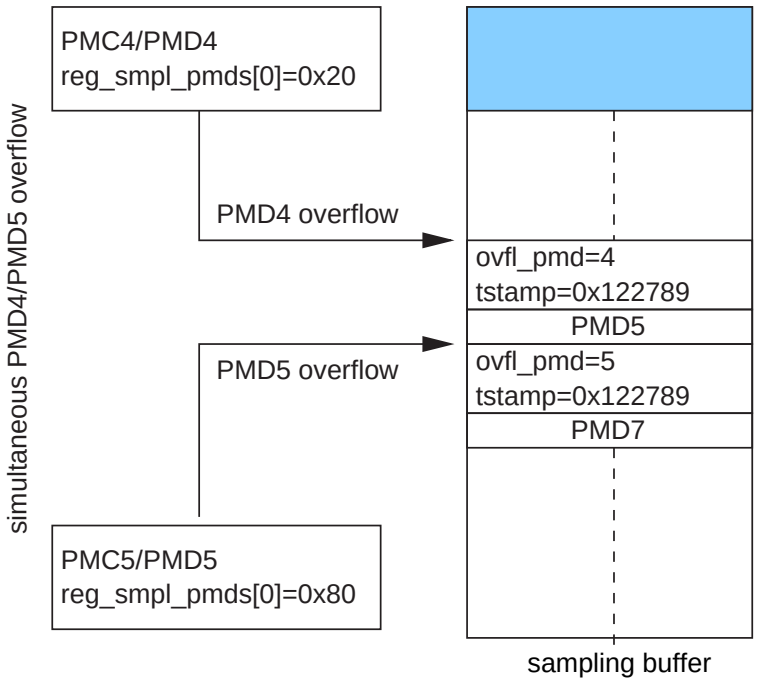


Figure 3.29: simultaneous overflow by two counters.

bitvector of its controlling PMC register, then each of the designated PMD register is also reset using the *short* reset value, if any. No notification is sent to the controlling process.

When the last sample is written, the buffer becomes full and the `hdr_overflows` is incremented by `1`. In this case, none of the overflowed PMD register is reset. This is deferred until the PFM_RESTART command is issued.

Given that the format supports variable size samples, the determination as to when the buffer is full is not necessarily trivial because it is not possible to know the size of the next sample. In order to avoid losing the last sample, the format uses a so-called, *post check* approach, where an estimate of the remaining space is computed after a sample is written. If the remaining space is smaller than the largest sample possible for the host PMU, then the buffer full condition is declared. With this technique, some space may be wasted at the end of the buffer, but no partial samples are ever recorded.

The maximum sample size is defined as follows:

$$\texttt{max\_sample\_size} = \texttt{sizeof(pfm\_default\_smpl\_entry\_t)} + 8 \times \texttt{num\_impl\_PMD}$$

The `num_impl_PMD` variable refers to the number of PMD registers implemented by host PMU. The eight factor comes from the fact that each PMD is represented by a 64-bit value.

When there are simultaneous overflows by multiple PMD registers, multiple samples are recorded. However there may not be enough space in the buffer for all samples. In that case, samples up to what can be accommodated by the buffer are written, the others are not written and therefore are lost. It may be possible to detect such condition by analyzing the bitvector in the PFM_OVFL_MSG message and by comparing it with the last few samples recorded in the buffer.

Figure 3.30 shows how the buffer full condition is detected. On the right-hand side the buffer is not full because the remaining space is bigger than the maximum sample size. The left-hand side shows what

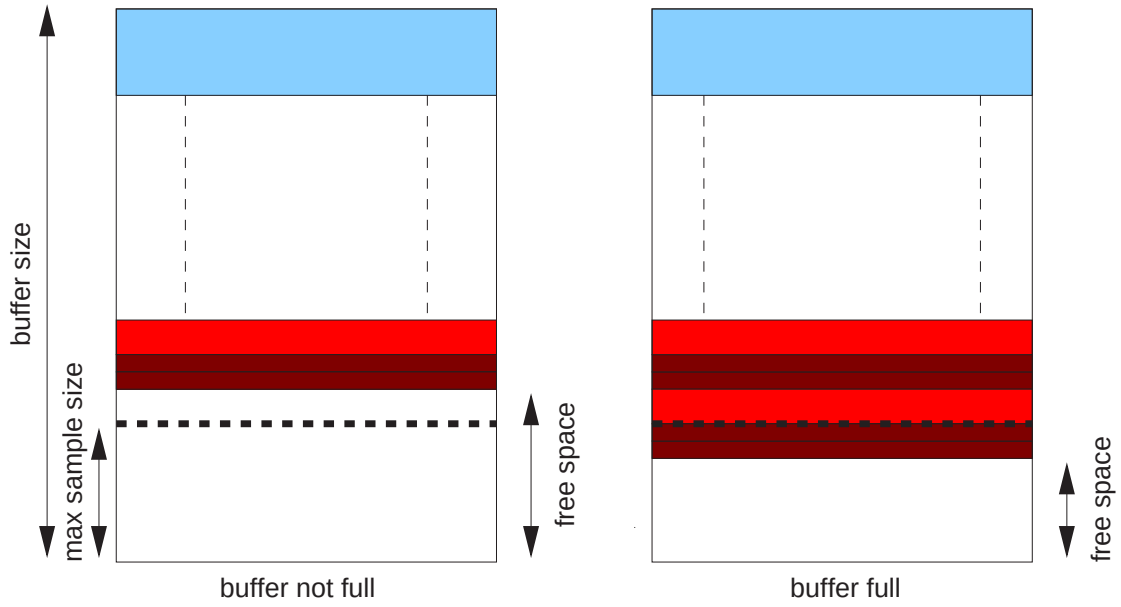


Figure 3.30: detection of a buffer full condition.

happens after the next sample is written, the remaining space is smaller than the maximum sample size and the buffer is considered full, what is left is wasted.

When the buffer becomes full the format systematically requests a user level notification. An actual notification happens only if any one of the overflowed PMD registers has the PFM_REGFL_OVFL_NOTIFY flag set in its controlling PMC register. Otherwise, the buffer is said to have *saturated*. In both cases, monitoring is masked, i.e., no qualified events are collected, thus no new samples are recorded. At this point, the buffer is in a *stable* state, it is safe to parse it, i.e., the information cannot change until the PFM_RESTART command is issued.

A buffer overflow occurs always as a consequence of a counter overflow, therefore the notification sent is a regular overflow notification. The information in the overflow message reports which counters overflowed. The monitoring tool knows it is using the sampling format, therefore it knows that the overflow notification must be interpreted as the signal that the buffer is now full.

When the notification is received, the tool should process the sample in whatever ways it needs. When this phase is completed, the tool signals that monitoring can restart by invoking the PFM_RESTART command. At that point, the default format resets the buffer. The `hdr_count` field in the buffer header is set to 0, the offset in `hdr_cur_pos` is set to point to the first byte after the buffer header. The reset only affects the buffer header, the rest of the buffer is not touched. In particular the previous samples are not cleared. The counters that overflowed last, i.e., the ones which caused the last sample to be recorded are reset using their *long* reset values. When randomization is used new pseudo-random values are chosen. Finally, monitoring is resumed and new samples can once again be collected.

When no notification is requested on buffer full condition, monitoring is masked and execution resumes. The tool is not informed that the buffer is full and no new samples are collected. This is the *saturation* point. This mode can be useful when a tool wants to collect up to $n$ samples for an entire run. At the end of run, the tool simply inspects the buffer and processes whatever is there. Issuing a PFM_RESTART command on a saturated buffer does reset the buffer and monitoring resumes.

The PFM_RESTART command can be issued at any time. If it is issued following a notification, the

behavior is as described above. When it is issued while monitoring is not masked, the command simply resets the buffer header such that `hdr_count` is 0 and `hdr_cur_pos` points to the first byte after the buffer header. This type of restart is called an *active restart*.

## 3.7 Support for event sets and multiplexing

### 3.7.1 Definition of an event set

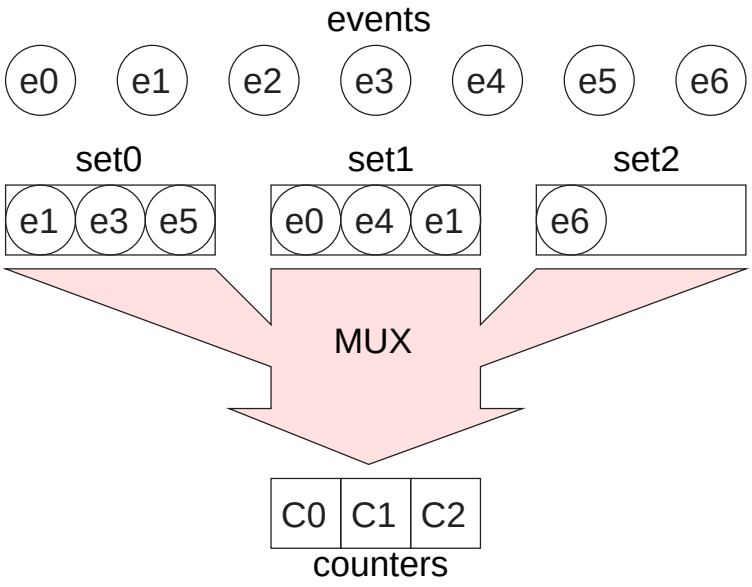


Figure 3.31: event sets and multiplexing principle.

An event set is a *software abstraction* which encapsulates the PMU machine state. An event set includes the values for all PMC and PMD registers and potentially the values of other registers used in conjunction with the PMU.

A context is allowed to define multiple event sets. Each set accommodates as many counters as provided by the hardware. Sets can then be multiplexed onto the actual PMU hardware. the basic principle is illustrated in figure 3.31, where at the top we have 7 events to measure but only 3 counters. Events are distributed among three sets. Notice that events can be distributed in any order that is supported by the PMU. Then the perfmon subsystem multiplexes one set at a time onto the actual counters.

