

Оптимизация библиотеки нитей NPTL в составе ОС Linux для систем жесткого реального времени

А.В.Федоров

Выпускник МФТИ 2010 года.

e-mail: Alexander.V.Fedorov@mcst.ru, alexfv@bk.ru

Получено 20.01.2011 г.

Аннотация: Библиотека NPTL полностью поддерживает нити POSIX – один из самых популярных интерфейсов для создания многопоточных приложений. Она постепенно перерабатывается с учетом требований систем реального времени, но на настоящее время подходит только для систем мягкого реального времени. В статье описываются причины, по которым библиотека не соответствует требованиям жесткого реального времени, и делается попытка их устранить.

1. Введение

В настоящее время отделением операционных систем ЗАО "МЦСТ" выполняется ряд проектов, ставящих целью адаптацию операционной системы Linux для использования в вычислительных комплексах (ВК) эльбрусской серии, работающих в режиме жесткого реального времени. В этом контексте важное значение имеет адаптированная к реальному времени реализация нитей (threads) в пользовательском интерфейсе POSIX (Portable Operating System Interface for Unix). Все примитивы этого интерфейса можно разделить на две группы: первая отвечает за создание и завершение нитей, вторая – за их синхронное исполнение. В статье рассматриваться будет только вторая группа: в системах жесткого реального времени она имеет куда большее значение, потому что всю инициализацию возможно провести при старте системы, а вот от синхронизации нитей при параллельных вычислениях никуда не деться.

В процессе работы автором была проанализирована библиотека NPTL (Native Posix Thread Library) на предмет ее пригодности для систем жесткого реального времени, а поскольку ему представляется, что этот анализ представляет наибольший интерес для читателя, он составляет основное содержание статьи. В начале дается обзор системного вызова `sys_futex`, затем рассказывается о деталях реализации объектов синхронизации, и в конце приводится измерение эффективности проведенной оптимизации.

2. Особенности реализации объектов синхронизации в NPTL

Библиотека NPTL разработана специально для Linux, что дает возможность вносить в интерфейс ядра изменения, ориентированные на ее поддержку. Таким образом в существующий только в Linux системный вызов `sys_clone` были внесены изменения, необходимые для создания эффективного механизма создания и завершения нитей, а для эффективной реализации любых объектов синхронизации (например, семафоров или условных переменных) был разработан системный вызов `sys_futex`, поддерживающий одноименный объект [1].

Конечно, самым простым решением было бы реализовать эти объекты полностью в адресном пространстве ядра и сделать системный вызов, который просто перенаправлял бы запросы приложения в ядро ОС. Более того, тогда можно было бы использовать уже имеющиеся в ядре реализации блокировок и семафоров – достаточно просто сопоставить каждому пользовательскому объекту "истинный" объект в ядре. Недостаток такого решения в том, что каждая операция будет использовать системный вызов, а это дорогое удовольствие (потеря нескольких сотен тактов и стирание содержимого кэша процессора).

Для данной проблемы разработчиками NPTL было найдено простое решение. Роль любого объекта синхронизации – задержать исполнение одной нити до наступления какого-либо события, однако, если в какой-то момент приостанавливать исполнение нет необходимости, то нет ее и в системном вызове. На практике такая оптимизация реализуется хранением состояния объекта в переменной, находящейся в пользовательском пространстве и являющейся его частью, тогда состояние можно изменять без системных вызовов.

Такая переменная и называется фьютексом. Приложение проверяет состояние объекта, считав значение фьютекса и, если не нужно блокироваться (например, когда закрывается открытый семафор) или будить ожидающие нити (когда открывается закрытый один раз семафор), просто записывает во фьютекс новое состояние. Чтобы не возникали состояния гонки, чтение и запись делаются одной атомарной инструкцией. Системный же вызов `sys_futex` вызывается либо для пробуждения ждущей нити (тогда он удаляет нить из очереди в ядре и помечает ее как готовую к исполнению), либо когда нужно блокироваться (тогда он добавляет нить в очередь, причем прямо перед добавлением значение фьютекса проверяется еще раз на тот случай, если оно успело

измениться и блокироваться уже не нужно). На его основе в NPTL реализуются блокировки и семафоры, а на их основе – все остальные объекты.

Именно из-за этой оптимизации в расшифровке слова `futex` (Fast Userspace MUTex) есть слово `Fast`.

