Calloutng: a new infrastructure for timer facilities in the FreeBSD kernel

Davide Italiano
The FreeBSD Project
davide@FreeBSD.org

Alexander Motin
The FreeBSD Project
mav@FreeBSD.org

ABSTRACT

In BSD kernels, timers are provided by the callout facility, which allows to register a function with an argument to be called at specified future time. The current design of this subsystem suffer of some problems, such as the impossibility of handling high-resolution events or its inherent periodic structure, which may lead to spurious wakeups and higher power consumptions. Some consumers, such as high-speed networking, VoIP and other real-time applications need a better precision than the one currently allowed. Also, especially with the ubiquity of laptops in the last years, the energy wasted by interrupts waking CPUs from sleep may be a sensitive factor. In this paper we present a new design and implementation of the callout facility, which tries to address those long standing issues, proposing also a new programming interface to take advantage of the new features.

Categories and Subject Descriptors

[**Operating systems**]; [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Kernel, Timers, C, Algorithms, Data structures, POSIX

1. INTRODUCTION

A certain number of computer tasks are generally driven by timers, which acts behind the scenes and are invisible for user. For example, network card driver needs to periodically poll the link status, or the TCP code needs to know when it is time to start retransmission procedures after acknowledgement not received in time. In FreeBSD [1] and other BSD kernels, the interface that allows programmer to register events to be called in the future takes the name of callout. The design of this subsystem was not modified for years and suffers of some issues, such as measuring time in fixed periods, waking up the CPU periodically on each period even if there are no events to process, inability group together close events from different periods, etc.
The calloutng project aims to create a new timer infrastructure with the following objectives:

- Improve the accuracy of events processing by removing concept of periods
- Avoid periodic CPU wakeups in order to reduce energy consumption
- Group close events to reduce the number of interrupts and respectively processor wakeups
- Keep compatibility with the existing Kernel Programming Interfaces (KPIs)
- Don't introduce performance penalties

The rest of the paper is organized as follows: Section 2 gives a relatively high-level description of the design as originally proposed and some improvements have been done during the years in FreeBSD-land, while Section 3 deeply analyzes the problems present in the current approach. In Section 4 the new design and implementation are explained and its validity is shown via experimental results in the successive section. The work is concluded with some considerations and future directions.

2. STATE OF THE ART

The callout code in FreeBSD is based on the work of Adam M. Costello and George Varghese. A detailed description can be found in [2]. The main idea is that of maintaining an array of unsorted lists (the so-called callwheel, see Figure 1) in order to keep track of the outstanding timeouts in system. To maintain the concept of time, the authors used a global variable called *ticks*, which keeps track of the time elapsed since boot. Historically, the timer was tuned to a constant frequency hz, so that it generated interrupts *hz* times per second. On every interrupt function *hardclock()* was called, incrementing the ticks variable.
When a new timeout is registered to fire at time t (expressed in ticks), it's inserted into the element of the array, addressed by (t *mod* callwheelsize), where callwheelsize is the size of the array and *mod* indicates the mathematical modulo operation. Being the lists unsorted, the insertion operation takes minimal constant time. On every tick, the array element pointed by (ticks *mod* callwheelsize) is processed to see if there are expired callouts, and in such cases the handler functions are called. If callwheelsize is chosen to be comparable to the number of outstanding callouts in the system, lists will be (in average) short.
The main advantage of using this strategy relies on its simplicity: the callout code has O(1) average time for all the operations, and O(1) worst case for most of them. In order to process events, the code needs only to read a global kernel variable, which is very cheap, in particular compared to hardware timecounter, which can be quite expensive. Also, this design requires neither specific hardware architecture

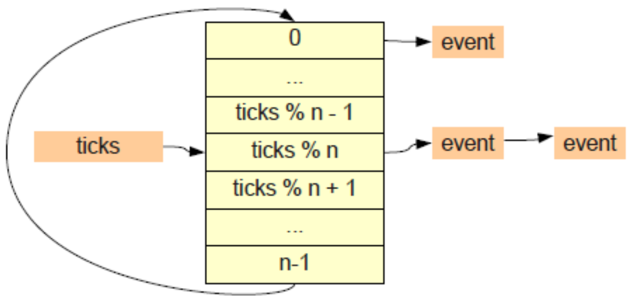

Figure 1: The callwheel data structure

nor specific OS infrastructure. It needs only a timer (no restrictions, because almost any timer has the ability to periodically generate interrupts or simulate this behaviour), and a subsystem for handling interrupt.

