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

Alexander Motin <mav@FreeBSD.org>
Davide Italiano <davide@FreeBSD.org>
What's callout?

- Kernel interface that allows a function (with argument) to be called in the future
- Widely used in FreeBSD (and *BSD in general):
  - TCP retransmission
  - Network card drivers
  - System calls dealing with time
Callout clients (some of them)

```
nanosleep(2)  select(2)  poll(2)
```

```
sleep(9)  sleepqueue(9)  callout(9)
```

```
condvar(9)
```

KERNEL

USERLAND
Current API (userland)

- `int nanosleep(const struct timespec *req, struct timespec *rem);`
- `int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`
- `int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);`
Current KPI (1)

- void sleepq_set_timeout(void *wchan, int timo);
- int cv_timedwait(struct cv *cvp, lock, int timo);
- int msleep(void *chan, struct mtx *mtx, int priority, const char *wmesg, int timo);
- int tsleep(void *chan, int priority, const char *wmesg, int timo);
Current KPI (2)

- void callout_init(struct callout *c, int mpsafe);
- int callout_stop(struct callout *c);
- int callout_reset(struct callout *c, int ticks, timeout_t *func, void *arg);
- int callout_schedule(struct callout *c, int ticks);
Granularity of tick

- *int ticks* is a global kernel variable which keeps track of time elapsed since boot.
- Historically timers generated interrupts *hz* times per second (tunable, generally equals to 1000 on most systems).
- On every interrupt *hardclock()* is called and ticks updated by one unit.
Callwheel data structure

- Array of \( n \) unsorted lists
- \( O(1) \) average time for most of the operations
- Every tick the bucket pointed by ticks \( mod \ n \) is scanned for expired callouts
- SWI scheduled to execute callback function
Recent'ish changes

- Single callwheel replaced by a per-CPU callwheel to improve scalability and performances
- Migration system introduced
- KPI extended:
  - int callout_reset_on(struct callout *c, int ticks, timeout_t *func, void *arg, int cpu)
Current design analysis

• Goodies
  - No hardware assumptions
  - Reading a global variable is cheap

• Drawbacks
  - Intervals rounded to the next tick
  - CPU woken up on every interrupt
  - No way to defer/coalesce callouts
  - All callouts running in SWI context
Calloutng goals

- Improve the accuracy of events removing the 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 KPIs
- Don’t introduce performance penalties
New API/KPI

• Userland services provide a fair enough level of precision (microseconds)
  – They can't be touched at all due to POSIX
• Kernel API built around the concept of tick:
  – Hz = 1000 means 1 millisecond granularity
  – 32-bit tick can't represent microseconds without quickly overflowing
  – Need some re-thinking
New API/KPI

- There are three data-types in FreeBSD to represent time:
  - struct timespec (time_t + long, 64-128 bits, decimal)
  - struct timeval (time_t + long, 64-128 bits, decimal)
  - struct bintime (time_t + uint64_t, 96-128 bits, fixed point)

- Math with bintime is easier, but ...

- 128 bits are overkill
  - Hardware clocks have short term stabilities approaching 1e-8, but likely as bad as 1e-6.
  - Compilers don’t provide a native int128_t or int96_t type.
sbintime_t type

- Think of it as a 'shrinked bintime'
  - 32 bit integer part
  - 32 bit fractional part
- Easily fit in int64_t (readily available in the C language)
- Math/comparisons are trivial
  - SBT_1S ((sbintime_t)1 << 32)
  - SBT_1M (SBT_1S * 60)
  - SBT_1MS (SBT_1S / 1000)
  - if (time1 <= time2)
KPI changes

- Try to avoid breakages
  - int callout_reset_sbt_on (..., sbintime_t sbt, sbintime_t precision, int flags);
  - int callout_reset_flags_on (..., int ticks, ..., int flags);
- Also kernel consumers KPI need to be extended:
  - 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);
KBI: struct callout (before and after)

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

```c
struct callout {
    ...
    sbintime_t c_time;
    sbintime_t c_prec;
    void *c_arg;
    void (*c_func)(void *);
    struct lock_object *c_lock;
    int c_flags;
    volatile int c_cpu;
};
```
Changes to the backend (1)

• Initially considered a switch to a tree-based structure
  – $O(\log n)$ insert/removal impact on overall performances
  – Lots of timeouts frequently rearmed but never fire (e.g. ahci(4))
  – Reallocation during insert difficult/impossible with callout locking policy
• Maintained the wheel and refreshed the code
Changes to the backend (2)

- Hash function revisited to take a subset of bits from integer part of `sbintime_t` and the others from fractional part
- Designed in a way key changes approximately every 4ms
- Rationale behind this choice:
  - 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.
Obtaining current time

- Time passed to callout is not anymore relative but absolute
- Need to know current time
- Two ways to obtain it:
  - `binuptime()`: goes directly to the hardware
  - `getbinuptime()`: read a cached variable updated from time to time
- `sbinuptime()` and `getsbinuptime()` implemented as wrappers to these two functions
Accuracy

• Callout structure augmented
• New KPI specifies a precision argument
• Default level of accuracy for kernel services: estimation based on timeout value passed and other global parameters (hz)
• Tunable using the SYSCTL interface
• Aggregation checked when the wheel is processed:
  - Precision + time fields of callout used to find a set of events which allowed times overlap
CPU-affinity/cache effects

- SWI complicates the job of the scheduler
  - Possibility to wake up another CPU (it may be expensive from deep sleep state)
  - Useless context switch
  - Other CPU caches unlikely contains useful data
- Allow to run from hw interrupt context specifying C_DIRECT flag
  - Eliminates the above problem
  - Enforces additional constraints in locking
CPU-affinity: an example

**SWI context:**

<table>
<thead>
<tr>
<th>CPU0</th>
<th>PROCESS</th>
<th>IDLE</th>
<th>IRQ</th>
<th>SWI</th>
<th>IDLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU1</td>
<td>IDLE</td>
<td>IDLE</td>
<td>IDLE</td>
<td>IDLE</td>
<td>PROCESS</td>
</tr>
</tbody>
</table>

**HWI context:**

<table>
<thead>
<tr>
<th>CPU0</th>
<th>PROCESS</th>
<th>IDLE</th>
<th>IRQ</th>
<th>PROCESS</th>
<th>PROCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU1</td>
<td>IDLE</td>
<td>IDLE</td>
<td>IDLE</td>
<td>IDLE</td>
<td>IDLE</td>
</tr>
</tbody>
</table>
Experimental results (amd64)
Experimental results (arm)