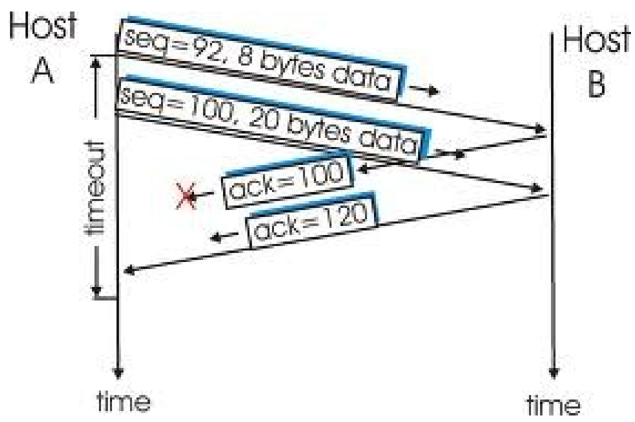
An Overview of the Timing Facilities in the FreeBSD Kernel

Davide Italiano
davide@FreeBSD.org

Institute of Biostructures and Bioimaging
Naples, Italy
April 6, 2013

Timers: why you should care

- Many modern computerized tasks are driven by timers



- Kernel drivers/system calls dealing with time need a mechanism to call functions at later time

Callout(9)

- KPI that allows a function (with argument) to be called in the future
- *Future* time expressed in number of ticks (relative value)

```
void callout_init(struct callout *c, int mpsafe);

int callout_reset(struct callout *c, int ticks,
timeout_t *func,   void *arg);

void callout_stop(struct callout *c);
```

Callout consumers (kernel)

- Kernel API relying more or less directly on callout(9)

```
int msleep(void *chan, struct mtx *mtx,
    int priority, const char *wmesg, int timo);

int cv_timedwait(struct cv *cvp, lock, int timo);


void sleepq_set_timeout(void *wchan, int timo);
int sleepq_timedwait(void *wchan);
```

Callout consumers (userland)

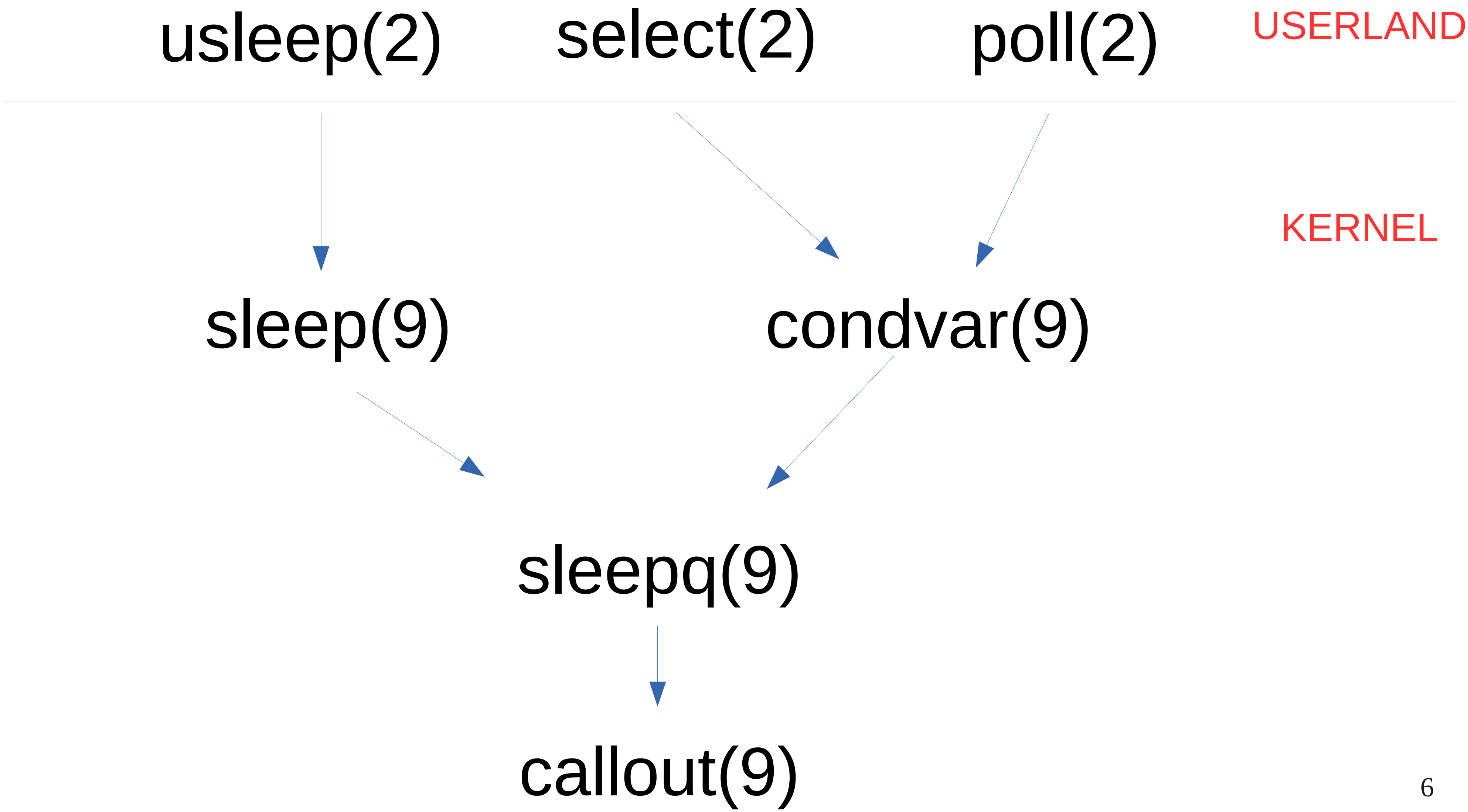
- Userland API relying more or less directly on callout(9)

```
void usleep(unsigned long usec);

int select(int nfds, fd_set *readfds,
    fd_set *wriztefds, fd_set *exceptfds,
    struct timeval *timeout);

int kevent(int kq, const struct kevent *changelist,
    int nchanges, struct kevent *eventlist,
    int nevents, const struct timespec *timeout);
```

