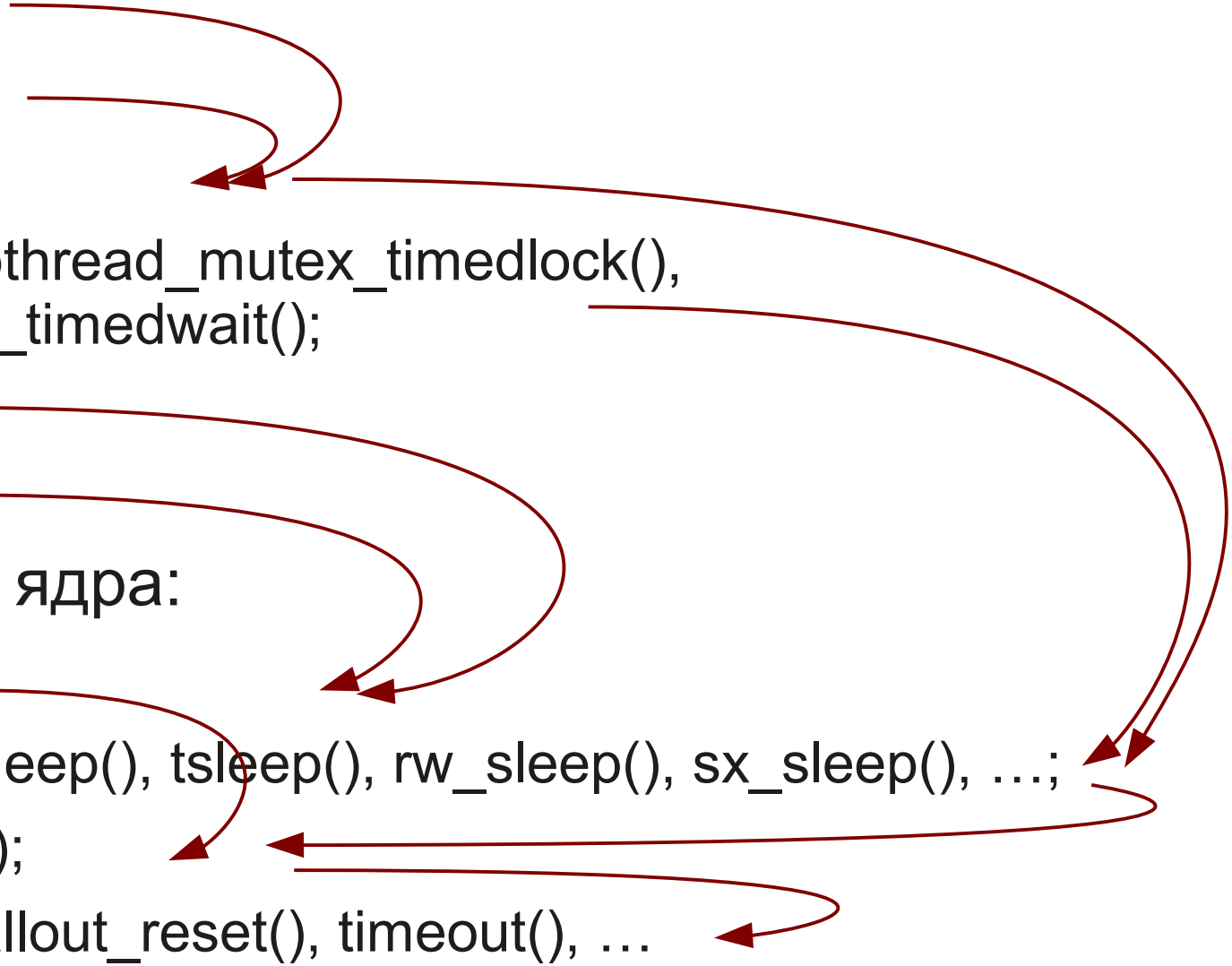
Переработка подсистемы таймеров ядра FreeBSD

Google Summer of Code 2012: Re-enginer the wheel: a rejuvenation of BSD callout(9) and timer facilities

Davide Italiano <davide@FreeBSD.org>
Alexander Motin <mav@FreeBSD.org>

- API/KPI событий времени во FreeBSD:
  - Пространство пользователя:
    - sleep(3);
    - usleep(3);
    - nanosleep(2);
    - pthread(3) – pthread_mutex_timedlock(), pthread_cond_timedwait();
    - poll(2);
    - select(2);
  - Пространство ядра:
    - condvar(9);
    - sleep(9) – msleep(), tsleep(), rw_sleep(), sx_sleep(), …;
    - sleepqueue(9);
    - callout(9) – callout_reset(), timeout(), ...

