



0,00

Рейтинг

Семинары Станислава Сидристого

CLRium #7: Parallel Computing Practice



sidristij 16 февраля 2020 в 22:48

Планирование потоков в Windows. Часть 1 из 4

Блог компании Семинары Станислава Сидристого, Программирование, .NET, Параллельное программирование

Ниже представлена не простая расшифровка доклада с семинара CLRium, а переработанная версия для книги .NET Platform Architecture. Той её части, что относится к потокам.

CLRium #6: Потоки, приоритеты, планирование

Потоки и планирование потоков

Что такое поток? Давайте дадим краткое определение. По своей сути поток это:

- Средство параллельного относительно других потоков исполнения кода;
- Имеющего общий доступ ко всем ресурсам процесса.

Очень часто слышишь такое мнение, что потоки в .NET — они какие-то абсолютно свои. И наши .NET потоки являются чем-то более облегчённым чем есть в Windows. Но на самом деле потоки в .NET являются самыми обычными потоками Windows (хоть Windows thread id и скрыто так, что сложно достать). И если Вас удивляет, почему я буду рассказывать не-.NET вещи в хэбе .NET, скажу вам так: если нет понимания этого уровня, можно забыть о хорошем понимании того, как и почему именно так работает код. Почему мы должны ставить volatile, использовать Interlocked и SpinWait. Дальше обычного lock дело не уйдёт. И очень даже зря.

Давайте посмотрим из чего они состоят и как они рождаются. По сути поток — это средство эмуляции параллельного исполнения относительно других потоков. Почему эмуляция? Потому, что поток как бы странно и смело это ни звучало — это чисто программная вещь, которая идёт из операционной системы. А операционная система создаёт этот слой эмуляции для нас. Процессор при этом о потоках ничего не знает вообще.

Задача процессора — просто исполнять код. Поэтому с точки зрения процессора есть только один поток: последовательное исполнение команд. А задача операционной системы каким-либо образом менять поток т.о. чтобы эмулировать несколько потоков.

Поток в физическом понимании

"Но как же так?", — скажите вы, — "во многих магазинах и на различных сайтах я вижу запись "Intel Xeon 8 ядер 16 потоков". Говоря по-правде это — либо скудность в терминологии либо — чисто маркетинговый ход. На самом деле внутри одного большого процессора есть в данном случае 8 ядер и каждое ядро состоит из двух логических процессоров. Такое доступно при наличии в процессоре технологии Hyper-Threading, когда каждое ядро эмулирует поведение двух процессоров (но не потоков). Делается это для повышения производительности, да. Но по большому счёту если нет понимания, на каких потоках идут расчёты, можно получить очень не приятный сценарий, когда код выполняется со скоростью, ниже чем если бы расчёты шли на одном ядре. Именно поэтому раздача ядер идёт $+2$ в случае Hyper-Threading. Т.е. пропуская парные ядра.

Технология эта — достаточно спорная: если вы работаете на двух таких псевдо-ядрах (**логических процессорах**, которые эмулируются технологией Hyper-Threading), которые при этом находятся на одном *физическом* ядре и работают с одной и той-же памятью, то вы будете постоянно попадать в ситуацию, когда второй логический процессор так же пытается обратиться к данной памяти, создавая блокировку либо попадая в блокировку, т.к. поток, находящийся на первом ядре работает с той же памятью.

Возникает блокировка совместного доступа: хоть и идёт эмуляция двух ядер, на самом-то деле оно одно. Поэтому в наихудшем сценарии эти потоки исполняются по очереди, а не параллельно.

