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 3

Глава 1. Обзор существующих моделей многопоточности и методов параллельного программирования 7

1.1. Многозадачность и ее типы 7

1.2. Потоки выполнения 16

1.3. Модели многопоточности 17

1.4. Методы параллельного программирования 18

1.5. Проблемы и недостатки высоконагруженных процессов 23

1.6. Способы решения и их обоснование 23

1.7. Постановка задачи диссертационных исследований 23

Выводы по главе 1 24

Глава 2. Разработка метода повышения производительности высоконагруженных процессов 26

2.1. Закон Амдала 26

2.2. Формализация работы высоконагруженного процесса 26

# 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 математического моделирования, методы теории массового обслуживания, а также научный 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.

#### Структура и объем работы

Диссертация состоит из введения, трех глав, заключения и списка литературы. Работа содержит 95 страниц основного текста, 42 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 распределённых 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 распознаёт 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

#### Трудности реализации многозадачной среды

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.

#### 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.

#### 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.

#### 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.

#### Вытесняющая или приоритетная многозадачность (режим реального времени)

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.

Голодание (starvation)

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).

#### Гонка (race condition)

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).

#### 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 (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.

#### 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.

#### 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

#### 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.

#### Потоки данных, Будущие результаты и Обещания (Futures & Promises)

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...

#### 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 и недостатки высоконагруженных процессов

## Способы решения и их обоснование

## 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 по главе 1

Проанализировав существующие алгоритмы балансировки нагрузки, можно сделать вывод, что каждый алгоритм имеет свои достоинства и недостатки.

Достоинствами основных cache-unaware алгоритмов является их простота реализации, низкие накладные расходы, накладываемые на работу алгоритма, и отсутствие проблем при добавлении или удалении web-сервера. Среди основных недостатков можно отметить неравномерную загрузку серверов, а также тот факт, что cache-unaware алгоритмы не направляют запросы для максимизации попадания в локальный кэш.

Cache-unaware алгоритмы нашли широкое применение в устройствах балансировки нагрузки, производимых такими компаниями как Cisco, RedHill и другие.

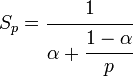
Достоинствами cache-aware алгоритмов является максимизация попадания в локальный кэш web-серверов а также простота реализации некоторых из алгоритмов (алгоритмы простого и консистентного хеширования). Среди недостатков можно отметить неравномерную загрузку серверов, а также возможные проблемы уменьшения частоты попадания в кэш при добавлении или удалении сервера.

# Разработка метода повышения производительности высоконагруженных процессов

## Закон Амдала

Закон Амдала (англ. Amdahl's law, иногда также Закон Амдаля-Уэра) — иллюстрирует ограничение роста производительности вычислительной системы с увеличением количества вычислителей. Джин Амдал сформулировал закон в 1967 году, обнаружив простое по существу, но непреодолимое по содержанию ограничение на рост производительности при распараллеливании вычислений: «В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента». Согласно этому закону, ускорение выполнения программы за счёт распараллеливания её инструкций на множестве вычислителей ограничено временем, необходимым для выполнения её последовательных инструкций.

Предположим, что необходимо решить некоторую вычислительную задачу. Предположим, что её алгоритм таков, что доля от общего объёма вычислений может быть получена только последовательными расчётами, а, соответственно, доля может быть распараллелена идеально (то есть время вычисления будет обратно пропорционально числу задействованных узлов ). Тогда ускорение, которое может быть получено на вычислительной системе из процессоров, по сравнению с однопроцессорным решением не будет превышать величины



## Формализация работы высоконагруженного процесса

В формализованном представлении работы высоконагруженного процесса многопоточного приложения, схематически изображенного на рис. 2.1. можно выделить следующие элементы:

* множество параллельных задач;
* множество пользовательских потоков;
* множество потоков ядра ОС;
* множество процессоров SMP системы.

Характеристиками такой системы будут являться:

* – время выполнения каждой задачи одним процессором SMP системы;
* – общее время простоя каждой задачи;
* ­– совокупность множеств временных отрезков между соседними блокирующими операциями
* – частота принудительного переключения контекста ядром ОС;
* – длительность переключения контекста ядром ОС;
* – длительность переключения контекста ядром пользовательского уровня.

Известно, что в современных ОС частота переключения контекста величина непостоянная. Смена контекста может произойти либо по таймеру (принудительная), либо при выполнении блокирующей операции. Таким образом, количество переключений контекста ядром для каждой задачи:

где dt – временной отрезок множества fi, а div - функция деления нацело.

Если условиться, что блокирующие операции распределены равномерно, то получим:

где qi – количество элементов множества fi, dt – промежуток времени между двумя соседними блокирующими операциями.

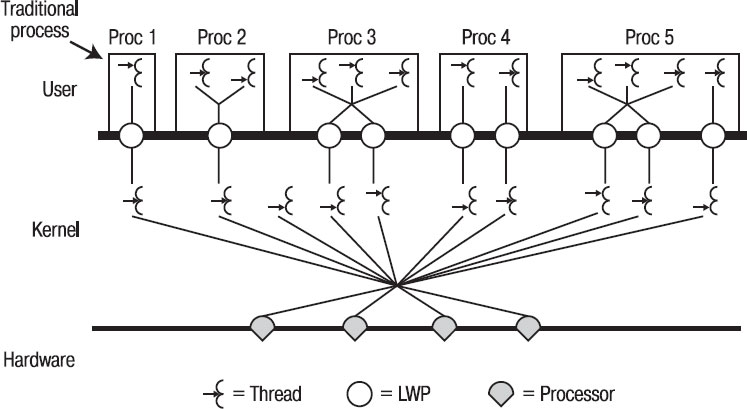


Figure Обобщенная схема строения процессов SMP системы

Ядро пользовательского уровня переключает контекст только при выполнении блокирующей операции или по требованию алгоритма выполняемой задачи (инструкция yield). Инструкцию yield можно отнести к операциям множества блокирующих, которая приостанавливает выполнение программы на 0 секунд.

Таким образом, получаем длительность выполнения процесса:

Для упрощения будем рассматривать выполнение однотипных задач в разных пользовательских потоках, т.е. в каждом потоке будет выполняться некоторая задача k. Тогда можно полагать, что

где (множество номеров строк кода задачи k).

Поэтому длительность выполнения процесса примет вид:

расписать