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

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. Будем далее рассматривать только определенный класс таких систем, состоящих из множества узлов - *распределенные вычислительные системы*.

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. Здесь приемы пакетов и будут событиями. Данные, принятые разными сетевыми адаптерами могут быть взаимосвязаны, например, могут являться частью какой-либо структуры данных: объекта класса, файла базы данных, и чтобы правильно эту связь отследить, необходимо знать, в каком порядке эти пакеты пришли в систему: в случаях файла базы данных и объекта класса - для того, чтобы правильно их собрать.

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

Примечание [1]: Плохое предложение, надо явно самому сделать переход, что-то вроде «Такие системы из множеста частей называют РВС.»

Примечание [2]: Не очень как-то звучит, может лучше «части которого разнесены на значительное расстояние друг от друга»

Примечание [3]: Ссылка на источник определения

Примечание [4]: Лучше «Зачастую необходимо».
«Иногда случается» — это что-то про ТерВер.

Примечание [5]: И что? Надо пример, объясняющий зачем нужны метки времени. Что-то про мониторинг трафика по нескольким каналам с отслеживанием взаимосвязей между пакетами.

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.

~~- Время течет непрерывно, в связи с чем получить абсолютно точно значение астрономического времени возникновения какого-либо события в системе невозможно, а иногда и не нужно. Поэтому будем говорить, что метки времени должны максимально точно определять момент возникновения события в системе относительно некоторого времени, принятого за эталонное.~~

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. Их 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 ее динамическая коррекция.

Примечание [6]: Переход непонятный.
Нужно сделать переход
Счетчик → время узла в какой-то общей шкале и его несоответствие астрономическому времени. Здесь надо явно упомянуть функцию пересчета f(x) → T

Примечание [7]: Получить точное значение времени невозможно не потому, что время непрерывно, а счетчик дискретен, а потому, что «скорость» счетчика плавает.

Примечание [8]: Смысл этого предложения от меня несколько ускользает.

Примечание [9]: Немножко странное на первый взгляд утверждение.
Чтобы было понятнее, я бы переформутировал примерно так:
0.сразу оговоримся, что это это один из основных подходов, но не единственный.
1.сказал бы что для определенности примем одни из часов в РВС за эталонные.
2.временные метки всех событий в системе пересчитываются в значения эталонных часов.
То есть для каждого аппаратного счетчика надо задать функцию f(x) → T которая пересчитывает в некоторое эталонное время.
Так как есть флуктуация хода локального счетчика, то эта функция не постоянна, а меняется с ходом времени.То есть, задача синхронизации часов, это задача построения для всех узлов системы таких функций f(x) → T и динамическая в процессе работы системы модификация этих функций.
(Либо можно как-то сказать, что эти функции параметризованы некоторым дополнительным параметром, который приходится в динамике подкручивать.)

Переходя к конкретным примерам распределенных 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.

Мы рассмотрим третий из описанных выше примеров, то есть задачу *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. Эти события мы можем помечать метками времени в моменты обработки прерываний адаптера, вызванных приходами 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. Таким образом приходим к *задаче 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.

Примечание [10]: Списки надо оформлять списками, иначе форматирование очень странное получается.

Примечание [11]: Надо расшифровать в чем проблема. Что есть несколько адаптеров, события на них мы можем помечать моментами обработки прерывания от них, но задержка обработки прерывания не детерминирована + адаптер может сгенерировать одно прерывание а при чтении с него данных, обнаружится, что у него в буферах уже несколько событий. Поэтому надо чтобы адаптер ставил метки по своему внутреннему счетчику в моменты аппаратного возникновения события.
Поэтому возникает задача пересчета/синхрониазции.

2 Постановка задачи

Примечание [12]: Целью работы является изучение …, а именно:
1.
2.
3.

По итогам выполнения работы должны быть получены следующие результаты:
1.
2.
3.

