Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра Автоматизации Систем Вычислительных Комплексов

ШПИЛЕВОЙ Владислав Дмитриевич

Исследование механизмов синхронизации времени от различных аппаратных источников в ядре Linux

КУРСОВАЯ РАБОТА
3 курс

Научный руководитель:
А.В.Герасёв

Москва, 2015

Аннотация

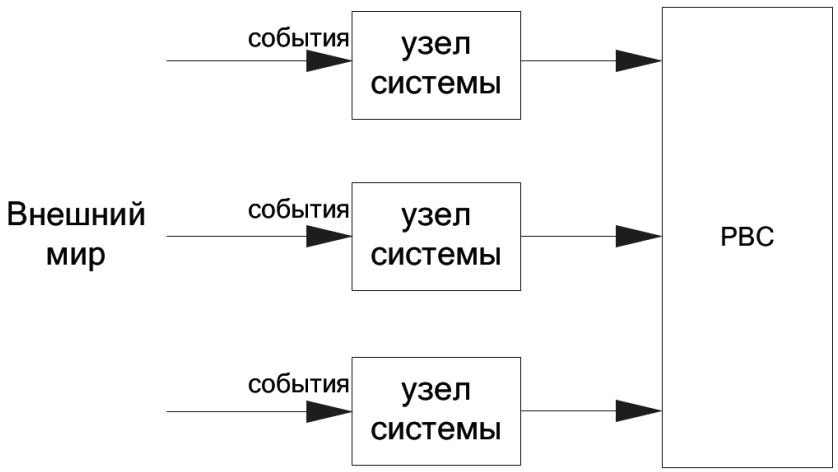
В распределенных вычислительных системах иногда возникает необходимость синхронизации времени на узлах таких систем, и сегодня задача синхронизации решается отдельно для каждой системы, что делает невозможным использование существующих решений для других систем. В данной работе представлен обзор существующих алгоритмов синхронизации, и представлен обобщенный алгоритм, который может быть использован различными системами. В рамках данной работы представленный алгоритм был программно реализован на языке C, а на построенную реализацию портирован существующий драйвер сетевого адаптера, на котором были проведены численные исследования эффективности алгоритма.

Оглавление

1 Введение

В последнее время системы, используемые в самых разных отраслях науки, промышленности, бизнеса и развлечений, все больше приобретают тенденцию развиваться не "вертикально", то есть наращивая мощности своих компонент, а "горизонтально", увеличивая количество этих компонент, что обходится дешевле и во многих случаях эффективнее. Но вместе с ростом масштабов систем растет сложность управления и работы с ними. Такие системы, состоящие из множества узлов, называют распределенными вычислительными системами.

Распределенная вычислительная система (РВС) - набор независимых компьютеров, объединенных средой передачи данных, представляющийся пользователям единой объединенной системой, и направленный на решение определенной общей задачи [1]. В таких системах, будь то большой программно-аппаратный комплекс, части которого разнесены на значительное расстояние друг от друга, или один компьютер с подключенными к нему устройствами (мониторы, внешние диски, модемы и т.д.), иногда возникает необходимость взаимодействия различных узлов таких систем с внешним миром. На узлах системы происходят различные события.



Зачастую необходимо уметь определять порядок во времени событий, возникающих на разных узлах системы, взаимодействующих с внешним миром.

Например, рассмотрим прием пакетов данных сетевыми адаптерами: пусть в некоторой РВС есть несколько узлов - сетевых адаптеров, каждый из которых взаимодействует с внешним миром, принимая и отправляя пакеты данных. Пусть на один из узлов пришел пакет, после прочтения которого этот узел выслал еще один пакет на другой узел. То есть прием первого пакета вызвал

отправку другого. В таком случае эти пакеты взаимосвязаны, и чтобы правильно эту связь отследить, необходимо знать, в каком порядке эти пакеты пришли к узлам-получателям. Здесь приемы пакетов и будут событиями.

Как правило, для определения порядка каждому событию ставят в соответствие некоторую метку, которая должна однозначно определить порядок этого события относительно остальных. Будем называть такие метки *метками времени* или *временными метками*. Метки времени можно получать, фиксируя в определенные моменты времени *время узла* - значение некоторого аппаратного счетчика, связанного с этим узлом, который инкрементируется через примерно равные промежутки времени. Счетчик не может инкрементироваться через точно равные промежутки времени, так как его инкрементация связана с частотой некоторого генератора, а частота генератора всегда колеблется в относительно небольшом диапазоне, зависящем от внешних условий, таких, например, как температура воздуха и давление.

Значение счетчика, когда бы оно ни было измерено, не будет соответствовать астрономическому времени, так как оно измеряется в некоторой шкале, специфичной для каждого устройства. Например, в документации устройства может быть написано, что частота его счетчика равна 37 наносекунд, тогда "шаг" шкалы этого счетчика - 37 наносекунд. Более того, даже если устройства одинаковы, то значения их счетчиков так же не будут совпадать, так как они могли стартовать в разные моменты времени, а так же условия их работы не могут быть абсолютно одинаковы, что ведет к расхождению в частотах их генераторов. Но значения счетчиков все равно необходимо уметь сравнивать, для чего можно задать для каждого узла функцию $f_i: t \to T$ для определения относительного порядка событий, которая бы пересчитывала значения счетчика в некоторую общую для всех устройств шкалу.

Таким образом, каждому событию на каждом узле системы ставится в соответствие метка времени. Это позволяет получить относительный порядок событий в рамках одного узла. Чтобы сравнить значения разных счетчиков, необходимо делать это относительно некоторой общей шкалы. Для этого примем часы одного из узлов в РВС за эталонные и будем пересчитывать временные метки всех событий системы в значения эталонных часов. То есть для каждого аппаратного счетчика зададим функцию $f_i: t \to T$, которая будет пересчитывать значения этого счетчика в эталонное время. Так как скорость счетчика непостоянна, то некоторые параметры функции f_i нужно изменять динамически для более точного пересчета. Из всех выше перечисленных задач и проблем

возникает задача *синхронизации часов*: задача построения для каждого аппаратного счетчика времени в РВС такой функции $f_i: t \to T$ пересчета, и динамическая коррекция ее параметров.