### 3.7.2 Motivations

There are several reasons why having support for event sets and multiplexing is very interesting for monitoring tools. We review them in the following sections.

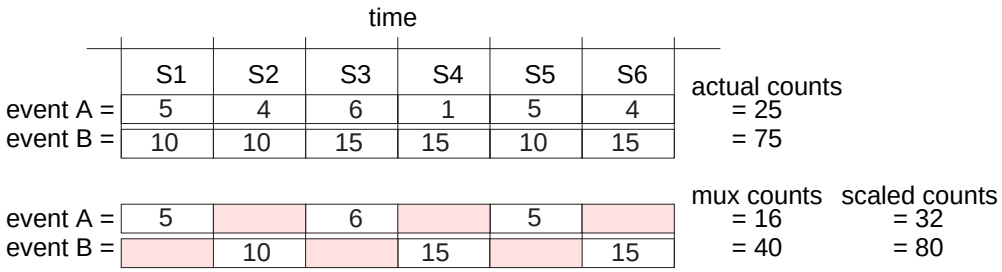


Figure 3.32: event multiplexing and its limitations.

**Limited number of counters**

First and foremost, sets can be used to overcome the limitations on the number of available counters. Depending on the PMU, the number of counters may be too small to collect certain measurements in one run.

A first solution to the problem is to use multiple runs but then the tool needs to deal with possibly a lot of fluctuations between runs. This is not very convenient and sometimes it can quite difficult if the workload is hard to restart.

A second solution is to multiplex the counters between multiple sets of events. For instance, if there is only 1 counter but 2 events to measure, the first event is loaded and measured for a certain period of time, then its value is saved and the second event is loaded and measured for a certain period of time, then the value of the counter is saved and the first event and its saved counter value are reinstalled and so on. In the end the two accumulated values can be scaled up to obtain an estimate of the total count for both events covering the entire run. It is important to realize that this is just an *estimate* and that it may not necessarily reflect what would have been collected, had the host PMU had 2 counters.

We illustrate the limitations of multiplexing in figure 3.32 where we show an example of a PMU with 1 counter and two events, A and B, to measure.

The top half of the figure shows what is collected for event A and B if they are run one after the other. For the complete execution we have 25 occurrences of event A and 75 of event B. Collecting this information required two runs.

The bottom half of the figure shows what happens if we use multiplexing, effectively using two sets of one event each. The execution time is sliced into small periods $S_i$. During period $S_1$, event A is measured, then its value is saved and event B is measured during period $S_2$. Both events alternate until the end of the execution. At that point we have collected *estimates* for A and B. The scaling operation takes these estimates and extrapolates what would have been the total value if each event had run for the entire duration of the execution. The scaling uses a simple average such that for event A:

$$A_s = \frac{A}{slice_A} * \sum_{i=1}^{m} slice_i$$

Where $A$ represents the value of event A at the end of the run. $slice_A$ is the number of time slices used for event A. $slice_i$ represents the total number of time slices for the execution. Basically the scaling takes the average number of occurrences of an event for one time slice and scales it to the total number of slices. We obtain the results shown on the right hand side of the figure. For event A, we have 3 time

slices with a total number of time slices equal to 6, hence the scaled count is:

$$A_s = \frac{16}{3} * 6 = 32$$

Obviously, there are overestimated because of the average calculation which does not take into account possible drop in the number of occurrences. This is clearly visible for event A where in all *blind spots*, the count is smaller. Such behavior can happen if the workload is fluctuating and the switching interval is too long.

It is possible to minimize those blind spots by having smaller time slices. But of course, that implies more switching which raises the overhead because the PMU state must be saved and a new state must be installed each time.

#### Constraints on event combinations or counters

Sometimes, having a large number of counters does not necessarily alleviate the need for multiplexing. For many PMU implementations, counters come with constraints on how events can be measured. It is rare when all counters are symmetrical and fully generic, i.e., any one counter can accept any one event. Due to underlying hardware resource constraints, it is very common to have restrictions such as:

- event A can only be measured by counter C
- event A and event B cannot be measured at the same time
- if event A is measured then only event B, C, or D can be measured on the other counters

Trying to maximize the use of the all counters while minimizing the number of runs for given set of events can be quite challenging.

Event set multiplexing can be used to overcome some of the restrictions by allowing measurements to be collected in a single run. This is achieved by placing incompatible events into different event sets and then multiplexing the sets during the execution.

#### Cascading of measurements

Another reason for supporting event sets is that it makes it easy to build *cascading* measurements. For instance, let us suppose a measurement is constructed as follows: *"after $n$ retired instructions, start counting the number of cache misses"*. In this case, once a counter has reached a certain threshold, another counter must start counting. With event sets, cascading can be constructed simply with two sets. Switching from the first set to the second when the $n$ threshold is reached. The threshold must be expressed such that the counter overflows after $n$ occurrences of the event. This can be done with overflow-based switching.

### 3.7.3 Why a kernel-level interface?

Support for event set and multiplexing can very well be implemented at the user level using the exising perfmon interface. On a timeout, a monitoring tool could completely reprogram the PMU using PFM_READ_PMDS, PFM_WRITE_PMCS, and PFM_WRITE_PMDS. This works well in the case of a self-monitoring applications but it becomes quite expensive when an application is monitoring the thread

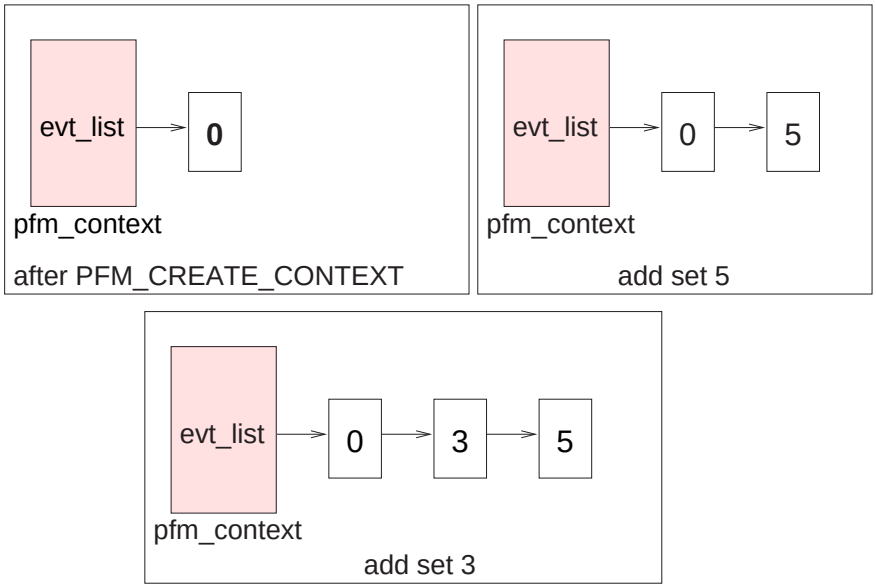


Figure 3.33: creating event sets.

of another process. A lot of context-switches are necessary. In fact, two context-switches per reprogramming. Some implementations may also impose that the monitored thread be stopped to allow reprogramming. The overhead generated by this approach limits the rate at which sets can be mutiplexing and can lead to large *blind spots* as described in the example of figure 3.32.

As a consequence, the interface directly provides event sets and multiplexing at the kernel level. For a per-thread measurement, it implies that switching always occurs in the context of the thread that is monitored, thereby providing an important performance boost which, in turn, allows faster switching and more accurate results. Other optimizations are possible especially for saving and restore the PMU machine state quickly.

### 3.7.4 Operating on event sets

The interface provides commands to create, modify, delete sets. Each of these operation can only be executed when the context is detached. This restriction is not very limiting because it is expected that sets are created at the beginning of a measurement and remain throughout the collection period. In the following sections, we also describe how it is possible to create sublists of sets making it possible to program multiple distinct measurements in advance.

### 3.7.5 Creating event sets

An event set is uniquely identified by a simple number ranging from 0 to 65535. Therefore up to 65536 distinct sets can be created. When a context is created, the initial set is automatically created. It is always assigned the number 0. Set0 cannot be deleted, therefore a context always has, at least, one set. Extra sets can be created using the PFM_CREATE_EVTSETS command.

New sets are allocated and placed in an ordered list. The order is determined by the set identification number. Sets are placed in the list in increasing set identification number. This is illustrated in

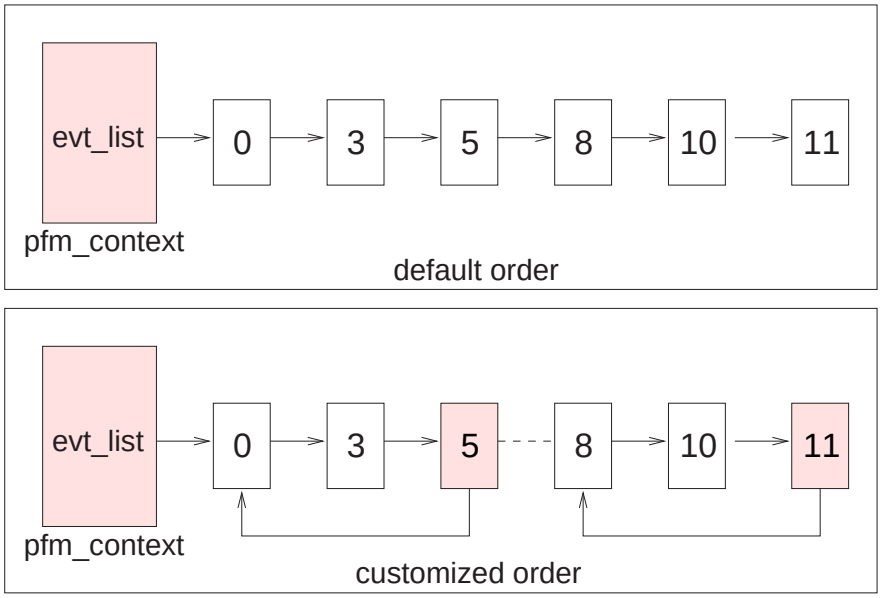


Figure 3.34: customized set ordering.

figure 3.33, where set5 is first inserted and placed after set0. Then set3 is added and placed after set0 but before set5.