Целью работы является изучение задачи *синхронизации часов*, а именно:

1) Изучение предметной области, существующих механизмов синхронизации времени в распределенных вычислительных системах
2) Изучение существующих механизмов пересчета временных меток в ядре Linux
3) Написание подсистемы пересчета и тестового модуля, который будет пересчитывать время источника в реальное время при помощи созданной подсистемы

По итогам выполнения работы должны быть получены следующие результаты:

1) Разработка обобщенной подсистемы пересчета временных меток, которая может быть использована различными драйверами оборудования, и тестового окружения для нее
2) Портирование на новую подсистему драйверов *ta1-usb*, *afc*

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 множества значений счетчика* T_{A_i} *и набора динамически изменяемых параметров в множество 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, учитывающего цикличность значений счетчика (он может переполняться), 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 будут 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

Рассмотрим алгоритм, реализованный в сетевой подсистеме ядра Linux. Для этого сначала определим алгоритм PTP.

Для точной синхронизации времени между несколькими компьютерами сети используется алгоритм *PTP (Precision Time Protocol)* [10]. PTP может быть реализован полностью программно или с аппаратной поддержкой - так называемый *Hardware-assisted PTP* [11]. Они отличаются тем, что полностью программная реализация ставит событиям метки времени программно в моменты обработки прерываний, а реализация с аппаратной поддержкой ставит метки в моменты аппаратного возникновения событий, что существенно точнее программной версии. Например, полностью программной реализацией является Linux *PTPd (Precision Time Protocol daemon) версии 1*, который реализует алгоритм PTP и служит для синхронизации времени компьютеров в сети LAN.

Для работы алгоритма, реализованного в сетевой подсистеме ядра Linux, необходимо точно регистрировать моменты времени прихода пакетов из сети, поэтому метки лучше ставить аппаратно, то есть использовать *Hardware-assisted PTP*. Таким образом возникает задача *синхронизации часов*. В ядре Linux до версии 3.8 существовали вспомогательные структуры для решения этой задачи для синхронизации часов сетевых адаптеров *(Network Interface Controller)*. Алгоритм был реализован специально для случая, когда эталонным временем считается время устройства, к которому подключены сетевые адаптеры, а узлами являются сетевые адаптеры, но так же алгоритм мог быть использован для решения других задач синхронизации. 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

Примечание [13]: Написать внятно историю:
1. для точной синхронизации времени между несколькими компьютерами в сети используется PTP
2. Для работы необходимо точно регистрировать моменты времени прихода пакетов по сети.
3. Метки можно считать программно в момент обработки прерываний. Именно так реализовали в Linux ptpd версии 1.
4. Лучше, использовать метки от адаптеров (т. н. Hardware-assisted PTP), которые они умеют проставлять, но тогда возникает задача пересчета этих меток. В Linux это было решено в версии X.
5. В версии Y это выпилили, потому что сочли решение слишком специфичным, а функцию пересчета вообще вынесли в userspace.

Тем не менее рассмотрим использоыванное решение:

Примечание [14]: Непонятно кого с кем синхронизируют, кто источник, а кто цель и что всё это значит.

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:

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

,где *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 75% 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 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.

4.3 Алгоритм Кристиана

Алгоритм Кристиана[7] служит для синхронизации часов на узлах системы 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 самое точное, 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:

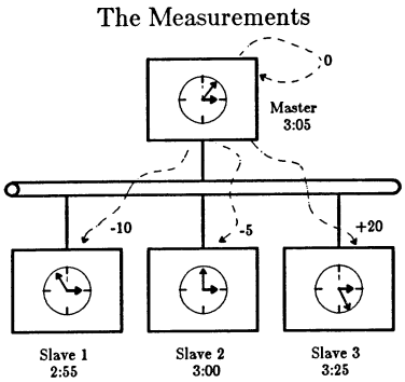
1) передача данных по сети занимает недетерминированное время, и полностью застраховаться от неточностей при его вычислении нельзя лишь усреднением
2) требуется наличие некоторого главного сервера, а значить при выходе его из строя вся система выйдет из строя

4.4 Алгоритм Беркли

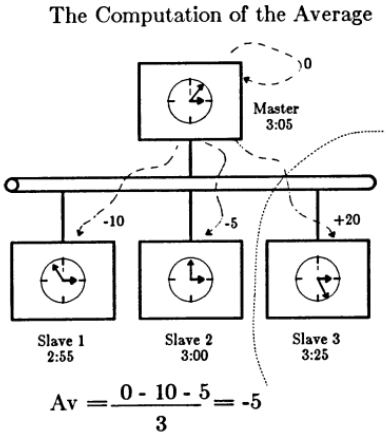
Алгоритм Беркли[8] может быть использован для синхронизации часов на узлах системы, в которой нельзя или слишком трудно узнать значение эталонного времени у сервера.