Так если процессор ничего не знает о потоках, как же достигается параллельное исполнение потоков на каждом из его ядер? Как было сказано, поток — средство операционной системы выполнять на одном процессоре несколько задач одновременно. Достигается параллелизм очень быстрым переключением между потоками в течение очень короткого промежутка времени. Последовательно запуская на выполнение код каждого из потоков и делая это достаточно часто, операционная система достигает цели: делает их исполнение псевдопараллельным, но параллельным с точки зрения восприятия человека. Второе обоснование существования потоков — это утверждение, что программа не так часто срывается в математические расчёты. Чаще всего она взаимодействует с окружающим её миром: различным оборудованием. Это и работа с жёстким диском и вывод на экран и работа с клавиатурой и мышью. Поэтому чтобы процессор не простаивал, пока оборудование сделает то, чего хочет от него программа, поток можно на это время установить в состояние блокировки: ожидания сигнала от операционной системы, что оборудование сделало то, что от него просили. Простейший пример этого — вызов метода `Console.ReadKey()`.

Если заглянуть в диспетчер задач Windows 10, то можно заметить, что в данный момент в вашей системе существует около 1,5 тысячи потоков. И если учесть, что квант на десктопе равен 20 мс, а ядер, например, 4, то можно сделать вывод, что каждый поток получает 20 мс работы 1 раз в 7,5 сек... Ну конечно же, нет. Просто почти все потоки чего-то ждут. То ввода пользователя, то изменения ключей реестра... В операционной системе существует очень много причин, чтобы что-либо ждать.

Так что пока одни потоки в блокировке, другие — что-то делают.

Создание потоков

Простейшая функция создания потоков в пользовательском режиме операционной системы — `CreateThread`. Эта функция создаёт поток в текущем процессе. Вариантов параметризации `CreateThread` очень много и когда мы вызываем `new Thread()`, то из нашего .NET кода вызывается данная функция операционной системы.

В эту функцию передаются следующие атрибуты:

1) Необязательная структура с атрибутами безопасности:

- Дескриптор безопасности (`SECURITY_ATTRIBUTES`) + признак наследуемости дескриптора.

В .NET его нет, но можно создать поток через вызов функции операционной системы;

2) Необязательный размер стека:

- Начальный размер стека, в байтах (система округляет это значение до размера страницы памяти)

Т.к. за нас размер стека передаёт .NET, нам это делать не нужно. Это необходимо для вызовов методов и поддержки памяти.

3) Указатель на функцию — точка входа нового потока

4) Необязательный аргумент для передачи данных функции потока.

Из того, что мы **не** имеем в .NET явно — это структура безопасности с атрибутами безопасности и размер стека. Размер стека нас мало интересует, но атрибуты безопасности нас могут заинтересовать, т.к. сталкиваемся мы с ними впервые. Сейчас мы рассматривать их не будем. Скажу только, что они влияют на возможность изменения информации о потоке средствами операционной системы.

Если мы создаём любым способом: из .NET или же вручную, средствами ОС, мы как итог имеем и `ManagedThreadId` и экземпляр класса `Thread`.

Также у этой функции есть необязательный флаг: `CREATE_SUSPENDED` — поток после создания не стартует. Для .NET это поведение по умолчанию.

Помимо всего прочего существует дополнительный метод `CreateRemoteThread`, который создаёт поток в чужом процессе. Он часто используется для мониторинга состояния чужого процесса (например программа `Snoop`). Этот метод создаёт в другом процессе поток и там наш поток начинает исполнение. Приложения .NET так же могут заливать свои потоки в чужие процессы, однако тут могут возникнуть проблемы. Первая и самая главная — это отсутствие в целевом потоке .NET runtime. Это значит, что ни одного метода фреймворка там не будет: только WinAPI и то, что вы написали сами. Однако, если там .NET есть, то возникает вторая проблема (которой не было раньше). Это — версия runtime. Необходимо: понять, что там запущено (для этого необходимо импортировать не-.NET методы runtime, которые написаны на C/C++ и разобраться, с чем мы имеем дело). На основании полученной информации подгрузить необходимые версии наших .NET библиотек и каким-то образом передать им управление.