The order in the list is important as it may determine the order in which sets are switched to and from.

### Default switch ordering of sets

By default, switching does follow the way sets are ordered in the list. In the example from figure 3.33, switching from set0 leads to set3 and switching from set3 leads to set5. Switching always happens in a *round-robin* fashion. This means that, in the figure, the set following set5 is set0.

The organization of sets shown in figure 3.33 represents the logical view and does not preclude any kind of implementation of the list of sets.

### Customized switch ordering of sets

It is possible to modify the default switch ordering of sets. The main motivation is to build groups of sets. This allows an application to program all the event sets for multiple distinct measurements all at once and then multiplex the counters on selected group of sets without having to reprogram and possibly add or delete sets on the fly.

Modifying the default switch order is accomplished using the PFM_SETFL_EXPL_NEXT flag on a per set basis. When this flag is specified, the *next set*, i.e., the set to switch to, is not determined by the default switch order but instead by the set identification number specified in the `set_id_next` field, the *explicit link*, when the set is created. In practice, this mechanism is used mostly to terminate a *chain of sets* early as shown in figure 3.34 where both set5 and set11 have the flag set. From set5 switching goes to set0, from set11, it goes back to set8. In this example we have logically created two groups, or *sublists*: 0-5 and 8-11. Using the PFM_LOAD_CONTEXT or PFM_START command, it is possible to select which of the two sublists is active. The dashed line between set5 and set8 indicates the default order which is

not used in this case. Note that an explicit link does not have to be going backward, any set number is accepted as long as the set exists when the context is attached.

### 3.7.6 Set switching

Switching is the action of moving from one set to another. The interface supports two methods for switching:

- *time-based* : the switch from one set to another occurs after a certain interval of time
- *overflow-based*: the switch from one set to another occurs after some counters have overflowed a certain number of times

The method for switching is chosen on a per-set basis. For a context with multiple sets, it is possible to use a mix of methods. Only one method is allowed per set. By default, no method is selected, which means that no switching occurs. This is how set0 is created. If this is not satisfactory, it is always possible to modify the properties of a set using the PFM_CREATE_EVTSETS command.

#### Time-based switching

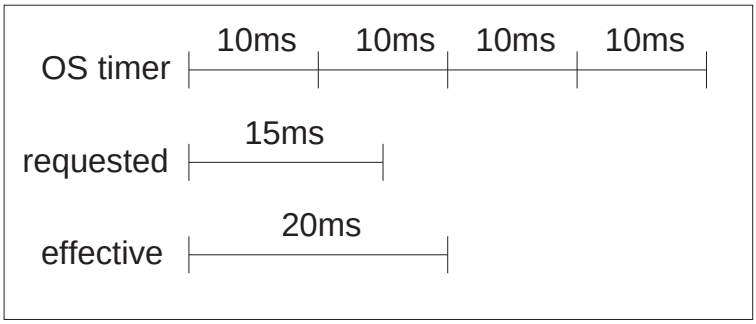


Figure 3.35: *requested* vs. *effective* switch timeout.

Time-based switching is selected when the PFM_SETFL_TIME_SWITCH flag is enabled. With this method, switching occurs at a regular time interval indicated in the `set_timeout` field when the set is created or modified. The timeout is expressed in seconds and nanoseconds.

Each implementation should drive the timeout from the lowest level timer of the operating system, i.e., most likely the timer tick in order to get the best granularity possible. However, even the timer tick may not necessarily have a small enough resolution. Oftentimes, the granularity is only 10ms, i.e., there is a timer interrupt every 10ms. For this reason, the timeout requested by an application may not necessarily be possible. Each implementation is expected to approximate to the best it can. The timeout returned in `set_timeout` after a call to the PFM_CREATE_EVTSETS command is the *effective* timeout. In figure 3.35, we show an example where the timeout is driven by the timer tick which has a granularity of 10ms. The *requested* timeout is 15ms, but the *effective* timeout will be 20ms, i.e., the timeout is rounded up to the next multiple of the timer tick. By returning the *effective* timeout, the interface informs the application that the *requested* timeout cannot be achieved. It is up to the application to either accept, reject, or re-adjust the timeout to suit its needs.

Time-based switching effectively starts once the context is attached and monitoring is activated via PFM_START. It stops when PFM_STOP or PFM_UNLOAD, or *close()* is issued. The timeout is not decremented when monitoring is masked. This can happen on a counter overflow followed by a user level notification when no custom sampling format is used. When a custom sampling format is used, it depends on whether the format masks monitoring or not. Refer to the format specification for more details. In any case, the timeout is *reactivated* whenever monitoring is resumed via PFM_RESTART. The timeout restarts from where it was stopped, i.e., it is not reinitialized. The behavior is illustrated in figure 3.36. While monitoring is masked, the timeout is stopped.

For implementations which make it possible to start and stop monitoring completely from the user level, the behavior described in the previous paragraph must be identical.

In per-thread mode, the timeout is *running* only when the monitored thread is running. When a thread is context switched out, the timeout is stopped. In other words, the timeout does not measure wall-clock time. As a consequence switching can only occur when the monitored thread is running. When the thread is running but monitoring is masked, the timeout is also stopped.

By construction, in system-wide mode, the timeout is not quite measuring wall-clock time because the timeout is stopped when monitored is masked. Switching occurs at the end of the timeout, even when the controlling process is not actually running on any processor.

In system-wide mode, the timeout is active when the idle thread is excluded from the measurement with the PFM_SETFL_EXCL_IDLE flag and the idle thread is the active thread. In other words, time-based set switching may occur while the idle thread is running even though no qualified event is captured.

During the PFM_LOAD_CONTEXT command and when the designated set is using time-based switching, its timeout is reset to the value indicated when the set was created or last modified.

An example of time-based switching for a per-thread context is given in figure 3.37. A context with two sets is attached to a thread. When the thread is stopped because it explicitly blocks or its quantum of time runs out, it is context switched out and the set timeout is stopped. When the thread is rescheduled, the rest of the timeout is restarted.

The timeout is not randomized however there may be enough fluctuations and noise in the system to produce actual randomization. In particular it is likely that when interrupts are masked, no timer interrupts are generated and the timeout is effectively extended.

### Overflow-based switching

Overflow-based switching is selected when the PFM_SETFL_OVFL_SWITCH flag is enabled for the set. With this method, switching occurs when certain counters of the set overflow. Those counters are called *triggers* and there can be more than one per set. If the host PMU supports $n$ counters with overflow interrupt capability then there can be up to $n$ triggers. An application can control after how many overflows of each counter, switching occurs. In other words, switching does not necessarily happen

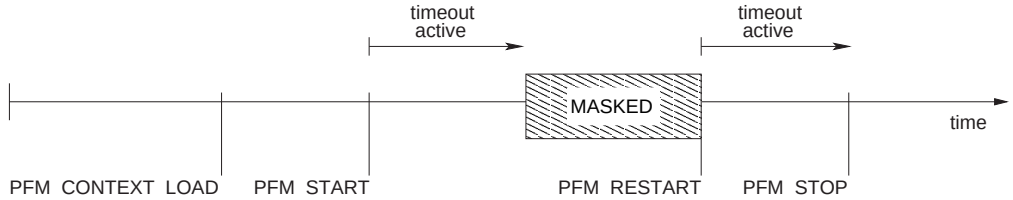


Figure 3.36: timeout activation periods.

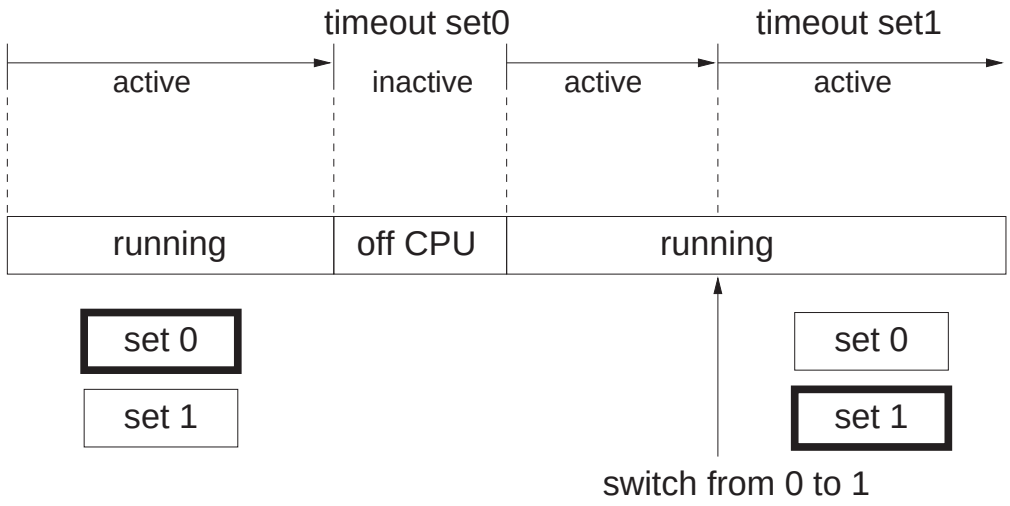


Figure 3.37: a time-based switching example.

at the first overflow of a counter. Similarly, not all counters which overflow trigger a switch. The counters which can trigger a switch must have their `reg_ovflsw_thres` field set to a non zero value. This field is modified with the PFM_WRITE_PMDS command. The value of this field determines the overflow *threshold* at which switching occurs. At each overflow, the value is decremented by one, when it reaches zero, a switch is performed. Because there can be multiple triggers, there can be multiple thresholds, each specific to a counter. A switch occurs as soon as one of the thresholds reaches the value of 0. It is possible that more than one threshold reaches zero at the same time in case multiple counters overflow at the same time. Next time the set becomes active, the value is reloaded with the same threshold. This mechanism alleviates the need to dedicate a counter as a trigger. This is useful when sampling as a counter can be used as the sampling period as well as the switch trigger, thereby maximizing the use of the counters.