While the current FreeBSD implementation shares basic concepts with the mechanisms described above, it evolved quite a bit during the years, introducing substantial improvements in term of performances/scalability and specifying a different programming interface. About the former, the single callwheel was replaced by a per-CPU callwheels, introducing respective migration system (2008).
For what concerns the latter, it is provided via an additional function that consumers can use to allocate their callouts, namely *callout_init()* and its locked variants (*callout_init_mtx()*, *callout_init_rw()* ...). This way the callout subsystem doesn't have to deal internally with object lifecycle, passing this responsibility to the consumer.
Also, KPI was extended, in order to reflect the introduction of per-CPU wheels, adding a new *callout_reset_on()* function, that allows to specify a CPU to run the callout on. It worth to say that the callout processing threads are not hard bound to each CPU, so they can't be starved into not processing their queues. They are *medium* bound by ULE and probably tend to run where they are scheduled on 4BSD but there is no guarantee. The ULE will run them on their native CPU if it's running something of lower priority or idle. If it's running something of greater priority, it will look for a more idle CPU. This prevents the softclock thread from being starved by other interrupt thread activity.

3. PROBLEMS WITH CURRENT IMPLEMENTATION

. The constant frequency *hz* determines the resolution at which events can be handled. On most FreeBSD machines this value is equal to one thousand, which means that one ticks corresponds to 1 millisecond. This way, even a function which specify timeout using accurate time intervals with resolution of one nanosecond (e.g. nanosleep) will see this interval rounded up to the nearest tick, i.e. it will be like the resolution is one millisecond. As soon as many callers require interval to be no shorter than specified, the kernel has to add one more tick to cover possibility that present tick is almost over. As result, the best we can get is resolution of one millisecond with one millisecond of additional latency.
Also, independently from the time for which the next event is scheduled, CPU is woken up on every tick to increment the global ticks variable and scan the respective bucket of the callwheel. These spurious and useless CPU wakeups directly translate in power dissipation, which e.g. from a laptop user point of view results in a reduction of the on-battery time.
Stiil talking about power consumption, the actual mechanism is unable to coalesce or defer callouts. Such a mechanism could be useful for istance in low power modes, where the system could postpone check for a new CD in the drive if CPU is sleeping deep at the moment.
The last but not the least, all the callouts currently run from software interrupt thread context. That causes additional context switches and in some cases even wakeup of other CPUs.

4. A NEW DESIGN

In order to address the first of the present problems, the interfaces to services need some rethinking. For what concern the kernel, all the Kernel Programming Interfaces dealing with time events are built around the concept of 'tick'. All consumers specify their timeouts via the callout_*() interface as relative number of ticks. 'ticks' variable is a 32-bit integer, and it is not big enough to represent time with resolution of microseconds (or nanoseconds) without quickly overrunning (overflowing). Change is definitely needed there in order to guarantee higher granularity.
For what concern the userland, the Application Programming Interfaces (APIs) and data types are standardized by Portable Operating System Interface (POSIX), so they cannot be changed without losing compliance. Luckily, most of the functions provide variants with acceptable degree of precision.
Let's take, as an example, the nanosleep() system call. This service asks the kernel to suspend the execution of the calling thread until either at least the time specified in the first argument has elapsed, or the call will be aborted by signal delivered to the thread. The function can specify the timeout up to a nanosecond granularity.
Other services (e.g. select()), which monitor one or more file descriptor to see if there's data ready, can limit the amount of time they monitor. This time is specified in one function argument. Differently from what happen in the previous case, the granularity is that of microseconds. Unfortunately, as long as all those system calls dealing with timeouts rely on callout(9) at the kernel level, they're limited by the precision offered by the kernel. So, as first step a new data type is needed to represent more granular precision.
From a system-wide perspective, FreeBSD has the following data types to represent time intervals:

```
struct timespec {
    time_t tv_sec;
    long tv_nsec;
};

struct timeval {
    time_t tv_sec;
    long tv_usec;
};

struct bintime {
    time_t bt_sec;
    uint64_t bt_frac;
};
```

