Министерство образования и науки Российской Федерации

Национальный исследовательский университет “МИЭТ”

Кафедра “Информатика и программное обеспечение вычислительных систем”

Сарычев Валентин Сергеевич

**Исследование и разработка методов повышения производительности высоконагруженных многопоточных приложений в SMP системах**

ДИССЕРТАЦИЯ

на соискание степени магистра техники и технологии по направлению   
552800 “Информатика и вычислительная техника”

(магистерская программа 552822 “Распределенные автоматизированные системы”)

Научный руководитель

к.т.н., профессор

кафедры ИПОВС

Илюшечкина Л.В.

Москва 2013

Введение 5

Глава 1. Обзор существующих моделей многопоточности и методов параллельного программирования 9

1.1. Многозадачность и ее типы 9

1.1.1. История многозадачных операционных систем 11

1.1.2. Невытесняющая многозадачность 13

1.1.3. Совместная или кооперативная многозадачность 13

1.1.4. Вытесняющая или приоритетная многозадачность 14

1.1.5. Голодание 16

1.1.6. Гонка 17

1.1.7. Инверсия приоритета 17

1.2. Потоки выполнения 18

1.3. Модели многопоточности 19

1.3.1. 1:1 (потоки выполнения на уровне ядра) 19

1.3.2. N:1 (потоки выполнения уровня пользователя) 19

1.3.3. M:N (смешанная потоковость) 20

1.4. Методы параллельного программирования 20

1.4.1. STM (Software Transactional Memory) 20

1.4.2. Потоки данных, «будущие результаты» и «обещания» 22

1.4.3. Асинхронный обмен сообщениями 24

1.5. Проблемы высоконагруженных многопоточных процессов 25

1.6. Способы решения и их обоснование 26

1.7. Постановка задачи диссертационных исследований 27

Выводы 28

Глава 2. Разработка гибридной модели потоков выполнения для повышения производительности высоконагруженных процессов 29

2.1. Моделирование и формализация работы процесса 30

2.2. Расчет погрешности измерений 39

2.3. Выбор средств анализа результатов 40

2.3.1. Microsoft Concurrency Visualizer 40

2.3.2. Intel VTune Amplifier 41

2.3.3. AMD CodeAnalyst Performance Analyzer 42

2.3.4. Apple XCode Instruments 43

2.3.5. GNU Valgrind 43

2.3.6. Сравнение средств профилирования 44

2.4. Разработка системы тестов для моделирования работы потоков 45

2.4.1. Создание/удаление потоков 47

2.4.2. Переключение контекста 47

2.4.3. Короткие вычисления 48

2.4.4. Длительные вычисления 49

2.4.5. Общая память 49

2.4.6. Примитивы синхронизации 50

2.4.7. Операции чтения/записи 50

2.5. Разработка гибридной модели потоков 51

2.6. Анализ результатов моделирования 53

2.6.1. Создание/удаление потоков 53

2.6.2. Переключение контекста 56

2.6.3. Короткие вычисления 58

2.6.4. Длительные вычисления 59

2.6.5. Общая память 60

2.6.6. Примитивы синхронизации 61

2.6.7. Операции чтения/записи 63

2.7. Анализ свойств гибридной модели 64

2.8. Парадокс производительности 65

Глава 3. Верификация и применение результатов разработки 68

3.1. Реализация гибридной модели 68

3.1.1. Принципы разработки 68

3.1.2. POSIX Threads 72

3.1.3. State Threads 73

3.2. Анализ работы полученной реализации 75

Заключение 77

Список литературы 78

Список иллюстраций 79

# Введение

#### Актуальность исследования

С каждым годом параллельно с развитием качества Интернет-соединения индустрия программного обеспечения делает все больший уклон в сторону развития высоконагруженных систем, способных обслуживать тысячи пользователей в секунду, обрабатывая терабайты данных. К таким системам можно смело отнести продукты Google, Youtube, Yandex, Twitter, NYSE и большое множество серверов электронной торговли.

Методология разработки высоконагруженного ПО опирается на принципы масштабируемости, как вертикальной, так и горизонтальной.

Очевидно, чем больше аппаратных ресурсов задействует ПО, тем острее встает проблема вертикального масштабирования. В большинстве случаев она решается использованием современных приемов программирования, таких как active object, balking pattern, thread pool pattern, reactor patter.

Однако на сегодняшний день существуют программные комплексы, разрабатываемые более десятка лет (NYSE, ROX, Plaza), написанные с использованием традиционного подхода, когда одного клиента обслуживает один поток выполнения. Практика показывает, что с точки зрения производительности эффективность такого подхода сильно снижается при большом количестве клиентов, а техническая сложность этих проектов не позволяет провести их модернизацию в виду ее дороговизны.

Также существует спектр задач, для которых разработку ПО с экономической точки зрения более приемлемо вести в рамках традиционного подхода параллельного программирования.

В эпоху однопроцессорных ЭВМ эти проблемы прекрасно решались бы заменой вытесняющих потоков выполнения на кооперативные потоки. Но на тот момент не существовало такого количества высоконагруженных систем, как в наши дни. А с появлением многоядерных процессоров неэффективность применения кооперативных потоков стала пропорциональна количеству процессоров SMP системы.

#### Объект исследования

Объектом исследования диссертационной работы являются современные SMP системы. К ним относится подавляющее большинство аппаратно-программных комплексов от ПК до серверных ЭВМ.

#### Предмет исследования

Предметом исследования диссертационной работы являются схемы реализации потоков выполнения в современных ОС c поддержкой SMP. К ним относятся все популярные многозадачные ОС: Windows 7, Debian Linux, Mac OS, FreeBSD, Windows Server 2008 и прочие.

#### Цели и задачи исследования

Цель работы заключается в исследовании схем реализации многопоточности в современных многозадачных ОС и языках программирования.

В соответствии с целью и предметом исследования в диссертации необходимо решить следующие задачи:

* исследовать модели многопоточности в современных операционных системах и языках программирования;
* выявить и проанализировать узкие места в работе высоконагруженных процессов;
* разработать гибридную модель потоков выполнения с учетом проблем существующих моделей;
* исследовать полученную модель, проведя математическое моделирование работы высоконагруженного процесса;
* реализовать модель на языке программирования;
* провести научный эксперимент;
* сопоставить и проанализировать результаты моделирования и эксперимента.

#### Методы исследования

Для решения задач диссертационной работы использовались методы имитационного моделирования, теоретического анализа, методы математической статистики, опрос и изучение литературы, а также эксперимент.

#### Положения, выносимые на защиту

На защиту выносятся следующие положения:

* результаты исследования современных моделей потоков выполнения;
* гибридная модель многопоточности;
* результаты исследования гибридной модели.

#### Научная новизна исследования

В диссертации были получены следующие результаты:

* разработана гибридная модель многопоточности для решения проблем производительности рассматриваемого спектра высоконагруженного ПО;
* теоретическое обоснование высокой эффективности гибридных потоков выполнения в SMP системах в сравнении с потоками выполнения, предоставляемыми ОС;
* произведена оценка эффективности применения гибридных потоков выполнения;
* разработана библиотека гибридных потоков, отвечающая требованиям POSIX;
* доказана экономическая выгода использования разработанной библиотеки гибридных потоков в Unix-подобных ОС.

#### Обоснованность и достоверность результатов

Обоснованность и достоверность полученных в диссертации результатов подтверждается:

* адекватностью разработанной имитационной модели;
* использованием реальных исходных данных при расчетах;
* верификацией имитационной модели.

#### Практическая значимость результатов

Практическая значимость результатов, полученных в диссертационной работе, заключается в возможности применения разработанной гибридной модели многопоточности для оптимизации работы высоконагруженных процессов при разработке ПО, а также в экономической выгоде применения разработанной библиотеки гибридных потоков выполнения при модернизации существующего ПО.

#### Структура и объем работы

Диссертация состоит из введения, трех глав, заключения, списка литературы и иллюстраций. Работа содержит 85 страниц основного текста, 34 страницы с рисунками и таблицами и список литературы из 26 наименований.

# Обзор существующих моделей многопоточности и методов параллельного программирования

## Многозадачность и ее типы

*Многозадачность (англ. multitasking)* — свойство операционной системы или среды программирования обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких процессов. Истинная многозадачность операционной системы возможна только в распределенных вычислительных системах.

Существует 2 типа многозадачности:

* *Процессная многозадачность (основанная на процессах — одновременно выполняющихся программах).* Здесь программа — наименьший элемент кода, которым может управлять планировщик операционной системы. Более известна большинству пользователей (работа в текстовом редакторе и прослушивание музыки).
* *Поточная многозадачность (основанная на потоках).* Наименьший элемент управляемого кода — поток (одна программа может выполнять 2 и более задачи одновременно).

*Многопоточность* — специализированная форма многозадачности.

Примитивные многозадачные среды обеспечивают чистое «разделение ресурсов», когда за каждой задачей закрепляется определённый участок памяти, и задача активизируется в строго определённые интервалы времени.

Более развитые многозадачные системы проводят распределение ресурсов динамически, когда задача стартует в памяти или покидает память в зависимости от её приоритета и от стратегии системы. Такая многозадачная среда обладает следующими особенностями:

* Каждая задача имеет свой приоритет, в соответствии с которым получает процессорное время и память
* Система организует очереди задач так, чтобы все задачи получили ресурсы, в зависимости от приоритетов и стратегии системы
* Система организует обработку прерываний, по которым задачи могут активироваться, деактивироваться и удаляться
* По окончании положенного кванта времени ядро временно переводит задачу из состояния выполнения в состояние готовности, отдавая ресурсы другим задачам. При нехватке памяти страницы невыполняющихся задач могут быть вытеснены на диск (своппинг), а потом через определённое системой время, восстанавливаться в памяти
* Система обеспечивает защиту адресного пространства задачи от несанкционированного вмешательства других задач
* Система обеспечивает защиту адресного пространства своего ядра от несанкционированного вмешательства задач
* Система распознает сбои и зависания отдельных задач и прекращает их
* Система решает конфликты доступа к ресурсам и устройствам, не допуская тупиковых ситуаций общего зависания от ожидания заблокированных ресурсов
* Система гарантирует каждой задаче, что рано или поздно она будет активирована
* Система обрабатывает запросы реального времени
* Система обеспечивает коммуникацию между процессами

Основной трудностью реализации многозадачной среды является её надёжность, выраженная в защите памяти, обработке сбоев и прерываний, предохранении от зависаний и тупиковых ситуаций.

Кроме надёжности, многозадачная среда должна быть эффективной. Затраты ресурсов на её поддержание не должны: мешать процессам проходить, замедлять их работу, резко ограничивать память.

* + 1. История многозадачных операционных систем

Поначалу реализация многозадачных операционных систем представляла собой серьёзную техническую трудность, отчего внедрение многозадачных систем затягивалось, а пользователи долгое время после внедрения предпочитали однозадачные.

В дальнейшем, после появления нескольких удачных решений, многозадачные среды стали совершенствоваться, и в настоящее время употребляются повсеместно.

Впервые многозадачность операционной системы была реализована в ходе разработки операционной системы Multics (1964 год). Одной из первых многозадачных систем была OS/360 (1966[2]), используемая для компьютеров фирмы IBM и их советских аналогов ЕС ЭВМ. Разработки системы были сильно затянуты, и на начальное время фирма IBM выдвинула однозадачный DOS, чтобы удовлетворить заказчиков до полной сдачи OS/360 в эксплуатацию. Система подвергалась критике по причине малой надёжности и трудности эксплуатации.

В 1969 году на основе Multics была разработана система UNIX с достаточно аккуратным алгоритмическим решением проблемы многозадачности. В настоящее время на базе UNIX созданы десятки операционных систем.

На компьютерах PDP-11 и их советских аналогах СМ-4 использовалась многозадачная система RSX-11 (советский аналог — ОСРВ СМ ЭВМ), и система распределения времени TSX-PLUS, обеспечивающая ограниченные возможности многозадачности и многопользовательский режим разделения времени, эмулируя для каждого пользователя однозадачную RT-11 (советский аналог — РАФОС). Последнее решение было весьма популярно из-за низкой эффективности и надёжности полноценной многозадачной системы.

Аккуратным решением оказалась операционная система VMS, разработанная первоначально для компьютеров VAX (советский аналог — СМ-1700) как развитие RSX-11.

Первый в мире мультимедийный персональный компьютер Amiga 1000 (1984 год) изначально проектировался с расчётом на полную аппаратную поддержку вытесняющей многозадачности реального времени в ОС AmigaOS. В данном случае разработка аппаратной и программной части велась параллельно, это привело к тому, что по показателю квантования планировщика многозадачности (1/50 секунды на переключение контекста) AmigaOS долгое время оставалась непревзойдённой на персональных компьютерах.

Многозадачность обеспечивала также фирма Microsoft в операционных системах Windows. При этом Microsoft выбрала две линии разработок — на базе приобретённой ею Windows 0.9[источник не указан 1306 дней], которая после долгой доработки системы, изначально обладавшей кооперативной многозадачностью, аналогичной Mac OS, вылилась в линейку Windows 3.x, и на основе идей, заложенных в VMS, которые привели к созданию операционных систем Windows NT. Использование опыта VMS обеспечило системам существенно более высокую производительность и надёжность. По времени переключения контекста многозадачности (квантование) только эти операционные системы могут быть сравнимы с AmigaOS и UNIX (а также его потомками, такими, как ядро Linux).

Интересно, что многозадачность может быть реализована не только в операционной, но и языковой среде. Например, спецификации языков программирования Modula-2 и Ada требуют поддержки многозадачности вне привязки к какой-либо операционной системе. В результате, популярная в первой половине 1990-х годов реализация языка программирования TopSpeed Модула-2 от JPI/Clarion позволяла организовывать различные типы многозадачности (кооперативную и вытесняющую — см. ниже) для потоков одной программы в рамках такой принципиально однозадачной операционной системы, как MS-DOS. Это осуществлялось путём включения в модуль программы компактного планировщика задач, содержащего обработчик таймерных прерываний. Языки программирования, обладающие таким свойством, иногда называют языками реального времени.

* + 1. Невытесняющая многозадачность

Тип многозадачности, при котором операционная система одновременно загружает в память два или более приложений, но процессорное время предоставляется только основному приложению. Для выполнения фонового приложения оно должно быть активизировано. Подобная многозадачность может быть реализована не только в операционной системе, но и с помощью программ-переключателей задач. В этой категории известна программа DESQview, работавшая под DOS и выпущенная первый раз в 1985 году.

* + 1. Совместная или кооперативная многозадачность

Сюда перенаправляется запрос «Кооперативная многозадачность». На эту тему нужна отдельная статья.

Тип многозадачности, при котором следующая задача выполняется только после того, как текущая задача явно объявит себя готовой отдать процессорное время другим задачам. Как частный случай, такое объявление подразумевается при попытке захвата уже занятого объекта mutex (ядро Linux), а также при ожидании поступления следующего сообщения от подсистемы пользовательского интерфейса (Windows версий до 3.x включительно, а также 16-битные приложения в Windows 9x).