In figure 3.38, we show a simple example with two sets: set0 and set1. The left side of the figure depicts what is going on for set0 which is initially the active set. The right side depicts what is going on with set1 which is initially inactive. The arrow in the middle of the figure denotes the elapsed time. In each set a PMD register is configured as a simple counter: PMD5 for set0 and PMD4 for set1. We assume their controlling PMC registers are setup properly to measure certain events. For set0, PMD4 is used as the overflow trigger and is measuring event A. It is initialized to overflow after 1000 occurrences of the event. The threshold is set to 2, meaning that after two overflows, i.e., after 2000 occurrences of event A, the measurement switches to set1. After 1000 occurrences of event A, the first overflow occurs, the threshold is decreased to 1, the short sampling period, here -1000, is reloaded into PMD4 and monitoring resumes. After the second overflow the threshold reaches 0 and switching to set1 occurs. For this set, PMD5 is the trigger and the threshold is set to 1. Hence after one overflow, a switch to set0 will occur.

A PMD register used as a trigger is expected to be setup just as it would be for sampling. The `reg_long_reset` and `reg_short_reset` must be setup as for any regular sampling period. Unless overflow notification is requested for the counter or a specific custom sampling buffer format is used, the short reset value is reloaded in the counter on overflow.

The overflow threshold is reloaded each time the set becomes active. Each time the same threshold value is re-used. It can be modified with the PFM_WRITE_PMDS command. As part of the switch, the indexes of counters which triggered the switch are recorded into a bitmask. It may be extracted with the PFM_GETINFO_EVTSETS command. In the `pfarg_setinfo_t` structure, the `set_switch_pmds` field

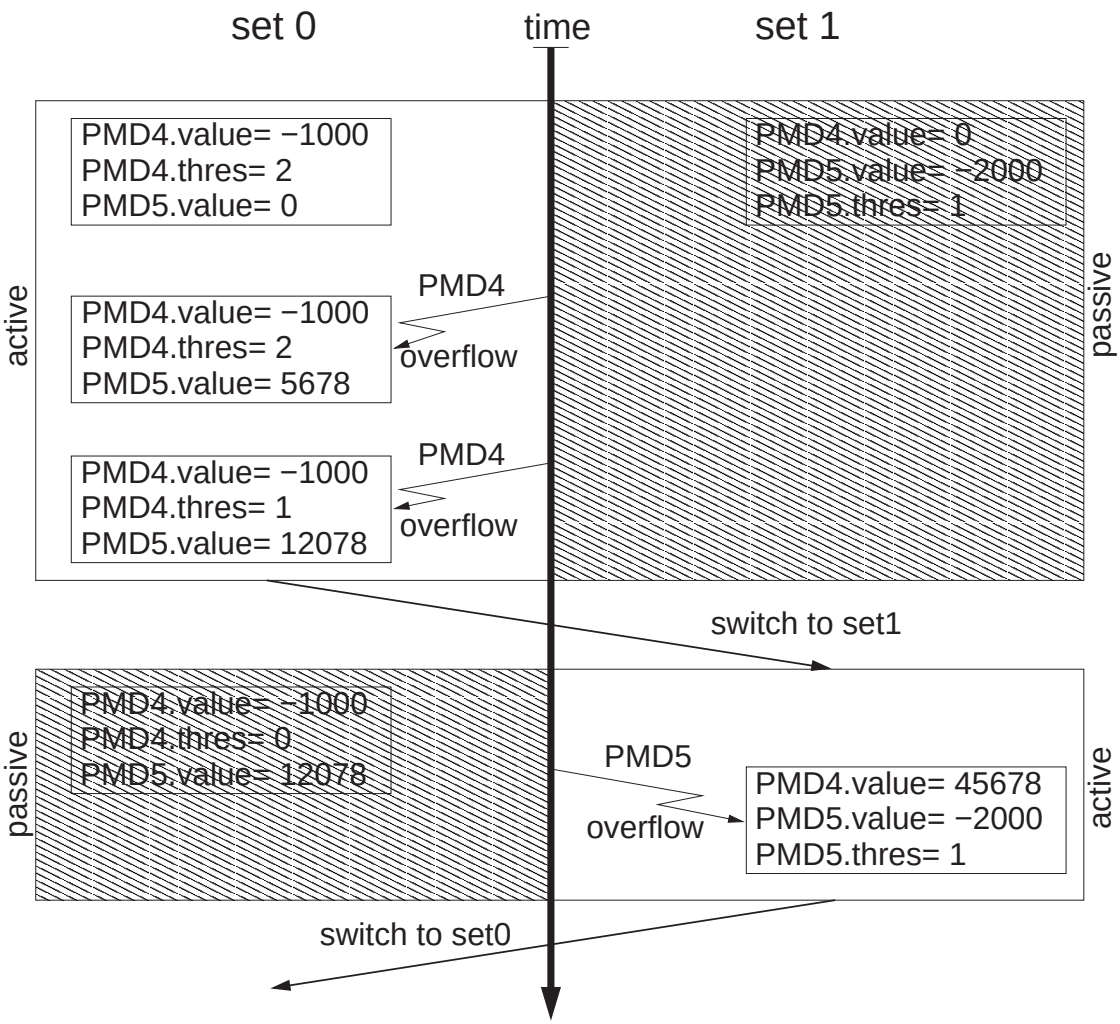


Figure 3.38: Example of overflow-based switching.

is the bitmask containing the set of PMD registers which last caused a switch from this set.

Overflow-based switching cannot occur when the idle thread is running and the active set has the PFM_SETFL_EXCL_IDLE flag is set. This comes from the fact that monitoring is effectively stopped, as if a PFM_STOP had been issued while the thread runs.

During the PFM_LOAD_CONTEXT and when the designated set is using overflow-based switching, its counting monitors have their switch thresholds reset to the value indicated when the corresponding PMD registers were last programmed with the PFM_WRITE_PMDS command.

### Cascading counters

Overflow-based switching can be used to easily implement counter cascading. Cascading is used when an application needs to start measuring certain events only once a certain threshold has been reached. The threshold is typically expressed as a number of occurrences of a particular event.

Cascading requires at least two event sets. The first set contains the threshold event. Once the threshold is reached, a switch to the second set occurs. If the threshold is never reached, the second set is never activated.

The threshold is implemented by programming a counter in the first set to overflow at the threshold value. For instance, if the threshold is 1000 occurrences, then the counter is armed at -1000. Then the set enables PFM_SETFL_OVFL_SWITCH. The second set is setup not to switch at all. After the setup phase, the context is attached to the thread to monitor and monitoring is activate via PFM_START. At the end of the execution, the values of interest are in the second set and can be retrieved with PFM_READ_PMDS. If they are 0, then there are two possible explanations:

1. no switch happened. In other words, the threshold was not reached
2. the switch happened, but there was no occurrence of the event of interest in the second set.

Using the PFM_GETINFO_EVTSETS, it is possible to sort out whether or not the second set ever got activated simply by looking at the number of activations in the `set_runs` field. If it has not changed since the context was last attached, then the set never became active.

### 3.7.7 Event Sets and sampling

The interface fully supports the combination of multiple event sets with sampling.

#### Interactions with user level sampling

It is not necessary to use a sampling format to create a sampling measurement. Sampling can completely be driven from the user level via the overflow notification mechanism.

In this case, it is important to retrieve the identity of the active set at the time of the overflow. The active set number is always included with the overflow message in the `msg_active_set` field. Using the value of this field, an application can then call PFM_READ_PMDS to retrieve the values of some PMD registers of interest.

#### Interaction with custom sampling formats

All custom sampling formats do not necessarily support multiple event sets. In that case, they advertise this problem by using the PFM_FMTFL_NOSET flag in their `pfm_smpl_fmt_t` structure. When a context is using such a format, it is not possible to create additional event sets beyond the default set, i.e., *set0*. As such, there is no risk of incompatibility with the format.

Each format is free to store the samples coming from different sets in any way it wants. in particular, one can envision a format where samples from different sets are saved into distinct regions of a sampling buffer.

#### Behavior with the default sampling format

The default sampling format does support multiple event sets. Each sample header includes a field, namely the `set` field, which identifies the active set number at the time of the overflow.

The configuration of each monitor can vary widely between sets. As such, samples may have different sizes. Parsing the buffer must be done from the top and in sequential order using the `hdr_count` as the number of valid samples. Each sample must be *decoded* based on the set number and overflowed PMD register.

### Behavior with time-based switching

For both type of contexts, when an overflow occurs, time-based switching is stopped if and only if monitoring is masked. Depending on the sampling format, this may not necessarily be the case for every overflow. The interface guarantees that no time-based switching does occur during the execution of the PMU interrupt handler.

In the case of a self-monitoring thread, each implementation must ensure that the timeout cannot expire while the thread is in the middle of a call into the interface. In other words, switching cannot occur while executing perfmon-related code.

### Behavior with overflow-based switching

By definition, overflow-based switching occurs when there is a counter overflow. For both types of context switching happens in the PMU interrupt handler if and only if monitoring is not masked as a consequence of the overflow. Otherwise, switching is typically deferred until monitoring is resumed with the PFM_RESTART command. This ensures that a consistent view of the sets is presented to the controlling process. Depending on the sampling format, monitoring may not not necessarily be masked for every overflow.

When switching occurs in the PMU interrupt handler, the new set is loaded. The PMD registers which last overflowed in this new set and their associated PMD registers of interest are reset using their *short* reset values.

When switching occurs during the PFM_RESTART command, the new set is loaded The PMD registers which last overflowed in this set and their associated PMD registers of interest are reset using their *long* reset values. Note that for non self-monitoring per-thread sessions, the new set may not be reloaded by the time the command returns.