- API/KPI событий времени во FreeBSD:
  - Пространство пользователя:
    - sleep(3);
    - usleep(3);
    - nanosleep(2);
    - pthread(3) – pthread_mutex_timedlock(), pthread_cond_timedwait();
    - poll(2);
    - select(2);
  - Пространство ядра:
    - condvar(9);
    - sleep(9) – msleep(), tsleep(), rw_sleep(), sx_sleep(), …;
    - sleepqueue(9);
    - callout(9) – callout_reset(), timeout(), …



… все реализованы через callout(9).

- Но есть одно “но” – разрешение:
  - Пространство пользователя:
    - sleep(int) – 1 секунда;
    - usleep(u_int) – 1 микросекунда;
    - nanosleep(struct timespec) – 1 наносекунда;
    - pthread_mutex_timedlock(struct timespec), pthread_cond_timedwait(struct timespec) – 1 наносекунда;
    - poll(int) – 1 миллисекунда;
    - select(struct timeval) – 1 микросекунда;
  - Пространство ядра:
    - cv_timedwait(int) – 1 tick (1/hz секунд);
    - msleep(int) – 1 tick (1/hz секунд);
    - sleepqueue(int) – 1 tick (1/hz секунд);
    - callout_reset(int) – 1 tick (1/hz секунд).

- Время в tick'ах:
  - Один tick исторически соответствует одному вызову функции hardclock() по прерыванию таймера, настроенного на постоянную частоту hz. Функция hardclock() вызывает callout_tick() hz раз в секунду для обработки событий времени.
  - Даже самый точный интервал времени с разрешением 1нс (например, из nanosleep()) будет округлен ядром вверх до ближайшего tick.
  - Так-как неизвестно сколько прошло с последнего tick'а, интервал на всякий случай увеличивается еще на 1 tick, чтобы не вызвать событие раньше.
  - Таким образом, при стандартном значении hz=1000, точность обработки событий времени составляет 1.5мс при минимальном интервале 2мс.

- Время в tick'ах – преимущества:
  - Высокая производительность:
    - Чтение аппаратный таймеров может быть дорого, а чтение глобальной переменной ticks очень дешево.
  - Простые и компактные типы данных:
    - В большинстве случаев используется int.
  - Минимальные требования к аппаратуре
    - Требуются лишь периодические прерывания.

- Цели данного проекта:
  - Обеспечить большую точность обработки событий;
  - Сохранить совместимость API;
  - Сохранить производительность;
  - Уйти от использования периодических событий ради снижения энергопотребления;
  - По возможности группировать близкие события ради снижения числа прерываний и соответственно пробуждений процессора.