The first type allows to specify time up to microseconds granularity, while the second up to nanosecond one. The common part between the two representation is that they're pseudo-decimal. Conversely, the bintime structure is a binary number, and this has some advantages, first of all making easier mathematical operations on the type. Depending on the underlying architecture, these struct have different sizes. In particular, they range respectively from 64 to 128 bit (for timeval and timespec) and from 96 to 128 bit (for bintime) [3].
The binary nature of bintime could make it suitable for our purposes (in fact it was originally chosen as default data type), but there are some problems that should be considered. First of all, using 128 bit (on amd64, probably the most widely spread platform) is overkill, because at best hardware clocks have short term stabilities approaching 1e-8, but likely as bad as 1e-6. In addition, compilers don't provide a native *int128_t* or *int96_t* type.
The choice made was that of using A 32.32 fixed point format, fitting it into an *int64_t*. Not only this makes mathematical operations and comparison trivial, but allows also to express 'real world' units for time such as microseconds or minutes trivially.

```
typedef sbintime_t   int64_t;
#define SBT_1S  ((sbintime_t)1 << 32)
#define SBT_1M  (SBT_1S * 60)
#define SBT_1MS (SBT_1S / 1000)
#define SBT_1US (SBT_1S / 1000000)
#define SBT_1NS (SBT_1S / 1000000000)
```

Now that the type is decided, the programming interface in the kernel requires to be adapted. Changing existing functions arguments is considered discouraging and intrusive, because it creates an huge KPI breakage for third-party software, considering the popularity of callout in many kernel subsystems, including device drivers. So it's been decided to maintain the old set of functions, introducing a new set of alternative functions, which rely on the aforementioned chosen type to represent time. Other than changing a field in order to specify the expiration time, the new field has been introduced in order to specify precision tolerance. This field may be provided by the consumer to specify a tolerance interval it allows for the scheduled callout. In other words, specifying a non-zero precision value, the consumer gives an hint to the callout backend code about how to group events. To sum up, the resulting KPI takes the following form:

```
int callout_reset_sbt_on (..., sbintime_t
     sbt, sbintime_t precision, int flags
    );
```

Some functionalities, e.g. the precision-related bits, may be useful also in the old interface. A new KPI for old customers also has been proposed, so that they can take advantage of the newly introduced features.

```
int callout_reset_flags_on (..., int
    ticks, ..., int flags);
```

Thus, an extension was made to the existing condvar(9), sleepqueue(9), sleep(9) and callout(9) KPIs by adding to them functions that take an argument of the type sbintime_t instead of int (ticks to represent a time) and a new argument to represent desired event precision. In particular:

```
int cv_timedwait_sbt (..., sbintime_t sbt
    , sbintime_t precision);
int msleep_sbt (..., sbintime_t sbt,
    sbintime_t precision);
int sleepq_set_timeout_sbt (...,
    sbintime_t sbt, sbintime_t precision)
    ;
```

In parallel to the KPI changes, there's need to adapt the callout structure in order to store the added bits. The struct callout prior to changes had the following form:

```
struct callout {
    ...
    int c_time;
    void  *c_arg;
    void  (*c_func)(void *);
    struct lock_object *c_lock;
    int c_flags;
    volatile int c_cpu;
};
```

As the reader can easily imagine, the c_time field, which stores the expiration time need to be replaced to sbintime_t in order to keep track of the new changes. Similarly, to store precision, a new field need to be added. Although the callout interface doesn't require direct access to the elements of the 'struct callout', which is an opaque type, clients need to allocate memory for it using the callout_init functions.. The result is that the size of the struct callout is a part of the KBI. The use of larger data types leads to an increase in the size of the structure and inevitably break the existing KBI. This change requires rebuilding of almost all the kernel modules and makes impossible to merge these changes to the existing STABLE branch, because the FreeBSD policy requires (or at least recommends) to preserve binary compatibility among different releases with the same major number (in this case FreeBSD-9).