# Chapter 4

# Security

## 4.1 Introduction

This interface is designed to be generic and serve a variety of performance monitoring tools. As such, it cannot assume that tools will be well behaved and follow the proper usage model. Furthermore, it is expected that the interface will be implemented as a built in service of the operating system. As such, it must be incorporate the same security features as the rest of the kernel interface.

For those reasons, security is an important aspect of the interface. Security is a shared burden between the interface specification and the implementation. There may be several operating system specific security issues which can only be resolved at the implementation level. In general, the following general security problems must be avoided:

- leaking unauthorized information from the kernel
- leaking unauthorized information from other processes and threads
- bringing the system down using a denial-of-service style of attack

Throughout the document, we have pointed out *perfmonctl()* calls where an implementation may impose additional restrictions to satisfy a specific security constraint.

The challenge is to enforce reasonable security while allowing a maximum number of users to access the interface without too much troubles. It is certainly not acceptable to use an *all or nothing* model where only a *super-user* can collect measurements.

In this section, we will go over each aspect of the security of the interface.

## 4.2 Accessing the interface

The interface is expected to be built into the operating system and, as such, be readily available via the *perfmonctl()* system call.

The interface provides two categories of commands:

- commands with no file descriptor

- commands requiring a valid file descriptor identifying a context

In the first category where no file descriptor is needed, there is no real security problem for the PFM_GET_CONFIG, PFM_SET_CONFIG, PFM_GETINFO_PMCS, PFM_GETINFO_PMDS commands. Only non-sensitive information is returned by these commands. The PFM_SET_CONFIG is restricted to the system administrator because it sets some of the security parameters. The PFM_GET_CONFIG command returns the global properties of the interface. This is useful for some tool which may need to size certain data structures based upon the information returned by the command.

The second category of commands includes all the calls that operate on an existing context. Without a valid context, the commands cannot be used. The key command is PFM_CREATE_CONTEXT because it creates a new context and therefore opens up the use of the other commands. It should be noted that the command does not access any hardware and no sensitive information can really be extracted until the context is attached to a thread or CPU core.

The interface has two modes of operations: per-thread and system-wide. Both have different risks which we examine in the next two sections.

### Per-thread mode

In per-thread mode, a context is monitoring only a single thread at a time. The context is attached to a specific thread via the PFM_LOAD_CONTEXT command. Each implementation may perform some security checks during this command to ensure that the caller has permission to access the designated thread. Those checks are very specific to each operating system. The interface mandates that a context is never inherited across *fork()*, *vfork()*, or *pthread_create()*. This implies that every monitored thread must be explicitly attached to with the PFM_LOAD_CONTEXT command and thus it must go through the security checks of this command.

By default, any user can create a per-thread context. In environments where this level of security is not enough, the interface provides a way for the system administrator to restrict the creation of a per-thread context to a specific group of users with the PFM_SET_CONFIG command. The identification of the group of users is passed in the `cf_thread_group` field of the `pfarg_config_t` structure. Only users from the group are allowed to create a per-thread context.

By default, there is no group restrictions for a per-thread context.

### System-wide mode

In system-wide, by definition, a measurement collects information about all processes running on a particular processor. Not all processes necessarily belong to the same user. Therefore it is possible to extract information that would otherwise not be visible to the owner of the monitoring application. One solution would be to filter out the processes which cannot be accessed when they are running on the monitored processor but that would defeat the purpose of the system-wide mode. Instead, the interface provides a way of controlling the creation of system-wide contexts.

The interface allows a system administrator to restrict the creation of a system-wide context to a group of users. The group is specified with the PFM_SET_CONFIG command. The identification is passed in the `cf_sys_group` field of the `pfarg_config_t` structure. Only users from the group are allowed to create a system-wide context.

By default, there is no group restrictions for a system-wide context.

## 4.3 Protection of the user

In this section, we describe the security features provided by the interface to protect the privacy of the users. Note that the mechanisms described here may be in addition to the regular operating system protections.

### 4.3.1 Identification of contexts

Each context is uniquely identified with a file descriptor. The validity of the descriptor follows the regular semantics of the operating system. On a POSIX-based system, a file descriptor is valid in the process that created it, its children processes and all of its threads. Similarly, the file descriptor remains valid across the *exec()* system call unless explicitely closed. A process may also receive a descriptor from another process via an explicit transmission in which case the sender is responsible for security.

Outside of this scope, the descriptor has no meaning and therefore it cannot be used to gain access to the perfmon context.

### 4.3.2 PMU machine state

The PMU state includes the values of the PMC and PMD registers and potentially other machine registers. The values stored in the PMC and PMD registers could reveal what an application is measuring. Some PMD registers may contain code and data addresses. As such they must be protected to avoid leaking unauthorized information from one thread to another or from the kernel to a user application.

There are two aspects to this problem:

- reading the hardware PMU registers directly
- reading the software-maintained state of the PMU

#### Accessing the hardware PMU registers

The interface guarantees that it is not possible to read the values of the PMD registers of a monitored thread when the controlling process does not allow it.

In per-thread mode, it is not possible to access the actual PMU registers of the monitored thread while it is running. When a thread is inactive, the PMU state is also protected. Similarly, once the thread has died it is impossible to read the leftover state in the actual PMU hardware. A self-monitored thread should be allowed to access its own PMU registers. On some architecture it may even be possible to do so without calling into the kernel by directly using a machine instruction.

In system-wide mode, the behavior is equivalent to self-monitoring by construction with the exception that any thread with access to the file descriptor and running on the monitored processor core can effectively access the PMU registers either by calling into the kernel or by directly using a machine instruction if the architecture supports it.

Depending on the capability of the hardware, it may be necessary to clear the PMU registers on context switch. However for most architectures reading the PMU registers is a privileged operation therefore this should not be necessary. For those which do allow reading there is typically a provision to disable that feature. Reading from the most privileged level is always possible, hence access must be controlled by software.

**Accessing the PMU software state**

The kernel must maintain the state of the PMU in software. To a bare minimum, it is necessary to provide some storage area to save the PMU state during a context switch.

Getting access to the context requires getting access to the file descriptor that identifies it. Without the file descriptor, the interface guarantees that no access is possible.

In system-wide mode, there is an additional restriction because the caller must run on the processor that is being monitored with the context.

### 4.3.3 The sampling buffer

Depending on which sampling buffer format is used, the interface may be able to re-map the sampling buffer into the address space of the controlling process. Sampling buffer formats may chose not to use the interface buffer allocation and re-mapping services, in which case, it is important to refer to the appropriate documentation for security related issues. The default format does use the allocation and re-mapping services.

The re-mapping is private to the process that creates the context. However the normal POSIX semantics do apply on *fork()* and *pthread_create()*. The mapping is inherited across *fork()*, therefore the buffer is visible in the children processes. It is important to note that the buffer is always re-mapped as read-only. If the buffer should not be visible in a child process, it is always possible to explicitly remove the mapping via a call to *munmap()*.

For all threads inside a process, by construction, the buffer is always visible because the address space is shared.

On *exec()*, the whole address space is torn down and recreated, therefore the buffer is not visible after the call.

## 4.4 Protecting the system

In this section we describe how the system as a whole is protected against applications using the interface as a way to compromise the integrity of a machine.

### 4.4.1 Visibility of kernel level information

Normally, kernel level information is never leaked to an application unless it is safe to do so. In the context of performance monitoring, some information about kernel level execution of a process may become visible to an application. Here are some typical examples:

- it is possible to extract kernel level instruction pointer (IP) when sampling is enabled
- some PMU implementations have the ability to capture branches including source and destination, code and data cache misses addresses
- for many PMU implementations, it is possible to control the privilege level at which qualified events are captured.

All of this can be valuable information to tune an application or the kernel. As such, monitoring kernel level execution should not be forbidden by default.

However in certain security sensitive environments this may be required. The interface does not have provision to limit the privilege level at which events can be monitored. It is too fine grain and would require extended knowledge of the structure of PMC registers at the kernel level.

Each PMU implementation may provide additional information which may be considered potentially sensitive. There is no way to predict what can be extracted by the PMU in the future. Therefore the interface uses a different approach. By default, the information is available to all users, however in security sensitive environments, the creation of a context can be restricted to groups of *trusted* users. Details on this protection mechanism are given in section 4.2.

### 4.4.2 The vector arguments

Many commands take a vector of arguments. Typically the vector is copied from the user address space to the kernel address space for security purposes. This implies that a kernel buffer is allocated to copy the vector into. That buffer could have any size because it needs to accommodate the vector. This could be dangerous because the buffer could consume a lot of system memory.

To counter such an attack, the interface provides a parameter to limit the size of the vector arguments. The maximum size is expressed in bytes and is set in the `cf_arg_size_max` field of the `pfarg_config_t` structure passed for the PFM_SET_CONFIG command. This limit is checked for all commands which take a vector argument.

### 4.4.3 The size of the sampling buffer

In this section, we assume the use of the default sampling format.

The interface of the default format does not impose a particular size limitation on the allocated buffer. The sampling buffer requires non-pageable physical memory therefore it can be quite taxing on the resources of a system. Each implementation must check the size of the buffer against certain resource limits. For instance, on Linux, it is possible to check against the resource limits provided by the *getrlimit()* system call. In particular, an implementation could check against the RLIMIT_MEMLOCK threshold. Of course, this is not a complete protection because these resource limits are per-process.

To overcome this difficulty, the interface provides a global parameter to limit the overall memory used by all sampling buffers. This is a system-wide limit which includes all the sampling buffers active at any one time. The limit is set the PFM_SET_CONFIG command and the `cf_smpl_buf_size_max` field of the `pfarg_config_t` structure.

The size limit is checked when the buffer is allocated and not when the context is attached, i.e., during PFM_CREATE_CONTEXT and not during PFM_LOAD_CONTEXT.