Примерами распределенных вычислительных систем можно назвать:

1) Вычислительный кластер, который может иметь несколько сотен узлов, где машины могут выходить из строя по несколько штук в неделю, где обрабатываются сотни терабайт данных, а центральных узлов может быть несколько [2]. (Например, кластеры компаний Mail.ru, Yandex, Google)
2) Информационно-управляющие системы (ИУС) летательных аппаратов[3].
3) Компьютер под управлением операционной системы Linux, с подключенными к нему устройствами, такими как, например, сетевые адаптеры.

В данной работе рассматривается третий из описанных выше примеров, то есть задача синхронизации часов в рамках одно компьютера под управлением операционной системы Linux с подключенными к нему сетевыми адаптерами, на которые приходят пакеты данных из сети - здесь приход очередного пакета и будет событием.

События получения сообщения можно помечать метками времени в моменты обработки прерываний адаптера, вызванных приемом пакетов. Задержка обработки прерывания не детерминирована, а так же за одно прерывание может быть обработано несколько событий, поэтому каждый адаптер должен ставить метки по своему счетчику в моменты аппаратного возникновения события. Здесь необходимость помечать события метками времени в некоторой общей шкале приводит к возникновению задачи синхронизации часов.

Эта задача уже имеет частные решения в рамках отдельных драйверов, и каждый раз, когда эта задача возникает снова, ее решают заново. В рамках данной работы рассматривается общее решение этой задачи в виде обобщенной подсистемы, представляющей собой загружаемый модуль ядра Linux.

2 Постановка задачи

Целью работы является исследование механизмов синхронизации времени от различных аппаратных источников в ядре Linux, а именно:

- Обзор существующих алгоритмов синхронизации временных меток в распределенной вычислительной системе.
- Обзор существующих реализаций решения исследуемой задачи в рамках ядра Linux.
- Проектирование и реализация обобщенной подсистемы синхронизации временных меток в ядре Linux.

В результате выполнения данной работы должно быть сделано следующее:

- Разработана подсистема синхронизации временных меток.
- Разработан тестовый драйвер виртуального устройства для проверки работоспособности разработанной подсистемы.
- Осуществлено портирование на подсистему одного из существующих драйверов бортовых интерфейсов.
- Проведена апробация разработанной подсистемы и численные исследования временных характеристик.

3 Математическая модель

Рассмотрим РВС, состоящую из K узлов, с каждым из которых связан некоторый аппаратный счетчик времени. Введем следующие обозначения для обозначенной РВС:

- A_i - i -й узел рассматриваемой РВС, $1 \leq i \leq K$
- T_{A_i} - аппаратный счетчик, соответствующий узлу A_i
- $D(T_{A_i})$ - множество значений счетчика T_{A_i}
- t - эталонное время в некоторой общей шкале
- $\forall t \geq 0, T_{A_i}(t)$ - значение счетчика T_{A_i} , соответствующее эталонному времени t, то есть это значение было доступно на счетчике в тот же момент времени, что эталонное время было равно t.

Тогда математически можно описать задачу синхронизации часов узлов РВС следующим образом:

$\forall i: 1 \leq i \leq K$ требуется построить функцию $f_i: D(T_{A_i}) \times (\alpha_1, \alpha_2, \dots \alpha_n) \to t$, то есть такую, которая переводит пары из элемента множества значений счетчика T_{A_i} и набора динамически изменяемых параметров в элемент множества значений эталонного времени. Причем $\forall t_1, t_2: 0 \leq t_1 < t_2$ необходимо выполнение $f_i(T_{A_i}(t_1)) \leq f_i(T_{A_i}(t_2))$

Под динамически изменяемыми параметрами подразумеваются параметры, используемые для более точного пересчета значений счетчика, с учетом цикличности значений счетчика (он может переполняться), а так же неравномерность его хода. Рассмотрим пример, показывающий необходимость наличия параметров $(\alpha_1, \alpha_2, \dots \alpha_n)$:

Пусть есть устройство, частота которого равна 30 нс, а множество значений его счетчика - [0, 4 294 967 295], то есть счетчик 32-х разрядный. Если каждые 30 нс счетчик инкрементируется, то его переполнение и сброс до нуля будут происходить примерно каждые 129 секунд, следовательно мы можем получить пару одинаковых значений счетчика, если возьмем их с интервалом между замерами больше, чем 129 секунд, но очевидно, что соответствующие им значения эталонного времени должны различаться, следовательно, необходимо наличие дополнительных параметров, по которым подобные случаи будут правильно обрабатываться.

4 Существующие алгоритмы синхронизации

4.1 Решение в сетевой подсистеме ядра Linux

Для работы алгоритма, реализованного в сетевой подсистеме ядра Linux, необходимо точно регистрировать моменты времени получения пакетов из сети, поэтому метки нужно ставить аппаратно. Алгоритм [12] используется для пересчета уже проставленных меток и предполагает, что проставление меток реализовано в виде алгоритма *PTP (Precision Time Protocol)* [10] с аппаратной поддержкой - *Hardware-assisted PTP* [11].

Алгоритм PTP служит для синхронизации времени между компьютерами в сети и может быть реализован полностью программно или с аппаратной поддержкой. Программная реализация дает меньшую точность (порядка сотен микросекунд), чем с аппаратной поддержкой, так как программно метка ставится не точно в момент получения пакета, а когда начинается обработка прерывания, вызванного приходом этого пакета. Например, полностью программной реализацией является Linux *PTPd (Precision Time Protocol daemon) версии 1*, служащий для синхронизации компьютеров в сети LAN.

Вернемся к рассмотрению алгоритма, реализованного в сетевой подсистеме ядра Linux. В ядре Linux до версии 3.8 реализация существовала в виде вспомогательных структур для решения задачи синхронизации часов сетевых адаптеров *(Network Interface Controller)*. В версии 3.8 реализация этого алгоритма была удалена из ядра в силу малочисленности его пользователей, слабости первоначальной идеи и узконаправленности алгоритма на сетевые адаптеры. Тем не менее рассмотрим использованное решение.

