## Введение

В этой статье рассматривается алгоритм синхронизации времени внутри секунды, его назначение, а так же приводится описание существующего в ядре Linux алгоритма и сложности, возникающие при его реализации.

Рассмотрим следующую систему: имеется машина с установленной Linux-based операционной системой и различными подключенными внешними устройствами. На машине запущены одна или несколько программ, которым для работы требуется, чтобы системные часы шли согласно некоторому эталонному источнику времени. Например, такой программой может быть драйвер сетевого адаптера, который должен регистрировать приходящие в систему извне пакеты данных и ставить в соответствие каждому из них некоторую метку времени, а источником могут быть часы другой системы, или одно из устройств, подключенных к рассматриваемой системе.

То есть, таким программам требуется, чтобы системные часы были *синхронизированы* с эталонным источником. Это означает, что в каждый момент времени значения часов источника и того, кто с ним синхронизируется, должны как можно меньше отличатся. Чем меньше разница, тем выше точность синхронизации.

Источников, как и программ, нуждающихся в синхронизации, может быть несколько. Для синхронизации значений часов системы может быть использован NTP (Network Time Protocol), дающий относительно неплохую точность синхронизации, порядка миллисекунд. Но иногда такой точности недостаточно, и тогда помогают алгоритмы, точность которых достигает порядка наносекунд.

Рассматриваемый далее в статье алгоритм позволяет с помощью некоторого высокоточного генератора PPS (Pulse per second) сигналов узнавать о начале каждой новой секунды и сообщать об этом системе, в которой, в свою очередь, регистрируются приложения - PPS-потребители, которым требуется реагировать на эти сигналы. Каждый PPS-сигнал вызывает в системе рассылку текущего значения системных часов всем зарегистрированным потребителям.

То есть для получения высокой точности можно использовать какой-нибудь относительно грубый алгоритм, вроде NTP, в совокупности с PPS. И тогда если время будет синхронизироваться с помощью основного алгоритма до хотя бы секунд, то уже внутри секунды синхронизацию обеспечит PPS.

Рассмотрим далее более конкретно, что такое метки времени, откуда берется PPS-сигнал, и что это такое, какие сложности возникают при реализации алгоритмов, работающих с PPS, а так же конкретный пример - алгоритм, использующийся в ядре Linux для синхронизации системного времени с PPS-источником.

## Время в ядре Linux

В компьютере время представлено как значение некоторого постоянно возрастающего аппаратного счетчика, значение которого увеличивается через примерно равные промежутки времени. Счетчик не может инкрементироваться через просто равные промежутки времени, так как частота его инкрементации связана с частотой некоторого генератора, а его частота колеблется в относительно небольшом диапазоне, зависящем от условий, в которых находится генератор, например, от давления и температуры, от энергопотребления и других причин.

Значение этого счетчика в каждый момент времени - целое число, которое используется системой для перевода его в человеческие единицы измерения времени. Говоря о человеческих единицах времени будем здесь и далее иметь ввиду наносекунды. Для перевода значения счетчика в наносекунды требуется знать, с примерно какой частотой счетчик инкрементируется - для каждой системы устройств эта частота известна заранее.

Значение этого счетчика - это так называемое *monotonic time -* "сырое" значение времени, к которому не применялись никакие алгоритмы для его коррекции. Так же в системе есть понятие *real time -* это время, получаемое из monotonic с помощью применения различных алгоритмов синхронизации времени - это то время, которое доступно в пользовательской области операционной системы для приложений вне ядра, и которое здесь, в статье, называется *системными часами*.

Когда говорят о коррекции времени, то имеют ввиду не прямое изменение значения аппаратного счетчика, а способ его преобразования в real time из monotonic time. То есть алгоритмы синхронизации служат для построения правил, по которым вычисляются коэффициенты для пересчета monotonic time.

Как инкрементируется значение счетчика времени? Для этого существует несколько механизмов, вот одни из них:

1. Аппаратные прерывания
2. Time stamp counter (TSC)
3. High precision event timer (HPET)

Прерывание - это сигнал, сообщающий процессору о наступлении какого-либо события, управление при возникновении прерывания передается его обработчику. Прерывание может быть как аппаратным, в случае счетчика времени, так и программным, если оно инициировано специальной инструкцией в коде. То есть через примерно равные промежутки времени счетчик инкрементируется с помощью аппаратного прерывания. Но этот способ устарел, поскольку обработка прерывания - это дополнительная нагрузка на систему.

Time stamp counter - специальный регистр процессора, который считает количество тактов, прошедшее с запуска компьютера. Его значение можно получить специальной инструкцией в коде.

HPET - аппаратный таймер, используемый в персональных компьютерах. Фактически, это отдельное устройство, подключенное к процессору. Метку времени можно читать с его регистров. Так же это устройство можно программировать на прерывания.