Кооперативную многозадачность можно назвать многозадачностью «второй ступени» поскольку она использует более передовые методы, чем простое переключение задач, реализованное многими известными программами (например, DOS Shell из MS-DOS 5.0 при простом переключении активная программа получает все процессорное время, а фоновые приложения полностью замораживаются. При кооперативной многозадачности приложение может захватить фактически столько процессорного времени, сколько оно считает нужным. Все приложения делят процессорное время, периодически передавая управление следующей задаче.

Преимущества кооперативной многозадачности: отсутствие необходимости защищать все разделяемые структуры данных объектами типа критических секций и mutex’ов, что упрощает программирование, особенно перенос кода из однозадачных сред в многозадачные.

Недостатки: неспособность всех приложений работать в случае ошибки в одном из них, приводящей к отсутствию вызова операции «отдать процессорное время». Крайне затрудненная возможность реализации многозадачной архитектуры ввода-вывода в ядре ОС, позволяющей процессору исполнять одну задачу в то время, как другая задача инициировала операцию ввода-вывода и ждет её завершения.

Реализована в пользовательском режиме ОС Windows версий до 3.х включительно, Mac OS версий до Mac OS X, а также внутри ядер многих UNIX-подобных ОС, таких, как FreeBSD, а в течение долгого времени — и Linux.

* + 1. Вытесняющая или приоритетная многозадачность

Вид многозадачности, в котором операционная система сама передает управление от одной выполняемой программы другой в случае завершения операций ввода-вывода, возникновения событий в аппаратуре компьютера, истечения таймеров и квантов времени, или же поступлений тех или иных сигналов от одной программы к другой. В этом виде многозадачности процессор может быть переключен с исполнения одной программы на исполнение другой без всякого пожелания первой программы и буквально между любыми двумя инструкциями в её коде. Распределение процессорного времени осуществляется планировщиком процессов. К тому же каждой задаче может быть назначен пользователем или самой операционной системой определенный приоритет, что обеспечивает гибкое управление распределением процессорного времени между задачами (например, можно снизить приоритет ресурсоёмкой программе, снизив тем самым скорость её работы, но повысив производительность фоновых процессов). Этот вид многозадачности обеспечивает более быстрый отклик на действия пользователя.

Преимущества: возможность полной реализации многозадачного ввода-вывода в ядре ОС, когда ожидание завершения ввода-вывода одной программой позволяет процессору тем временем исполнять другую программу. Сильное повышение надежности системы в целом, в сочетании с использованием защиты памяти — идеал в виде «ни одна программа пользовательского режима не может нарушить работу ОС в целом» становится достижимым хотя бы теоретически, вне вытесняющей многозадачности он не достижим даже в теории. Возможность полного использования многопроцессорных и многоядерных систем.

Недостатки: необходимость особой дисциплины при написании кода, особые требования к его реентрантности, к защите всех разделяемых и глобальных данных объектами типа критических секций и mutex’ов.

Реализована в таких ОС, как:

* VMS
* Linux
* в пользовательском режиме (а часто и в режиме ядра) всех UNIX-подобных ОС, включая версии Mac OS X, iOS; Symbian OS
* в режиме ядра ОС Windows 3.x — только при исполнении на процессоре 386 или старше, «задачами» являются только все Windows-приложения вместе взятые и каждая отдельная виртуальная машина ДОС, между приложениями Windows вытесняющая многозадачность не использовалась
* Windows 95/98/ME — без полноценной защиты памяти, что служило причиной крайне низкой, на одном уровне с MS-DOS, Windows 3.x и Mac OS версий до X — надежности этих ОС
* Windows NT/2000/XP/Vista/7 и в режиме ядра, и в пользовательском режиме.
* AmigaOS — все версии, до версии 4.0 без полноценной защиты памяти, что на практике для системных программ почти не сказывалось на надёжности из-за высокой стандартизированности, прозрачных API и SDK. Программы ориентированные на «железо» Амиги, наоборот не отличались надёжностью.
  + 1. Голодание

Задержка времени от пробуждения потока до его вызова на процессор, в течение которой он находится в списке потоков, готовых к исполнению. Возникает по причине присутствия потоков с большими или равными приоритетами, которые исполняются все это время.

Негативный эффект заключается в том, что возникает задержка времени от пробуждения потока до исполнения им следующей важной операции, что задерживает исполнение этой операции, а следом за ней и работу многих других компонентов.

Голодание создаёт узкое место в системе и не дает выжать из неё максимальную производительность, ограничиваемую только аппаратно обусловленными узкими местами.

Любое голодание вне 100 % загрузки процессора может быть устранено повышением приоритета голодающей нити, возможно — временным.

Как правило, для предотвращения голодания ОС автоматически вызывает на исполнение готовые к нему низкоприоритетные потоки даже при наличии высокоприоритетных, при условии, что поток не исполнялся в течение долгого времени (~10 секунд). Визуально эта картина хорошо знакома большинству пользователей Windows — если в одной из программ поток зациклился до бесконечности, то переднее окно работает нормально несмотря на это — потоку, связанному с передним окном, Windows повышает приоритет. Остальные же окна перерисовываются с большими задержками, по порции в секунду, ибо их отрисовка в данной ситуации работает только за счет механизма предотвращения голодания (иначе бы голодала вечно).

* + 1. Гонка

Недетерминированный порядок исполнения двух путей кода, работающих с одними и теми же данными и исполняемыми в двух различных нитях. Приводит к зависимости порядка и правильности исполнения от случайных факторов.

Устраняется добавлением необходимых блокировок и примитивов синхронизации. Обычно является легко устраняемым дефектом (забытая блокировка).

* + 1. Инверсия приоритета

Поток L имеет низкий приоритет, поток M — средний, поток H — высокий. Поток L захватывает mutex, и, выполняясь с удержанием mutex’а, преемптивно прерывается потоком M, который пробудился по какой-то причине, и имеет более высокий приоритет. Поток H пытается захватить mutex.

В полученной ситуации поток H ожидает завершения текущей работы потоком M, ибо, пока поток M исполняется, низкоприоритетный поток L не получает управления и не может освободить mutex.

Устраняется повышением приоритета всех нитей, захватывающих данный mutex, до одного и того же высокого значения на период удержания mutexa. Некоторые реализации mutex’ов делают это автоматически.

## Потоки выполнения

*Поток выполнения (англ. thread — нить)* — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени). В качестве аналогии потоки выполнения процесса можно уподобить нескольким вместе работающим поварам. Все они готовят одно блюдо, читают одну и ту же кулинарную книгу с одним и тем же рецептом и следуют его указаниям, причём не обязательно все они читают на одной и той же странице.

На одном процессоре многопоточность обычно происходит путём временного мультиплексирования (как и в случае многозадачности): процессор переключается между разными потоками выполнения. Это переключение контекста обычно происходит достаточно часто, чтобы пользователь воспринимал выполнение потоков или задач как одновременное. В многопроцессорных и многоядерных системах потоки или задачи могут реально выполняться одновременно, при этом каждый процессор или ядро обрабатывает отдельный поток или задачу.

Многие современные операционные системы поддерживают как временные нарезки от планировщика процессов, так и многопроцессорные потоки выполнения. Ядро операционной системы позволяет программистам управлять потоками выполнения через интерфейс системных вызовов. Некоторые реализации ядра называют потоком ядра, другие же — *лёгким процессом (англ. light-weight process, LWP),* представляющим собой особый тип потока выполнения ядра, который совместно использует одни и те же состояния и данные.

Программы могут иметь пользовательское пространство потоков выполнения при создании потоков с помощью таймеров, сигналов или другими методами, позволяющими прервать выполнение и создать временную нарезку для конкретной ситуации (Ad hoc).

## Модели многопоточности

* + 1. 1:1 (потоки выполнения на уровне ядра)

Потоки выполнения, созданные пользователем в модели 1-1, соответствуют диспетчируемым сущностям ядра. Это простейший возможный вариант реализации потоковости. В Windows API этот подход использовался с самого начала. В Linux обычная библиотека C реализует этот подход (через библиотеку потоков POSIX, а в более старших версиях через LinuxThreads). Такой же подход используется ОС Solaris, NetBSD и FreeBSD.

* + 1. N:1 (потоки выполнения уровня пользователя)

В модели N:1 предполагается, что все потоки выполнения уровня пользователя отображаются на единую планируемую сущность уровня ядра, и ядро ничего не знает о составе прикладных потоков выполнения. При таком подходе переключение контекста может быть сделано очень быстро, и, кроме того, он может быть реализован даже на простых ядрах, которые не поддерживают многопоточность. Однако, одним из главных недостатков его является то, что в нём нельзя извлечь никакой выгоды из аппаратного ускорения на многопоточных процессорах или многопроцессорных компьютерах, потому что только один поток выполнения может быть запланирован на любой момент времени. Эта модель используется в GNU Portable Threads.

* + 1. M:N (смешанная потоковость)

В модели M:N некоторое число N прикладных потоков выполнения отображаются на некоторое число M сущностей ядра или «виртуальных процессоров». Модель является компромиссной между моделью уровня ядра («1:1») и моделью уровня пользователя («N:1»). Вообще говоря, «M:N» потоковость системы являются более сложной для реализации, чем ядро или пользовательские потоки выполнения, поскольку изменение кода как для ядра, так и для пользовательского пространства не требуется. В M:N реализации библиотека потоков отвечает за планирование пользовательских потоков выполнения на имеющихся планируемых сущностях. При этом переключение контекста потоков делается очень быстро, поскольку модель позволяет избежать системных вызовов. Тем не менее, увеличивается сложность и вероятность инверсии приоритетов, а также неоптимальность планирования без обширной (и дорогой) координации между пользовательским планировщиком и планировщиком ядра.

## Методы параллельного программирования

* + 1. STM (Software Transactional Memory)

Первый нетрадиционный способ обеспечения обмена состоянием между потоками — *программная транзакционная память, сокращённо STM (Software Transactional Memory)*. Один из самых популярных примеров такой параллельности — компилятор GHC для языка Haskell. Википедия содержит хорошо подобранные факты и ссылки по многим вопросам компьютерных наук, потому мы тоже будем опираться на факты, изложенные в ней.

*STM* — это механизм контроля аналогичный транзакциям баз данных для контроля доступа к общей памяти при параллельных вычислениях. Он работает, как альтернатива синхронизациям на основе замков (lock), и часто сделан без них в «беззамочной форме» (lock free). В этом контексте, транзакция — это фрагмент кода, выполняющий операции чтения и записи из общей памяти, и этот ряд операций происходит логически как бы в один момент времени, другой код не может вклиниться и выполнить доступ к общей памяти между ними. По завершении транзакции следующий, конкурирующий, фрагмент кода проверяет, подходят ли данные в памяти для записи результатов его работы? Если данные в памяти не соответствуют его ожиданиям, результат отбрасывается и операция переделывается заново.

(Пример с бейсбольным мячом — в оригинальной статье автор рассылает бейсбольный мяч членам любимой команды, чтобы они оставили подписи на нём. Мы же используем более близкий нашей молодёжи пример с «Обходным листом» университета).

Пример с Обходным листом. Допустим, вы студент, и у вас на руках имеется Обходной Лист, который надо пронести по всем факультетам, библиотеке, ректорату и проч. и собрать подписи всех ответственных лиц. Допустим, что это очень особенный лист, с которого я могу снимать копии и передавать кому следует на подпись. Я не даю им копию навсегда, я просто передаю копию на подпись и забираю обратно. Таким образом 10 деканатов, библиотека и ректорат получают копии моего чистого обходного листа. Внезапно, деканат математического факультета первым подписал мой Обходной Лист и вернул мне их копию. Я заменяю чистый лист в своей руке на эту подписанную копию. Через секунду мне присылают мой лист с подписью физического факультета. Я сравниваю его с листом в моей руке и замечаю, что они различаются. Ох ведь, так не годится! Я выбрасываю пришедший лист с подписью физиков, и отсылаю им копию моего листа с подписью математиков. Мне ведь нужны все подписи на одном листе, а не раздельно! Эта операция повторяется много раз, по мере возвращения подписанных копий моего листа до тех пор, отбрасываем несовпадающие листы и отправляем повторно мою копию до тех пор, пока все подписи не собраны на одном листе. По сути я контролирую тот факт, что у меня на руках всегда свежайшая копия моего Обходного Листа.

STM имеет несколько преимуществ. Первое и самое главное преимущество не сразу бросается в глаза при просмотре примера выше. STM — это оптимистическая модель. Каждый поток делает своё дело не задумываясь о том, работает ли другой поток с теми же данными или нет. В конце манипуляции, если всё в порядке и ничего на принимающей стороне не поменялось, результат отправляется на принимающую сторону. Если возникает конфликт версий или данных, то результат отбрасывается и операция проводится повторно, уже с новыми данными.

Хорошее свойство этой модели — никто не ждёт никаких ресурсов. Потоки могут писать в разные поля структуры не заботясь о «замках». Плохое свойство — работу иногда придётся переделывать. Также имеются некоторые, ощутимые накладные расходы на систему транзакций, которые влияют на скорость выполнения. В некоторых ситуациях может возникнуть перерасход памяти: если N процессов работают над M байтами памяти — понадобится N\*M байт для хранения N копий этих данных. В целом, для непрограммиста и для новичков этот подход к параллельности намного более понятен и прост, чем традиционный способ с общей памятью и замками (shared memory & locks) и может быть неплохим способом погрузиться в мир разработки параллельных программ.

* + 1. Потоки данных, «будущие результаты» и «обещания»

Ещё один подход к параллельному программированию это *«Обещания результатов» (futures, promises).* Примеры такого подхода могут быть найдены в Mozart-Oz, в Java, в С++ (библиотека ASIO, Boost.Asio), Python (библиотека Twisted). Снова, Википедия даёт исчерпывающее описание по данной теме, и мы построим повествование на этом описании.

В компьютерных науках, обещания результатов (futures and promises) — это родственные конструкции, используемые для синхронизации в некоторых параллельных языках программирования. Оба они ссылаются на объект, который является посредником в предоставлении результата, который не является немедленно доступным, потому что его вычисление ещё не завершилось.

Вернёмся к нашему Обходному Листу. Мне снова надо подписать мой Обходной Лист, на этот раз на факультете Информационных Технологий. Я просто иду в их деканат и передаю им мой Обходной Лист, но поскольку в данный момент они немного заняты (пьют кофе, к примеру), они дают мне *Расписку (в данном случае future),* в которой мне обещают вернуть мой подписанный Обходной Лист в аккурат тот момент, когда они освободятся. В шоке я сижу под деканатом и читаю эту Расписку, пока не выходит секретарь деканата и не вручает мне мой Обходной Лист с подписью деканата. Расписка мне больше не нужна, её можно выбросить.