Алгоритм позволял синхронизировать *источники* времени с *целевым* временем, принятым за эталонное. Источник времени должен был представлять из себя монотонно увеличивающийся счетчик, тогда как целевое время могло увеличиваться скачками. Целевое время, соответствующее времени источника определялось следующим образом:

1) считывалось целевое время, время источника и затем снова целевое время
2) предполагалось, что среднее по считанному целевому времени соответствует считанному времени источника

Последнее предположение было основано на том, что считывание времени источника выполняется медленно и включает время на отправку запроса

на считывание и получение ответа со значением, тогда как считывание целевого времени предположительно выполняется быстро.

Затем при возникновении необходимости конвертации времени источника в целевое время результат считался по формуле:

$$result\ time = (source\ timestamp + offset) + (source\ timestamp - last\ update) * skew$$

- *result time* - целевое время, соответствующее времени источника, которое требовалось конвертировать
- *source timestamp* - время источника, которое требуется конвертировать
- *offset* - *среднее* отклонение времени источника относительно целевого времени по некоторому количеству замеров подряд. То есть для обновления offset производится многократное обращение к источнику, и каждый раз считается отклонение полученного значения от текущего целевого времени, а затем за offset принимается среднее по некоторому проценту самых небольших отклонений для отбрасывания неправдоподобных отклонений.
- *last update* - последнее значение времени источника, измеренное тогда же, когда измерен последний *offset*.
- *skew* - коэффициент смещения, который считается следующим образом:
  1) Пусть *skew* считается первый раз, тогда оно будет принято равным 0
  2) Пусть теперь *skew* считается не первый раз, тогда оно будет принято как 4/16 от старого значения *skew* + 12/16 от отношения разниц двух последних *offset* и двух последних *last update*. Такая разница отношений 4/16 и 12/16 берется от того, что чем измерения свежее, тем больший вес им придается при вычислениях.

4.2 Решение в драйвере ta1 usb интерфейса MIL-STD1553-B

В драйвере *ta1 usb* решается задача пересчета значения аппаратного счетчика устройства в максимально соответствующее ему значение эталонного времени. За эталонное время принята функция, аппроксимирующая ход часов устройства.

В процессе работы устройства алгоритм накапливает в структуру - кольцо значения локального счетчика этого устройства. Когда размер кольца, то есть количество накопленных в нем значений аппаратного счетчика, достигает некоторого фиксированного размера, то самые старые значения поочередно

отбрасываются, и на их место записываются новые. Новое значение для кольца считывается со счетчика с некоторым заранее заданным периодом.

Когда возникает необходимость пересчитать значение счетчика в эталонное время, оно вычисляется следующим образом:

1) Вычисляется подмножество кольца, содержащее структуры с наилучшими значениями счетчика, то есть такими, которые по абсолютному значению наименее отличаются от соответствующих им значений эталонного времени. Такое подмножество получается из всего кольца отбрасыванием некоторого заранее заданного количества процентов от всех самых больших и самых малых значений. Этот ход позволяет отбросить неадекватные значения счетчика, которые могут быть результатами некоторого сбоя, слишком долгого обмена с устройством или резкого изменения условий работы устройства.
2) Далее вычисляется значение *линейного коэффициента*. Из уже выделенного подмножества кольца берутся самое старое и самое новое значение счетчика и соответствующие им значения системного времени. Затем линейный коэффициент считается, как отношение разностей выбранных значений системного времени и выбранных значений локального счетчика.
3) Затем результирующее значение эталонного времени, соответствующее значению счетчика x_i, вычисляется по формуле:

$$result = t_0 + \alpha * (x_i - x_0)$$

где t_0 - значение системного времени, соответствующее полученному в пункте (1) значению счетчика, α - линейный коэффициент, вычисленный в пункте (2), $x_i и x_0$ - значения счетчика, которое надо перевести в эталонное время, и которое получено в пункте (1) соответственно.

Алгоритм достаточно эффективен и высокоточен, но реализация требует хранения большого количества вспомогательных данных, а так же выполнения большого количества действий при каждом обновлении параметров. Так же алгоритм направлен не на синхронизацию с некоторой общей для нескольких узлов шкалой, а на аппроксимацию счетчика одного узла.

4.3 Алгоритм Кристиана

Алгоритм Кристиана [7] служит для синхронизации часов на узлах системы с некоторым эталонным временем. Условия работы алгоритма таковы: в системе

есть некоторый сервер, часы на котором считаются эталонными, и связанные с ним узлы, синхронизирующиеся с этим сервером. Каждый узел с некоторой частотой, специальной для него, опрашивает сервер о его текущем времени. В качестве первого приближения, когда отправитель получает ответ от сервера, он может просто выставить свои часы в полученное значение. Такая реализация имеет два недостатка, решения которых в алгоритме Кристиана рассмотрены ниже:

Первый недостаток в том, что такая реализация не учитывает то, что время никогда не должно идти назад, то есть значение счетчика не должно уменьшаться. Если часы отправителя спешат, то полученное от сервера время может оказаться меньше текущего значения времени отправителя. Простая подстановка времени сервера способна вызвать серьезные проблемы - например, объектные файлы, скомпилированные после того, как было изменено время узла, становятся помечены, временем более ранним, чем модифицированные исходные тексты, которые поправлялись до изменения времени узла, что может привести к невозможности дальнейшей трансляции исходных кодов.

Для решения этой проблемы изменения могут вноситься постепенно. Предположим, что таймер узла, на котором производится синхронизация, настроен так, что он генерирует 100 прерываний в секунду. В нормальном состоянии каждое прерывание будет добавлять ко времени по 10мс. При запаздывании часов сервера процедура прерывания будет добавлять каждый раз лишь по 9мс. Соответственно часы узла должны быть замедлены так, чтобы добавлять при каждом прерывании по 9мс.

Второй недостаток в том, что доставка сообщения от сервера к узлу занимает некоторое время, которое следует учитывать. Алгоритм Кристиана вычисляет это время следующим образом: за время доставки принимается половина разницы между временем отправки запроса и получения ответа. После получения ответа, чтобы получить приблизительное текущее время сервера, значение, содержащееся в сообщении следует увеличить на это число. Эта оценка может быть улучшена, если приблизительно известно, сколько времени сервер обрабатывает прерывание и работает с пришедшим сообщением.

