УДК 004.451.45

Фёдоров А.В.

Московский физико-технический институт

ЗАО «МЦСТ»

Разработка библиотеки нитей POSIX реального времени

I. Что такое NPTL

NPTL – стандартная библиотека, реализующая нити (threads) POSIX, которая поставляется практически во всех дистрибутивах Linux’а в составе системной библиотеки glibc. Она разрабатывается исключительно для Linux и использует присутствующий только в Linux'е системный вызов clone(). Основное внимание я уделяю реализации объектов синхронизации, поскольку для систем жёсткого реального времени критична именно их производительность.

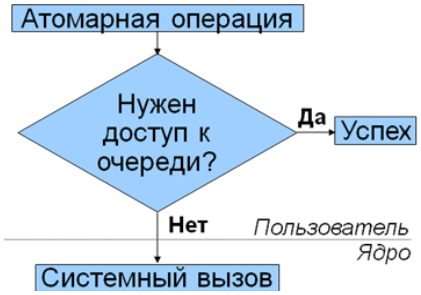
NPTL разработана специально для Linux’а, поэтому у её разработчиков есть замечательная возможность вносить в интерфейс ядра ОС изменения, ориентированные исключительно на её поддержку. Для эффективной реализации всех объектов синхронизации был добавлен системный вызов futex (Fast Userspace MUTex.), реализующий одноимённый объект синхронизации.

Конечно, самым простым решением было бы реализовать синхронизацию полностью в адресном пространстве ядра и сделать системный вызов, который просто «переводил» бы запросы приложения в пространство ядра. Более того, тогда можно было бы использовать уже имеющиеся в ядре реализации блокировок и семафоров. Недостаток такого решения – каждая операция будет использовать системный вызов, а это дорогое удовольствие (потеря нескольких сотен тактов и стирание содержимого кэша процессора).

К счастью, у этой проблемы есть 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 не необходимости и в системном вызове. На практике такая оптимизация реализуется хранением состояния объекта в переменной, находящейся в пользовательском пространстве. Чтобы не возникали состояния гонки, она модифицируется только атомарными инструкциями.

Такая переменная и называется фьютексом. Системный вызов futex либо удаляет нить из очереди и помещает её в очередь готовых к исполнению задач, либо добавляет нить в очередь в ядре, ещё раз проверив состояние объекта, чтобы избежать состояния гонки. На его основе в NPTL реализованы блокировки и семафоры, а на их основе – все остальные объекты.

На картинке схематично изображена эта оптимизации (под очередью понимается очередь нитей, ждущих на данном объекте).



Именно из-за поддержки такой вот оптимизации в названии фьютексов есть слово Fast.

II. Требования систем реального времени

Различают два вида систем реального времени – мягкого реального времени и жёсткого. В обоих случаях операции должны гарантировать своё завершение за определённое время, только в системах жёсткого РВ, если операция не успевает выполниться в срок, вся система может стать неработоспособной. Например, для систем ПВО, если при расчёте траектории перехвата какая-то операция слишком задержится, ракета не попадёт в цель.

Помимо гарантированных времён исполнения, операции должны иметь малое среднее время исполнения, а также скорость реакции системы на внешние события должна быть минимальной.

Итого получаем три параметра, которые надо минимизировать:

1) Максимальное время исполнения операций.
2) Среднее время исполнения операции.
3) Задержка между внешним событием и реакцией на него.

Обеспечиваются эти требования следующими способами:

1) Оптимизация с использованием атомарных инструкций, улучшает среднее время исполнения.
2) Ограничение длительности отключения прерываний улучшает задержки.
3) Использование не универсального системного вызова futex, а специализированного улучшает всё.