*Обещания (promise)* очень похожи на только что описанную *Расписку деканата (future).* К примеру, я прихожу в ректорат, где собрались все деканы, кроме декана факультета ИТ, чья подпись мне и нужна. У меня забирают мой лист и мне вручают Письменное Обещание, что как только он появится, ему передадут мой лист на подпись. В шоке я сижу под ректоратом и смотрю на эту записку, пока случайный декан мне не выносит мой Обходной Лист, подписанный деканом факультета ИТ. Записка более не нужна и её я выбрасываю.

*Будущий результат (future)* это некий контракт, что определённый поток в какой-то момент времени обязательно возвратит мне результат, согласно данному контракту. В отличие от этого Обещание (promise) это обещание, что в какой-то момент времени, какой-то поток, неважно какой, исполнит мою задачу и вернёт мне обещанный результат. По сути это стиль программирования основанный на потоке данных и концептуально достаточно простой. Этот принцип делает обмен данными в параллельных системах простым и понятным, и служит хорошей основой для построения более сложных структур вроде каналов данных. Однако, хотя Будущие результаты и Обещания позволяют обойти самые известные проблемы параллельного программирования, это всё ещё общая память со всеми скрытыми проблемами.

По сути *Будущие результаты и Обещания* — это общее состояние разных потоков. Они хорошо обходят с самые крупные проблемы, но не полностью решают их. Следующий механизм подходит к проблеме с другой стороны. Что позволяет ему ловко решить все известные проблемы общей памяти, в основном ценой дополнительных расходов памяти и времени на копирование данных. Но не будем забегать вперёд...

* + 1. Асинхронный обмен сообщениями

По этой модели параллельности работает язык Erlang, но при желании несложно настроить подобную модель в любом другом языке, главное понять идею. Процессы содержат своё состояние в себе, и общаются друг с другом только посредством сообщений, которые не хранятся в общей памяти, а копируются от процесса к процессу, и ложатся в специальный, личный для этого процесса, «почтовый ящик». Такая коммуникация является полностью асинхронной, никто никого не ожидает и сообщения обрабатываются другими процессами по мере возможности, когда удобно им. Более сложные модели коммуникации могут быть построены на этом примитиве. Такая модель совместной работы иногда ещё называется Actor Model.

Вернёмся к нашему Обходному Листу. На этот раз вместо того, чтобы ножками бегать по университету и подписывать везде наш лист, мы передаём наш лист в специальную бесплатную службу, которая по чистой случайности специализируется на Обходных Листах. Мы размещаем наш лист в конверте с подробной запиской, что с ним нужно сделать, и возвращаемся к своим делам. Через две недели в мой почтовый ящик приходит ответ с моим Обходным Листом, подписанным всеми инстанциями, и сопроводительной запиской с подробностями исполнения моего задания.

Вся суть этого метода в том, чтобы наладить сообщение между независимыми процессами, которые посылают друг другу запросы и ответы. Семантически эти сообщения никак не связаны с пославшими их процессами. Это означает, что если вы пишете проект на языке А, вас совершенно не затруднит связать эту программу с другой, написанной на языке Б, главное договориться о формате и обработке сообщений. Всё, о чём следует позаботиться, это о потоках сообщений, текущих через части вашей системы.

Само собой, это изящное решение достаётся нам совсем не бесплатно. Во многих случаях передача сообщений между параллельными процессами сделана с помощью глубокого копирования (включая все поля и их под-поля) исходящего сообщения. Проблема в том, что для больших структур, цена копирования становится ощутимой. Дополнительный расход памяти может негативно повлиять на вашу систему, особенно если у вас жёсткие ограничения по памяти, а сообщения имеют большой размер. На практике вам придётся следить за тем, чтобы посылаемые и принимаемые сообщения имели по возможности небольшой размер.

Подобно Будущим Результатам и Обещаниям, самая страшная проблема параллельного программирования — взаимная блокировка — практически решена, пусть всё ещё возможна, но очень маловероятна.

## Проблемы высоконагруженных многопоточных процессов

Современные операционные системы, реализуя модель потоков 1:1 довольно неэффективно планируют работу высоконагруженных многопоточных процессов с большим количеством единовременных потоков выполнения.

На основании результатов проведенного исследования можно утверждать, что на потерю производительности многопоточного процесса влияет ряд факторов:

* системные вызовы, которые возникают, например, на операциях чтения/записи при работе с файловыми дескрипторами, при захватах примитивов синхронизации, при работе с устройствами и т.д.
* очистки буфера ассоциативной трансляции TLB, которые возникают при кросс-ядерных переключениях контекста потока выполнения

## Способы решения и их обоснование

В подавляющем большинстве случаев проблемы такого рода успешно решаются современными методами разработки, которые отходят от канонов традиционного многопоточного программирования.

Для однопроцессорных систем с одним логическим ядром решение состоит в замене системной библиотеки потоков на совместимую с ней реализацию кооперативной модели N:1.

Для систем с несколькими логическими ядрами можно применить модель потоков N:M с многоуровневыми планировщиками. К тому же влияние фактора, связанного с большим количеством системных вызовов, для модели потоков N:M в будущем, вероятно, будет значительно снижено применением технологии агрегирования, описанной работе Марка Риттингхауса «System Call Aggregation for a Hybrid Thread Model».

Во всех случаях влияние отрицательных факторов сильно снижается за счет уменьшения частоты их проявления. Однако, применение современных подходов разработки или модели потоков N:M влечет за собой изменение кода программы, что не подходит для решения проблемы диссертационной работы. Также для подавляющего большинства популярных ОС не существует актуальной реализации модели N:M. Решение для однопроцессорных систем с одним логическим ядром в общем случае не подходит для SMP-систем.

В связи с этим напрашивается решение – разработать гибридную модель потоков выполнения N:M, реализовать которую можно было бы с учетом совместимости с системной библиотекой потоков. Данная задача выполнима, т.к. уровень планировщика вытесняющих потоков можно сделать прозрачным для пользователя. В итоге пользователь получит потоки N:X, где X будет контролироваться автоматически. Таким образом, получаем взаимозаменяемые потоки 1:1, N:1 и N:X.

Также хочется отметить удачное применение одной гибридной модели многопоточности в современном ПО Microsoft SQL Server. Начиная с 2008 года в данный программный была включена возможность работы в режиме волокон, повышающая производительность в полтора раза. Результаты испытаний этой модели, взятые из исследования авторитетного в сети Интернет разработчика Линчи Ши, коротко изображены на Рис. 1.1.

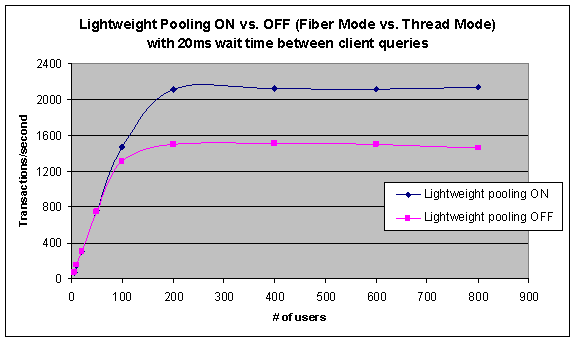


Рис. .. График зависимости режима работы Microsoft SQL Server 2008 от пользовательской нагрузки.

## Постановка задачи диссертационных исследований

Цель работы заключается в исследовании схем реализации многопоточности в современных многозадачных ОС и языках программирования.

В соответствии с целью и предметом исследования в диссертации необходимо решить следующие задачи:

* исследовать модели многопоточности в современных операционных системах и языках программирования;
* выявить и проанализировать узкие места в работе высоконагруженных процессов;
* разработать гибридную модель потоков выполнения с учетом проблем существующих моделей;
* исследовать полученную модель, проведя математическое моделирование работы высоконагруженного процесса;
* реализовать модель на языке программирования;
* провести научный эксперимент;
* сопоставить и проанализировать результаты моделирования и эксперимента.

## Выводы

Современные операционные системы, реализуя модель потоков 1:1 довольно неэффективно планируют работу высоконагруженных многопоточных процессов с большим количеством единовременных потоков выполнения. Однако, применение современных подходов разработки влечет за собой изменение кода программы, что не подходит для решения проблемы диссертационной работы. Было показано и обосновано, что проблему можно решить применением гибридных потоков с программным интерфейсом, совместимым с интерфейсом системных потоков. В виду отсутствия модели и реализации таких потоков выполнения их нужно будет разработать.

# Разработка гибридной модели потоков выполнения для повышения производительности высоконагруженных процессов

Как уже было замечено в первом разделе, моя исследовательская работа посвящена вопросам, связанным с потерями производительности, вызванные низкой эффективностью планирования этих потоков, и не затрагивает вопросы оптимизации их количества. В этом разделе будет разработана имитационная модель, представляющая ортогональную систему тестов, проведено моделирование работы двух основных моделей реализации многопоточности: вытесняющей и кооперативной – с целью анализа их свойств и разработана гибридная модель, обладающая характеристикам основных моделей.

Испытания будут проводиться в самых популярных на сегодняшний день ОС GNU Linux 3.5, Microsoft Windows 7, Apple Mac OS X Lion. Все системы предоставляют низкоуровневое API на языке C, поэтому реализация разработанной модели будет на C/C++.

Информационная закрытость коммерческих ОС заставила прибегнуть именно имитационному (ситуационному) моделированию. А результаты исследования, проведенного в первом разделе, послужат критерием проверки адекватности разработанной модели.

*Имитационное моделирование (ситуационное моделирование)* — метод, позволяющий строить модели, описывающие процессы так, как они проходили бы в действительности. Такую модель можно «проиграть» во времени как для одного испытания, так и заданного их множества. При этом результаты будут определяться случайным характером процессов. По этим данным можно получить достаточно устойчивую статистику.

*Имитационное моделирование* — это метод исследования, при котором изучаемая система заменяется моделью, с достаточной точностью описывающей реальную систему, с которой проводятся эксперименты с целью получения информации об этой системе.

*Имитационное моделирование* — это частный случай математического моделирования. Существует класс объектов, для которых по различным причинам не разработаны аналитические модели, либо не разработаны методы решения полученной модели. В этом случае аналитическая модель заменяется имитатором или имитационной моделью.

## Моделирование и формализация работы процесса

Данная глава посвящена разработке обобщенной имитационной модели работы процесса. Конечная имитационная модель будет представлять собой систему нескольких моделей (во избежание путаницы в дальнейшем буду называть ее системой тестов), построенную на обобщенной.

Схематично строение процесса многопоточного приложения изображено на :



Рис. .. Обобщенная схема строения процессов SMP системы

В целом многопоточный процесс занимается вычислением такой задачи, скорость выполнения которой упирается в ресурсы ОС. Многопоточный высоконагруженный процесс делает то же самое, но оперирует при этом множеством потоков выполнения, что повышает эффективность выполнения хорошо распараллеливаемых задач в SMP-системах.

В соответствии с концепцией, изложенной в предисловии к разделу, целесообразно рассматривать задачи с большим количеством параллельных подзадач, выполняющихся в некотором множестве потоков.

Входными и выходными данными алгоритма моделирования являются количество потоков в каждый момент времени, время тестирования, количество выполненных/выполняемых задач, задача и тип модели многопоточности. Причем, количество выполняемых задач может являться входным или выходным параметром. В противоположность ему время тестирования может являться выходным или входным параметром алгоритма. Какой из параметров является входным, а какой выходным зависит от выбранной методики тестирования, т.е. методики оценки производительности процесса. Можно выделить два способа:

1. зафиксировать объем задачи (количество подзадач) и сравнивать время ее выполнения разными процессами;
2. зафиксировать время выполнения задач (подзадач) и сравнивать выполненное количество.

Нетрудно убедиться в равнозначности этих методов, т.е. в том, что на одной и той же задаче оба метода покажут одинаковый прирост производительности в разах. Формально требуется доказать:

* – время выполнения задачи по первому методу;
* – количество выполненных задач за время соответственно.

Так как , где t – время выполнения одной подзадачи, то



Рис. .. Блок-схема алгоритма работы экспериментальной программы, подпрограммы A

Второй метод, очевидно, более приемлемый, т.к. позволяет зафиксировать время выполнения теста. Однако в общем случае производительность процесса зависит от объема работы, времени выполнения и объема подзадачи, т.к. планировщик некоторых ОС оптимизирует производительность, отдавая предпочтение более коротким задачам. Этот факт будет учтен в ходе проведения исследования. Таким образом, алгоритм экспериментальной программы будет выглядеть, как показано на .

Принимая во внимание закон Амдала:

* a – доля от общего объема вычислений, которая может быть получена только последовательными расчетами;
* p – количество процессоров.

получаем, что для полностью распараллеливаемого кода прирост производительности составит:

Учитывая тот факт, что в каждый момент времени на одном логическом процессорном ядре может находиться только один поток, прирост производительности при выполнении большого количества небольших задач в n потоках:

Тогда количество задач, выполняемых в p потоках, составит:

* – время выполнения задачи;
* – задержки (синхронизация, вытеснение, простои и пр.);
* – количество процессоров (или ядер, включая гипертрейдинг)
* – количество потоков выполнения.

Следовательно, производительность процесса:

В свою очередь задержка зависит от факторов:

* время и количество переключений контекста потока (задержка вытеснения)
* время создания и удаления потока
* накладные расходы примитивов синхронизации
* издержки на операции ввода-вывода

Из проведенного исследования ясно, что модель кооперативных потоков не позволяет процессу задействовать больше одного логического вычислительного ядра процессора. В связи с этим понятно, что сравнивать производительность моделей кооперативных и вытесняющих потоков (потоков ядра ОС) в SMP система некорректно. Поэтому кооперативные потоки нужно будет распределять по нескольким процессам так, чтобы обе модели в одинаковых тестовых ситуациях задействовали равное количество ядер. В конечном итоге, согласно разрабатываемой гибридной модели, пользовательские потоки должны отображаться на системные. Т.е. разница между гибридной моделью потоков и ее имитационной моделью будет заключаться в использовании различных сущностей ядра – процессов или потоков.

Значит, используя приведенный выше подход в имитационной модели, нужно учитывать различия между процессами и потоками, т.е. отсутствие общего пространства памяти и сложность контекста. Также необходимо будет установить оптимальное количество процессов для каждой тестовой ситуации.

Таким образом, алгоритм работы экспериментальной программы усложняется, как показано на рис. Рис. 2.3. Выходными данными по-прежнему является количество выполненных задач за время t, а к входным данных добавляется параметр x – количество запускаемых дочерних процессов.



Рис. .. Блок-схема алгоритма работы экспериментальной программы, подпрограммы B.

Для анализа результатов экспериментов необходима их визуализация, например, в виде графиков зависимости количества тестовых задач от количества потоков. Изначально мной был выбран путь написания скрипта командной оболочки ОС, который выдавал результаты измерений в текстовом виде, предназначенном для копирования в скрипты MATLAB. Но проводимых экспериментов оказалось так много, что данный способ стал непродуктивным. Тогда я пришел к идее написания MEX-расширения, который позволяет забирать результаты без участия пользователя, см. рис. Рис. 2.4. Входные данные остаются такими же, что у программы B, а выходными является количество выполненных задач в единицу времени.