Для повышения точности Кристиан предложил производить не одно измерение, а серию. Все измерения, в которых время доведения превосходит некоторое пороговое значение, отбрасываются, как недостоверные. Оценка делается по оставшимся замерам, которые могут быть усреднены для получения наилучшего значения, например, можно взять среднее арифметическое. Другим

подходом является рассмотрение только сообщения, пришедшего быстрее всех, как самого точного, поскольку оно предположительно попало в момент наименьшей загрузки сети, и потому более точно отражает время прохождения. Таким образом описанный алгоритм имеет следующие достоинства и недостатки:

Достоинства:

1) время передачи сообщения по сети частично учитывается и компенсируется
2) легок в реализации
3) достаточно эффективен в небольших системах

Недостатки:

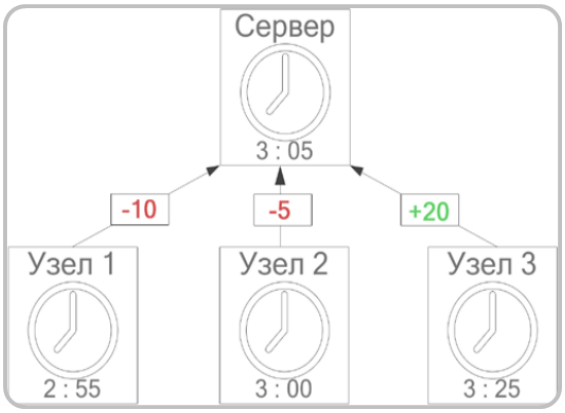
1) передача данных по сети занимает недетерминированное время, и полностью застраховаться от неточностей при его вычислении нельзя лишь усреднением
2) требуется наличие некоторого главного сервера, а значит при выходе его из строя вся система станет неработоспособна

4.4 Алгоритм Беркли

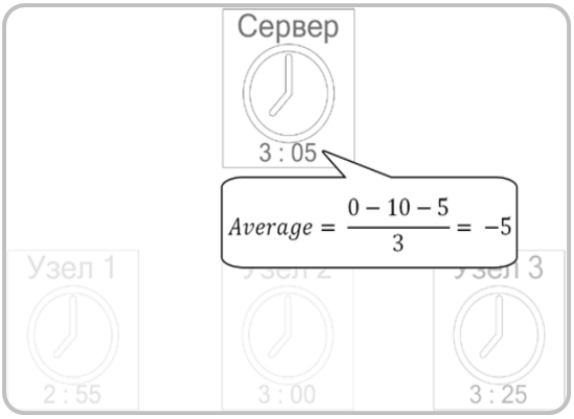
Алгоритм Беркли [8] может быть использован для синхронизации часов на узлах системы, в которой нельзя или слишком трудно узнать значение эталонного времени у сервера.

Определим условия работы алгоритма: в системе есть некоторый сервер, задающий эталонное время, и связанные с ним узлы. На каждом узле системы запущен процесс-демон, взаимодействующий с сервером. Когда в системе появляется новый узел, он запрашивает у главного сервера время, устанавливает его себе и становится равноправной частью системы.

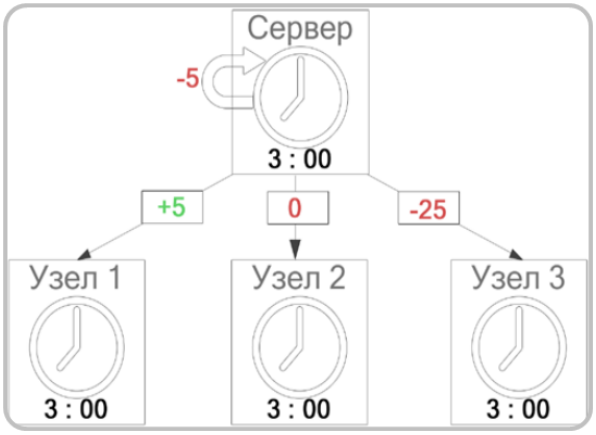
Через некоторые заранее выбранные промежутки времени сервер опрашивает все узлы сети о значениях их часов и запоминает, на сколько часы каждого узла отклоняются от времени сервера.



Затем происходит вычисление среднего отклонения, с отбрасыванием самых больших отклонений, как недостоверных.



После чего оно добавляется к часам главного сервера, и всем узлам производится рассылка нового времени.



Возможны модификации алгоритма, учитывающие время прохождения по сети сообщений для установки локальных часов узлов в новое время. Отметим достоинства и недостатки рассмотренного алгоритма:

Достоинства:

1) высокая эффективность для систем, некритичных к отклонениям времени
2) легок в реализации

Недостатки:

1) не учитывается время передачи данных по сети
2) требуется наличие некоторого главного сервера, а значить при выходе его из строя вся система выйдет из строя
3) "плохие" узлы (медленные, неустойчивые по частоте) оказывают влияние на общее время
4) время может отбрасываться назад

4.5 Отметки времени Лампорта

Отметки времени Лампорта [9] - простой алгоритм, служащий для определения порядка возникновения событий в РВС. Алгоритм требует минимальных ресурсов, а так же является основой для другого алгоритма - *Vector Clocks*. Отметки Лампорта позволяют определить порядок с помощью счетчиков, которые есть в каждом процессе. Каждый счетчик работает следующим образом:

1) процесс увеличивает значение своего счетчика перед каждым событием, которое в нем возникает
2) когда процесс взаимодействует с другими процессами посредством сообщений, он прикрепляет к каждому сообщению текущее значение счетчика
3) когда процесс получает сообщение, он устанавливает значение своего счетчика в наибольшую из двух величин: текущее значение своего счетчика и значение счетчика в пришедшем сообщении, и затем увеличивает это значение на 1.

Однако у этого алгоритма есть особенность: отметки Лампорта могут обеспечивать лишь *частичный* порядок событий между процессами. Введем обозначение *"событие A произошло до события B"* : A → B. Тогда условие непротиворечивости счетчика *C* можно сформулировать так:

Если A → B, *то* C(A) < C(B)

Сильное условие непротиворечивости имеет вид:

A → B тогда и только тогда, когда C(A) < C(B)

Частичный порядок означает выполнение только обычного условия непротиворечивости, но вовсе не гарантирует выполнение сильного условия. Отметим достоинства и недостатки рассмотренного алгоритма:

Достоинства:

1) высокая эффективность при малом числе узлов

2) высокая степень синхронизации - время переводится только вперед и нет необходимости ждать эффекта замедления на спешащих узлах
3) выход из строя одного или нескольких узлов не разрушит систему

Недостатки:

1) не учитывается время передачи данных по сети
2) сильное условие непротиворечивости не гарантируется к выполнению

5 Предложенный алгоритм

Разработанный алгоритм предназначен для синхронизации часов нескольких узлов РВС с эталонным временем, измеряемым в некоторой общей шкале. Эталонным считается время главного узла системы.