Выше мы рассматривали систему, часы которой надо синхронизировать, и устройство, являющееся источником PPS сигналов. У системы есть свой счетчик времени, значение которого менять нельзя, но можно менять коэффициенты и правила, по которым считается real time. Вычислением таких коэффициентов и занимается алгоритм синхронизации, использующий PPS.

## Pulse per second

Pulse per second (PPS) - сигнал, генерируемый некоторым высокоточным источником (PPS-source) с частотой максимально близкой к одной секунде. Для рассматриваемого в статье алгоритма важно уметь "ловить" моменты, когда возникает PPS-сигнал, то есть получать метки времени точно в эти моменты.

PPS-источник подключается к вычислительной системе посредством контакта интерфейса последовательной линии (modem-control pin on a serial-line interface) или параллельной . Система узнает о возникновении сигнала, получая прерывание, и как можно скорее запоминает текущую метку времени. Далее эта метка становится доступна с относительно небольшой задержкой после ее получения для ПО, работающего со временем - PPS-потребителей.

PPS-потребители - это приложения, заинтересованные в обработке возникающих PPS-сигналов, которые ранее зарегистрировались в системе, как получатели PPS-сигналов. С помощью метки, полученной с PPS-сигналом, PPS-потребитель может, например, получить расхождение между источником точного времени и системными часами.

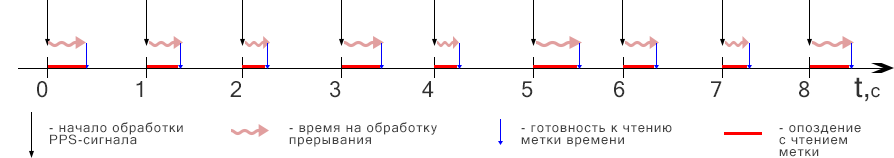
К одной системе может быть подключено несколько PPS-источников, каждый из которых генерирует собственные PPS-сигналы независимо от других. Когда очередной PPS-потребитель регистрируется на прием сигналов, он может указать, какой PPS-источник ему нужно слушать, причем слушать можно несколько источников.

Кроме пользовательских программ в ядре есть еще один потребитель сигналов: специальная процедура ядра - *hardpps*. Эта процедура на основании получаемых сигналов пересчитывает коэффициенты, по которым monotonic time пересчитывается в real time, в результате чего изменяется системное время. Причем эта процедура может слушать единовременно только один PPS-источник. Как раз с ним рассматриваемая система и будет синхронизировать свои часы.

Рассмотрим, какие возникают сложности при реализации алгоритмов синхронизации, основанных на PPS.

1. Что ставить как метку времени для PPS-сигнала?
2. Как правильно учесть время, которое пройдет с момента генерации сигнала и возникновения прерывания до вызова его обработчика и запоминания текущего времени для его передачи PPS-потребителям?
3. Как использовать получаемые метки для коррекции системного времени в процедуре hardpps, описанной выше?

Первая задача - это выбор между меткой real time и monotonic time. Ее решает API ядра, предоставляющий программисту самому выбрать, какая метка ему нужна - каждому сигналу ставятся в соответствие сразу две метки - и real time, и monotonic time, так же называемый raw time.

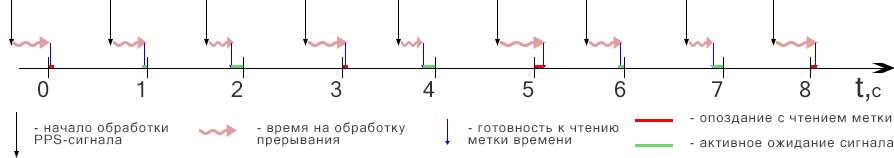
Рассмотрим, в чем суть второй задачи, с помощью следующего рисунка:

На рисунке изображена шкала времени по секундам, где каждое деление на шкале обозначает возникновение сигнала PPS. Черные стрелки - это момент, когда мы понимаем, что прошел сигнал PPS; волнистые стрелки - это время, которое тратится на обработку прерывания, и, наконец, синие стрелки - это момент, когда мы готовы прочитать метку времени, чтобы поставить ее в соответствие возникшему сигналу. Как видно из рисунка - если обработчик прерывания проснется только в момент возникновения сигнала, то метка времени, которая будет считана к нему, будет получена с отставанием, причем каждый раз с разным, как видно из красных полос на рисунке.

Такое неравенство отставаний получается в силу того, что для вызова обработчика прерывания нужно прервать выполнение текущего кода. И время, за которое это прерывание осуществится, зависит от нескольких факторов, например:

* уровень энергопотребления процессора (процессор постоянно пытается его снизить). Если энергопотребление снижено, то необходимо время, чтобы "пробудить" процессор.
* если в данный момент в ядре выполняется код критической секции - выполнение такого кода прервать нельзя. Вход в критическую секцию запрещает прерывания.
* могут возникать более приоритетные прерывания.