3. Об универсальности использования фьютексов

В системах жесткого реального должно быть гарантировано завершение операций за определенное время. Помимо этого, следует добиться улучшения следующих основных параметров:

- 1) Максимального времени исполнения операций.
- 2) Среднего времени исполнения операций.
- 3) Задержки между внешним событием и реакцией на него (для ОС она напрямую зависит от длины критических секций, защищенных спинлоками).

Под "операциями" здесь и далее будут пониматься функции, работающие с объектами синхронизации: блокировками, барьерами, условными переменными. (Еще в стандарте определены блокировки чтения-записи, но в системах реального времени они практически не используются из-за серьезных проблем с инверсией приоритетов – ситуацией, когда нить с приоритетом большим, чем у владельца, но меньшим, чем у ожидающей нити, занимает процессор и не дает владельцу освободить блокировку, тем самым блокируя и ожидающую нить [2].)

В библиотеке нити POSIX реализованы полностью, и уже многие годы идет их переработка с учетом требований систем реального времени. Однако эти изменения всегда тормозились из-за трений между пользователями, которым не нужна поддержка реального времени, и теми, кому она нужна, а поскольку первых большинство, внесение правок встречает препятствия. Из-за этого в библиотеке до сих пор остаются узкие места, однако даже если их устранить, есть более глобальная проблема: универсальность фьютексов, на основе которых сделана вся синхронизация.

Каким же образом универсальность оказывается недостатком? Реализовать барьеры и условные переменные только на фьютексах очень сложно, поэтому они были сделаны на основе блокировок, которые реализовать через фьютексы гораздо проще (в конце концов,

недаром слово "фьютекс" означает "быстрая блокировка"). В итоге все объекты действительно оказываются основаны только на фьютексах, но такая иерархия (в противоположность реализации напрямую через спинлоки ядра) приводит к определенным проблемам в системах жесткого реального времени:

- 1) Внутренняя блокировка, защищающая состояние объекта, должна следовать протоколу защиты от инверсии приоритетов, что в настоящее время не сделано.
- 2) В ряде случаев увеличиваются задержки и накладные расходы, например, если при реализации "напрямую" требуется один системный вызов, то в текущем варианте условных переменных в худшем случае нить сделает по системному вызову в начале и в конце двух критических секций и один вызов для собственно выполнения операции – итого пять вызовов, к тому же, даже если внутренняя блокировка объекта окажется свободна, ее захват и освобождение не бесплатны – это четыре лишние атомарные инструкции.
- 3) Проблема, указанная в предыдущем пункте, усугубляется особенностями исторического развития фьютексов и ядра Linux. Для защиты внутренних объектов ядра всегда использовались спинлоки, однако в системах реального времени это не годится: чтобы получить детерминированные задержки при работе со спинлоками, длина критических секций, которые они защищают, должна быть ограничена, потому что внутри таких секций задача не может быть вытеснена. Разработчики ядра не всегда заботятся об этом, и во многих случаях длина этих секций ничем явно не ограничена. Поэтому в патче для систем реального времени от Инго Молнара очень многие спинлоки в ядре были заменены на блокировки [3], в результате чего барьеры и условные переменные в NPTL реализованы на основе пользовательских блокировок, которые реализованы на основе фьютексов, которые реализованы на основе блокировок ядра, которые реализованы на основе спинлоков ядра. Понятно, что такая длинная иерархия отрицательно влияет и на среднее, и на максимальное время исполнения операций.
- 4) Теряется возможность некоторых оптимизаций (подробней об этом будет сказано, при описании конкретных объектов).

Помимо этого, хотя вызов `sys_futex` и задумывался как универсальное средство, часть функциональности, нужной для конкретных объектов (например, атрибут устойчивости или протокол наследования приоритета для блокировок), реализовать только на уровне пользователя невозможно, и в итоге со временем реализация фьютексов все больше

усложнялась добавлением кода, предназначенного для решения специфических задач, а сам фьютекс все дальше уходил от первоначальной концепции описываемого несколькими небольшими правилами простого объекта синхронизации, и в настоящее время вызов `sys_futex` выглядит гораздо сложнее, чем когда появлялся.

Итого получаем две проблемы: реализация всех объектов через фьютексы и то, что библиотека изначально не ориентировалась на системы реального времени. Наиболее существенные аспекты второй проблемы рассматриваются в следующей главе.

4. О соответствии NPTL требованиям систем реального времени

Инициализация объектов

Каждому объекту синхронизации ставится в соответствие очередь ждущих на нем нитей в ядре. В NPTL память для этой очереди выделяется и освобождается автоматически по мере необходимости, и хотя используется небольшой кэш, он не гарантирует полное отсутствие выделений памяти, которые могут занимать неопределенно долгое время. Такой подход упрощает использование фьютексов: нет нужды их инициализировать и устраняется опасность утечки памяти. Это даже позволяет вообще не вызывать деструкторы для объектов.