Согласно алгоритму, главный узел периодически опрашивает подключенные узлы о значениях их счетчиков для обновления параметров связанной с каждым узлом функции пересчета времени.

Введем такие же обозначения, как в математической модели, тогда параметры, ассоциированные с каждой функцией пересчета f_i, можно описать следующим образом:

- пары $P_k = \left(T_{A_i}(t_k), t_k\right),\ 2 \le k \le N$, то есть время счетчика узла A_i, измеренное в момент времени t_k и соответствующее ему эталонное время t_k. Количество пар $N \ge 2$.
- шаг шкалы счетчика узла, то есть частота его инкрементации (остается неизменным)
- α_i - коэффициент пересчета, вычисляемый на основе пар $P_k, 2 \le k \le N$, и связанный с узлом A_i

Частота опроса каждого конкретного узла выбирается алгоритмом, как минимальное между двумя значениями: частота инкрементации счетчика узла, помноженная на некоторый коэффициент, и заранее фиксированная минимальная частота.

Например, пусть частота счетчика устройства - 40 наносекунд, заранее фиксированная частота таймера - 1 секунда (10^9нс), а коэффициент равен 1000. Тогда получим, что частота обращений к узлу будет выбрана, как минимальное между 1000 * 40 и 10^9, то есть будет равна 1 секунде.

Такое отсечение по минимальной частоте необходимо для того, чтобы снизить нагрузку на главный узел, а так же, чтобы между обращениями к узлам значения их счетчиков инкрементировались достаточное количество раз для возможности сделать некоторые средние выводы об их работе.

Когда возникает необходимость пересчитать время узла в эталонное, то вычисляется коэффициент пересчета α_i на основе значений в парах $P_k, 2 \le k \le N$; затем на основе множества $P_k, 2 \le k \le N$ получается пара (T_P, t_P), которая должна характеризовать связь значений счетчика узла и времени главного узла. Например, можно взять усреднение значений в каждой паре, или, отбросив

некоторый процент пар с самыми большими и самыми малыми значениями счетчика узла, взять из оставшихся последнюю пару.

Затем результат - эталонное время, соответствующее значению счетчика $T_{A_i}(t)$, вычисляется по формуле:

$$result = \alpha_i * T_{A_i}(t) + (t_P - \alpha_i * T_P)$$

Здесь слагаемое $\alpha_i * T_{A_i}(t)$ переводит текущее значение счетчика в шкалу эталонного времени, а затем уже в этой шкале с помощью слагаемого $(t_P - \alpha_i * T_P)$ компенсируется разница в моментах времени старта счетчиков узла и главного узла, а так же учитывается переполнение счетчика узла.

Достоинства:

1) время на узлах не меняется, алгоритм лишь переводит конкретные значения счетчиков времени в общую шкалу. Время переводится только вперед и нет необходимости ждать эффекта замедления на спешащих узлах
2) узлы друг с другом не связаны и не могут влиять на время других узлов
3) алгоритм учитывает ход часов каждого узла за некоторый промежуток времени
4) точность пересчета можно выбирать, заранее задав необходимое количество пар P_k

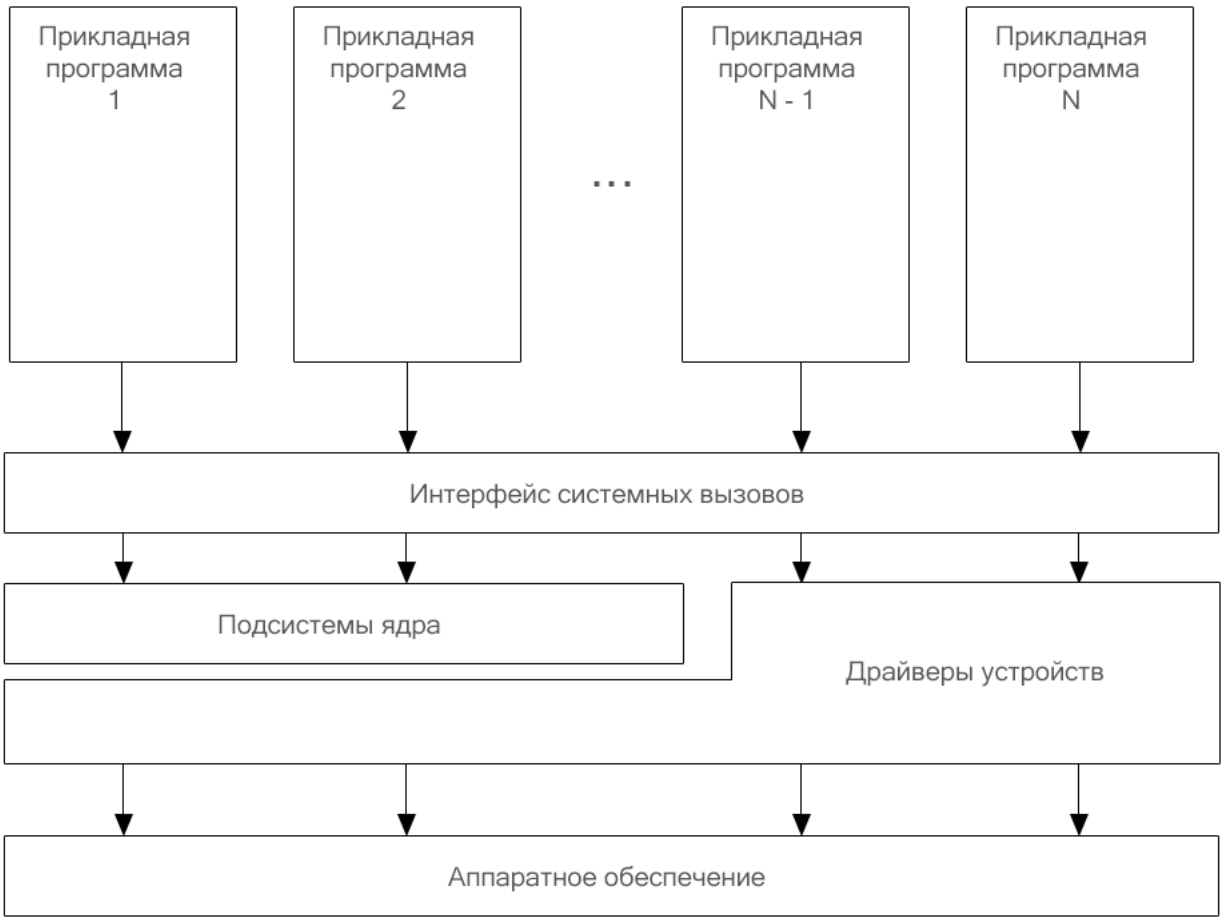
Алгоритм построен так, что на любом этапе его работы достаточно легко вносить модификации, то есть алгоритм легко поддается расширению и настройке.