Changes to the backend data structure

During the initial planning stage we've analyzed possibility of using different tree-based data structures to store events. They can provide much better results on operation of fetching the first event, but have more complicated insertion and removal operation. As soon as many events in system are cancelled and never really fire (for example, device timeouts, TCP timeouts, etc.), performance of insert/remove operations is more important. But tree-like algorithms typically have O(log(n)) complexity for such operations, while callwheel has only O(1). Also tree-like algorithms may require memory (re-)allocation during some insert operation. That is impossible to do with present callout locking policy, that allows to use it even while holding spin-locks.
As result, the underlying design has been maintained, but slightly refreshed. The wheel is going to use as hash key subset of time bits, picking some of them from the integer part of sbintime_t and some from the fractional, so that the key changes sequentially approximately every millisecond. The new hash function is translated into code as follows:

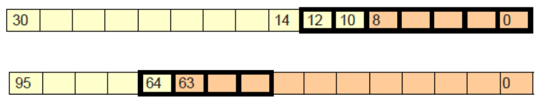

Figure 2: Old and new callout hash keys compared

```
#define CC_HASH_SHIFT   10

static inline int
callout_hash(sbintime_t sbt)
{

    return ((int)(sbt >> (32 -
        CC_HASH_SHIFT))) & callwheelmask;
}
```

where the last bitwise AND operation realizes modulo being callwheel size constrained to be a power-of-two. The compared behaviour of the hash in the two cases is shown in Figure 2. The following motivations affect callwheel parameters now:

- The callwheel bucket should not be too big to not rescan events in current bucket several times if several events are scheduled close to each other.
- The callwheel bucket should not be too small to minimize number of sequentially scanned empty buckets during events processing.
- The total number of buckets should be big enough to store (if possible) most of events within one callwheel turn. It should minimize number of extra scanned events from distant future, when looking for expired events.

As result, for systems with many tick-oriented callouts, bucket size can be kept equal to the tick time. For systems with many shorter callouts it can be reduced. For mostly idle low-power systems with few events it can be increased.

Obtaining the current time

The new design poses a new challenge. The conversion to struct bintime instead of tick-based approach to represent time makes impossible to use low-cost global variable ticks to get the current time. There are two possibilities to get time in FreeBSD: the first, going directly to the hardware, using binuptime() function, the second, using a cached variable which is updated from time to time, using getbinuptime(). It was decided to use a mixed approach: in cases where high precision needed (e.g. for short intervals) use heavy but precise function binuptime(), and use light but with 1ms accuracy getbinuptime() function in cases where precision is not essential. This allows to save time by using better accuracy only when necessary.

Accuracy

In particular situations, e.g. in low-power environments, for the correct and effective use of resources, events need to have information about the required accuracy. The new functions introduced in kernelspace allow to specify it as absolute or relative value. In order to store it, the callout structure has been augmented by adding a 'precision' field of type struct bintime, which represents the tolerance interval, allowed by the consumer of callout. Unfortunately, this is not possible for the old interface. None of existing functions in kernel space and none functions in userspace have such (or similar) argument, and therefore only an estimation might be done, based on the timeout value passed and other global parameters of the system (e.g. hz value). The default level of accuracy (in percentage, for example) can be set globally via the syscontrol interface.
The possibility of aggregation for events is checked every time the wheel is processed. When the callout_process() function is executed, the precision field is used together with the time field to find a set of events which allowed times overlap, which can be executed at the same time.

