#### 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)









callout(9)

sleep(9)



## 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 time);
- 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)</pre>
  - SBT\_1M (SBT\_1S \* 60)
  - SBT\_1MS (SBT\_1S / 1000)
  - if (time1 <= time2)</pre>



#### **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)

0.0.0

};

struct callout {

#### int c\_time;

void \*c\_arg; void (\*c\_func)(void \*); struct lock\_object \*c\_lock; int c\_flags; volatile int c\_cpu; };

#### 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(lg 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: extimation 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:

CPU0	PROCESS	IDLE	IRQ	SWI	IDLE
CPU1	IDLE	IDLE	IDLE	PROCESS	PROCESS

HWI context:

CPU0	PROCESS	IDLE	IRQ	PROCESS	PROCESS
CPU1	IDLE	IDLE	IDLE	IDLE	IDLE



#### Experimental results (amd64)





#### Experimental results (arm)