Рис. .. Блок-схема алгоритма работы экспериментальной программы, подпрограммы C.



Рис. .. Блок-схема алгоритма работы экспериментальной программы, подпрограммы D.



Рис. .. Блок-схема взаимодействия подпрограмм экспериментальной программы.

Заключительным этапом является подпрограмма D (см. Рис. 2.5), скрипт для MATLAB, выполняющий моделирование работы высоконагруженного многопоточного процесса, по результатам которого строятся графики зависимости количества выполненных задач task и прироста производительности пользовательских потоков к системным от количества потоков выполнения. Измерения проводятся C раз в течение времени t, рассчитывается стандартное отклонение и относительная погрешность измерений. В качестве модели кооперативной потоков были выбраны реализации State Threads и GNU Portable Threads. Первая по заявлению разработчиков является более производительной, чем Portable Threads, но менее универсальной.

## Расчет погрешности измерений

## Выбор средств анализа результатов

В этой главе будут перечислены средства получения дополнительных параметров работы процесса, которые позволят проанализировать результаты работы модели. Инструмент, используемый для анализа работы, называют профилировщиком.

*Профилирование* — сбор характеристик работы программы, таких как время выполнения отдельных фрагментов (обычно подпрограмм), число верно предсказанных условных переходов, число кэш промахов и т. д.

2. 2. 1. Microsoft Concurrency Visualizer

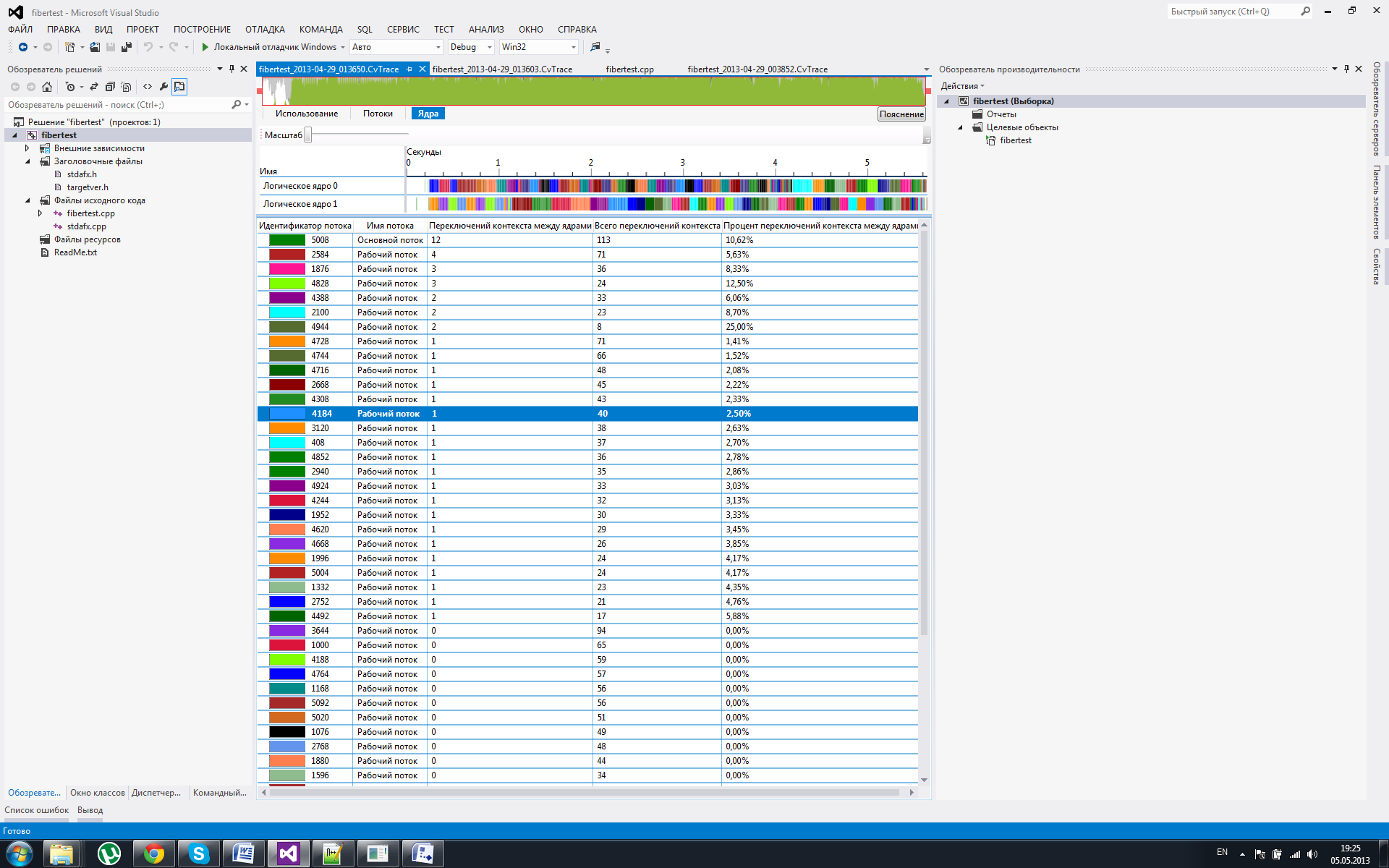


Рис. .. Снимок части окна программы Concurrency Visualizer

*Concurrency Visualizer* – средство профилирования, которое было добавлено в Visual Studio, начиная с версии 2010 года, позволяющее значительно облегчить сбор сведений о производительности параллельных программ, и предоставляет разработчикам возможность анализировать последовательные приложения на предмет возможности их распараллеливания.

Средство включает несколько инструментов для визуализации и создания отчетов. Поддерживается три основных представления (режима просмотра) для просмотра отчетов:

* CPU Utilization (использование процессора),
* Threads (потоки),
* Cores (ядра процессора).
  + 1. Intel VTune Amplifier

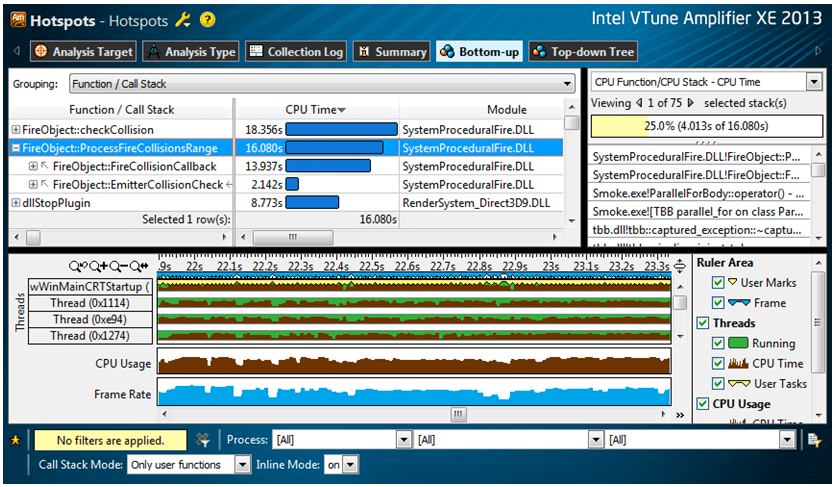


Рис. .. Снимок окна программы Intel VTune Amplifier XE 2013

*VTune Amplifier* – инструмент профилирования производительности от компании Intel, встраиваемое в Intel Parallel Studio и интегрируемое с Microsoft Visual Studio. На сегодняшний день разрабатывается для ОС Linux и Windows.

Данное средство позволяет провести:

* *Hotspot-анализ*. «На что моя программа тратит вычислительное время процессора?» Нам необходимо знать те места в программе, Hotspot-функции, где больше всего тратится вычислительных ресурсов при исполнении, а также тот путь, по которому мы в эти места попали, т.е. стэк вызовов.
* *Concurrency-анализ*. «Почему моя программа плохо распараллеливается?» Часто бывает, что ожидаемый прирост производительности, например, при переходе от 4-ядерной системы к 8-ядерной, так и не достигается. Поэтому тут нужна оценка эффективности параллельного кода, которая дала бы представление о том, насколько полно используются ресурсы микропроцессора.
* *Lock & Wait - анализ*. «Где моя программа простаивает в ожидании синхронизации или операции ввода-вывода?» Поняв, что наша программа плохо масштабируется, мы хотим найти, где именно и какие именно объекты синхронизации стали на пути к хорошей параллельности. Возможно, необходимо пересмотреть реализацию алгоритмов, а может, и всю параллельную инфраструктуру приложения.
  + 1. AMD CodeAnalyst Performance Analyzer

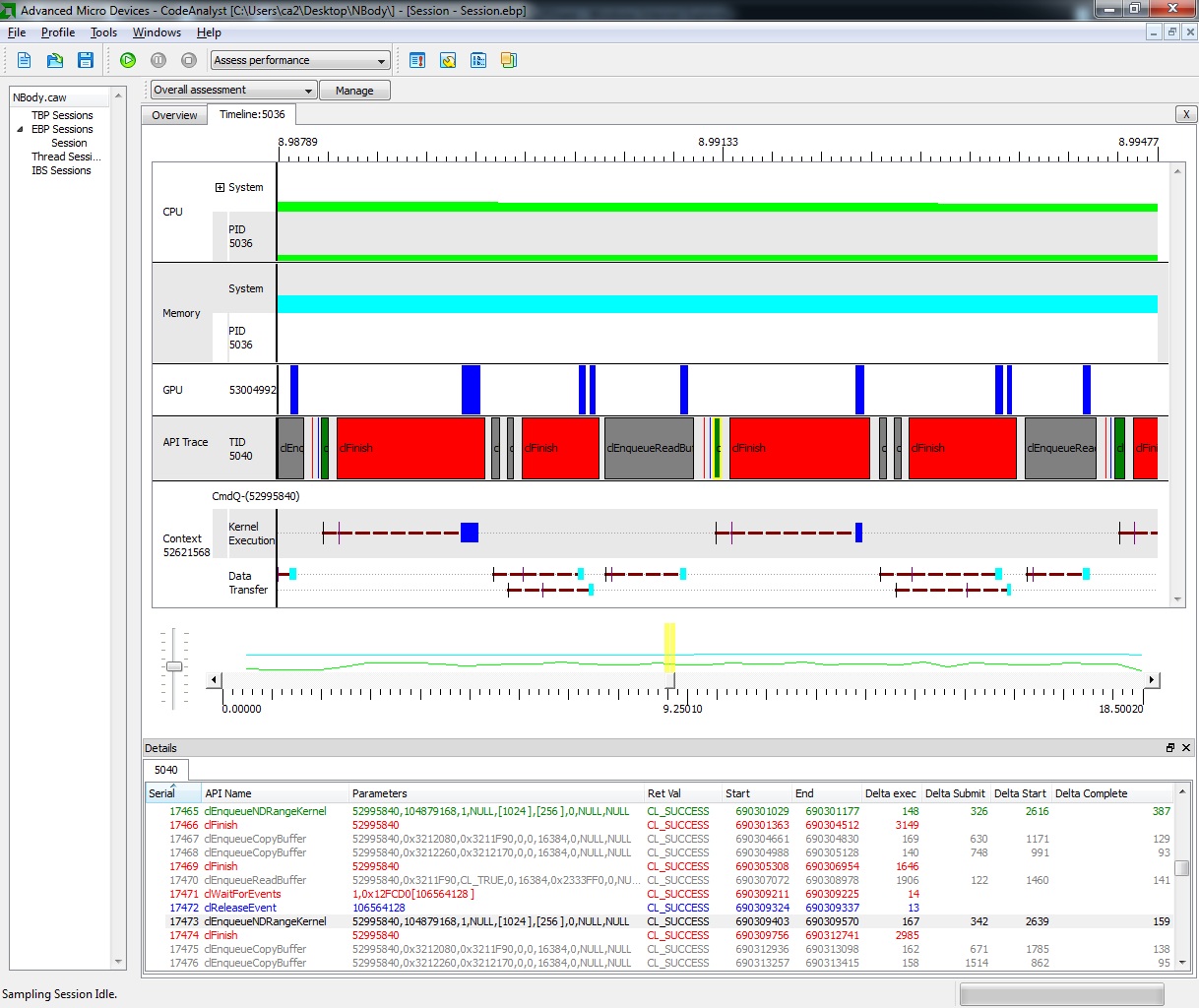


Рис. 2.9. Снимок окна программы AMD CodeAnalyst Performance Analyzer

Аналогичное Intel VTune Amplifier средство профилирование. Так же, как и средство от Intel работает в ОС Windows и Linux, интегрируется с Microsoft Visual Studio.

* + 1. Apple XCode Instruments

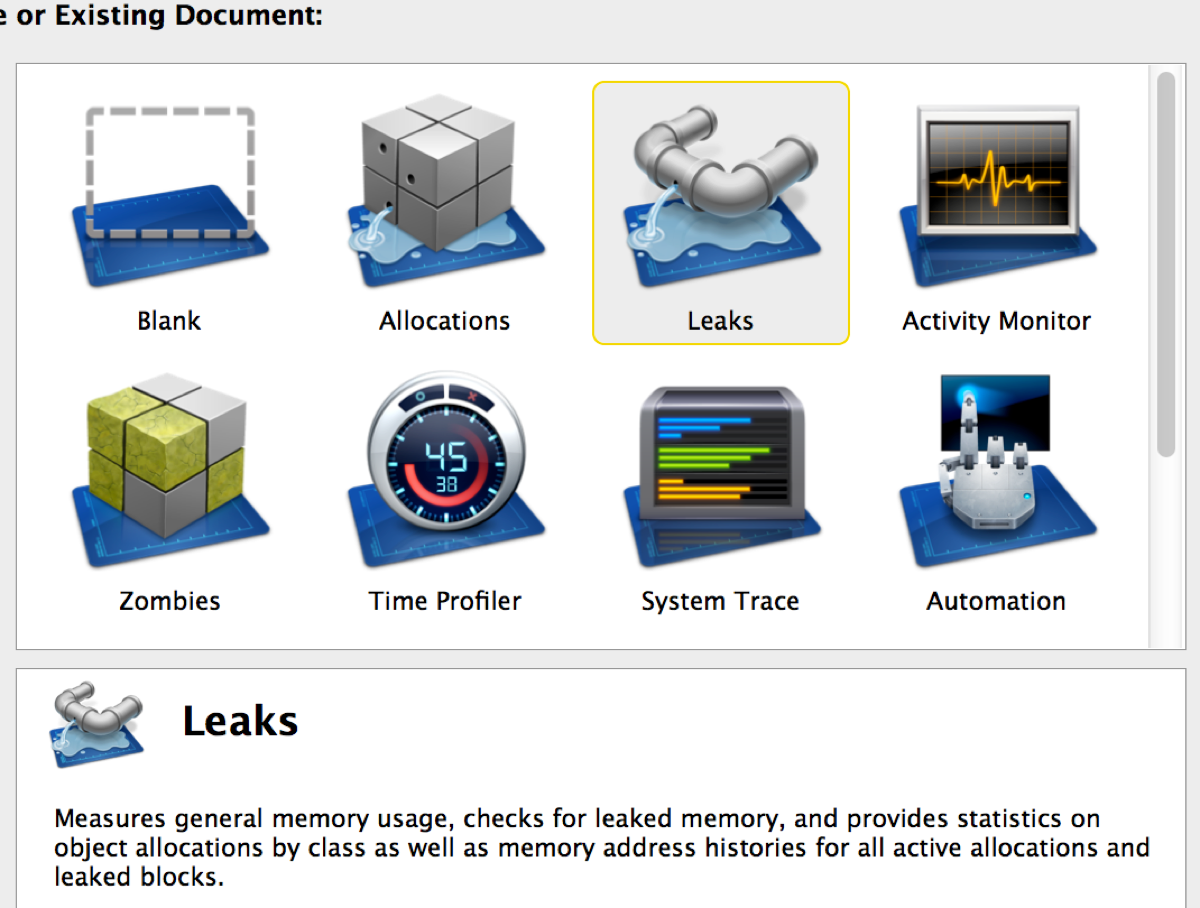


Рис. .. Снимок экрана с набором инструментов анализа работы приложения Apple XCode Instruments

*Instruments* – набор инструментов профилирования компании Apple, разработанный для IDE XCode. Помимо средств анализа производительности включает в себя утилиты детектирования утечек памяти и взаимных блокировок потоков.