Однако из-за неопределенности времени выделения памяти инициализацию в системах жесткого реального времени необходимо делать заранее, то есть сначала создать все нити и объекты, и только когда все подготовлено, начать работу системы. В то же время динамическое выделение и освобождение памяти для очереди противоречит такому подходу и отрицательно влияет на худшее и среднее время исполнения операций.

Еще одной причиной использования статической инициализации очередей является ускорение работы разделяемых между процессами объектов. При исполнении программы необходимо уметь различать - какая очередь в ядре ОС соответствует определенному объекту синхронизации в программе пользователя. Для этого следует сопоставить каждый объект с уникальным числом, которое будет служить индексом в массиве очередей или ключом в хэш-таблице очередей. При статической инициализации, используемой в системах реального времени, возможно вычислить его один раз при инициализации и сохранить внутри объекта в программе пользователя. Но при работе NPTL могут в разные моменты времени использоваться различные очереди для одного и того же объекта, поэтому приходится вычислять ключ каждый раз, когда нить блокируется на объекте, что

увеличивает худшее время исполнения (для вычисления ключа необходимо в ядре закрыть семафор на чтение).

Блокировки

В системах реального времени обычно используются блокировки, следующие протоколу наследования приоритета, согласно которому нить, ожидающая освобождения блокировки, должна поднять приоритет владельца до своего. Благодаря этому, становится невозможным появление в системе нитей с приоритетом большим, чем у владельца, и меньшим, чем у ожидающей нити. Особенностью таких блокировок является необходимость всегда знать, какой именно нити принадлежит в данный момент блокировка, чтобы поднять ее приоритет. А поскольку одним из основных способов ускорения работы блокировок является минимизация числа дорогих (по сравнению с атомарными операциями) системных вызовов, нужно найти способ узнавать идентификатор нити без захода в ядро. Этот способ называется Thread Local Storage, и NPTL использует его для реализации таких блокировок. Собственно же алгоритм наследования используется тот же, что и для ядерных блокировок, – это получается автоматически благодаря тому, что для пользовательских блокировок с наследованием приоритета `sys_futex` служит только дополнительной прослойкой между ними и ядерными блокировками. Но поскольку ядерные блокировки не поддерживают всю функциональность пользовательских (в частности, протокол защиты приоритета и атрибут устойчивости), использование их кода вместо специализированного не позволяет реализовать пользовательские блокировки оптимальным образом.

Помимо протокола наследования приоритета в системах жесткого реального времени используется протокол защиты приоритета. Он дает более предсказуемые времена исполнения, так как при наследовании приоритета могут возникать цепочки из нитей, ожидающих освобождения блокировок, принадлежащих нитям, которые тоже ожидают освобождения каких-нибудь блокировок, и так далее, а такие цепочки быть очень длинными (или даже заикливаться). При защите приоритета с каждой блокировкой связывается максимальный приоритет, который должен быть не меньше, чем максимальный из приоритетов нитей, использующих эту блокировку. При каждом захвате нить временно повышает свой приоритет до заданного значения, а при освобождении – понижает обратно (отсюда видно, что захват и освобождение таких блокировок всегда реализуются системным вызовом, поскольку менять приоритет может только ядро). В NPTL это сделано посредством двух дополнительных системных вызовов, повышающих приоритет при захвате и понижающих при освобождении. Понятно, что это не лучшим

образом сказывается на производительности, и гораздо эффективней поддержать защиту приоритета в ядре.

Еще один момент, требующий внимания, – атрибут устойчивости. Необходимый для его реализации список захваченных нитью блокировок можно помещать или в пространство ядра, или в пространство пользователя. Проблема в том, что делать, когда владелец блокировки умирает. Если у него имеется список захваченных блокировок, то он может просто пройти по нему, освобождая их все, тем самым делая невозможным возникновение "подвисших" очередей, ссылающихся на несуществующего владельца. В NPTL так делать нельзя, потому что список находится в пространстве пользователя [4] и, следовательно, ненадежен, поэтому у каждой очереди заведен счетчик использования, который при захвате соответствующей блокировки увеличивается на 1, а при освобождении – уменьшается (так же этот же счетчик необходим для работы динамического создания и удаления очередей – когда он становится равен 0, очередь можно удалять). Эти два атомарных сложения – не самые быстрые инструкции, примерно по сотне тактов каждая (на x86), поэтому имеет смысл перенести список в ядро, чтобы их не делать. Тогда не получится реализовать атрибут устойчивости для блокировок без защиты от инверсии приоритета, но системы жесткого реального времени только выиграют – для них такие блокировки не критичны.