Callout consumers (some of them)



Granularity of tick

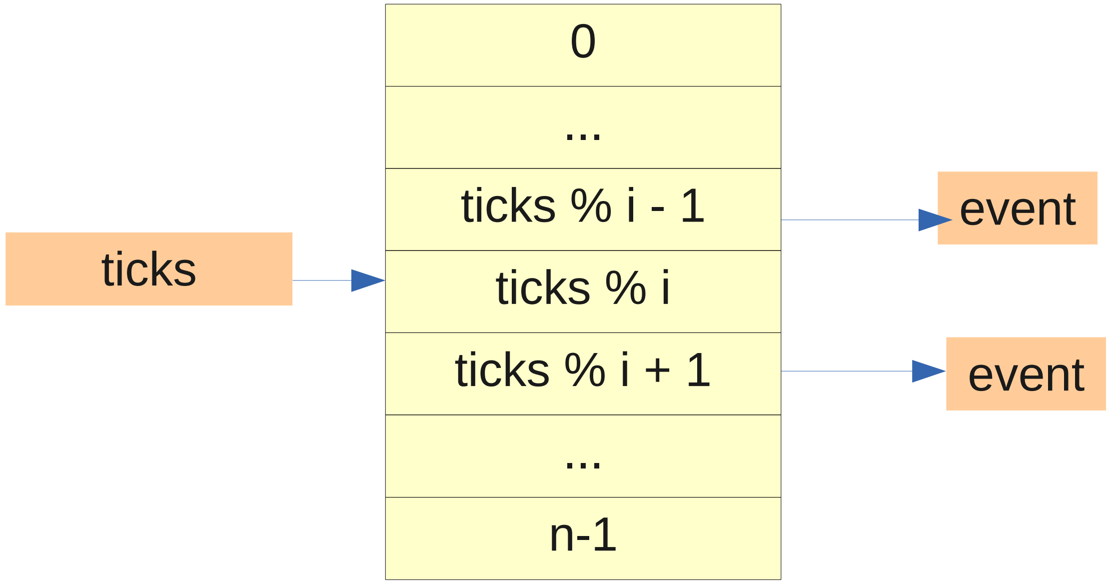
- *int ticks* is a global kernel variable which keeps track of time elapsed since boot
- Two kind of hw timers: periodic/one-shot
- *(Periodic)* timers generated interrupts *hz* times per second (tunable, generally equals to 1000 on most systems)
- On every interrupt *hardclock()* is called and ticks updated by one unit