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.

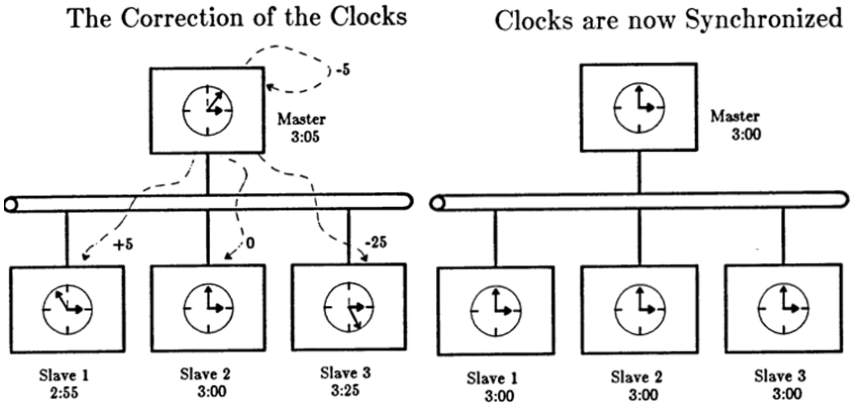
Через некоторые заранее выбранные промежутки времени сервер опрашивает все узлы сети о значениях их часов и запоминает, на сколько часы каждого узла отклоняются от времени сервера.



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

4.5 Отметки времени Лампорта

Отметки 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:

$A \rightarrow B$ 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:

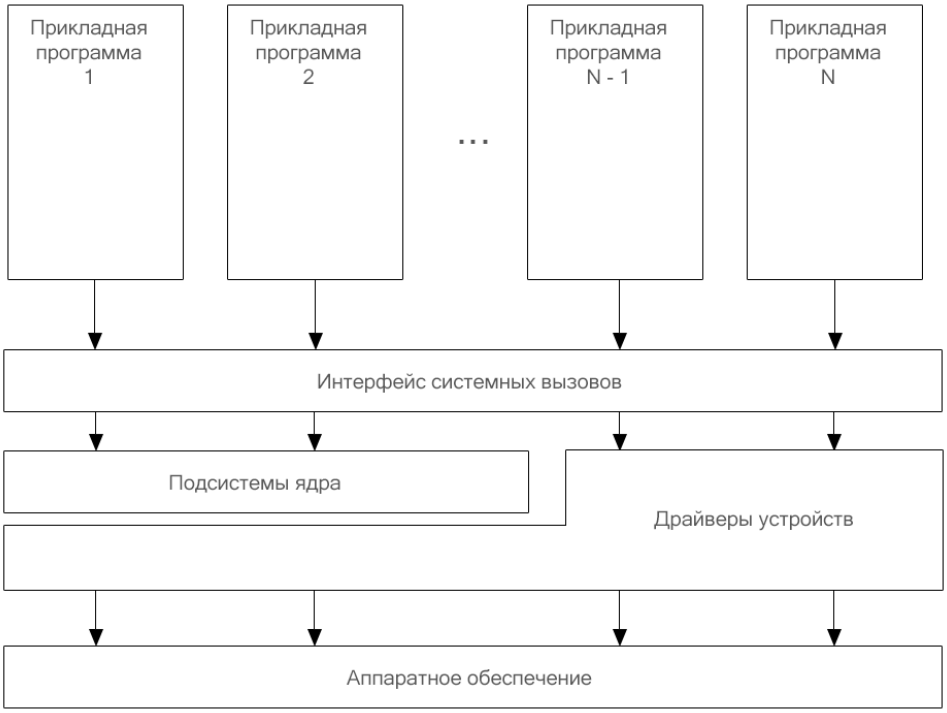
1) высокая эффективность при малом числе узлов
2) высокая степень синхронизации - время переводится только вперед и нет необходимости ждать эффекта замедления на спешащих узлах
3) выход из строя одного или нескольких узлов не разрушит систему

Недостатки:

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

5 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, а отличается *hrtimer* от существующих в ядре обычных таймеров тем, что их точность существенно выше.