Будем далее рассматривать только простой случай:

- количество пар P_k равно двум, то есть множество P_k , $2 \le k \le N = \{(T_{A_i}(t_k), t_k)\}_{k=1}^{2}$, и $T_{A_i}(t_2) < T_{A_i}(t_1)$, $t_2 < t_1$
- $\alpha = \frac{t_1 - t_2}{T_{A_i}(t_1) - T_{A_i}(t_2)}$

6 Архитектура ядра Linux

Ядро Linux - это программное обеспечение, которое предоставляет базовые функции для всех остальных частей операционной системы, занимается управлением аппаратурой и распределяет системные ресурсы [5]. Ядро имеет доступ ко всем областям защищенной памяти и полный доступ к аппаратному обеспечению. Состояние системы, в котором находится ядро, и область памяти, в которой оно расположено, вместе называются *пространством ядра*. Прикладные программы, работающие в системе, взаимодействуют с ядром посредством интерфейса *системных вызовов*.



В Linux поддерживается возможность динамической загрузки в ядро внешних бинарных файлов - модулей ядра, которые так же динамически могут выгружаться из него. После загрузки модуль размещается и работает в пространстве ядра. Для работы с аппаратурой используется особый вид модулей - *драйверы устройств*. Драйвер предназначен для того, чтобы скрыть детали работы его устройства, и предоставить некоторый программный интерфейс для работы с устройством [4].

Будучи состоящим из динамически загружаемых и выгружаемых модулей, ядро является объектной системой. Проводя аналогию с объектно-ориентированным программированием, модули являются реализациями классов, и драйверы в том числе. При загрузке драйвера в ядро, он *инициализируется* - создает некоторый объект, хранящий информацию о поддерживаемых устройствах, и регистрирует его в ядре. Затем при каждом обнаружении нового устройства ядро вызывает его драйвер, чтобы он создал объект, хранящий служебную информацию об этом устройстве и зарегистрировал его в различных подсистемах ядра. Например, в подсистеме, организующей взаимодействие драйвера с пользователем через файл-устройство или объекты в *sysfs* - виртуальной файловой системе в ОС Linux.

При разработке модулей ядра иногда получается, что у нескольких модулей появляется некоторая общая функциональность, которую удобно выделить в отдельный модуль - *подсистему*. Подсистема может быть одного из двух видов:

1) реализовывать некоторый обобщенный интерфейс для пользователя и вызывать драйвер только для взаимодействия с устройствами.
2) являться служебной подсистемой - предоставлять любому использующему ее модулю некоторую функциональность. Например, подсистема для использования высокоточных таймеров *hrtimer* [6], предоставляющая функционал для запуска таймеров, обработки их срабатываний, перезапуска, отмены.

В данной работе мы рассмотрим решение задачи синхронизации часов, выполненное в виде подсистемы второго типа, так как необходимо создать обобщенную подсистему, которую смогут использовать различные драйвера.

7 Описание программной реализации

7.1 Общий принцип работы

Алгоритм реализован в виде подсистемы ядра Linux, предоставляющей функционал различным драйверам для пересчета временных меток событий на их устройствах. Подсистема представляет из себя управляющий сервер, к которому могут подключаться драйверы устройств, а реализована она в виде загружаемого модуля ядра Linux. Реализован простой случай алгоритма, рассмотренный в конце описания общего алгоритма.

Каждый подключенный драйвер должен включать в себя заголовочный файл, где определены функции для взаимодействия с подсистемой. Для регистрации в подсистеме нового устройства его драйвер заполняет структуру с информацией об этом устройстве и посылает в подсистему запрос на регистрацию, после принятия которого драйвер может транслировать локальное время устройства в эталонное.

7.2 Взаимодействие с подсистемой

1) Регистрация в подсистеме

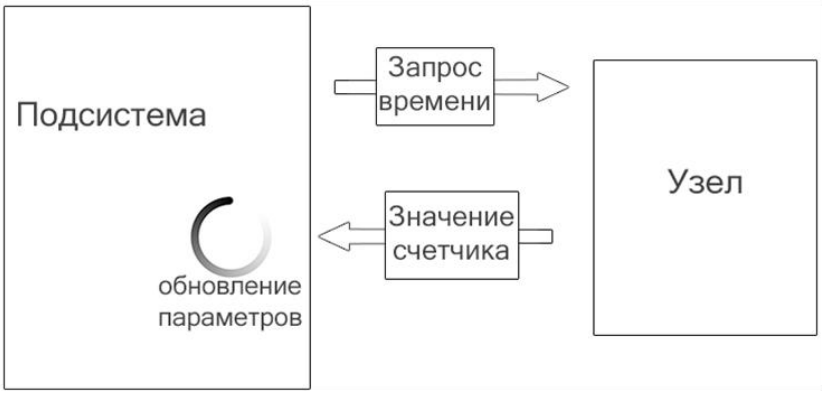
Рассмотрим регистрацию в подсистеме очередного узла. Для регистрации узел посылает запрос подсистеме, содержащий частоту его счетчика времени, его имя (опционально) и указатель на функцию, с помощью которой подсистема будет периодически опрашивать этот узел о значениях его счетчика.