### 4.4.4 Throttling PMU interrupts

The interface does not impose minimal thresholds on the sampling rates because that would require kernel knowledge about each event. Different events happen at different rates. The kernel would have to know what a *reasonable* rate would be for every possible event. That is clearly unrealistic.

Having a sampling period of one, could potentially overload the system especially when coupled with high frequency events such as the one counting elapsed cycles, for instance. Basically for every cycle,

there would be a counter overflow leading to an interrupt. In practice however, this kind of extreme measurement may be mitigated by implementation dependencies. For instance, operating system kernels have periods of time where all interrrupts are masked and that would keep the PMU interrupts pending and thereby lower their rate.

However this can only mitigate the problem and not really solve it. There are two ways to possibly limit the damages:

- restrict access to a trusted group of users who know what they are doing
- throttle the rate at which interrupts are generated

The second solution looks more flexible but the difficulty is to determine what is a *reasonable* rate really to ensure forward progress. As for the first solution, it is too coarse, it certainly prevents the random user from causing problems, yet even the most expert users make mistakes.

One solution is to rate-limit based on time and not on the number of occurrences of an event. For instance, the threshold could be set to no more than 1000 PMU interrupts per second. In the PMU interrupt handler and on a per thread basis, each interrupt would be time-stamped. A new time-stamp would be compared to the time-stamp of the previous interrupt. If the difference between the two is less than an administrator specified threshold, monitoring could be masked until an application issues a PFM_RESTART command. Without notification, such approach would require some sort of polling on the part of the application.

*This solution needs more thoughts. Especially in system-wide mode. More to come in this section...*

### 4.4.5 Custom sampling formats

Any code added to the operating system kernel is a potential risk. Typically all operating systems require special privileges to insert a new module into the kernel.

There is not much the perfmon core can do to verify that a module is not malicious or buggy. In this area, the interface relies entirely on the operating system or administrator to ensure that only users with the right set of permissions can register a custom sampling buffer format.

# Chapter 5

# Itanium Processor Family specific interface

In this chapter we describe the features to the interface that are specific to the Itanium Processor Family (IPF).

By convention, all model-specific constants have the ITA prefix in their name. This allows for fast recognition between generic and model-specific constants.

## 5.1 Itanium specific register mappings

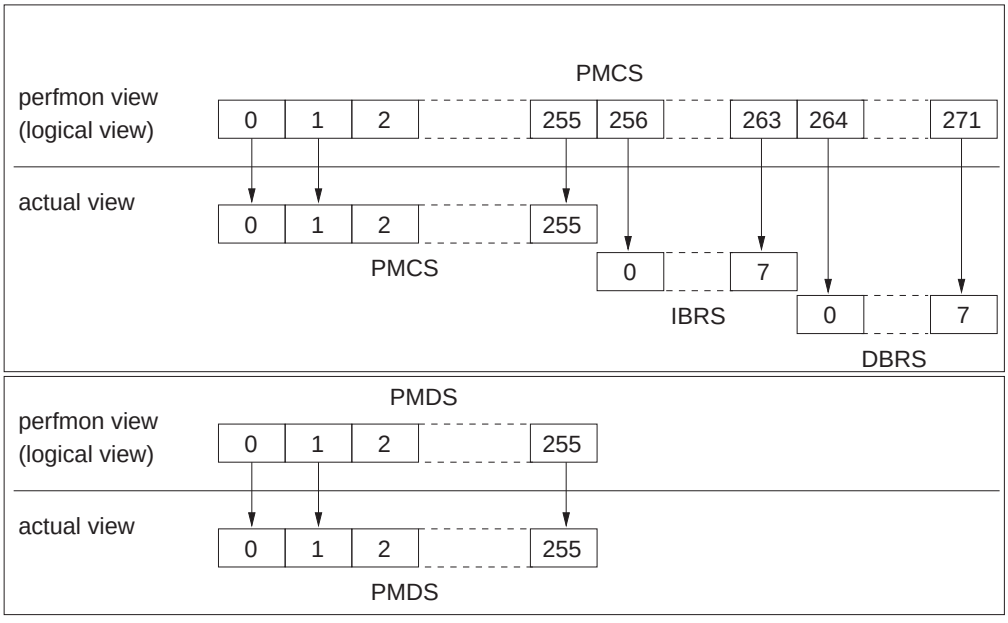


Figure 5.1: Itanium PMU register mappings.

The Itanium architecture is very nice in that is defines the framework in which the PMU can be implemented. The framework defines the minimum and maximum number of PMC and PMD registers that can ever be implemented. The role of each PMC/PMD can still fluctuate yet there is enough information to build a generic mapping for all existing and future Itanium processors.

The PMC and PMD registers naming used by the interface maps directly onto the actual PMD and PMC registers used by the Itanium PMU. The architecture specifies that up to 256 PMC and 256 PMD registers can be implemented. As such, the mapping is trivial as PMC$x$ maps to the actual PMC$x$ registers, same thing with the PMD registers.

The interface defines extra PMC registers to access the debug registers which are used on some Itanium PMU models to implement code and data range restrictions, see section 5.6.

Figure 5.1 shows a global view of the register mappings for all Itanium processors.

## 5.2 Privileged versus user monitors

The Itanium architecture supports two types of monitors: *user* and *privileged*. Each counting PMC register, and possibly model-specific extensions, such as the Data Event Registers on Itanium® 2, can be configured for one type or the other via the `PMC.pm` field.

Starting and stopping each type of monitors is controlled by two distinct bits in the `psr` register: `psr.up` and `psr.pp`. This potentially, allows simultaneous distinct measurements to start and stop independently of each other. Privileged monitors also offer extended settings for dealing with interrupt-triggered execution via the `dcr.pp` bit.

The interface is designed such that *user* monitors are selected for all per-thread contexts and *privileged* monitors for all system-wide contexts.

The `PMC.pm` bit is automatically set depending on the type of context. Applications are not required to set the bit. No error is generated when the bit set by an application does not match the type of the context.

## 5.3 Secure monitoring

The Itanium architecture provides a bit, `psr.sp`, to control access to the PMD registers and the `psr.up` bit for code running at a non-zero privilege level. When this bit is cleared, it is possible to read the PMD registers via a simple `mov` instruction. Similarly, it is possible to start and stop monitoring of *user monitors* with the `rum/sum` instructions. When the bit is set, reading a PMD register with a `mov` instruction returns `0` and `rum` and `sum` instructions have no effect on `psr.up`. This bit has no effect on *privileged monitors*. Being able to read a counter, start and stop monitoring with a simple instruction is very interesting when measuring small sections of code where monitoring overhead must be minimized. This is typically the case for programs that are monitoring themselves.

By default, the interface sets `psr.sp` to `1`, i.e., in secure monitoring mode, except for self-monitoring per-thread context. This is where it could be very useful to monitor small sections of code and where overhead must be minimized, i.e., no system call to start and stop monitoring. However when monitoring another thread, it may also be interesting to allow *insecure* monitoring because it becomes possible to use `rum` and `sum` instructions in the monitored program to measure specific portions of code. In many cases, it may be much easier to inject `rum/sum` instructions into a binary than to modify the source code to construct a self-monitoring program.

## 5.4 Context flags

In this section, we describe the context flags that are specific to the Itanium Processor Family (IPF).

### 5.4.1 The PFM_ITA_FL_INSECURE flag

The PFM_ITA_FL_INSECURE flag overrides secure monitoring for non-self monitoring contexts. It is only useful for per-thread contexts using *user monitors*. This flag is invalid for a system-wide context.

## 5.5 Event set flags

The following Itanium-specific set flags are defined.

### 5.5.1 The PFM_ITA_SETFL_EXCL_INTR flag

The PFM_ITA_SETFL_EXCL_INTR flag is used to exclude interrupt-triggered execution from monitoring. Interrupts include external interrupts but also software interrupts such as those triggered by `break.i`. This flag does not have any influence if all the events in the set are programmed to monitor at non zero privilege level. This flag only affects monitoring if the PMC registers of the set are setup as *privileged monitors*. Any attempt to set this flag when *user monitors* are selected will return an error. By default, this flag is not set, i.e., interrupt-triggered execution is monitored.

#### Interaction with the PFM_SETFL_EXCL_IDLE flag

The flag can be used in conjunction with PFM_SETFL_EXCL_IDLE. In that case this means that the idle thread is not monitored and that interrupts processed during its execution are not monitored either. In other words while the idle thread is active, nothing is monitored and when it is not active only non interrupt-triggered kernel level execution is monitored.

When only the PFM_SETFL_EXCL_IDLE flag is specified, interrupt-triggered execution is monitored when the idle thread is active. This is motivated by the fact that the system cannot be considered idle when it is processing interrupts.

### 5.5.2 The PFM_ITA_SETFL_INTR_ONLY flag

the PFM_ITA_SETFL_INTR_ONLY flag is used to monitor only interrupt-triggered execution. Interrupts include external interrupts but also software interrupts such as those triggered by `break.i`. This flag does not have any influence if all the events in the set are programmed to monitor only non zero privilege level. This flag only affects monitoring if the PMC registers of the set are setup as *privileged monitors*. Any attempt to set this flag when *user monitors* are selected will return an error. By default, this flag is not set, i.e., interrupt-triggered execution is monitored.

#### Interaction with the PFM_SETFL_EXCL_IDLE flag

Both flags can be specified at the same time. The PFM_SETFL_EXCL_IDLE flag excludes the idle thread from active monitoring. This flag monitors only during interrupt-triggered execution. While executing in the idle thread, interrupts can be processed and will be monitored. Non interrupt processing execution of the idle thread is not monitored.

#### Interaction with the PFM_ITA_SETFL_EXCL_INTR flag

Both flags cannot be used at the same time, i.e., for the same set.

## 5.6 Support for code and data range restrictions

Both the Itanium® [7] and Itanium® 2 [9] processors include a nice PMU feature, called range restriction, which uses registers outside the set of hardware PMC and PMD registers. For both processors, it is possible to restrict monitoring to a certain range of code or data addresses. This feature uses the debug registers to describe the address ranges. Obviously those registers are normally used for debugging applications. On Linux, they are setup with the *ptrace()* system call, other operating system have equivalent system calls.