Барьеры

Барьеры – объекты синхронизации, которые приостанавливают исполнения нитей, вызвавших функцию `pthread_barrier_wait()`, пока их не наберется заранее заданное число.

Как уже говорилось, барьеры в NPTL реализованы на основе пользовательских блокировок, которые реализованы на основе фьютексов, и так далее. Реализация напрямую через ядерные спинлоки проста и в то же время эффективна, поэтому несколько непонятно желание создателей NPTL сделать из фьютексов "ко всякой бочке затычку" и основывать все объекты синхронизации исключительно на них. Единственное исключение – блокировки, следующие протоколу наследования приоритета, которые сделаны прямолинейно, но и то лишь потому, что реализовать наследование на уровне приложения без серьезной помощи со стороны операционной системы невозможно.

Единственная небольшая проблема, возникающая в системах реального времени при реализации напрямую через спинлоки, связана с тем, что нитей в очереди блокировки может быть сколько угодно, а значит, обход всей этой очереди тоже может длиться сколько угодно. Поскольку нить внутри критической секции не может быть вытеснена,

для улучшения задержек приходится ограничивать длину критических секций. Поэтому надо будить ждущие на барьере нити не все сразу, а небольшими группами, причем в промежутке между пробуждением этих групп освобождать спинлок и открывать прерывания. Чтобы в этот момент в очереди не появились новые нити (ведь она уже заполнена), она "отсоединяется" от барьера, который получает новую пустую очередь.

В результате возможна ситуация, когда группа, в которой находится нить с высоким приоритетом, еще не была разбуждена, а будящая нить имеет малый приоритет и была вытеснена, то есть получаем инверсию приоритетов. Чтобы избежать ее, достаточно на время между пробуждением двух групп поднимать приоритет будящей нити до уровня первой нити в очереди.

Остается только заметить, что текущая реализации барьеров подвержена инверсии, для защиты от которой надо включить наследование приоритетов у пользовательской блокировки, на основе которой реализован барьер. Однако в ближайшем будущем такая возможность вряд ли будет включена в NPTL, поскольку она нужна лишь очень небольшому числу пользователей.

Условные переменные

Все, что было сказано про барьеры, в полной мере относится и к условным переменным: очереди у них также не ограничены в размерах, при обработке их частями возникает инверсия приоритетов, и на данный момент внутренняя блокировка не использует наследование. Правда, для них хотя бы есть соответствующий патч, включающий наследование, – он был опубликован Дарреном Хартом (Darren Hart) еще в мае [5].

Особенностью условных переменных является то, что в результате выполнения функции `pthread_cond_broadcast()` только одна нить может захватить блокировку, остальные нет нужды даже будить. Тем не менее, изначальная реализация просто будила все нити, сколько бы их ни было, и в результате небольшой "потасовки" между ними оставалась исполняться только одна, захватившая блокировку. Затем была добавлена операция `futex_queue`, которая будила только одну нить, а все остальные перемещала. Однако и здесь еще оставалось место для оптимизации: если блокировка в момент вызова `pthread_cond_broadcast()` была занята, то не нужно будить никакую нить. Для блокировок, использующих наследование приоритета, она была реализована, а вот для остальных – нет, и причина здесь все в той же универсальности фьютексов.

Не следует думать, что ситуация с захваченной блокировкой маловероятна. Следующий код, в котором захвата нет, вполне законен:

```
pthread_mutex_lock(mx);  
condition = true;  
pthread_mutex_unlock(mx);  
pthread_cond_broadcast(cv);
```

Впрочем, очень часто можно увидеть и такую реализацию:

```
pthread_mutex_lock(mx);  
condition = true;  
pthread_cond_broadcast(cv);  
pthread_mutex_unlock(mx);
```

В ней блокировка принадлежит нити, вызывающей `pthread_cond_broadcast()`, а значит, в очереди будить никакую нить не нужно в 100 % случаев.