Granularity of tick (2)

- Hz = 1000 means 1ms resolution
- Intervals rounded to the next tick!
- The fallout? (e.g. for userspace consumers)
  - Process ask to suspend execution for 1 musec
  - usleep(1) is called
  - Process awakened (at least) after 1 millisecond

Current callout(9) implementation

- 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 (even if empty)
- *SWI* scheduled to execute callback function



Some recent'ish changes

- Global callwheel data structure replaced with per-CPU callwheel
  - Scalability/performance greatly improved
- 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
  - Lack of precision
  - CPU woken up every tick
  - No way to defer/coalesce callouts
  - All the callouts run in SWI context

New design

- Improve the accuracy of events removing the concept of periods
- Avoid periodic CPU wakeups in order to reduce energy consumption
- Group events close in time to reduce the number of interrupts/processor wakeups
- Keep compatibility with existing KPI
- Don't introduce performance penalties

New 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:
  - 32-bit tick can't represent microseconds granularity without quickly overflowing
  - Need to switch to another type

New KPI (2)

- There are three 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.

A new type: sbintime_t

- 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)

New KPI proposed

```
int callout_reset_sbt(struct callout *c,
sbintime_t time, sbintime_t precision,
timeout_t *func, void *arg, int flags);
```

```
int callout_reset_sbt_on(struct callout *c,
sbintime_t time, sbintime_t precision,
timeout_t *func, void *arg, int cpu, int flags);
```

```
int callout_reset(struct callout *c, int ticks,
timeout_t *func,   void *arg, int flags);
```

Changes to the backend

- “Tickless“ callwheel:
  - If one-shot timer available, scan buckets in the future to find next event
  - Schedule next interrupt at that time
  - If CPU is idle, wake up every ½ second

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.

Changes to the backend (3)

- 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

Changes to the backend (4)

- Current time: take the best of the two worlds
- For small timeouts, use expensive but precise *binuptime()*
- After a given threshold, use cheap but less precise *getbinuptime()*
- Rationale: if the threshold is carefully chosen, error is bounded by a given percentage

Coalesce/defer events

- 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(3) 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

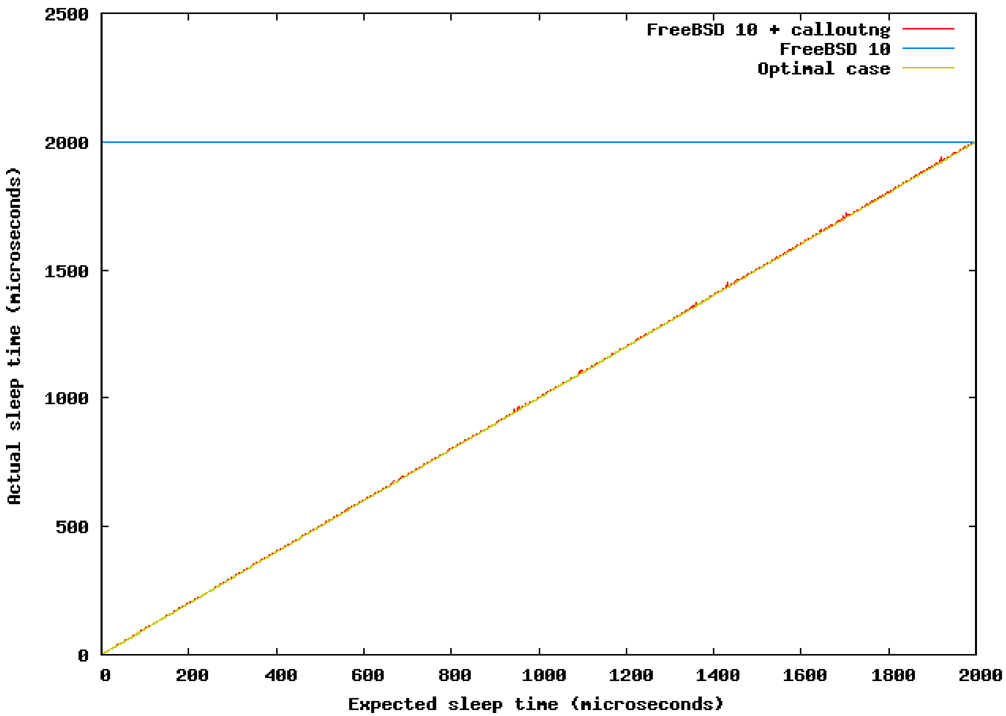
- SWI complicates the job of the scheduler
  - Possibility to wake up another CPU (it may be expensive from deep sleep state)
  - Useless context switch
  - Other CPU caches unlikely contains useful data
- Allow to run from hw interrupt context specifying C_DIRECT flag
  - Eliminates the problem above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