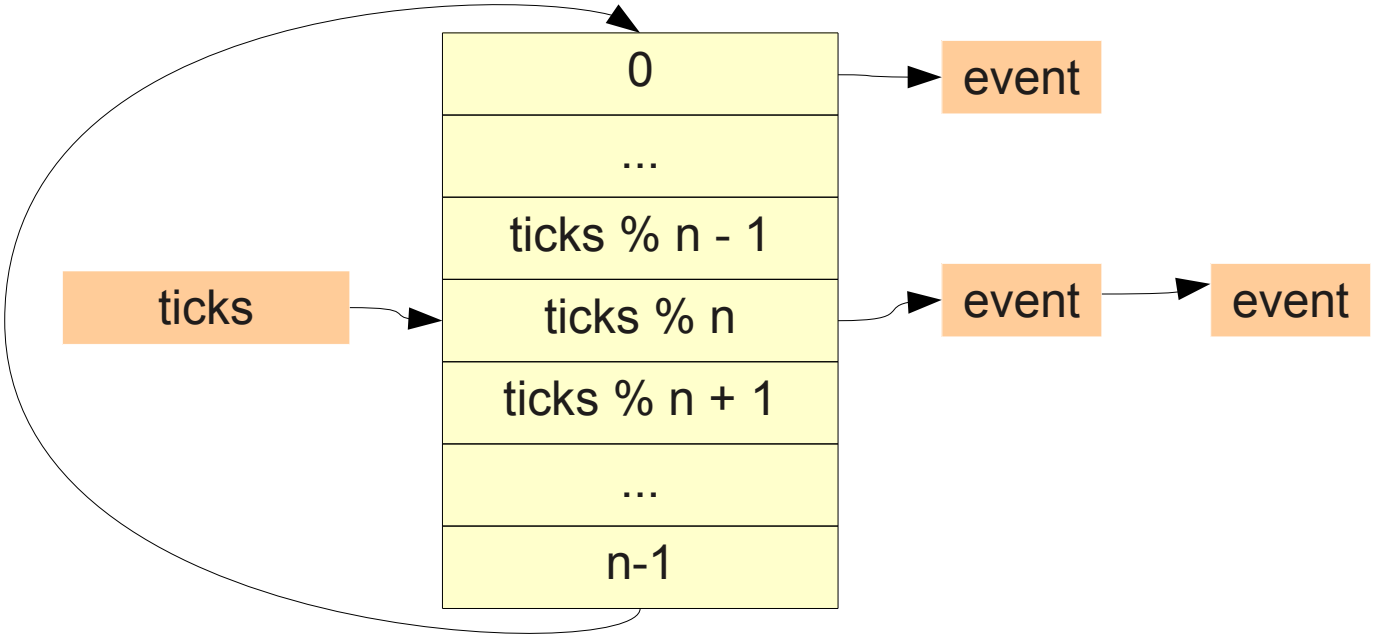
- API/KPI:
  - API пространства пользователя и используемые типы данных стандартизированы и не могут быть изменены. Но они по больше части обеспечивают приемлемую точность.
  - Все API пространства ядра (KPI) построены вокруг концепции tick'ов и требуют доработки. Дальнейшее использование типа int для представления времени с точностью хотябы до микросекунд (а тем более наносекунд) не представляется возможным из-за быстрого переполнения.
  - Так как изменение параметров существующих функций полностью нарушило бы совместимость и было бы крайне инвазивно, было принято решение добавить альтернативные функции.

- KPI (2):
  - В ядре уже существуют следующие типы данных для представления времени:
    - struct timeval {time_t, long} – 64-128bit, в микросекундах;
    - struct timespec {time_t, long} – 64-128bit, в наносекундах;
    - struct bintime {time_t, uint64_t} – 96-128bit, fixed point;
  - Так как базовые подсистемы timecounters(4) и eventtimers(4) используют struct bintime, а разница между типами не велика, было принято решение использовать struct bintime.
  - Linux использует специальный 64-битный тип для работы с таймерами. Во FreeBSD подобного типа нет и следуя результатам более ранних дискуссий, было принято решение его не создавать, а обойтись имеющимся.

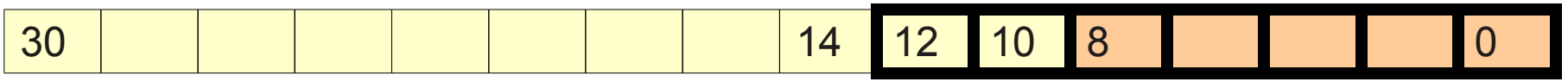
- KPI (3):
  - Таким образом, было выполнено расширение существующих KPI condvar(4), sleepqueue(4), sleep(4) и callout(4) путем добавления в них функций принимающих аругмент типа struct bintime * вместо int для представления времени и новый аргумент flags. В частности:
    - callout_reset_bt_on(..., struct bintime *bt, int flags);
    - cv_timedwait_bt(..., struct bintime *bt, int flags);
    - msleep_bt(..., struct bintuptime *bt, int flags);
    - sleepq_set_timeout_bt(..., struct bintime *bt, int flags);
    - ...

- KBI (Kernel Binary Interface):
  - Хотя KPI callout(9) и не предполагает прямого доступа к элементам struct callout, он требует от пользователя самостоятельно аллоцировать для нее память, а значит sizeof(struct callout) является частью KBI. Использование больших по размеру типов данных неизбежно ведет к увеличению размера данной структуры и нарушению KBI, что требует перекомпиляции практически всех модулей ядра и невозможности портирования данных изменений в существующие STABLE ветки.

- Внутреннее устройство callout(9):
  - Активные struct callout организованы в виде хеша несортированных списков.