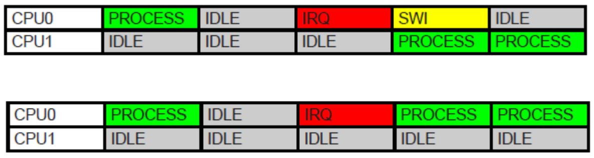

Figure 3: Traditional and SWI-less callout comparison

CPU-affinity and cache effects

The callout subsystem uses separate threads to handle the events. In particular, during callout initialization, a software interrupt threads (SWI) for each CPU are created, and they are used to execute the handler functions associated to the events. Right now, all the callouts are executed using these software interrupts. This approach has some advantages. Among others, that it is good in terms of uniformity and removing many of limitations, existing for code, executing directly from hardware interrupt context (e.g. the ability to use non-spin mutexes and other useful locking primitives). However, the use of additional threads complicates the job of the scheduler. If the callout handler function will evoke another thread (that is a quite often operation), the scheduler will try to run that one on a different CPU, as long as the original processor is currently busy running the SWI thread. Waking up another processor has significant drawbacks. First of all, waking up another processor from deep sleep might be a long (hundreds of microseconds) and energy-consuming process. Besides, it is ineffective in terms of caches usage, since the thread scheduled on another CPU, which caches unlikely contain any useful data.
Let's see an example. Some thread uses the tsleep(9) interface to delay its execution. tsleep(9) at a lower-level relies on the sleepqueue(9) interface, which uses callout(9) to awake at the proper time. So what's happening in the callout execution flow: the SWI thread is woken up for the single purpose of waking up the basic thread! And as described above, very likely wake it up on another processor!
The analysis shows, that example above is relevant to all customers using sleep(9), sleepqueue(9) or condvar(9) KPIs, and respectively all userland consumers which rely on those

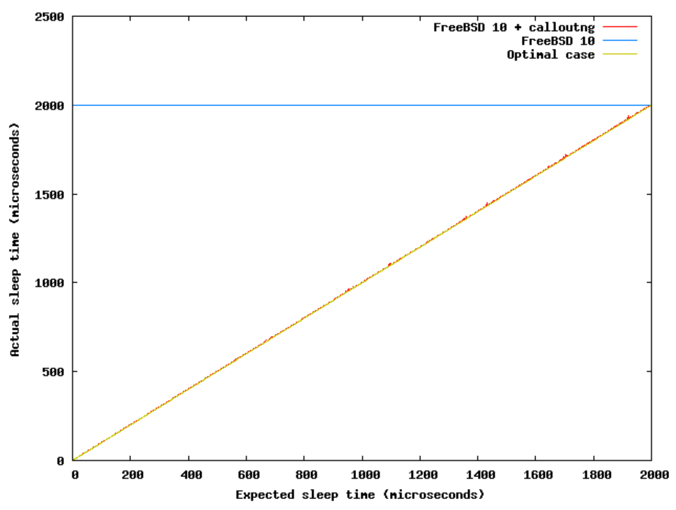

Figure 4: Calloutng performances on amd64: absolute precision

primitives: poll(2), select(2), nanosleep(2), etc.
In order to solve this problem the mechanism of direct execution has been implemented. Using the new C_DIRECT_EXEC flag the callout(9) consumer can specify that event handler can be executed directly in the context of hardware interrupt. This eliminates the use of SWI for this handler by the cost of enforcing additional constraints. According to our tests, use of this technique, where applicable, allows significantly reduce resource consumption and latency of the event processing, as well as, in theory, improve efficiency and reduce pollution of the CPU caches.

5. EXPERIMENTAL RESULTS

When using TSC timecounter and LAPIC eventtimer, the new code provides events handling accuracy (depending on hardware) down to only 2-3 microseconds. The benchmarks (real and synthetic) shown no substantial performance reduction, even when when used with slow timecounter hardware. Using the direct event handlers execution shown significantly increased productivity and accuracy, while also improving system power consumption.
The following tests have been run measuring the actual sleep time of the usleep() system call, specifying as input a sequential timeout starting from one microseconds and increasing it. The system used was Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz, equipped with 8GB of RAM. The couple LAPIC+TSC has been used. In the graphs the x-axis represent the input passed to usleep() and the y-axis represent the real sleep time, both expressed in microseconds. Figure 4 shows how calloutng behaves when the users requires absolute precision, while Figure 5 show the result when the precision tolerance has been set to 5 percent, in order to reduce the number of interrupts and exploit the benefits of aggregation. The same two tests have been repeated also on ARM hardware, SheevaPlug – Marvell 88F6281 1.2GHz, 512MB RAM, and the corresponding results are shown in Figure 6 and Figure 7.