* + 1. GNU Valgrind



Рис. .. Снимок окна терминала с результатами профилирования программы Valgrind

Данное средство относится к категории ПО с открытым исходным кодом, имеет консольный интерфейс, работает в Unix-подобных ОС, служит для выявления утечек памяти и детального профилирования программы.

* + 1. Сравнение средств профилирования

В таблице ниже приведено сравнение средств профилирования, перечисленных в главе 2.2.

Табл. .. Сравнение характеристик профилирующего ПО

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Название** | **Функц.** | **Удобное** | **Linux** | **Windows** | **Mac OS** | **Беспл.** |
| Concurrency Vizualizer | + | + | - | + | - | + |
| VTune Amplifier | + | + | + | + | - | - |
| Performance Analyzer | + | + | + | + | - | - |
| Instruments | +/- | + | - | - | + | + |
| Valgrind | +/- | +/- | + | - | + | + |

Следовательно, наиболее подходящими вариантами являются:

* Concurrency Vizualizer для Windows
* Instruments для Mac OS

Для Linux в своей работе я предпочел VTune Amplifier, т.к. оно вполне подходит для проведения исследовательской работы, учитывая возможность использования пробной 30-тидневной версии.

## Разработка системы тестов для моделирования работы потоков

Теперь необходимо выделить основной набор характеристик моделей многопоточности, изменение значений которых влиял бы на производительность потоков. Их достаточно много: задержка вытеснения и их количество, издержки на создание/удаление потока, чувствительность к среде выполнения, загрузка ЦП, производительность вычислений на процессоре и сопроцессоре – поэтому нужно выбрать только основные, которые отличали бы одну модель от другой. Также нужно, чтобы система тестов была ортогональной, т.е. не должно быть такого параметра, влияющего на производительность, значение которого варьировалось бы в двух различных тестах, формально должно выполняться:

где – множество из N тестов, p – функция, возвращающая множество параметров, охватываемых тестом.

Разумеется, влияния тех или иных факторов, анализируемых в каждом тесте должны быть различимы. К примеру, мы должны знать, что фактор A в тесте B внес 10% вклад, а фактор C – 90%. Анализировать результаты таких тестов довольно сложно, и, скорей всего, они будут давать только косвенные подтверждения выдвинутой гипотезы. По мере возможности их стоит избегать.

Наконец, стоит перейти к самому сложному вопросу – вопросу о полноте системы тестов. Самый простой способ сделать систему полной – перебрать все возможные варианты входных данных и проверить правильность получаемых результатов. К сожалению, воспользоваться этим способом почти никогда не удается. Даже для простейших программ количество вариантов входных данных оказывается астрономическим. Отсюда печальный вывод: исчерпывающее тестирование (т.е. перебор всех возможных вариантов выполнения) для любой нетривиальной программы невозможно. Критерии, по которым проводится классификация всех возможных вариантов выполнения программы с точки зрения проверки правильности программы, называются критериями полноты тестирования. Существует два подхода к формулированию критериев полноты тестирования: критерии «черного ящика» и критерии «белого ящика». Критерии черного ящика описывают тестирование с точки зрения поставленной задачи, не касаясь внутреннего устройства программы. Критерии белого ящика учитывают структуры программы. Воспользоваться последним довольно сложно в связи с закрытостью ОС Mac OS и Windows. А в том числе на этих платформах будет проводиться моделирование. Поэтому выбор падает на «черный ящик».

Учитывая вышеизложенное, можно перечислить тесты, отражающие важнейшие свойства потоков выполнения:

* Создание/удаление потоков. Вариантов реализации кооперативных потоков в отличии от системных может быть несколько. Например, потоки GNU Portable Threads рассчитаны на как можно больший охват UNIX-подобных систем и совместимость API, а State Threads на максимальную производительность, достигая этого в том числе путем упрощения возможностей. Поэтому расходы на создание/удаление потоков могут различаться.
* Переключение контекста. Планировщик кооперативных потоков тратит меньше системных вызовов на смену контекста, чем планировщик потоков ядра, поэтому этот фактор также должен быть отмечен.
* Работа с общей памятью. При работе с общей памятью возрастает вероятность кэш-промахов при использовании ее в нескольких потоках ядра. Этого же нельзя сказать в случае с кооперативными потоками, где доступ к памяти всегда последовательный.
* Длительные/короткие вычисления. В этом тесте будет изучено влияние частоты смены контекста потоков на производительность процесса. В отличии от предыдущего теста будет варьироваться количество переключений, а не сравниваться их длительность.
* Примитивы синхронизации. Требуется для исследования влияния издержек, относящихся к примитивам синхронизации: мьютексам, барьерам, семафорам, условиям, затворам.
* Операции чтения/записи, работа с файловыми дескрипторами. Эти тесты служат для анализа влияния фактора, связанного с работой потока с файловыми дескрипторами.
* Нагруженная среда выполнения. Тестирование проводится в среде, где существует несколько процессов, требующих большое количество вычислительных ресурсов процессора. Таким образом, происходит борьба за процессорное время, и в этой ситуации в SMP системах с несколькими логическими ядрами возрастает количество «дорогих» кросс-ядерных переключений контекста потоков выполнения.

## Создание/удаление потоков

Самый простой тест, проверяющий создание/удаление потоков.



Рис. .. Блок-схема алгоритма работы тестовой задачи 3, создание и удаление потока

* + 1. Переключение контекста

Основа теста – команда *yield*, передающая управление планировщику потоков для смены контекста потока выполнения. Для потоков pthread таковой является команда *sched\_yield*, для State Threads – *st\_yield*, для GNU Portable Threads – *pth\_yield*. Данная команда производит смену контекста до истечения кванта времени (это касается только системных потоков), что позволяет поставить обе модели в равные условия.

С целью сделать издержки на создание/удаление потоков незначительными в рамках данного теста было выбрано достаточно большое число итераций.



Рис. .. Блок-схема алгоритма работы тестовой задачи 2, переключение контекста потока

* + 1. Короткие вычисления



Рис. .. Блок-схема алгоритма работы тестовой задачи 1, короткие вычисления

В тесте проводится серия коротких измерений длительностью 30мс.

* + 1. Длительные вычисления

В тесте проводится серия коротких измерений длительностью 10с.



Рис. .. Блок-схема алгоритма работы тестовой задачи 5, длительные вычисления

* + 1. Общая память



Рис. .. Блок-схема алгоритма работы тестовой задачи 6, общая память

Аналогичен тесту с короткими вычислениями за исключением того, что переменная, куда записывается результат, статическая атомарная.

* + 1. Примитивы синхронизации

В качестве примитива синхронизации был взят мьютекс. Можно было выбрать любой другой: барьер, семафор, затвор – т.к. все они построены на базе одного и того же механизма. У потоков pthread, State Threads и GNU Portable Threads принципы работы с ним различаются.

С целью сделать издержки на создание/удаление потоков незначительными в рамках данного теста было выбрано достаточно большое число итераций.



Рис. .. Блок-схема алгоритма работы тестовой задачи 4, примитивы синхронизации

* + 1. Операции чтения/записи

Данный тест посвящен работе с дескрипторами чтения/записи. В тесте открываются 3 дескриптора чтения и 1 дескриптор записи для работы с символьными устройствами.

Аналогично предыдущим тестам с целью сделать издержки на создание/удаление потоков незначительными в рамках данного теста было выбрано достаточно большое число итераций.



.. Блок-схема алгоритма работы тестовой задачи 8, чтение/запись, работа с файловыми дескрипторами

## Разработка гибридной модели потоков

Согласно проведенному исследованию, пик производительности высоконагруженного процесса при использовании вытесняющих (системных) потоков в SMP системе достигается при совпадении количества выделенных потоков с количеством доступных логических ядер ЦП. Ниже этого количества не достигается полная загрузка всех ядер. Выше этого значения происходит борьба за процессорное время, и, как следствие этого, рост накладных расходов.

Таким образом, когда количество потоков меньше или равно количеству доступных логических ядер, должна происходить привязка 1:1, т.е. одного пользовательского потока к одному системному. При возрастании этого значения выделение потоков должно достигаться исключительно за счет пользовательских, причем привязка должна происходить к системному потоку с наименьшим количеством пользовательских для соблюдения равномерности распределения.

Забегая немного вперед, во всех тестах, кроме одного, издержками на создание/удаление можно пренебречь, поэтому теоретический расчет поведения гибридной модели сводится к разделению графиков на две части по границе, где количество потоков достигает количества логических ядер ЦП, и дальнейшему сведению левой части графика для системных с правой частью для кооперативных потоков путем масштабирования последней.

Логика теоретического расчета поведения гибридной модели в тесте на создание/удаление потоков такова: до 4-х потоков включительно при завершении одного из них нужно будет выделять 1 системный и 1 пользовательский поток, а при большем количестве потребуется выделение всего лишь одного пользовательского. Это значит, что при использовании свыше 4-х потоков гибридная модель будет вести себя, как кооперативная. А до 4-х потоков включительно формула расчета, например при гибридизации потоков Pthreads потоками State Threads, следующая:

где и – количество потоков выделяемых за 1с соответствующими моделями, единица в знаменателе – время, требуемое для выделения потоков Pthreads, дробь в знаменателе – время, требуемое для выделения потоков State Threads. Упростив получим,

## Анализ результатов моделирования

Ниже приведены результаты моделирования. Значения количества выполненных задач на графиках рассчитаны за единицу времени (1с). Для отмеченных точек на графике проведено 10 измерений по 10 секунд каждое. Вычислена относительная погрешность, приведены логические выкладки, объясняющие результаты. Все тесты проведены на Mac OS X 10.7.5 с процессором Intel Core i5 2.3 ГГц. Теоретический расчет гибридной модели проведен при условии гибридизации потоками State Threads.

Условные обозначения: *st* – State Threads, *pth* – GNU Portable Threads, *pthread* – системные потоки, *theoretical* – теоретическая модель гибридных потоков. Маленькими окружностями обозначены конкретные измерения, графики построены по математическому ожиданию.

## Создание/удаление потоков

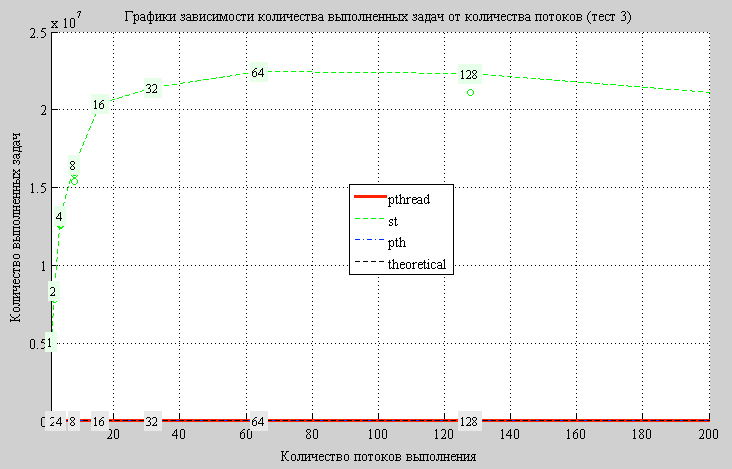


Рис. .. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, создание/удаление потока (а)

Относительная погрешность измерений 0.0048.

Из результатов эксперимента видно, что скорость создания/удаления потоков State Threads во много раз превосходит скорости у остальных моделей.

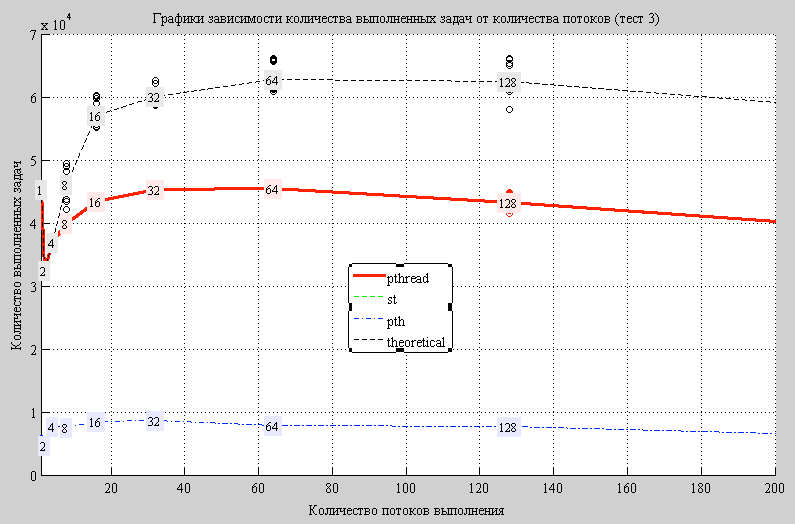


Рис. .. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, создание/удаление потока (б)

Тем не менее, даже результатами аутсайдера Portable Threads можно пренебречь для дальнейших тестов, т.к. скорость создания/удаления одного потока составляет около 120нс. Для системных потоков эти издержки не превысят 5мс на 1с времени, т.е. будут менее 0.5%.

Также стоит заметить, что вряд ли найдется хотя бы один программный продукт, которому необходимо выделение столь большого числа потоков (более 40 тыс.) за единицу времени.