В качестве решения алгоритм, рассматриваемый в статье, начинает ожидать сигнал немного раньше, чем он возникнет. Это делается путем оценки того, сколько примерно может занять времени получение метки после возникновения сигнала. И тогда если начать ожидать сигнал чуть раньше, то к тому моменту, как он появится, мы будем готовы сразу считать текущую метку времени.

Это можно осуществить, например, с помощью таймера, который будет запущен в середине секунды и срабатывать раз в секунду, то есть будет запас до предполагаемого появления PPS-сигнала будет в пол секунды. Описанный способ хорошо видно на следующем рисунке:

Здесь мы видим, что если подготовится к сигналу пораньше, то почти всегда метка будет считана достаточно точно. Зеленые полосы - это активное ожидание прихода сигнала.

Далее рассмотрим третью из задач, описанных выше, решение которой и является рассматриваемым в статье алгоритмом.

## Процедура ядра hardpps

*Hardpps* - так называется специальная процедура ядра, служащая для синхронизации системных часов с некоторым PPS-источником, и реализована она как С функция в коде ядра. hardpps вызывается при каждом возникновении сигнала и работает с двумя аргументами - метками времени real time и monotonic time, запоминаемыми в момент получения сигнала. Далее будем первый аргумент обозначать как *phase\_ts*, а второй как *raw\_ts*.

Поскольку весь механизм PPS-сигналов рассчитан на работу со временем внутри секунды, то и hardpps не исключение - абсолютное значение меток raw\_ts и phase\_ts для алгоритма значения не имеет.

Для чего нужен этот алгоритм - он позволяет на основании данных о последних метках времени, получаемых по PPS-сигналам, вычислять, с какой скоростью идут часы - то есть на сколько возрастает monotonic time за секунду в среднем, и на основании ранее запомненных скоростей пересчитывать коэффициенты, по которым считается real time. То есть происходит корректировка скорости хода часов.

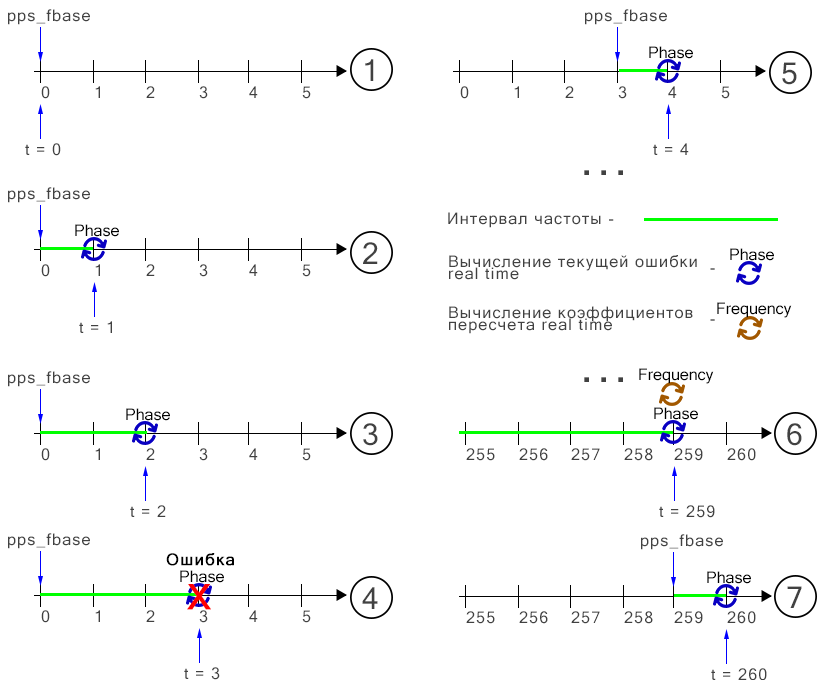
Опишем, какими параметрами и понятиями оперирует алгоритм:

* Интервал частоты - количество последних секунд, за которые измеряется скорость хода monotonic time. Значение интервала меняется по ходу алгоритма от 4-х до 256-ти. Когда в алгоритме фиксируются ошибки, то интервал уменьшается, а когда алгоритм работает без сбоев, то интервал увеличивается.
* pps\_fbase - точка, от которой считаются текущий интервал частоты, метка monotonic time. От текущей метки, raw\_ts, она должна отставать не более чем на некоторое количество секунд, которое является параметром алгоритма.