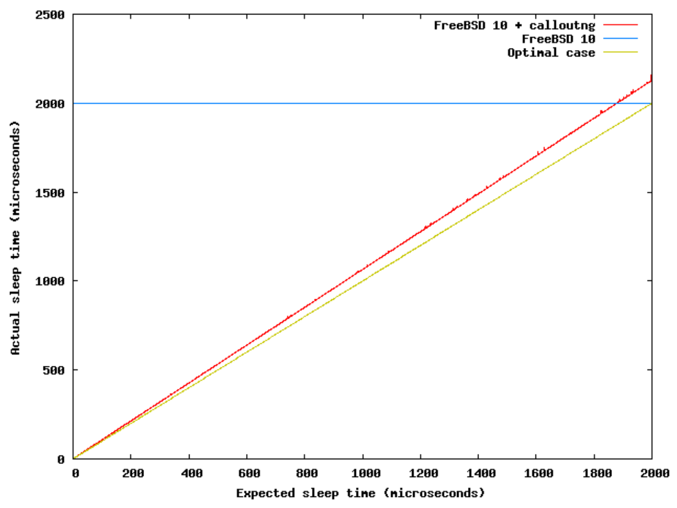

Figure 5: Calloutng performances on amd64: 5 percent precision tolerance

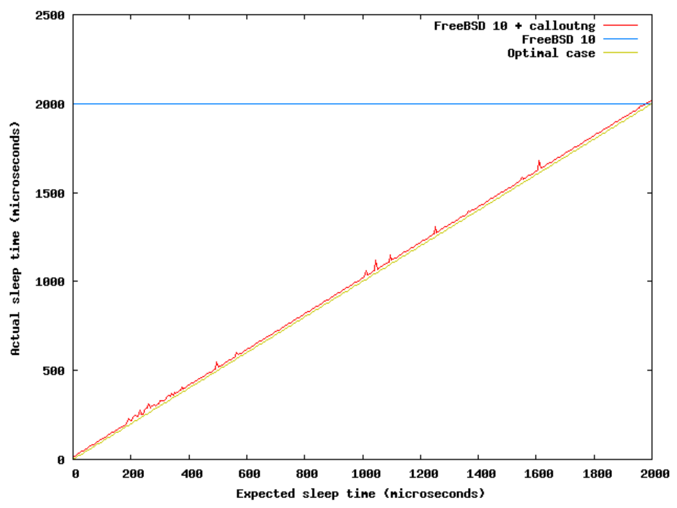

Figure 6: Calloutng performances on ARM: absolute precision

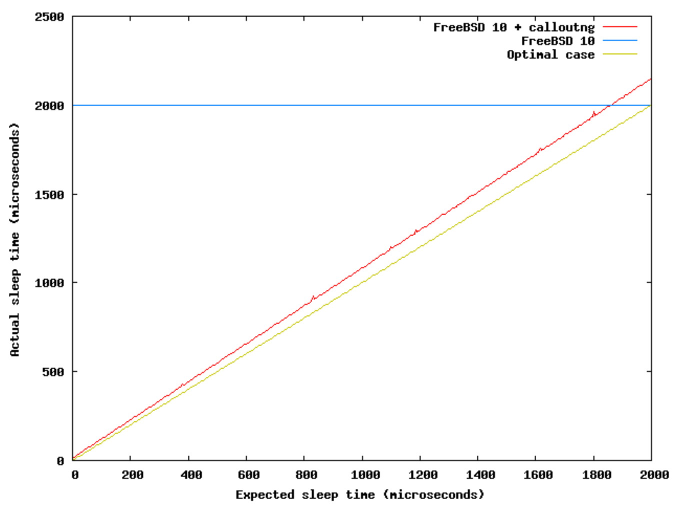

Figure 7: Calloutng performances on ARM: 5 precent precision tolerance

6. CONCLUSIONS

The code written as part of this project going to be the part of the forthcoming FreeBSD 10.0 release. Further wider deployment of newly implemented APIs in different kernel subsystems will provide additional benefits. Some user-level APIs could be added later to specify events precision for the specified thread or process.

7. ACKNOWLEDGMENTS

We would like to thank Google, Inc. (Google Summer of Code Program, 2012 edition) and iXsystems, Inc. for sponsoring this project, as well as the FreeBSD community for reviewing, discussing and testing the changes we made.

8. REFERENCES

[1] The FreeBSD Project. `http://www.freebsd.org/`.

[2] A. M. Costello and G. Varghese. Redesigning the BSD Callout and Timer Facilities. November 1995.

[3] P.-H. Kamp. Timecounters: Efficient and precise timekeeping in SMP kernels. June 2004.