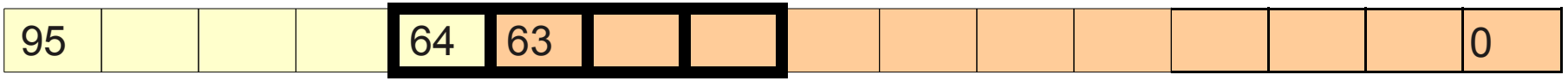
  - В качестве ключа хеша используются младшие биты времени события в tick'ах.



- Внутреннее устройство callout(9) (2):
  - Использование несортированных списков и простой хеш-функции делает вставку и удаление событий очень легкой операцией – O(1). Использование именно младших битов позволяет реализовать линейную выемку событий для обработки перебором только одного элемента хеша за раз, что при малом числе коллизий крупного хеша так-же достаточно легкая операция. С ростом числа коллизий стоимость выемки стремится к O(n), но это большая редкость.
  - Ввиду цикличности обхода элементов хеша данную организацию принято называть callwheel.

- Альтернативные варианты структур
  - С целью ухода от цикличности были рассмотрены другие варианты организации активных событий.
  - Различные древовидные структуры созволяют обеспечить стоимость выемки элементов порядка O(log n), что много лучше чем O(n) худшего случая callwheel. Однако стоимость вставки/выемки у них так-же O(log n), что хуже чем O(1) callwheel.
  - Необходимость обеспечить возможность использования API callout(9) совместно с mutex(9) блокировками делает проблематичной (ре-)аллокацию памяти во время добавления записей, что делает проблематичным использование части алгоритмов.

- Принятое решение
  - В результате анализа было решено сохранить callwheel, перейдя к использованию в качестве ключа хеша части битов времени, так чтобы ключ менялся примерно каждую миллисекунду:



  - Это с одной стороны позволяет зачастую ограничить линейный перебор при выборке следующего элемента одной миллисекундой, а с другой позволяет легко выполнить группировку событий близких по времени.
  - В результате, наиболее частые операции вставки и удаления по-прежнему имеют сложность порядка O(1), а операция выемки усложнилась незначительно.

- Получение текущего времени:
  - Переход к использованию struct bintime вместо tick'ов для представления времени делает невозможным дешевое использование глобальной переменной ticks для получения текущего времени.
  - Было решено использовать компромисный подход: использовать тяжелую но точную функцию binuptime() в случаях когда необходима высокая точность (например, для коротких интервалов), и легкую но с точностью не выше 1мс функцию getbinuptime() в случаях когда точность не существенна.
  - Такой подход позволяет в большей части случаев сохранить ресурсоемкость получения времени на существующем уровне, подняв точность когда это необходимо.

- Точность:
  - Для корректной и эффективной обработки событий нужно иметь информацию о требуемой точности.
  - Новые функции пространства ядра позволяют указать требуемую точность обработки событий используя поле flags.
  - Существующие функции пространства ядра и все функции пространства пользователя к сожалению не имеют подобных аргументов, а потому решение о точности принимается на основе разрешения аргумента задающего период и длины запращиваемого интервала. Требуемый уровень точности планируется сделать настраиваемым глобально при помощи sysctl.

- Оптимизация обработки событий
  - callout(9) использует нити SWI для обработки событий. Это позволяет вывести обработку из контекста аппаратного прерывания, что снимает с обработчиков множество ограничений.
  - Однако использование дополнительных нитей усложняет работу планировщика. Если обработчик события пробуждает другую нить, планировщик старается запустить ее на свободном процессоре, причем не текущем, так как текущий процессор в данный момент занят нитью SWI. Пробуждение другого процессора из состояния глубокого сна может быть долгой и энергоемкой операцией. К тому-же это ухудшает работу кешей, так как нить пробуждается на другом процессоре в кеше которого нет ее данных.

- Оптимизация обработки событий (2)
  - Типичный пример: некоторый процесс использует tsleep() для задержки своего исполнения. tsleep() использует sleepqueue(9), который использует callout(9) для пробуждения. Все что делает обработчик события – это пробуждает спящий процесс. То-есть нить SWI пробуждается лишь для того, чтобы пробудить основной процесс! Причем пробудить его на другом процессоре!

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

  - Анализ показывает, что этот пример актуален для всех потребителей использующих KPI sleep(9), sleepqueue(9) или condvar(9) и соответственно всех потребителей пространства пользователя.