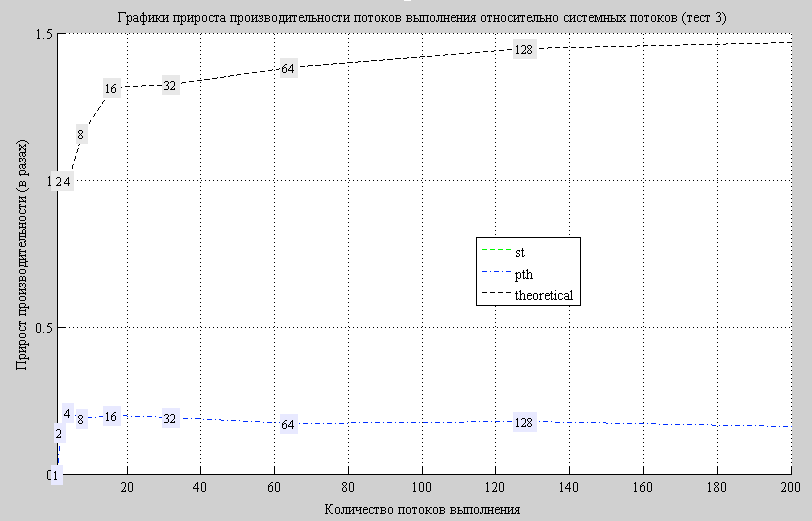


Рис. .. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, создание/удаление потока

На графике модели гибридных потоков можно наблюдать полуторакратный прирост производительности.

**Вывод**: полуторакратный прирост производительности, полученный в результате моделирования, станет крайне незначительным в реальных программах, однако стоит отметить, что издержи на применение гибридной модели также крайне малы.

* + 1. Переключение контекста

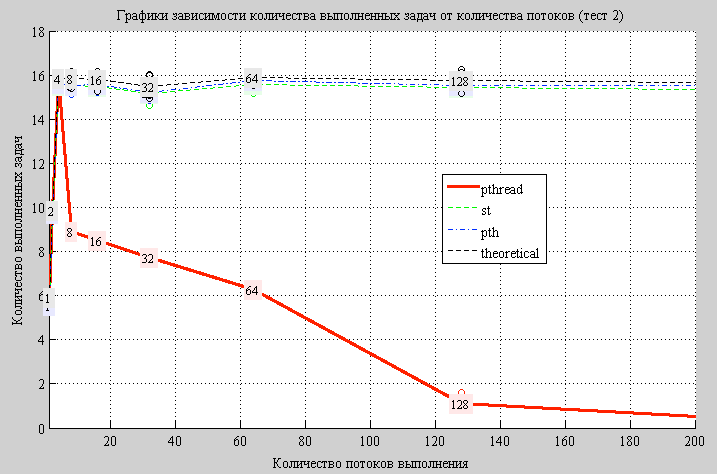


Рис. .. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, переключение контекста

Относительная погрешность измерений 0.0102.

На графиках можно наблюдать стабильность работы кооперативной модели и снижение производительности вытесняющих с ростом количества потоков выполнения.

В результате профилирования была выдвинута гипотеза, что снижению производительности системных потоков способствует пропорциональное возрастание количества кэш-промахов и сложности работы планировщика ядра.

За единицу времени системный тест для 4-х системных потоков совершает около 16 млн. переключений контекста. Однако это довольно большое число. Для сравнения, пользовательская ОС Ubuntu Linux совершает около 35 принудительных переключений в единицу времени, а серверная ОС Ubuntu Server – около 6.

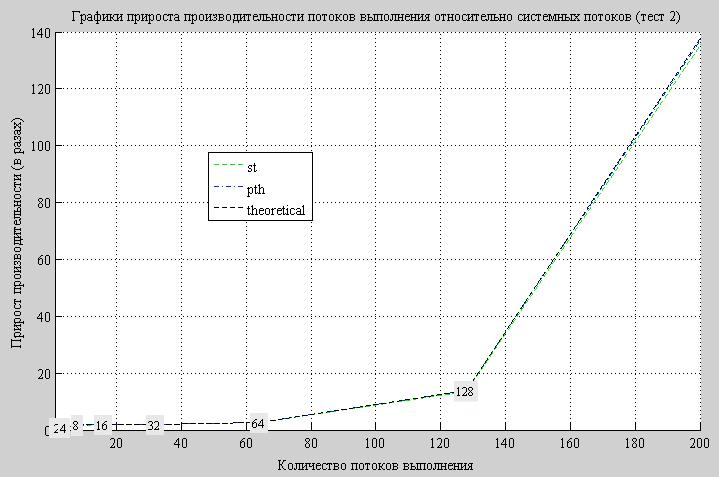


Рис. .. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, переключение контекста

**Вывод**: полуторакратный прирост производительности, полученный в результате моделирования, станет крайне незначительным в реальных программах, однако стоит отметить, что издержи на применение гибридной модели также крайне малы.

* + 1. Короткие вычисления

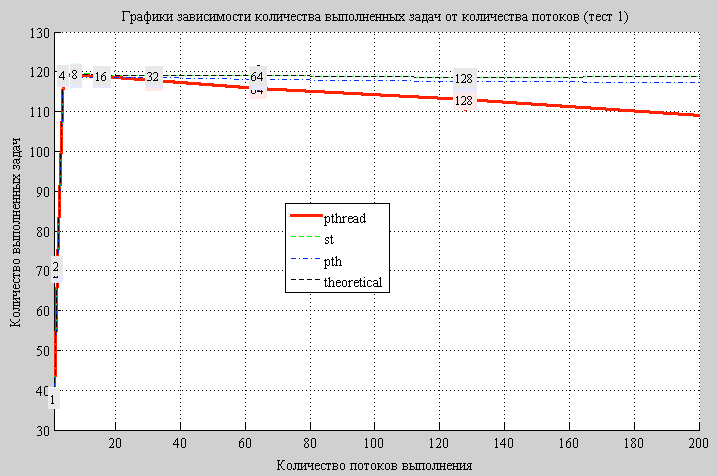


Рис. .. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, короткие вычисления

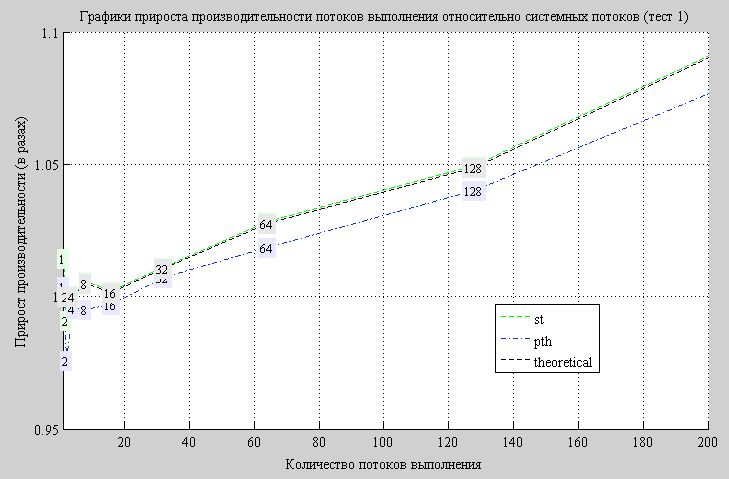


Рис. .. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, короткие вычисления

Относительная погрешность измерений 0.0113.

* + 1. Длительные вычисления

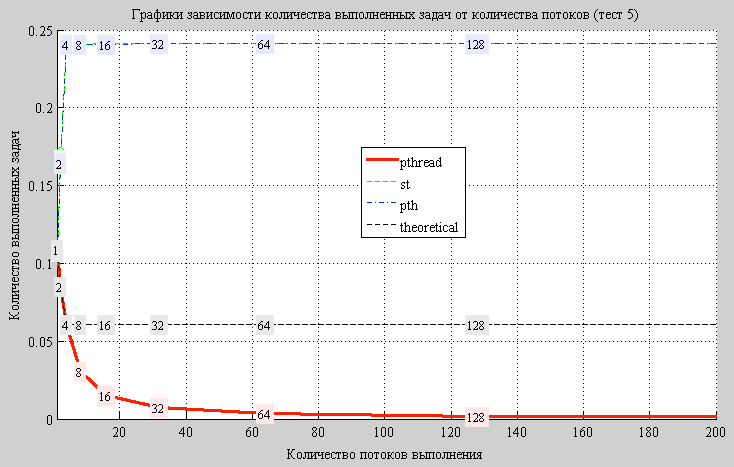


Рис. .. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, длительные вычисления

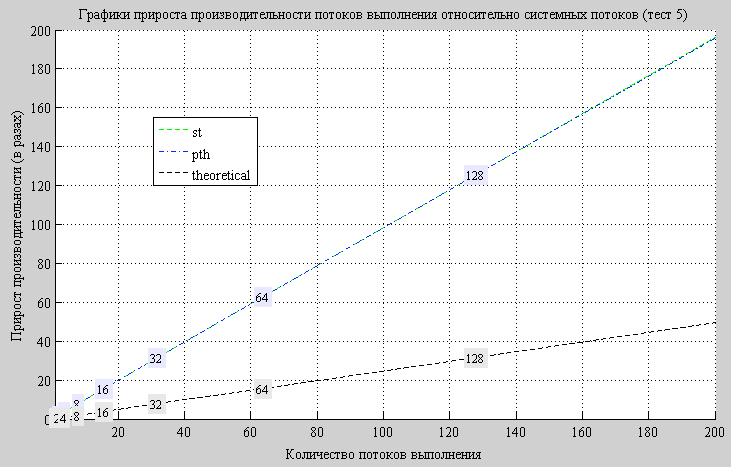


Рис. .. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, длительные вычисления

Относительная погрешность измерений 0.0011.

* + 1. Общая память

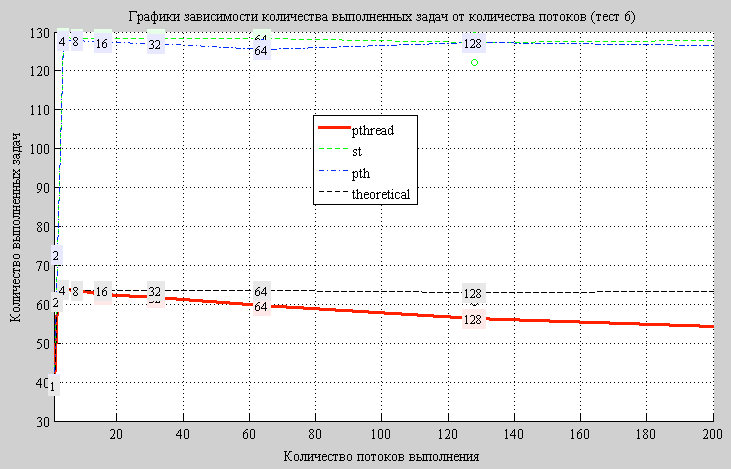


Рис. .. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, общая память

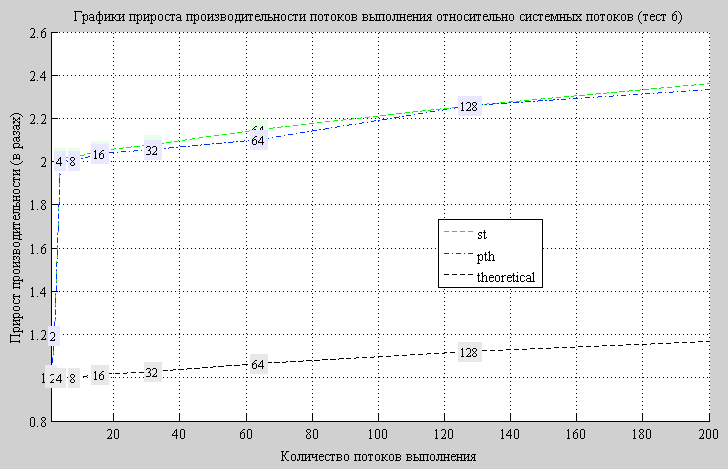


Рис. .. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, общая память

Относительная погрешность измерений 0.0044.

* + 1. Примитивы синхронизации

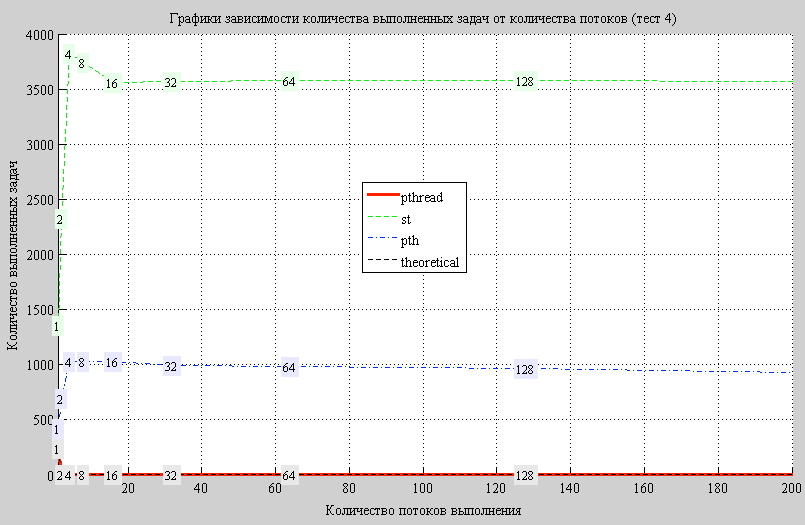


Рис. .. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, примитивы синхронизации (а)

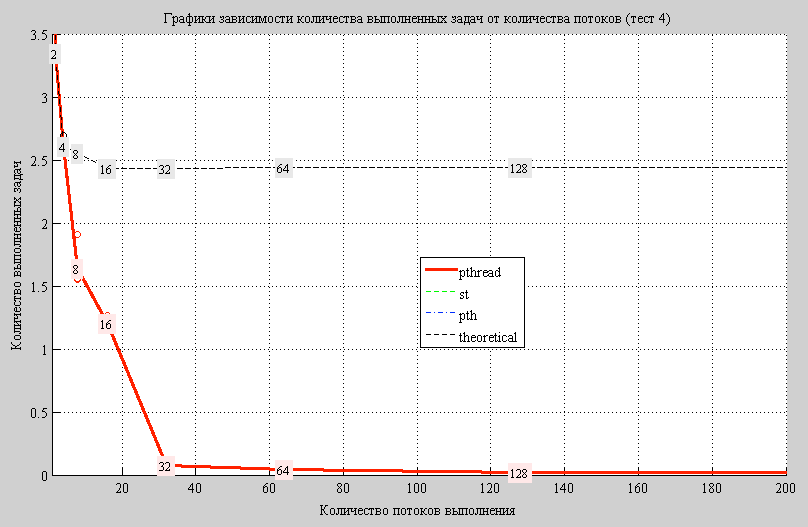


Рис. .. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, примитивы синхронизации (б)

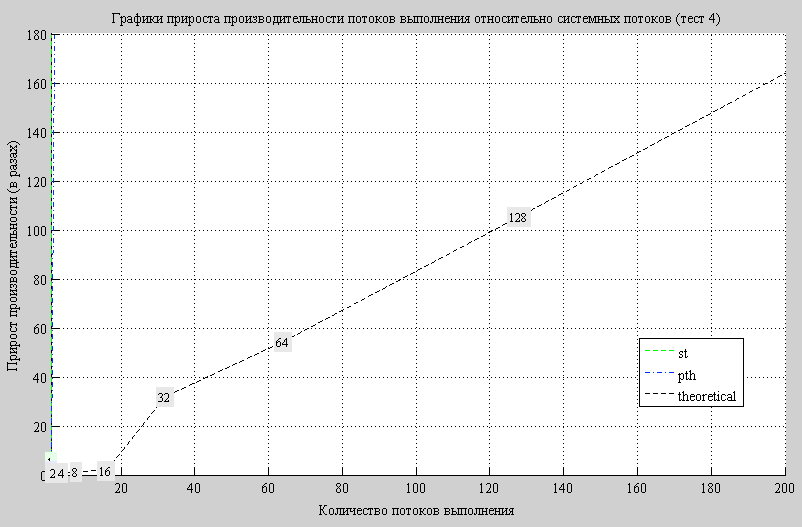


Рис. .. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, примитивы синхронизации

Относительная погрешность измерений 0.0017.