Если подсистема может принять этот узел (достаточно памяти под создание параметров для пересчета времени), то с узла считывается текущее значение его счетчика, затем считывается текущее время подсистемы и полученными значениями инициализируется первая пара t_1 и ts_1 - значение

счетчика и соответствующее ему время подсистемы, - которая сохраняется в структуру, в дальнейшем ассоциируемую с данным узлом.

В этой же структуре создается объект таймера, частота срабатывания которого вычисляется согласно общему алгоритму с минимальной частотой - 1 секунда, и коэффициентом, на который умножается частота счетчика узла, равным 1000. Затем таймер запускается и узел может конвертировать значения своего счетчика во время подсистемы.

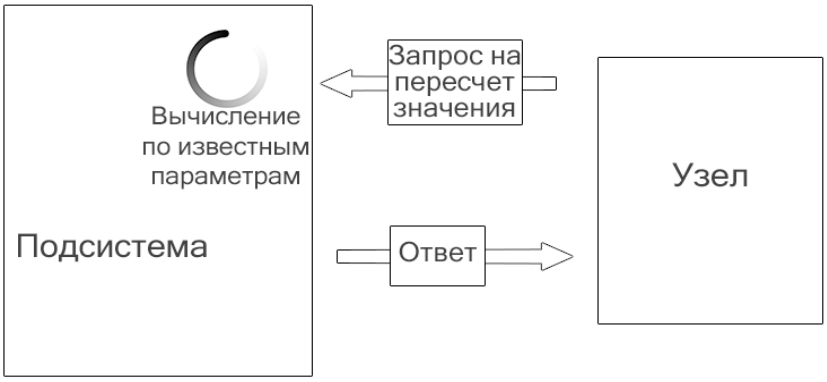
2) Обновление параметров



Рассмотрим один конкретный узел, зарегистрированный в подсистеме. По срабатыванию таймера, связанного с этим узлом и запущенного при регистрации, подсистема запрашивает у узла значение его локального счетчика посредством вызова функции, указатель на которую узел передал при регистрации.

После получения значения подсистема считывает собственное время и получается новая пара - t_{cur} и ts_{cur} - считанное значение времени узла и время подсистемы соответственно. Затем эта пара запоминается на место t_1 и ts_1, а t_2 и ts_2 получают старые значения t_1 и ts_1. То есть происходит сдвиг, при котором забывается самая старая пара значений, и запоминается новая. После такого обновления параметров таймер перезапускается.

3) Пересчет времени



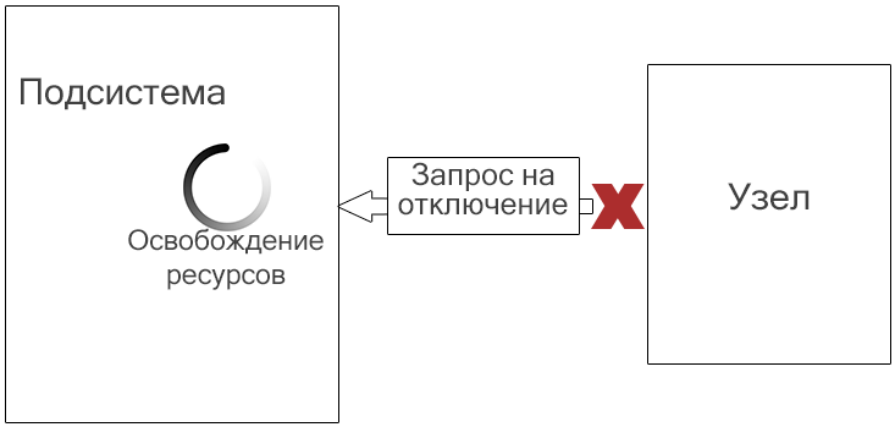
Когда у узла возникает необходимость конвертировать некоторое значение своего счетчика во время подсистемы, то он посылает подсистеме запрос на пересчет времени посредством вызова функции пересчета, которую подсистема делает доступной всем модулям ядра при своем запуске.

Подсистема, получив запрос, на основании известных на момент обращения параметров определяет, какое время соответствует значению счетчика. Пересчет значения выполняется согласно формуле, описанной в общем алгоритме, а за пару, используемую в формуле, принимается самая новая - t_1 и ts_1. Таким образом, формула, по которой будет посчитано результирующее время, имеет вид:

$$result = \alpha * t + (ts_1 - \alpha * t_1)$$

где t - значение счетчика, которое необходимо конвертировать.

4) Завершение работы с подсистемой



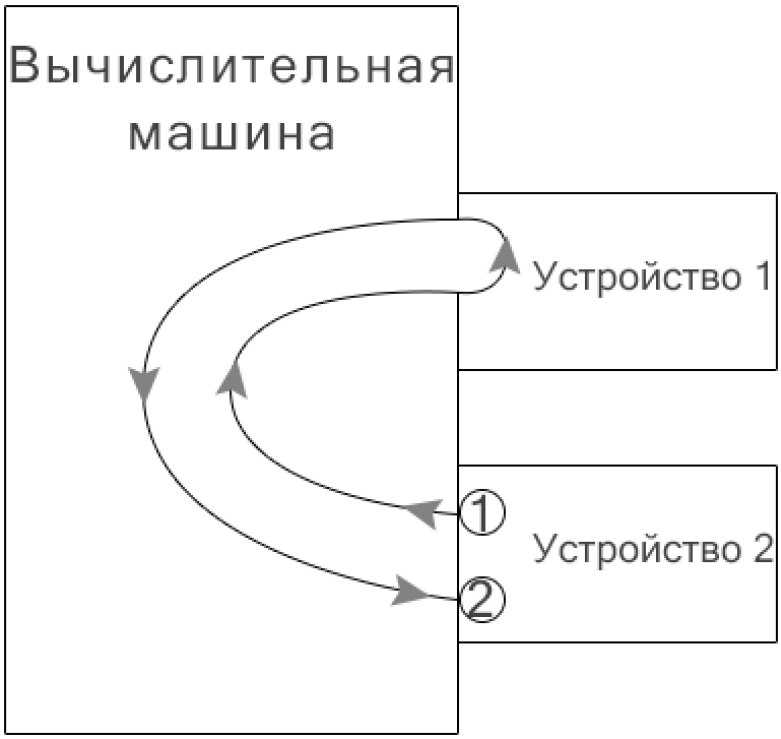
Когда у узла больше нет необходимости в использовании подсистемы, он должен отправить подсистеме запрос на отключение посредством вызова функции, которую подсистема делает доступной всем модулям ядра при своем запуске.