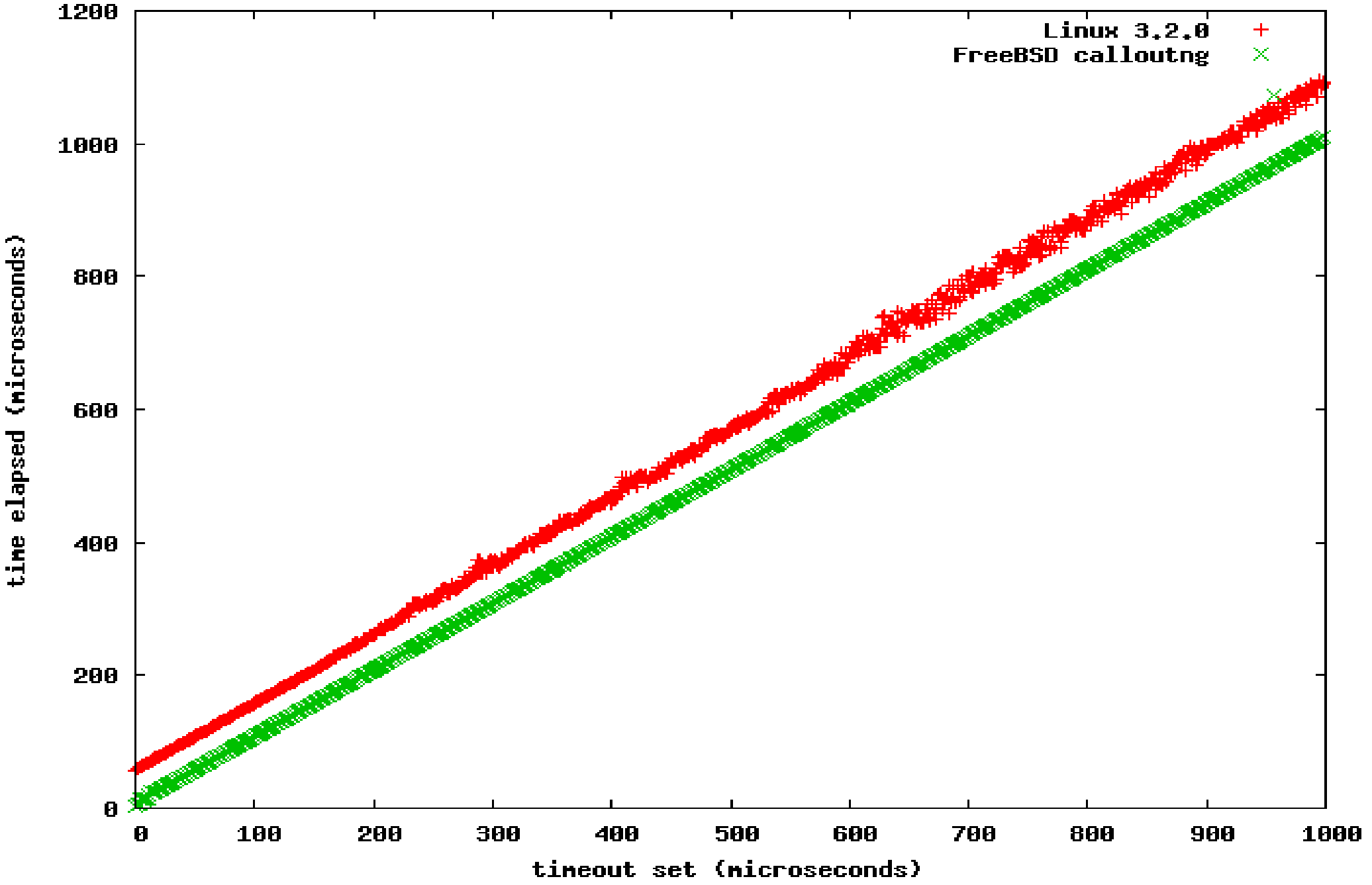
- Оптимизация обработки событий (3)
  - Для решения данной проблемы был реализован механизм прямого исполнения. Используя новый флаг C_DIRECT_EXEC можно указать, что обработчик события можно исполнять непосредственно в контекста аппаратного прерывания. Это исключает использование SWI для данного обработчика:

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

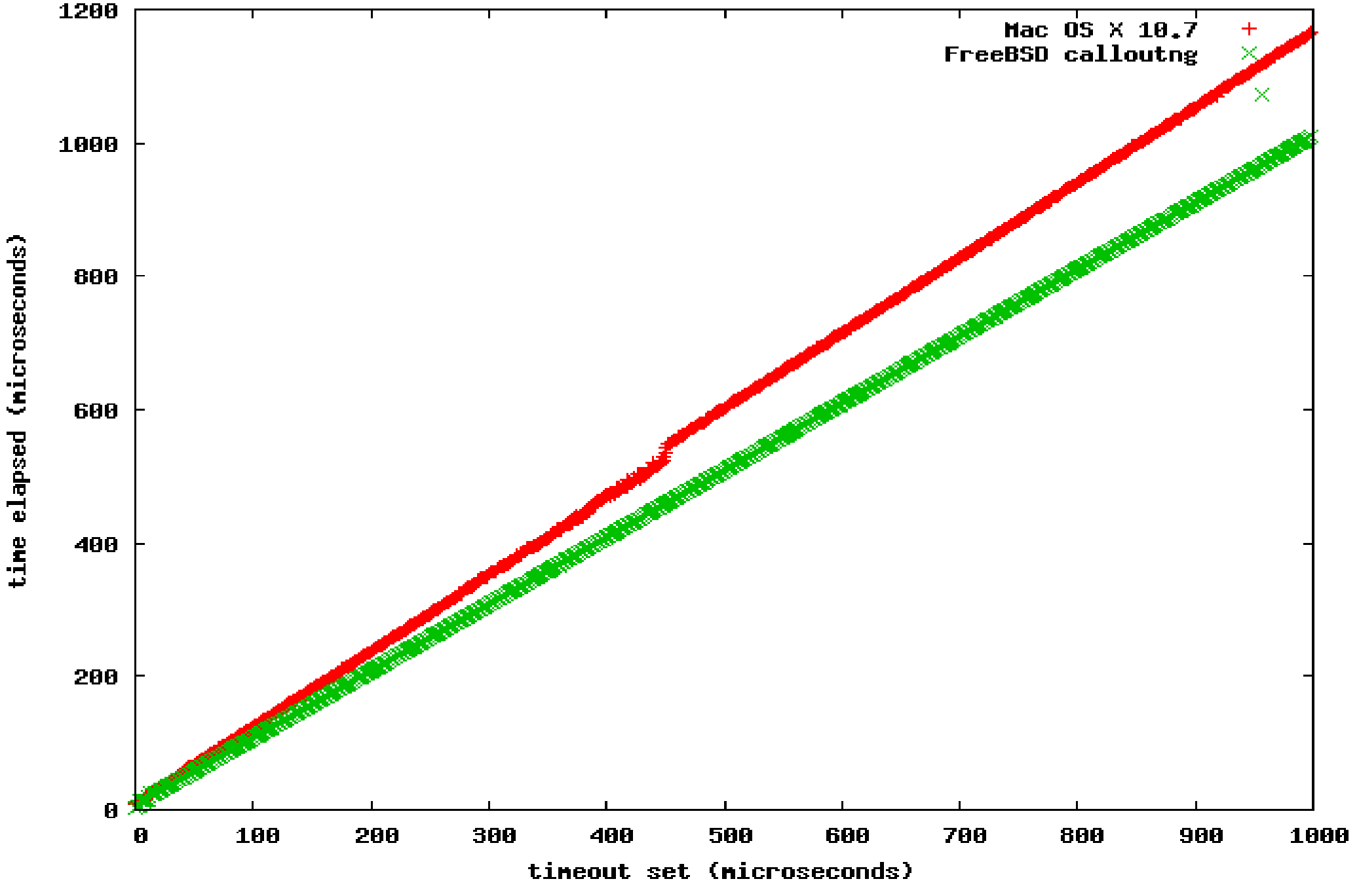
  - Использование данной техники, согласно тестам, позволило значительно снизить ресурсоемкость и задержку обработки событий, а так-же, в теории, улучшить эффективность работы кешей и снизить энергопотребление.

- Результаты
  - При использовании TSC timecounter и LAPIC eventtimer новый код обеспечивает точность обработки событий порядка 2-3 микросекунд.
  - Проведенные тесты (как реальные реальные, так и синтетические) не показали существенного снижени производительности даже при использовании медленных timecounter'ов.
  - Использование механизма прямого исполнения обрабочиков событий позволило значительно повысить производительность и точность в ряде тестов.