**Вывод**: высокий прирост производительности, полученный в результате моделирования, станет крайне незначительным в реальных программах, однако стоит отметить, что издержи на применение гибридной модели также крайне малы.

* + 1. Операции чтения/записи

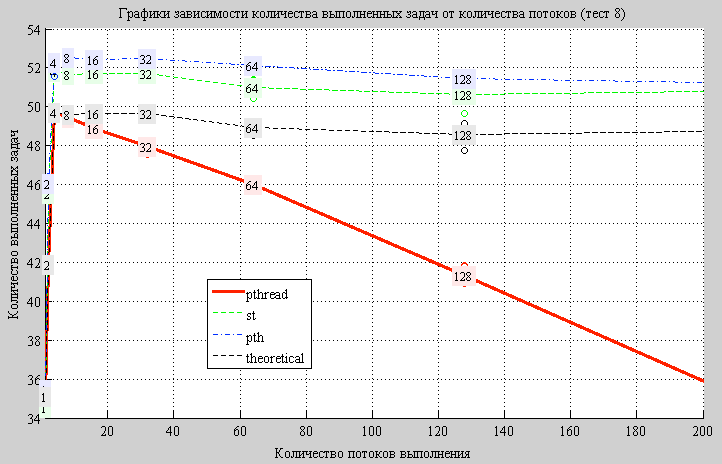


Рис. .. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, операции чтения/записи

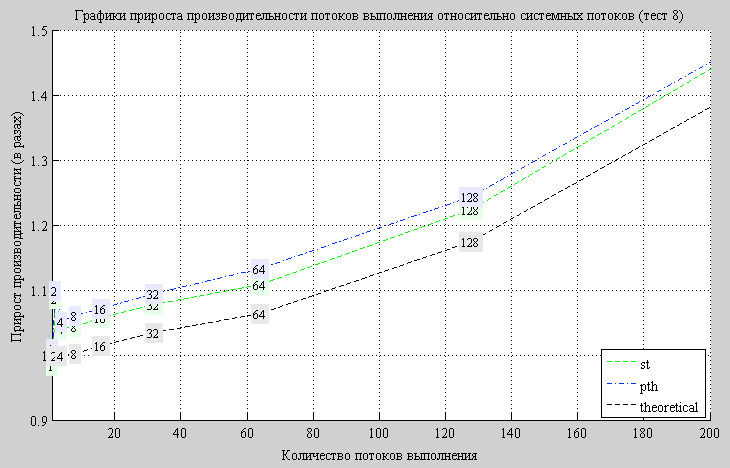


Рис. .. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, операции чтения/записи

Относительная погрешность измерений 0.0073.

## Анализ свойств гибридной модели

По результатам моделирования был сделан вывод, что наиболее ярко выраженный прирост будет достигаться в тестах на примитивах синхронизации и операциях чтения/записи.

Для конкретизации результатов необходимо провести моделирование поведения полученной модели гибридных потоков на практическом примере. Для этой цели по адресу <http://sourceforge.net/projects/state-threads/> был взят и адаптирован пример веб-сервиса, разработанный для демонстрации работы кооперативных потоков State Threads. Программа запускает процессы в количестве равным количеству логических ядер процессора, каждый из которых слушает 80-й порт. На каждое подключение создается поток-обработчик, который формирует и отправляет клиенту заголовки и тело HTTP-ответа. Генератор нагрузки был взят по адресу <http://www.spec.org/osg/web99/>.

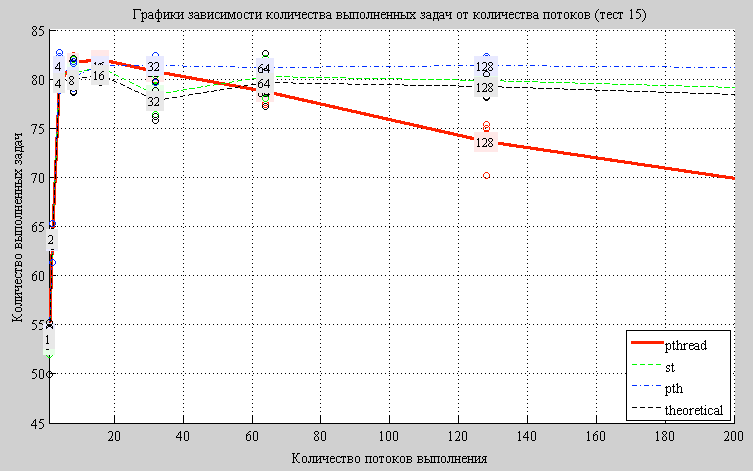


Рис. .. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, HTTP-сервер

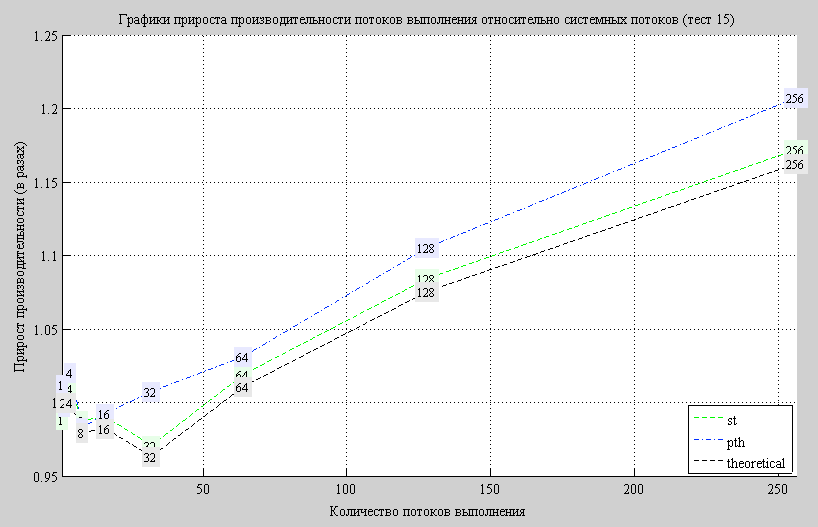


Рис. .. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, HTTP-сервер

Относительная погрешность измерений 0.0171.

## Парадокс производительности

Рассмотрим функцию на языке C, приведенную ниже.

DWORD WINAPI MyThreadFunction(LPVOID lpParam) {

volatile auto x = 1;

for (auto i = 0; i < 800000000 / MAX\_THREADS; ++i) {

x += i / 3;

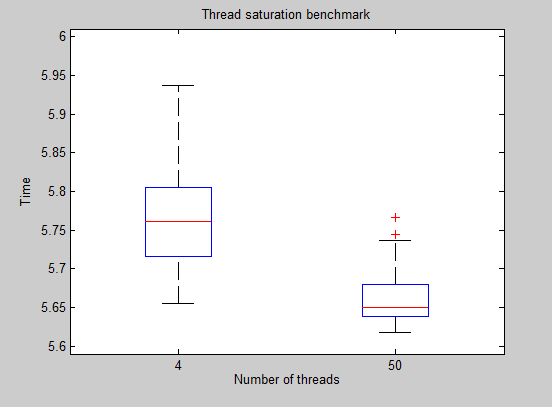
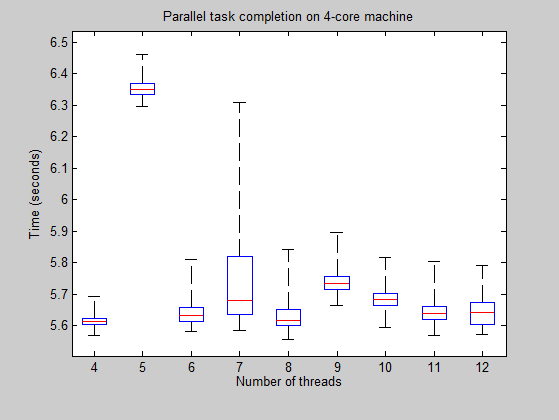
}

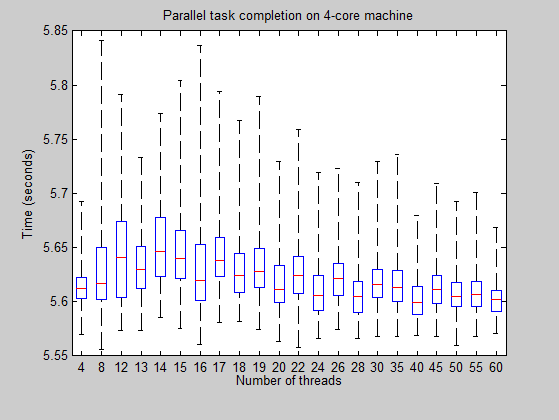
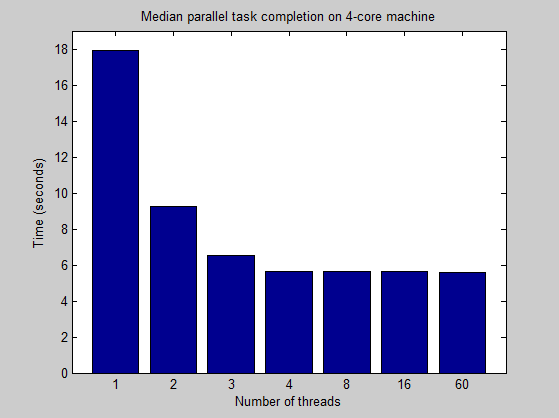
return 0;

}

Запустив ее, например, на Intel Core 2 Duo, MS Windows 7 c MAX\_THREADS равным 4 и 50, можно наблюдать, задача, выполняемая в 4 потока завершиться позже, чем задача, выполняемая в 50 потоков – 7.1с против 5.8с. Почему же так происходит?

Ниже приведены диаграммы с подробными результатами измерений.

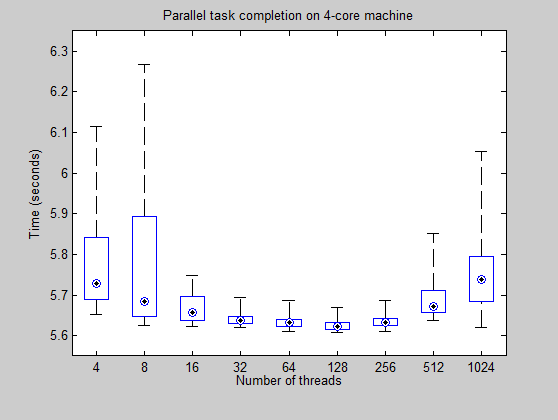


Рис. .. Парадокс производительности, диаграммы результатов

Все тесты проведены с большим количеством испытаний. Диаграммы включают вероятностные свечи и гистограмму.

Правильный ответ на поставленный вопрос связан с пониманием планирования потоков. Задача не выполнена, пока все потоки не завершат работу. В первом случае каждый поток должен выполнить четверть работы, во втором – одну пятидесятую часть. Т.к. рассматриваемый процесс работает в системе совместно с другими процессами, возникает рассинхронизация работы его потоков. И, как только один из потоков в случае с 4-мя потоками завершает работу, логическое ядро, на котором он исполнялся, начинает простаивать. При завершении работы одного из 4-х последних потоков в случае 50-ти потоков возникает аналогичная ситуация. Но максимальный простой во втором случае будет короче. В следствии этого тело и фитиль свечи в первом случае будет длиннее, чем во втором.

Данный эффект служит наглядной демонстрацией необходимости в тщательном продумывании деталей тестов сравнения производительности.

# Верификация и применение результатов разработки

## Реализация гибридной модели

Данная глава посвящена реализации полученной модели гибридных потоков выполнения. Модель разработана с учетом возможности реализации ее совместимой с системным API. Полагаю, что сделать кроссплатформенную реализацию довольно сложно, как и обеспечить совместимость API. Поэтому в рамках диссертационной работы я решил продемонстрировать реализацию c частично совместимым POSIX API. В виду этого, а также по причине отсутствия текстов программ, демонстрация применения разработки для биржевых серверов NYSE, ROX и других программных продуктов, решение проблем которых рассматривает данная работа, не представляется возможным.

В этой главе будут изложены основные принципы реализации разработанной гибридной модели многопоточности такие, как инверсия управления, «голливудский» принцип, частотный принцип, принцип модульности и принцип функциональной избирательности. Также будут рассмотрены программные особенности моделей потоков State Threads и POSIX Threads.

* + 1. Принципы разработки

*Инверсия управления (Inversion of Control, IoC)* — важный принцип объектно-ориентированного программирования, используемый для уменьшения связанности в компьютерных программах.

Формулируется принцип следующим образом:

* Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракции.
* Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Одной из реализаций IoC является *внедрение зависимостей (англ. Dependency Injection).* Инъекция зависимостей используется во многих framework'ах, которые называются IoC контейнерами.

К техникам реализации относятся:

* Фабричный метод (англ. Factory Method pattern)
* Service locator (англ. Service locator pattern)
* Внедрение зависимости (англ. Dependency injection)
  + Через метод класса (англ. Setter injection)
  + Через конструктор (англ. Constructor injection)
  + Через интерфейс внедрения (англ. Interface injection)
* IoC контейнер (англ. IoC-container)

*Фабричный метод (англ. Factory Method)* — порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, *Фабрика* делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне. Также известен под названием виртуальный конструктор (англ. Virtual Constructor).