Общую схему алгоритма, реализованного в hardpps, можно описать следующим образом:

1. На каждой итерации считается, на сколько увеличился monotonic time за текущий интервал частоты.
2. Если увеличение неправдоподобно велико, то за pps\_fbase принимается текущий raw\_ts и пересчета коэффициентов не происходит, а интервал начинается заново. Так отсеиваются слишком большие или слишком маленькие отклонения.
3. Если текущий интервал частоты завершен, то pps\_fbase сбрасывается к текущему raw\_ts и происходит вычисление новой частоты, с которой должны идти часы системы. То есть пересчитываются коэффициенты для real time.
4. Вычисляется, насколько и в какую сторону ошибается сейчас real time. Затем эти значения используются в шагах выше.
5. Далее к пункту 1

Следует остановиться подробнее на шагах (3) и (4).

1. Для вычисления средней скорости, с которой идут часы, делят разницу между pps\_fbase и текущей меткой времени monotonic time на размер закончившегося интервала частоты. То есть получают величину
2. Если слишком велико или слишком мало, то алгоритм уменьшает длину интервала частоты. А если скорость в пределах нормы, то длина интервала частоты увеличивается. То есть алгоритм обучается - когда ошибки редки, то учитывается больше последних меток времени, а когда ошибки учащаются, то меньше.
3. Вычисленная скорость передается алгоритму NTP для корректировки коэффициентов.
4. Обозначим за *error* - количество наносекунд в текущий момент времени. То есть, если сейчас время было бы N секунд и 123 наносекунды, то error было бы равно 123. Значение error высчитывается из текущей phase\_ts - так мы обозначили текущую метку real time.
5. error добавляется в *кольцо* последних error. Размер такого кольца в ядре, например, 4.3, равен трем. То есть в нем хранятся три последних error. Теперь на основании элементов этого кольца можно посчитать, как следует изменить коэффициенты пересчета real time.
6. Обозначим за *correction* то значение, которое будет извлечено из кольца. Алгоритм выбора здесь может быть разным, но в ядре 4.3 просто берется последнее значение. За *jitter* обозначим модуль разницы между последними двумя error. Теперь мы знаем текущую ошибку, и на сколько она изменилась с прошлой корректировки. За *pps\_jitter* обозначим максимально допустимый jitter - если jitter больше чем pps\_jitter, то коррекция времени не происходит, pps\_jitter увеличивается, и алгоритм завершает работу.
7. Если jitter меньше pps\_jitter, то количество наносекунд, на которое требуется скорректировать real time, считается как , где NTP\_FREQ\_INTERVAL - константа алгоритма NTP, а pps\_jitter уменьшается.

Наглядно увидеть работу алгоритма можно на следующем рисунке:

Что здесь происходит:

1. На первой итерации pps\_fbase инициализируется как текущий monotonic time
2. Далее приходит PPS-сигнал, и по текущим real time и monotonic time считаются текущая ошибка real time. Ведь если PPS-сигнал приходит раз в секунду, то и jitter, который был описан в пункте 4.3 должен быть всегда ноль. Если он не ноль, то время надо корректировать.
3. Пока алгоритм не обнаружил сильных отклонений меток времени, и интервал частоты еще не завершен, следовательно повторяется предыдущий шаг.
4. Алгоритм обнаружил ошибку - обновления коэффициентов не происходит, и интервал частоты начинается сначала.
5. То же, что в пункте 3.
6. Интервал частоты успешно завершился, происходит обновление коэффициентов пересчета на основании скорости времени на протяжении завершенного интервала частоты и накопленных отклонений, затем интервал перезапускается с увеличенной максимальной длиной.
7. То же, что в пункте 2.

## Заключение

## Как видно, алгоритм хоть и работает, но он несовершенен и может быть улучшен. Например, размер кольца для последних error можно увеличить, и усовершенствовать алгоритм выбора из него значения correction - можно брать медиану, или среднее арифметическое от элементов кольца, отбросив самые большие и самые маленькие.

## Так же при возникновении ошибки можно не игнорировать весь пройденный интервал частоты, а использовать его для менее значимого обновления времени - если при успешном завершении время корректируется на сотую часть от ошибки, то при неуспешном завершении можно корректировать, например, на двухсотую.

## Список литературы

1. Cochran R., Marinescu C., Riesch C. Synchronizing the Linux system time to a PTP hardware clock //Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2011 International IEEE Symposium on. – IEEE, 2011. – С. 87-92.
2. Mogul J. C. et al. Pulse-per-second api for unix-like operating systems, version 1.0 //Pulse. – 2000.
3. Gleixner T., Niehaus D. Hrtimers and beyond: Transforming the linux time subsystems //Proceedings of the Linux symposium. – 2006. – Т. 1. – С. 333-346.
4. Köker K., Hielscher K. S., German R. A Low-Cost High Precision Time Measurement Infrastructure for Embedded Mobile Systems //Robot Motion and Control 2007. – Springer London, 2007. – С. 445-452.