III. Реализация объектов синхронизации в NPTL

На основе системного вызова futex в NPTL реализуются блокировки и семафоры, и уже на их основе – все остальные объекты. Реализация же объектов не напрямую, а на основе блокировок и фьютексов, ведёт к серьёзным проблемам в системах реального времени (РВ):

1) Внутренняя блокировка, защищающая состояние объекта, должна следовать протоколу защиты от инверсии приоритетов, что в данный момент в NPTL не реализовано даже как опция, так как библиотека предназначена не только для систем реального времени (т.е. барьеры и условные переменные подвержены инверсии приоритетов).
2) Значительно увеличиваются накладные расходы и задержки: если при реализации «напрямую» потребуется один системный вызов, то при реализации через блокировки в худшем случается нить сделает по системному вызову при захвате блокировки, собственно операции и освобождении блокировки – целых три системных вызова.
3) Следующий недостаток вытекает из исторического развития фьютексов и ядра Linux'a. Для защиты своих внутренних объектов ядро первоначально использовало спинлоки, однако для систем реального времени это не годится: для получения детерминированных задержек при работе со спинлоками длина критических секций, которые они защищают, должна быть ограничена, а разработчики ядра редко думают о потребностях систем реального времени, и в худших случаях (например, при трассировке списков) длина этих секций ничем явно не ограничена. Поэтому в патче для систем реального времени от Инго Молнара очень многие спинлоки были заменены на блокировки, вот и получается, что барьеры реализованы на основе пользовательских блокировок, которые реализованы на основе фьютексов, которые реализованы на основе блокировок ядра, которые реализованы на основе спинлоков. Понятно, что это отрицательно влияет и на среднее, и на максимальное время исполнения операций.

Реализация блокировок, следующих протоколу наследования приоритета, в NPTL лучше

всего соответствует изложенным принципам: они сделаны на основе ядерных блокировок, т.е. хотя они и работают через системный вызов futex, мало связаны с собственно фьютексами. Однако использование одного и того же кода для ядерных и пользовательских блокировок, хотя и уменьшает объём ядра на десяток килобайт, отрицательно сказывается на производительности, ведь в ядре никак не используются протокол защиты приоритета, атрибут устойчивости блокировок и условные переменные. В результате эта функциональность реализуется “поверх” ядерных блокировок, и некоторые микрооптимизации становятся недоступны.

Ещё одна проблема — инициализация объектов. Она связана с выделением памяти под очередь для нитей, ждущих на данном объекте, которое может занимать неопределённо долгое время. В NPTL выделение происходит динамически, по мере необходимости, и хотя используется небольшой кэш, он не может гарантировать полное отсутствие задержек, связанных с выделением памяти. Такой подход упрощает использование фьютексов – не надо их инициализировать и устраняется опасность утечки памяти, но плохо влияет на худшее и среднее время исполнения операций.

Помимо этого, подход со статической инициализацией позволяет улучшить среднее и худшее время работы разделяемых между процессами (process shared) объектов синхронизации за счёт устранения необходимости динамически различать различные объекты, чтобы «на лету» сопоставить каждому свою очередь в ядре ОС.

Именно поэтому в приложениях, написанных для систем жёсткого реального времени, вся инициализация делается заранее, т.е. сначала создаются все нити и инициализируются все объекты синхронизации, и только когда всё подготовлено, система начинает работу.

Поэтому для систем реального времени необходимо сделать инициализацию и уничтожение объектов не автоматическими.

Приведу результат сравнения NPTL с написанной мною реализацией, в которой память под объекты выделяется заранее и реализация пользовательских блокировок отличается от реализации ядерных блокировок тем, что она поддерживает различные аспекты стандарта уже на уровне ядра операционной системы. В тесте были созданы две нити, каждая привязана к своему ядру процессора, которые в цикле захватывали и освобождали блокировку, использующую протокол наследования приоритета. Замерялось время исполнения критической секции, ниже приведены лучший и худший результаты:

	NTPL	Новая реализация
Мин. время, такты	172	192
Макс. время, такты	172288	74056

Литература

1. Архив Linux kernel mailing list.

2. Архив libc-alpha mailing list.