В данной работе мы рассмотрим решение задачи синхронизации часов, выполненное в виде подсистемы второго типа.

Примечание [15]: Я просил расписать следующее:
1.Ядро — объекная штука
2.модуль/драйвер — реализация класса
3.при инициализации драйвера он создает некоторый объект, хранящий информацию о поддерживаемых устройствах, и регистрирует его в ядре.
4.При обнаружении ядром устройства оно вызывает драйвер, чтобы тот создал объект, хранящий всю служебную информацию об устройстве и зарегистрировал его в различных подсистемах ядра: например в подсистеме, организующей взаимодействие драйвера с userspace через файл-устройство или объекты в sysfs

Примечание [16]: Здесь про два вида подсистем:
1. подсистема прослойка, которая реализует некоторый обобщенный интерфейс для userspace, и вызывает драйвер только для взаимодействия с железом.
2. «служебная» подсистема, предоставляющая любому использующему его модулю некоторую функциональность (найти/придумать пример)

6 Описание программной реализации

Примечание [17]: Нужно описание архитектуры, а не список функций.

6.1 Общий принцип работы

Алгоритм реализован в виде подсистемы ядра Linux, предоставляющей функционал различным драйверам для синхронизации часов их устройств. Подсистема представляет из себя управляющий сервер, к которому могут подключаться драйверы устройств.

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 во время, соответствующее эталонному.

6.2 Описание реализации подсистемы

Подсистему необходимо загрузить в ядро прежде, чем к ней попытается подключиться хотя бы один узел. Сразу после загрузки подсистема готова принимать запросы на регистрацию новых устройств и на отключение уже подключенных.

При получении запроса на регистрацию подсистема получает из него описание работы локальных часов устройства. Затем на основании этого описания для устройства запускается таймер, который при каждом срабатывании будет опрашивать устройство о значении его счетчика времени, и обновлять параметры, используемые для пересчета времени.

После регистрации, устройство может конвертировать свое время в эталонное вызовом функции, реализующей следующую формулу:

$$result = \alpha * source\_time + (target_1 - \alpha * source_1)$$

$$\text{где } \alpha = \frac{target_1 - target_2}{source_1 - source_2}$$

$source_1, source_2$ - последние два полученных значения локального счетчика устройства, соответствующие значениям эталонного времени $target_1$ и $target_2$. $source\_time$ - значение локального счетчика устройства, а $result$ - соответствующее ему эталонное.

При получении запроса на отключение устройства освобождаются ресурсы, выделенные для устройства подсистемой и останавливается соответствующий таймер.

6.3 Описание процесса работы с подсистемой

Драйвер, при регистрации устройства в подсистеме, предоставляет ей функцию, вызовом которой подсистема сможет получить значение локального счетчика устройства. Если регистрация прошла успешно, то с этого момента устройство может в любой момент конвертировать свое локальное время в эталонное. Когда узлу больше не требуется использовать подсистему, он должен отключить себя от подсистемы.

7 Экспериментальная апробация

В данном разделе приведены результаты исследования работы алгоритма, описанного в работе. Для тестирования подсистемы был написан тестовый модуль, представляющий собой драйвер устройства, который при загрузке в ядро запускает таймер, срабатывающий через заданные ранее равные промежутки времени и инкрементирующий значение своего локального программного счетчика.

После загрузки драйвер создает файл, представляющий устройство, в */dev*, и из которого можно прочитать значение локального счетчика в любой момент времени. Событиями являются попытки чтения значения локального счетчика устройства. При инициализации драйвер заполняет вспомогательную структуру для подсистемы и посылает запрос на регистрацию. После успешной регистрации выполняются операции чтения из файла устройства, созданного драйвером, и на каждое чтение локальное время счетчика конвертируется в эталонное время подсистемы.

8 Заключение

В ходе выполнения курсовой работы были получены следующие результаты:

1) Изучены существующие механизмы синхронизации времени в распределенных вычислительных системах и существующий алгоритм в ядре Linux
2) Разработана обобщенная подсистема пересчета временных меток, которая может быть использована различными драйверами оборудования
3) Разработан тестовый модуль, конвертирующий свое локальное время в эталонное время подсистемы

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.