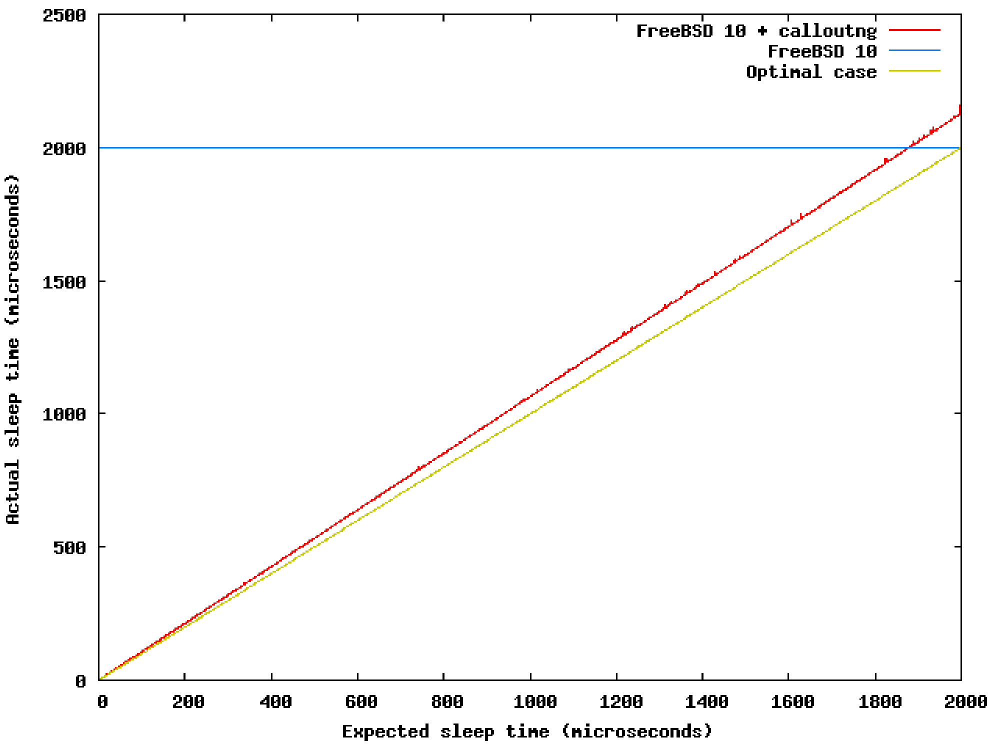
SWI vs direct dispatch

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

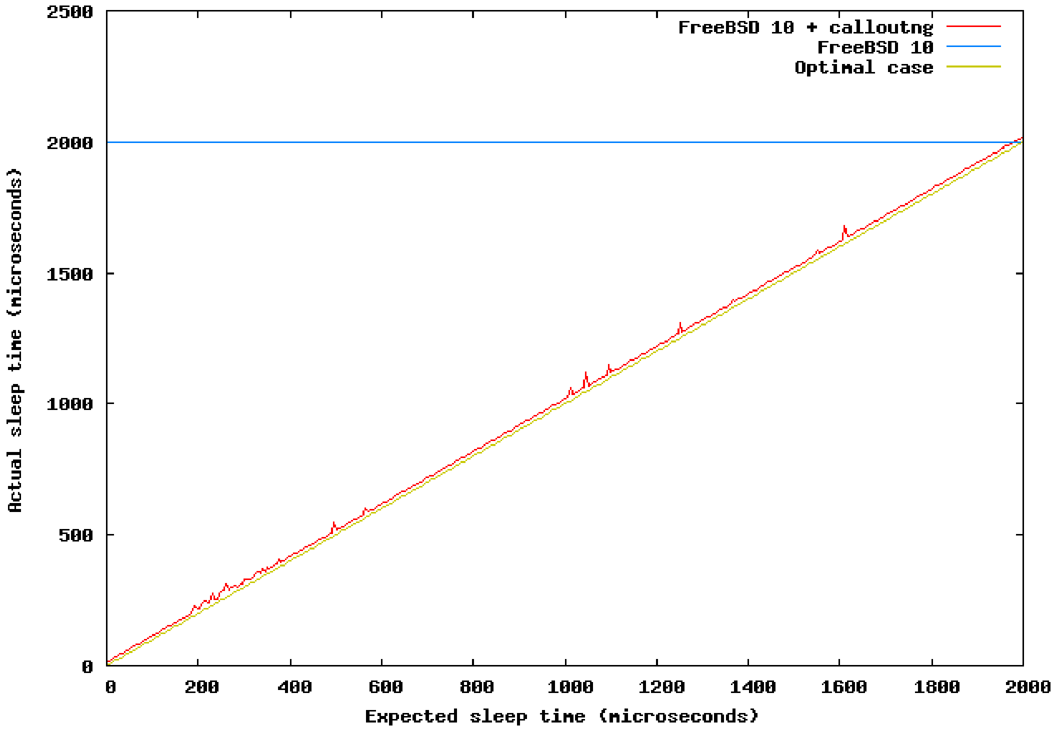
CPU0	PROCESS	IDLE	IRQ	PROCESS	PROCESS