*Внедрение зависимости (англ. Dependency injection)* — процесс предоставления внешней зависимости программному компоненту. Условно, если объекту нужно получить доступ к определенному сервису, объект берет на себя ответственность за доступ к этому сервису: он или получает прямую ссылку на местонахождение сервиса, или обращается к известному «сервис-локатору» и запрашивает ссылку на реализацию определенного типа сервиса. Используя же внедрение зависимости, объект просто предоставляет свойство, которое в состоянии хранить ссылку на нужный тип сервиса; и когда объект создается, ссылка на реализацию нужного типа сервиса автоматически вставляется в это свойство (поле), используя средства среды. Внедрение зависимости более гибко, потому что становится легче создавать альтернативные реализации данного типа сервиса, а потом указывать, какая именно реализация должна быть использована в, например, конфигурационном файле, без изменений в объектах, которые этот сервис используют. Это особенно полезно в юнит-тестировании, потому что вставить реализацию «заглушки» сервиса в тестируемый объект очень просто. С другой стороны, излишнее использование внедрения зависимостей может сделать приложения более сложными и трудными в сопровождении: так как для понимания поведения программы программисту необходимо смотреть не только в исходный код, а еще и в конфигурацию, а конфигурация, как правило, невидима для IDE, которые поддерживают анализ ссылок и рефакторинг, если явно не указана поддержка фреймворков с внедрениями зависимостей.

*Голливудский принцип* по Фаулеру – это синоним принципа инверсии управления. Согласно идеи, компоненты высокого уровня (например, интерфейсы) определяют за компоненты низкого уровня (реализации), как и когда им подключаться к системе. Авторы Head First Design Patterns допускают, что согласно этому принципу компоненты низкого уровня могут участвовать в вычислениях без формирования зависимостей с компонентами высокого уровня, и в этом состоит отличие от более жесткого принципа инверсии управления.

*Частотный принцип*. Принцип основан на выделении в алгоритмах и данных особых групп по частоте использования. Для действий, наиболее часто встречающихся при работе программ, создаются условия их быстрого выполнения. К часто используемым данным обеспечивается наиболее быстрый доступ. «Частые» операции стараются делать более короткими. Следует отметить, что лишь не более 5 % операторов программы оказывают ощутимое влияние на скорость выполнения программы. Этот факт позволяет значительную часть операторов программы кодировать без учета скорости вычислений, обращая основное внимание при этом на «красоту» и наглядность текстов.

*Принцип модульности*. Под модулем в данном контексте понимают функциональный элемент рассматриваемой системы, имеющий оформление, законченное и выполненное в пределах требований системы, и средства сопряжения с подобными элементами или элементами более высокого уровня данной или другой системы. Способы обособления составных частей программ в отдельные модули могут различаться существенно. В значительной степени разделение системы на модули определяется используемым методом проектирования программ.

*Принцип функциональной избирательности*. Этот принцип является логическим продолжением частотного и модульного принципов и используется при проектировании программ. В программах выделяется некоторая часть важных модулей, которые постоянно должны быть в состоянии готовности для эффективной организации вычислительного процесса. Эту часть в программах называют ядром или монитором. При формировании состава монитора требуется учесть два противоречивых требования. В состав монитора, помимо чисто управляющих модулей, должны войти наиболее часто используемые модули. Количество модулей должно быть таким, чтобы объем памяти, занимаемой монитором, был не слишком большим. Программы, входящие в состав монитора, постоянно хранятся в оперативной памяти. Остальные части программ постоянно хранятся во внешних запоминающих устройствах и загружаются в оперативную память только при необходимости, перекрывая друг друга также при необходимости.

*Принцип генерируемости.* Основное положение этого принципа определяет такой способ исходного представления программы, который бы позволял осуществлять настройку на конкретную конфигурацию технических средств, круг решаемых проблем, условия работы пользователя.

*Принцип функциональной избыточности.* Этот принцип учитывает возможность проведения одной и той же работы различными средствами. Особенно важен учет этого принципа при разработке пользовательского интерфейса для выдачи одних и тех же данных разными способами вызова из-за психологических различий в восприятии информации.

*Принцип «по-умолчанию»* применяется для облегчения  организации связей с системой как на стадии генерации, так и при работе с уже готовыми программами. Принцип основан на хранении в системе некоторых базовых описаний структур, модулей,  конфигураций оборудования и данных, определяющих условия работы с программой. Эту информацию программа использует в качестве  заданной по умолчанию, если пользователь забудет или сознательно не конкретизирует ее.

* + 1. POSIX Threads

*POSIX Threads* — стандарт POSIX реализации потоков (нитей) выполнения, определяющий API для создания и управления ими.

Библиотеки, реализующие этот стандарт (и функции этого стандарта), обычно называются Pthreads (функции имеют приставку «pthread\_»). Хотя наиболее известны варианты для Unix-подобных операционных систем, таких как Linux или Solaris, но существует и реализация для Microsoft Windows (Pthreads-w32).

Pthreads определяет набор типов и функций на языке программирования Си. Заголовочный файл — pthread.h.

Типы данных:

* pthread\_t: дескриптор потока.
* pthread\_attr\_t: перечень атрибутов потока.

Функции управления потоками:

* pthread\_create(): создание потока.
* pthread\_exit(): завершение потока (должна вызываться функцией потока при завершении).
* pthread\_cancel(): отмена потока.
* pthread\_join(): подключиться к другому потоку и ожидать его завершения; поток, к которому необходимо подключиться, должен быть создан с возможностью подключения (PTHREAD\_CREATE\_JOINABLE).
* pthread\_detach(): отключиться от потока, сделав его при этом отдельным (PTHREAD\_CREATE\_DETACHED).
* pthread\_attr\_init(): инициализировать структуру атрибутов потока.
* pthread\_attr\_setdetachstate(): указывает параметр «отделимости» потока (detach state), который говорит о возможности подключения к нему (при помощи pthread\_join) других потоков (значение PTHREAD\_CREATE\_JOINABLE) для ожидания окончания или о запрете подключения (значение PTHREAD\_CREATE\_DETACHED); ресурсы отдельного потока (PTHREAD\_CREATE\_DETACHED) при завершении автоматически освобождаются и возвращаются системе.
* pthread\_attr\_destroy(): освободить память от структуры атрибутов потока (уничтожить дескриптор).

Функции синхронизации потоков:

* pthread\_mutex\_init(), pthread\_mutex\_destroy(), pthread\_mutex\_lock(), pthread\_mutex\_trylock(), pthread\_mutex\_unlock(): с помощью мьютексов.
* pthread\_cond\_init(), pthread\_cond\_signal(), pthread\_cond\_wait(): с помощью условных переменных.
  + 1. State Threads

State Threads – небольшая библиотека потоков выполнения, которая обеспечивает основу для написания быстрых и масштабируемых интернет-приложений (таких, как веб-серверы, прокси-серверы, агенты передачи почты, и другие сетевые приложения, занимающиеся обработкой данных) на Unix-подобных платформах.

Эта библиотека сочетает в себе простоту парадигмы многопоточного программирования, в которой один поток поддерживает одновременное подключение каждого, с производительностью и масштабируемостью управляемой событиями архитектуры машины состояний. Другими словами, эта библиотека предлагает потоковое API для структурирования интернет-приложений в виде конечного автомата.

Библиотека State Threads является производной от Netscape Portable Runtime библиотеки (NSPR) и, следовательно, распространяется на условиях Mozilla Public License (MPL) версии 1.1 или GNU General Public License (GPL) версии 2 или более поздней.

State Threads определяет набор типов и функций на языке программирования Си. Заголовочный файл — st.h.

Типы данных:

* st\_thread\_t: дескриптор потока.
* st\_thread\_attr\_t: перечень атрибутов потока.

Функции управления потоками:

* st\_thread\_create(): создание потока.
* st\_thread\_exit(): завершение потока (должна вызываться функцией потока при завершении).
* st\_thread\_cancel(): отмена потока.
* st\_thread\_join(): подключиться к другому потоку и ожидать его завершения; поток, к которому необходимо подключиться, должен быть создан с возможностью подключения (PTHREAD\_CREATE\_JOINABLE).
* st\_thread\_detach(): отключиться от потока, сделав его при этом отдельным (PTHREAD\_CREATE\_DETACHED).
* st\_thread\_attr\_init(): инициализировать структуру атрибутов потока.
* st\_thread\_attr\_setdetachstate(): указывает параметр «отделимости» потока (detach state), который говорит о возможности подключения к нему (при помощи st\_thread\_join) других потоков (значение PTHREAD\_CREATE\_JOINABLE) для ожидания окончания или о запрете подключения (значение PTHREAD\_CREATE\_DETACHED); ресурсы отдельного потока (PTHREAD\_CREATE\_DETACHED) при завершении автоматически освобождаются и возвращаются системе.
* st\_thread\_attr\_destroy(): освободить память от структуры атрибутов потока (уничтожить дескриптор).

Функции синхронизации потоков:

* st\_thread\_mutex\_init(), st\_thread\_mutex\_destroy(), st\_thread\_mutex\_lock(), st\_thread\_mutex\_trylock(), st\_thread\_mutex\_unlock(): с помощью мьютексов.
* st\_thread\_cond\_init(), st\_thread\_cond\_signal(), st\_thread\_cond\_wait(): с помощью условных переменных.

## Анализ работы полученной реализации

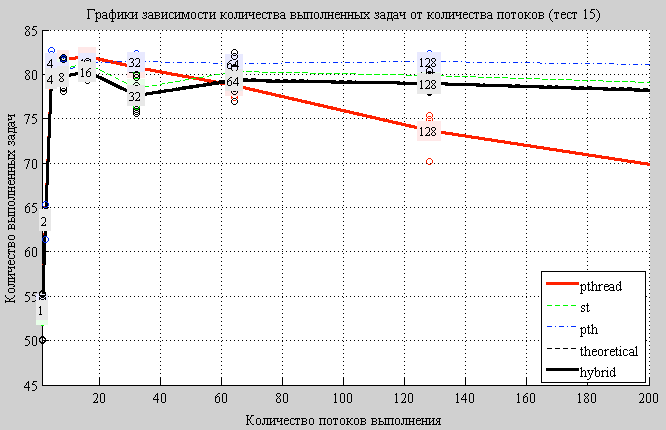


Рис. .. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, HTTP-сервер

Относительная погрешность измерений 0.0171.

График реализации гибридной модели почти совпал с графиком теоретического расчета поведения гибридной, что подтверждает адекватность построенной имитационной модели.

Прирост производительности наблюдается, начиная с 60 одновременных подключений. Ниже этого порога наблюдаются потери, ограниченные 5%. При 256 одновременных соединений можно отметить 12% прирост.

**Вывод**: эксперимент с тестовым HTTP-сервером подтверждает гипотезу об эффективности применения гибридной модели потоков выполнения для повышения производительности высоконагруженных многопоточных приложений, однако стоит отметить и издержи на применение гибридной модели при малом количестве потоков.

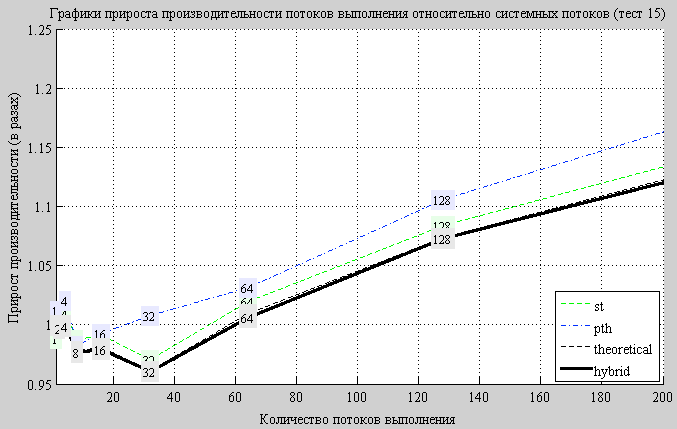


Рис. .. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, HTTP-сервер

# Заключение

В результате проведения исследования удалось решить следующие задачи:

* исследовать модели многопоточности в современных операционных системах и языках программирования;
* разработать имитационную модель работы высоконагруженного многопоточного процесса;
* выявить и проанализировать основные характеристики моделей потоков выполнения на имитационной модели;
* обосновать адекватность имитационной модели;
* разработать гибридную модель потоков выполнения с учетом проблем существующих моделей;
* реализовать модель на языке программирования;
* применить гибридную модель к тестовой программе и убедиться в приросте производительности;
* уточнить область применения разработанной гибридной модели.

# Список литературы

# Список иллюстраций

Рис. 1.1. График зависимости режима работы Microsoft SQL Server 2008 от пользовательской нагрузки. 27

Рис. 2.1. Обобщенная схема строения процессов SMP системы 30

Рис. 2.2. Блок-схема алгоритма работы экспериментальной программы, подпрограммы A 32

Рис. 2.3. Блок-схема алгоритма работы экспериментальной программы, подпрограммы B. 35

Рис. 2.4. Блок-схема алгоритма работы экспериментальной программы, подпрограммы C. 36

Рис. 2.5. Блок-схема алгоритма работы экспериментальной программы, подпрограммы D. 37

Рис. 2.6. Блок-схема взаимодействия подпрограмм экспериментальной программы. 38

Рис. 2.7. Снимок части окна программы Concurrency Visualizer 40

Рис. 2.8. Снимок окна программы Intel VTune Amplifier XE 2013 41

Рис. 2.9. Снимок окна программы AMD CodeAnalyst Performance Analyzer 42

Рис. 2.10. Снимок экрана с набором инструментов анализа работы приложения Apple XCode Instruments 43

Рис. 2.11. Снимок окна терминала с результатами профилирования программы Valgrind 43

Рис. 2.12. Блок-схема алгоритма работы тестовой задачи 3, создание и удаление потока 47

Рис. 2.13. Блок-схема алгоритма работы тестовой задачи 2, переключение контекста потока 48

Рис. 2.14. Блок-схема алгоритма работы тестовой задачи 1, короткие вычисления 48

Рис. 2.15. Блок-схема алгоритма работы тестовой задачи 5, длительные вычисления 49

Рис. 2.16. Блок-схема алгоритма работы тестовой задачи 6, общая память 49

Рис. 2.17. Блок-схема алгоритма работы тестовой задачи 4, примитивы синхронизации 50

2.18. Блок-схема алгоритма работы тестовой задачи 8, чтение/запись, работа с файловыми дескрипторами 51

Рис. 2.19. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, создание/удаление потока (а) 53

Рис. 2.20. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, создание/удаление потока (б) 54

Рис. 2.21. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, создание/удаление потока 55

Рис. 2.22. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, переключение контекста 56

Рис. 2.23. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, переключение контекста 57

Рис. 2.24. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, короткие вычисления 58

Рис. 2.25. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, короткие вычисления 58

Рис. 2.26. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, длительные вычисления 59

Рис. 2.27. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, длительные вычисления 59

Рис. 2.28. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, общая память 60

Рис. 2.29. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, общая память 60

Рис. 2.30. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, примитивы синхронизации (а) 61

Рис. 2.31. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, примитивы синхронизации (б) 61

Рис. 2.32. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, примитивы синхронизации 62

Рис. 2.33. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, операции чтения/записи 63

Рис. 2.34. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, операции чтения/записи 63

Рис. 2.35. Графики зависимости количества выполненных задач в единицу времени от количества потоков выполнения, HTTP-сервер 64

Рис. 2.36. Графики зависимости прироста производительности потоков выполнения относительно системных потоков, HTTP-сервер 65

Рис. 2.37. Парадокс производительности, диаграммы результатов 66