Given that it is never possible for a thread to debug itself, the existing kernel interface requires a second thread, usually in the parent process, for the setup and control of debugging. However it is totally reasonable for a self-monitoring thread to use address range restrictions. Similarly, some operating systems may restrict the addresses that can be stored in a debug register. For instance, it is likely that setting a breakpoint in the kernel is forbidden. Yet for monitoring purposes, it may be useful to restrict monitoring to an address range in the kernel address space.

In order to provide access to the debug registers for monitoring purposes, the number of logical PMC registers presented by the interface was extended to cover the eight code debug registers and the eight data debug registers. To program range restrictions, the PMU configuration registers and the debug registers can be modified using the PFM_WRITE_PMCS command.

### 5.6.1 Debug registers mappings

An application can access the debug registers through a set of logical PMC registers. Table 5.1 shows the mapping that is used by the interface. The Itanium architecture guarantees there will never be more than 256 PMC registers. As such, the interface allocated a range between 256 and 271 for the debug registers.

The range restriction feature may not be implemented on all Itanium PMU models. The interface guarantees that this range is dedicated for the debug registers. When the feature is not present, PMC256-PMC271 are not implemented.

It is possible to use PFM_GETINFO_PMCS to retrieve the mapping in the other direction, i.e., from PMC to IBR or DBR. See section 3.1.13 for more details.

| debug register | PMC register |
|---|---|
| IBR0 | PMC256 |
| IBR1 | PMC257 |
| IBR2 | PMC258 |
| IBR3 | PMC259 |
| IBR4 | PMC260 |
| IBR5 | PMC261 |
| IBR6 | PMC262 |
| IBR7 | PMC263 |
| DBR0 | PMC264 |
| DBR1 | PMC265 |
| DBR2 | PMC266 |
| DBR3 | PMC267 |
| DBR4 | PMC268 |
| DBR5 | PMC269 |
| DBR6 | PMC270 |
| DBR7 | PMC271 |

Table 5.1: Mapping of the debug registers to PMC for the Itanium Processor Family.

### 5.6.2 Interactions with debugging

For code debug registers, if the `ibr.x` is set, then the debug register pair is not used by the PMU and is considered an active debug breakpoint. Conversely, if `ibr.x` is cleared, the breakpoint is considered inactive but its content is valid for range restriction. The same behavior exists for data debug registers. The `dbr.r` and `dbr.w` field must be cleared for the register to be used by the PMU otherwise it is considered as an active breakpoint. In other words the PMU implementation allows a thread to be debugged and monitored with range restriction at the same time, subject to both subsystem using distinct debug registers.

However, there are certain operating system issues which limit this degree of freedom, in particular, the existing `ptrace` interface used to setup breakpoints. It is not possible to modify this interface used by debuggers. It is perfectly valid for a debugger to clear `ibr.x` or `dbr.r` or `dbr.w` to inactivate a breakpoint. But that should not be interpreted as a valid PMU range restriction. Conversely, the perfmon implementation cannot simply clear all debug registers as soon as it detects that they are used for range restrictions.

To prevent picking up stale debug registers state for range restriction, the interface ensures that it is not possible to use the debug registers for both debugging and performance monitoring at the same time. Such restriction applies to a system-wide context as well. It this case, the following rules are used:

- the PFM_LOAD_CONTEXT command fails if any thread across the entire system is using the debug registers for debugging. This constraint is necessary because processes can migrate from one CPU core to another.
- for a *loaded* context, writing a debug register with PFM_WRITE_PMCS fails if there is at least one thread using the debug registers for debugging across the entire system.

In both cases, if the operating system can detect that the debug registers are no longer used for debugging, then it becomes possible to use range restrictions.

### 5.6.3 Using PFM_WRITE_PMCS with the debug registers

The command is used just like for an actual PMC register, see section 3.1.2 for details. However the `pfarg_pmc_t` data structure is used as follows:

- `reg_num` : the PMC register index (in the range [256:271])
- `reg_set` : the event set for the PMC register.
- `reg_value` : the *raw* value of the corresponding debug register.
- `reg_flags` : there is no specific flag supported on input for a debug register. The bits which are not defined are *reserved* and must be cleared. It is only possible to set bits that are defined. Upon return, the field may be updated to reflect possible error conditions (see section 3.1.2).
- `reg_smpl_pmds` : ignored for debug registers.
- `reg_reset_pmds` : ignored for debug registers.
- `reg_smpl_eventid` : ignored for debug registers.

The value of a code debug register is checked for PMC257, PMC259, PMC261and PMC263, such that `ibr.x` is cleared. This is necessary to avoid using the interface to set rogue code breakpoints. Similarly, the value of a data debug register is checked PMC265, PMC267, PMC269 and PMC271 such that `dbr.r` and `dbr.w` bits are cleared.

## 5.7 Calling *perfmonctl()* from signal handlers

The interface supports *perfmonctl()* calls made from POSIX signal handler. It is possible to create, program, control and destroy a context from a signal handler. On an Itanium system, however, there is a small difficulty for certain commands due to the behavior of signal handler with regards to the machine state. The Processor Status Register, `psr`, may be modified during the execution of the handler. However, it is systematically restored upon return from that handler. This means that if an application modifies the `psr` in the handler, these modifications are lost outside of it. This affects the behavior of a per-thread context for the following commands:

- PFM_START: this command sets the user level `psr.up` bit.
- PFM_STOP : this command clears the user level `psr.up` bit.
- PFM_UNLOAD: this command stops monitoring and therefore invokes PFM_STOP internally. As such it clears `psr.up`.
- *close()* : this commands stops monitoring and therefore it may call PFM_UNLOAD internally. As such it may clear `psr.up`.
- `rum/sum`: when insecure monitoring is enabled, these instructions toggle the value of `psr.up`.

For all these, the modification to `psr` is lost when execution resumes after the signal.

In order for the modification to survive beyond the signal handler, it is necessary to propagate it using the `sigcontext` argument of the handler. This data structure contains part of the machine state that the user can modify. Upon return from the handler, the machine state described in the

`sigcontext` is copied back to the user level machine state. As such, it is possible to propagate some of the modifications to the *normal* execution of the program. The modifications to `psr.up` must be replicated in the `sc_um`, the *user mask*, field of the `sigcontext` structure. The `psr.up` bit corresponds to bit 2 of `sc_um`.

# Chapter 6

# Future extensions

In this chapter, we describe the extensions that are considered for the next revisions of the interface.

## 6.1 Alternative system call interface

As alluded to in section 3.1, for some operating systems, such as Linux, it is preferable to decompose the commands of the interface into multiple system calls. This approach offers several advantages such as:

- better type-checking of the parameters
- possibly faster implementation, because no demultiplexing is needed in the kernel

But it also has some disadvantages such as:

- potentially less code reuse and more code duplication
- more difficulties to extend interface while maintaining backward compatibility

In any case, the following sections present the conversion of each command into the corresponding system call. The conversion preserves all functionalities of the original interface.

### 6.1.1 Creation the perfmon context

The PFM_CREATE_CONTEXT command is converted into the following system call:

```
int pfm_create_context(pfarg_ctx_t *ctx, int n);
```

The call behaves exactly as described in section 3.1.1. The second argument `n` must be `1`. It is provided to ensure that in the future we would be able to create multiple contexts in one call, should this become useful. The `pfarg_ctx_t` structure is identical to that described in section 3.1.1.

### 6.1.2 Accessing the PMC registers

The PFM_WRITE_PMCS command is converted into the following system call:

```
int pfm_write_pmcs(int fd, pfarg_pmc_t *pmcs, int n);
```

The call behaves exactly as described in section 3.1.2.

The PFM_GETINFO_PMCS command is converted into the following system call:

```
int pfm_getinfo_pmcs(pfarg_pmcinfo_t *pmcs, int n);
```

The call behaves exactly as described in section 3.1.13.

### 6.1.3 Accessing the PMD registers

The PFM_WRITE_PMDS command is converted into the following system call:

```
int pfm_write_pmds(int fd, pfarg_pmd_t *pmds, int n);
```

The call behaves exactly as described in section 3.1.3.

The PFM_READ_PMDS command is converted into the following system call:

```
int pfm_read_pmds(int fd, pfarg_pmd_t *pmds, int n);
```

The call behaves exactly as described in section 3.1.4.

The PFM_GETINFO_PMDS command is converted into the following system call:

```
int pfm_getinfo_pmds(pfarg_pmdinfo_t *pmds, int n);
```

The call behaves exactly like the PFM_GETINFO_PMDS command, see section 3.1.14.

### 6.1.4 Starting and stopping monitoring

The PFM_START command is converted into the following system call:

```
int pfm_start(int fd, pfarg_start_t *start);
```

The `start` argument is optional, when not used the parameter must be `NULL`. The call behaves exactly as described in section 3.1.5.

The PFM_STOP command is converted into the following system call:

```
int pfm_stop(int fd);
```

The call behaves exactly like the PFM_STOP command, see section 3.1.6.

### 6.1.5 Attaching and detach a context

The PFM_LOAD_CONTEXT command is converted into the following system call:

```
int pfm_load_context(int fd, pfarg_load_t *load);
```

Only one context can be loaded at a time. The call behaves exactly as described in section 3.1.7.
The PFM_UNLOAD_CONTEXT command is converted into the following system call:

```
int pfm_unload_context(int fd);
```

The call behaves exactly as described in section 3.1.8.

### 6.1.6 Resuming monitoring

The PFM_RESTART command is converted into the following system call:

```
int pfm_restart(int fd);
```

The call behaves exactly as described in section 3.1.9.