CPU1	IDLE	IDLE	IDLE	IDLE	IDLE

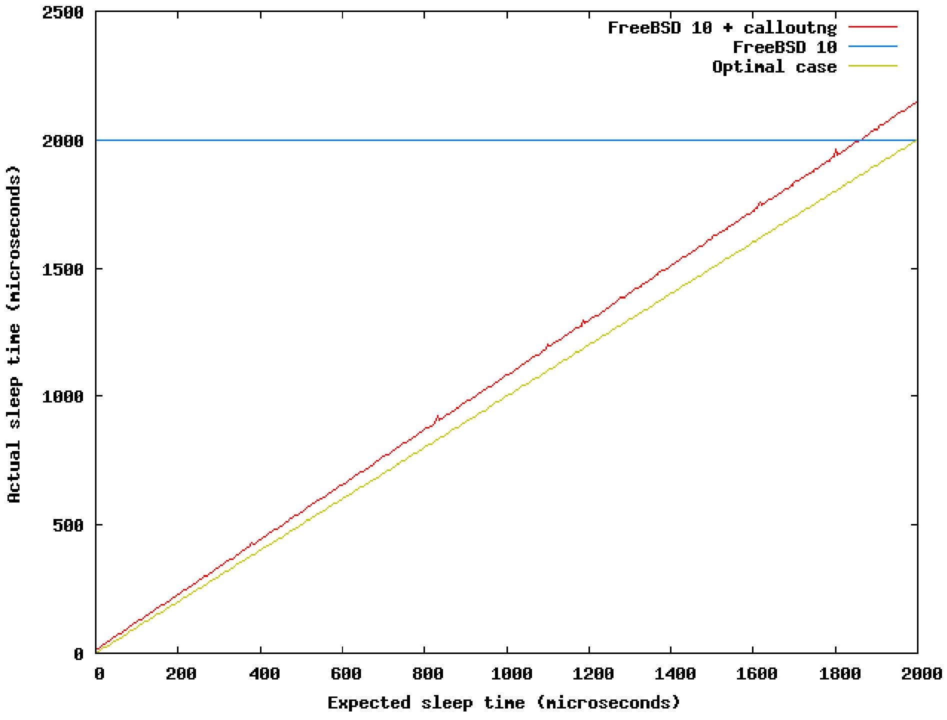
Experimental results (amd64)





Experimental results (arm)





Thank you for your attention!

flood me with questions :-)