Подсистема при получении запроса освобождает ресурсы, связанные с отключающимся узлом: останавливается его таймер, освобождается динамически выделенная память. После отключения узел больше не может конвертировать свое время в эталонное.

8 Экспериментальная апробация

В данном разделе приведены результаты численного исследования эффективности работы алгоритма, представленного в работе. Для тестирования подсистемы был написан тестовый модуль, представляющий собой драйвер виртуального устройства, который при загрузке в ядро запускает таймер, срабатывающий через заданные заранее равные промежутки времени и инкрементирующий значение своего локального программного счетчика. Так же на реализованную подсистему был портирован драйвер сетевого адаптера pc_t301. Здесь приведены результаты работы подсистемы на портированном драйвере.

Схема системы, на которой производилось тестирование



Подсистема запускается на главном узле - вычислительной машине, сетевые адаптеры *Устройство 1* и *Устройство 2* подключаются к вычислительной машине и регистрируются в подсистеме, на ней запущенной.

Пакеты данных посылаются с *Устройства 2* со входа ①, принимаются вычислительной машиной - главным узлом, время которого считается эталонным. Здесь время, аппаратно проставленное *Устройством 2* в пришедшем пакете данных, конвертируется подсистемой во время главного узла и сравнивается с программной меткой, которая проставлялась в драйвере до его портирования на

подсистему. Затем пакет посылается дальше - на *Устройство 1*, которое ставит ему свою метку времени и шлет обратно.

На обратном пути подсистема конвертирует метку времени пришедшего пакета во время главного узла, и результат снова сравнивается с программной меткой. Затем пакет возвращается к *Устройству 2*.

В процессе тестирования проводился обычный тест с редкими посылками пакетов, и стресс-тест из 1000 пакетов подряд.

Полученные результаты на простом тесте

	Среднее	Среднеквадратичное отклонение	Наибольшая разница	Наименьшая разница	Медиана
Разница программных меток с метками подсистемы	138,5 миллисекунд	254,1 миллисекунд	763,8 миллисекунд	182 микросекунды	329 микросекунд
Разница меток подсистемы у пакета до прохождения *Устройства 1* и после	831,6 микросекунд	181,7 микросекунд	1,3 миллисекунды	249,9 микросекунды	824,96 микросекунд

Значения в первой строке показывают, что предложенный алгоритм значительно эффективнее программных меток. Значения второй строки показывают разницу меток у пакета до достижения *Устройства 1* и после - она характеризует время, которое тратится на прохождение пакета по среде передачи данных, на аппаратное проставление меток и погрешность реализации алгоритма.

Полученные результаты на стресс-тесте

	Среднее	Среднеквадратичное отклонение	Наибольшая разница	Наименьшая разница	Медиана
Разница программных меток с метками подсистемы	3,1 миллисекунды	2,93 миллисекунд	7,6 миллисекунд	227 микросекунд	4,4 миллисекунды
Разница меток подсистемы у пакета до прохождения *Устройства 1* и после	325,7 микросекунд	254,1 микросекунд	980,2 микросекунд	140 микросекунд	209,9 микросекунд

Из второй таблицы видно, что предложенный алгоритм выигрывает у программных меток и на стресс-тесте, а средняя разница временных меток у пакетов до прохождения *Устройства 1* и после стала меньше в силу очень плотной загрузки среды передачи данных.

Стоит отметить, что точность значительно повысилась по сравнению с программными метками даже на реализации упрощенного алгоритма, из чего следует, что повышение количества пар временных меток, ассоциированных с каждым узлом, а так же более сложный выбор пары, используемой в формуле пересчета, могут дать еще более высокую точность.

9 Заключение

В результате выполнения курсовой работы достигнуты следующие результаты:

- Разработана подсистема синхронизации временных меток.
- Разработан тестовый драйвер виртуального устройства для проверки работоспособности разработанной подсистемы.
- Осуществлено портирование на подсистему драйвера сетевого адаптера pc_t301.
- Проведена апробация разработанной подсистемы и численные исследования временных характеристик.

Список литературы

1. Таненбаум Э., Ван Стеен М. Распределенные системы. Принципы и парадигмы. Санкт-Петербург, 2003 г.
2. Jin H., Buyya R., Baker M. Cluster computing tools, applications, and Australian initiatives for low cost supercomputing //MONITOR Mag.(The Institution of Engineers Australia). – 2001. – Т. 25. – №. 4.
3. Ключев А. О. и др. Программное обеспечение встроенных вычислительных систем //СПб.: СПбГУ ИТМО. – 2009.
4. Rubini A., Corbet J. Linux device drivers. – " O'Reilly Media, Inc.", 2001, 1 — 26с
5. Роберт, Лав Ядро Linux: описание процесса разработки = Linux Kernel Development: 3-е изд. / Лав Робертю – М.:Вильямс, 2012. 24 — 31 с.
6. Gleixner T., Niehaus D. Hrtimers and beyond: Transforming the linux time subsystems //Proceedings of the Linux symposium. – 2006. – Т. 1. – С. 333-346.
7. Cristian F. Probabilistic clock synchronization //Distributed computing. – 1989. – Т. 3. – №. 3. – С. 146-158.
8. Gusella R., Zatti S. The accuracy of the clock synchronization achieved by tempo in Berkeley Unix 4.3 BSD. – University of California. Computer Science Division, 1987.
9. Lamport L. Time, clocks, and the ordering of events in a distributed system //Communications of the ACM. – 1978. – Т. 21. – №. 7. – С. 558-565.
10. Correll K., Barendt N., Branicky M. Design considerations for software only implementations of the IEEE 1588 precision time protocol //Conference on IEEE. – 2005. – Т. 1588. – С. 11-15.
11. Ohly P., Lombard D. N., Stanton K. B. Hardware assisted precision time protocol. Design and case study //Proceedings of LCI International Conference on High-Performance Clustered Computing. Urbana, IL, USA: Linux Cluster Institute. – 2008. – Т. 5. – С. 121-131.
12. Исходные коды ядра Linux версии 3.7, файловый путь: /include/linux/timecompare.h, /include/linux/timecompare.c