### 6.1.7 Operating on event sets

The PFM_CREATE_EVTSETS command is converted into the following system call:

```
int pfm_create_evtsets(int fd, pfarg_setdesc_t *setdesc, int n);
```

The call behaves exactly as described in section 3.1.10.
The PFM_DELETE_EVTSETS command is converted into the following system call:

```
int pfm_delete_evtsets(int fd, pfarg_setdesc_t *setdesc, int n);
```

The call behaves exactly as described in section 3.1.11.
The PFM_GETINFO_EVTSETS command is converted into the following system call:

```
int pfm_getinfo_evtsets(int fd, pfarg_setdesc_t *setdesc, int n);
```

The call behaves exactly as described in section 3.1.12.

### 6.1.8 Configuring the perfmon interface

The PFM_GET_CONFIG command is converted into the following system call:

```
int pfm_get_config(pfarg_config_t *cfg);
```

The call behaves exactly like the PFM_GET_CONFIG command, see section 3.1.17.
The PFM_SET_CONFIG command is converted into the following system call:

```
int pfm_set_config(pfarg_config_t *cfg);
```

The call behaves exactly like the PFM_SET_CONFIG command, see section 3.1.16.

## 6.2 Command Extensions

### 6.2.1 The PFM_REGFL_NO_64BIT_EMUL flag

The PFM_REGFL_NO_64BIT_EMUL flag would be added to the PFM_WRITE_PMCS command. It would indicate that no 64-bit emulation is necessary on the associated counting PMD register. This can be useful if emulation is not necessary for certain measurements. This flag is ignored for non counting PMD registers.

### 6.2.2 The PFM_REGFL_READ_RESET flag

The PFM_REGFL_READ_RESET flag would be added to the PFM_READ_PMDS command. It would force a *long reset* of the PMD register during a PFM_READ_PMDS command. The flag is useful when polling the value at some interval of time. It would avoid having to issue a PFM_WRITE_PMDS command to reset the counter to zero. The PMD registers in the `reg_smpl_reset` field would also be reset.

### 6.2.3 The PFM_SETFL_EXCL_KERNEL_ONLY_THREADS flag

The PFM_SETFL_EXCL_KERNEL_ONLY_THREADS flag would be passed when the event set is created or changed. It would exclude all kernel only threads from active monitoring for a system-wide context. It would not be supported for pre-thread contexts. A kernel thread is defined as a thread created inside the kernel which has no user level machine state, i.e. lives only at the most privileged execution level. Such threads, sometimes called *kernel daemons*, are typically used to process kernel background jobs, such as flushing the buffer cache. The most famous kernel thread is the *idle thread* which runs when nothing else can. The nature and numbers of kernel threads depends on the kernel and potentially from the architecture. As such, it may be possible that for some kernels there would be nothing to exclude.

The effects of the flag are identical to that of the PFM_SETFL_EXCL_IDLE flag described in this document. It is simply more generic and therefore it is mutually exclusive with it.

## 6.3 Sampling support

### 6.3.1 Double-buffering sampling format

The default sampling format could be extended to support double-buffering. Such format would be implemented as a separate kernel module with its own UUID. This is motivated by the fact that:

- backward compatibility must be maintained for the default sampling format
- to keep the code fairly simple.

The buffer area is split into to two halves. Samples are written to the first half. When it becomes full, a notification is sent but monitoring remains active and samples are written in the second half of the buffer. The motivation behind *double-buffering* is to minimize *blind spots*. This is not necessarily bullet-proof and there are some side effects. For instance, in system-wide mode, the risk is to monitor

the performance tool itself as it is processing its half of the buffer. However, it is fairly easy to filter out the samples coming from the tool by using the `pid` stored in each sample header.

Compared to the default sampling format, the buffer header would be quite different to maintain information about each half. The sample header would remain identical to the one used by the default sampling format.

Once the first half becomes full, a notification is sent to the controlling process but monitoring is not masked. Overflowed PMD registers and associated PMD registers of interest are reset using their *short* reset periods. Then samples are written into the second half. Once the tool is done processing its half, it invokes the PFM_RESTART command which invokes the `fmt_restart` call-back. The first half is marked as free. When the second half becomes full, a notification is sent and the format switches to the first half.

In case of the tool does not invoke the PFM_RESTART command before the other half fills up there are two possible modes:

- cycle in the half of the buffer *owned* by the format
- saturate

The default behavior is to cycle in the half owned by the format. This can be altered with a format-specific flag:

- DOUBLE_FMT_MASK_WHEN_FULL: this flag forces saturation. Monitoring is masked until a controlling thread invokes the PFM_RESTART command.

This sampling format would be present by default in the kernel. Extensions to more than more than 2-way are possible.

### 6.3.2 Support for correlation of samples

In system-wide mode and for sampling measurements, it is very useful to correlate the information recorded in a sample to the actual program that generated it. In particular, it is interesting to correlated the `pid` to an actual binary to possibly gain access to source level information.

The default sampling format does not currently store enough information to make the correlation possible.

On Linux, tools may, instead, use the information provided by the `/proc` file-system. A common technique is to take snapshots of the `/proc` tree whenever there is an overflow notification or at some interval of time. The `/proc` snapshot contains information about each running process including names and mappings of shared libraries. Yet the issue with this technique is that it may miss short-lived processes which may generate samples. Depending on the workload this may be a significant problem to interpret the samples.

The default format could be extended, via a new format or possibly with a new flag to include sample correlation information. On Linux, it may be possible to use some of the existing hooks on *exec()* and *mmap()* to store useful information. It may also be possible to leverage the correlation support implemented by the OProfile [3] subsystem.

## 6.4 Extensions specific to the Itanium Processor Family

In this section, we describe the potential extensions that are specific to the Itanium Processor Family (IPF).

### 6.4.1 The PFM_ITA_FL_PRIV_MONITORS flag

As described in section 5.2, the interface selects, by default, *user* monitors for all per-thread contexts and *privileged* for all system-wide contexts. This is good enough for 90% of the measurements.

For very specific measurements, it is beneficial to use *privileged* monitors for a per-thread session. In particular it is interesting when privilege level 0 execution must be broken down between interrupt and non-interrupt triggered. This could easily be achieved by leveraging the `dcr.pp` and `psr.pp` mechanism whereby `psr.pp` receives the value of `dcr.pp` on interrupt.

This new flag, PFM_ITA_FL_PRIV_MONITORS, would override the default behavior of the interface and enable the use of *privileged* monitors for a per-thread context. In this mode, it becomes possible to enable the PFM_ITA_SETFL_EXCL_INTR or PFM_ITA_SETFL_INTR_ONLY flags on an event sets to tweak interrupt level monitoring.

There is no real benefit in using *user* monitors for a system-wide context, as such it is not possible to override the use of *privileged* monitors for this type of context.

## 6.5 Support for PMU preemption

As alluded to in the description of the PFM_LOAD_CONTEXT command, when the PMU cannot be shared between system-wide and per-thread contexts, the interface enforces mutual exclusion.

However exclusion may be too strong a policy for certain environments where a class of users or applications would be denied access to the PMU resource. When the PMU is needed for temporary measurements, that may be acceptable. However it is likely that future applications and managed runtimes, such as Java, will use the PMU to collect profiling information to adjust just-in-time optimizations each time they run. It would not be possible to deny them access to the PMU just because a system-wide monitoring tool is running in the background all the time.

The interface must provide some kind of *preemption* mechanism whereby an application could preempt the PMU from another one. The problem with this approach is that some tools and measurements may rely on the fact that they *effectively* run during a known period of time. As such, the preemption must be implemented as a *cooperative preemption* where an application indicates that it accepts to be preempted for some time. In exchange, it would receive detailed information when its access is restored, allowing the measurement to account for the *blind spot*.

# Chapter 7

# References

We would like to thank Curt Wolgemuth, Eric Gouriou, Dan Truong, David Mosberger, Dick Fowles, Jim Callister, Brad Chen, Hugh Caffey, and Bryan Wilkerson for their feedback on this interface specification.

# Bibliography


[1] David F. Carta. Two fast implementations of the minimal standard random number generator. *Com. of the ACM*, 33(1):87–88, 1990. http://doi.acm.org/10.1145/76372.76379.

[2] J. Anderson et al. Continuous profiling: Where have all the cycles gone, 1997. http://citeseer.ist.psu.edu/anderson97continuous.html.

[3] John Levon et al. Oprofile. http://oprofile.sf.net/.

[4] Robert Cohn et al. The PIN tool. http://rogue.colorado.edu/Pin/.

[5] Forschungszentrum Juelich GmbH. The Performance Counter Library (pcl). http://www.fz-juelich.de/zam/PCL/PCLcontent.html.

[6] Hewlett-Packard Company. The Caliper performance analyzer. http://www.hp.com/go/caliper/.

[7] Intel. *Intel Itanium Processor Reference Manual for Software Development and Optimization*, November 2001. http://www.intel.com/design/itanium/documentation.htm.

[8] Intel. *The IA-64 Architecture Software Developer's Manual*, October 2002. http://www.intel.com/design/itanium/documentation.htm.

[9] Intel. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, April 2003. http://www.intel.com/design/itanium/documentation.htm.

[10] Intel. *IA-32 Intel Architecture Software Developers' Manual: System programming Guide*, 2004. http://developer.intel.com/design/pentium4/manuals/index_new.htm.

[11] Intel Corp. The VTune™ performance analyzer. http://www.intel.com/software/products/vtune/.

[12] Mikael Pettersson. the Perfctr interface. http://user.it.uu.se/ mikpe/linux/perfctr/.

[13] Alex Tsariounov. The Prospect monitoring tool. http://prospect.sf.net/.

[14] University of Tenessee, Knoxville. Performance Application Programming Interface(PAPI) project. http://icl.cs.utk.edu/papi/.

[15] University of Tenessee, Knoxville. Performance Application Programming Interface(PAPI) project. http://icl.cs.utk.edu/papi/.

[16] University of Wisconcin, Madison. The Paradyn project. http://www.paradyn.org/index.html.

[17] POSIX working group. *Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API), Amendment 2: Threads Extension (C Language)*. IEEE/ANSI Std 1003.1c-1995, 1995.