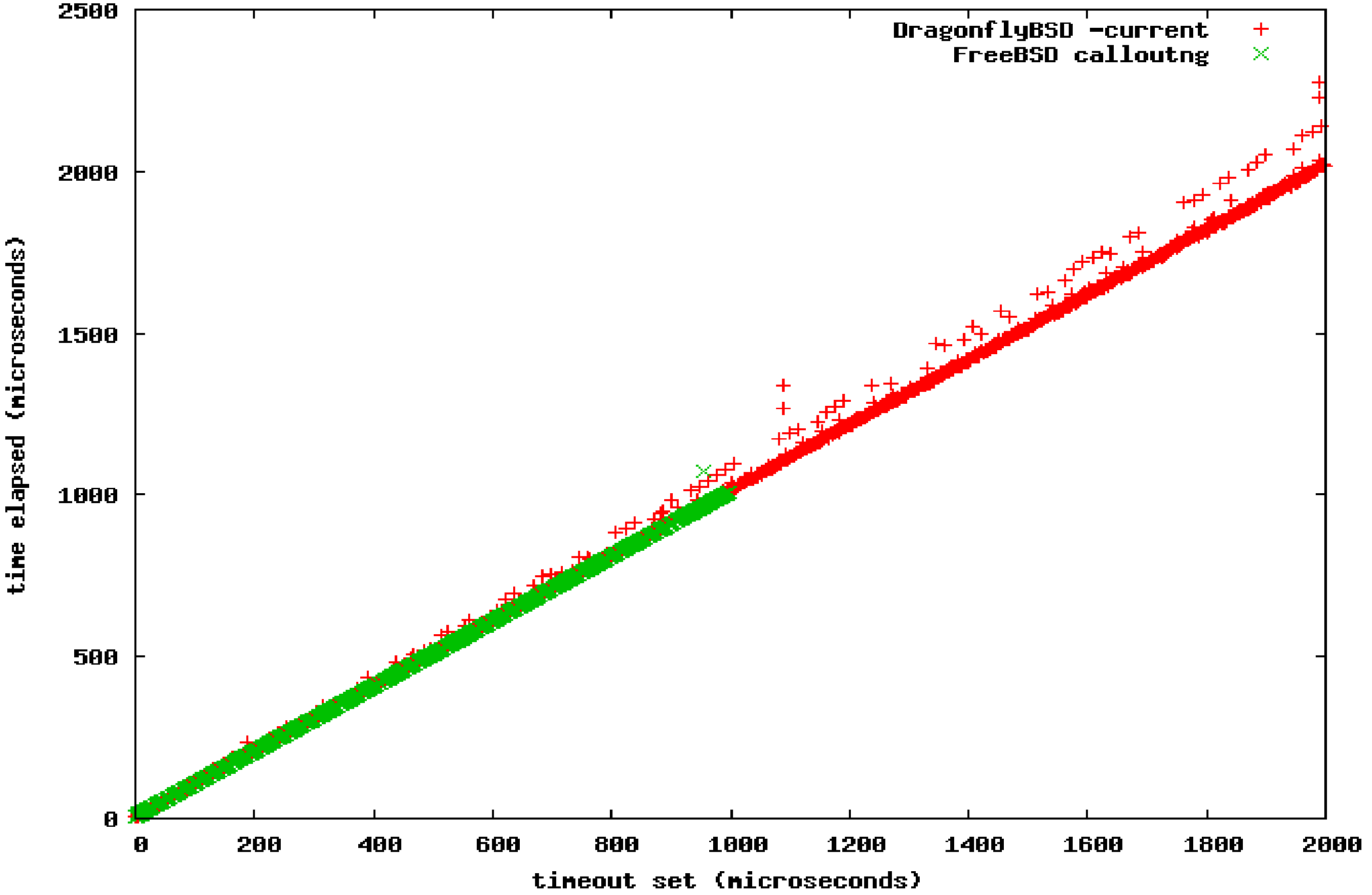
- Точность nanosleep() (calloutng vs Linux)



- Точность nanosleep() (calloutng vs MacOSX)



- Точность nanosleep() (calloutng vs Dragonfly)



- Ссылки
  - http://blogs.freebsdish.org/davide/
  - http://svnweb.freebsd.org/base/projects/calloutng/
  - SVN: svn.freebsd.org/base/projects/calloutng/

- Вопросы?