Еще одно тонкое место – применение этой оптимизации в случае разделяемых между процессами объектов. Поскольку в NPTL очередь объекта взаимно связана с фьютексом, то для этого надо подать в операцию `queuee` как параметры два фьютекса – условной переменной и блокировки. Но `pthread_cond_broadcast()` в качестве параметра получает только условную переменную, то есть соответствующую блокировку мы должны найти самостоятельно. В случае частных (`private`) объектов это достигается хранением в условной переменной указателя на блокировку, связанную с ней в данный момент. Однако разделяемые объекты могут иметь различные адреса в различных процессах, и в пространстве пользователя нет способа внутри функции `pthread_cond_broadcast()` быстро найти указатель на связанную блокировку. При реализации же напрямую, без фьютексов, это сделать можно.

5. Экспериментальная оценка

На основании сказанного можно перечислить, что именно в NPTL имеет смысл оптимизировать с точки зрения систем жесткого реального времени:

- 1) Использование статической инициализации позволит ускорить разделяемые объекты, а также улучшит максимальное время выполнения.

- 2) Реализация условных переменных и барьеров должна быть в ядре, а не основанной на пользовательских блокировках.
- 3) Использовать отдельный код для пользовательских блокировок.
- 4) Если выполнить п. 3, то возможно оптимизировать производительность разделяемых между процессами условных переменных и блокировок.
- 5) Разместить список захваченных блокировок в ядре (что вместе с п. 3 позволит ускорить блокировки, используя наследование приоритета).

Иными словами, необходимо практически полностью переписать реализацию объектов синхронизации в NPTL, причем как пользовательскую часть, так и ядерную. Это было выполнено автором в библиотеке под названием `elpthread`, что дало возможность сравнить производительность блокировок и условных переменных старой и новой версиях (принципы реализации барьеров аналогичны условным переменным, поэтому и разница в производительности будет такой же).

Измерения проводились на машине с Core Quad Q8400 @ 3,06 ГГц, дистрибутив Ubuntu 10.04, NPTL версии 11.1, ядро – модифицированное 2.6.33. Во всех тестах кроме `cond-perf` была отключена подкачка с помощью `mlockall` и все прерывания кроме 0-го и 2-го были отключены в тех ядрах, на которых производились замеры.

`ptsematest -a -t -p99 -i100 -d25 -i1000000`

Тест `ptsematest` из пакета `rt-tests`. У NPTL мин./ср./макс. результаты составили 1, 2 и 19 мкс соответственно, у `elpthread` – 1, 2 и 16 мкс.

Для замера производительности были запущены на разных ядрах две нити, которые в цикле захватывали и освобождали одну блокировку. Первая нить имела приоритет 98, вторая - приоритет 99 и измеряла суммарное время захвата и освобождения, всего 10000000 замеров. Приоритет первой нити был понижен чтобы исключить влияние так называемого `lock stealing`. У NPTL время составило от 176 до 137224 тактов процессора, у `elpthread` – от 192 до 49816 тактов.

Для оценки эффекта от оптимизации условных переменных использовался следующий тест. На одном ядре были запущены две нити с приоритетом 99, которые в цикле выполняли функции `pthread_cond_broadcast()` и `pthread_cond_wait()`, при этом измерялось время от входа в функцию `broadcast` первой нити до выхода из `wait` второй. Для NPTL

времена мин./ср./макс составили, соответственно, 4672, 4965 и 30904 тактов, для elpthread – 4016, 4255 и 25504 тактов.

Чтобы оценить общую производительность, использовался тест cond-perf, который входит в состав NPTL.

```
$ time ./cond-perf -r1000000  
real 0m30.887s  
user 0m5.383s  
sys 1m19.538s  
$ time ./cond-perf -r1000000 -k  
real 0m20.649s  
user 0m2.467s  
sys 0m19.972s  
$ export LD_PRELOAD=libelpthread.so.1  
$ time ./cond-perf -r1000000  
real 0m16.705s  
user 0m1.847s  
sys 0m15.076s  
$ time ./cond-perf -r1000000 -k  
real 0m16.150s  
user 0m1.887s  
sys 0m14.469s
```

Видно, что в основном улучшились худшие времена. Этого и следовало ожидать, так как в основном на это ориентированы проведенные оптимизации. Однако в случае условных переменных видно заметное улучшение и средних времен, в основном благодаря использованию более специализированного кода в ядре Linux'a.

6. Список литературы:

- 1) <http://people.redhat.com/drepper/nptl-design.pdf>
- 2) <http://lwn.net/Articles/146861>
- 3) <http://lwn.net/Articles/345076>
- 4) www.mjmwired.net/kernel/Documentation/robust-futexes.txt

5) <http://www.cygwin.com/ml/libc-alpha/2010-05/msg00054.html>

7. Автор: Фёдоров Александр Вадимович, закончил МФТИ в 2010-м году, магистр техники и технологий.