Я бы рекомендовал вам поиграться с задачей такого рода: вжиться в код любого .NET процесса и вывести куда-либо сообщение об удаче внедрения (например, в файл лога)

Планирование потоков

Для того чтобы понимать, в каком порядке исполнять код различных потоков, необходима организация планирования этих потоков. Ведь система может иметь как одно ядро, так и несколько. Как иметь эмуляцию двух ядер на одном так и не иметь такой эмуляции. На каждом из ядер: железных или же эмулированных необходимо исполнять как один поток, так и несколько. В конце концов система может работать в режиме виртуализации: в облаке, в виртуальной машине, песочнице в рамках другой операционной системы. Поэтому мы в обязательном порядке рассмотрим планирование потоков Windows. Это — настолько важная часть материала по многопоточке, что без его понимания многопоточка не встанет на своё место в нашей голове никоим образом.

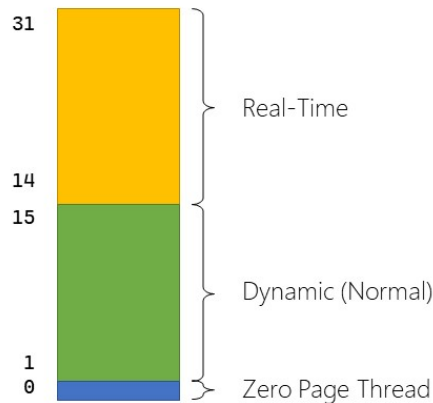
Итак, начнём. Организация планирования в операционной системе Windows является: гибридной. С одной стороны моделируются условия вытесняющей многозадачности, когда операционная система сама решает, когда и на основе каких условия вытеснить потоки. С другой стороны — кооперативной многозадачности, когда потоки сами решают, когда они всё сделали и можно переключаться на следующий (UMS планировщик). Режим вытесняющей многозадачности является приоритетным, т.к. решает, что будет исполняться на основе приоритетов. Почему так? Потому что у каждого потока есть свой приоритет и операционная система планирует к исполнению более приоритетные потоки. А вытесняющей потому, что если возникает более приоритетный поток, он вытесняет тот, который сейчас исполнялся. Однако во многих случаях это бы означало, что часть потоков никогда не доберется до исполнения. Поэтому в операционной системе есть много механизмов, позволяющих потокам, которым необходимо время на исполнение его получить несмотря на свой более низкий по сравнению с остальными, приоритет.

Уровни приоритета

Уровни приоритета

Windows имеет 32 уровня приоритета: от 0 до 31, где 31 — наивысший:

- 16 уровней — реального времени;
- 16 уровней — обычные, динамические приоритеты



CLRium #6

Windows имеет 32 уровня приоритета (0-31)

- 1 уровень (00 — 00) — это Zero Page Thread;
- 15 уровней (01 — 15) — обычные динамические приоритеты;
- 16 уровней (16 — 31) — реального времени.

Самый низкий приоритет имеет Zero Page Thread. Это — специальный поток операционной системы, который обнуляет страницы оперативной памяти, вычищая тем самым данные, которые там находились, но более не нужны, т.к. страница была освобождена. Необходимо это по одной простой причине: когда приложение освобождает память, оно может ненароком отдать кому-то чувствительные данные. Личные данные, пароли, что-то ещё. Поэтому как операционная система так и runtime языков программирования (а у нас — .NET CLR) обнуляют получаемые участки памяти. Если операционная система понимает, что заняться особо нечем: потоки либо стоят в блокировке в ожидании чего-либо либо нет потоков, которые исполняются, то она запускает самый низко приоритетный поток: поток обнуления памяти. Если она не доберется этим потоком до каких-либо участков, не страшно: их обнулят по требованию. Когда их запросят. Но если есть время, почему бы это не сделать заранее?

Продолжая говорить о том, что к нам не относится, стоит отметить приоритеты реального времени, которые когда-то давным-давно таковыми являлись, но быстро потеряли свой статус приоритетов реального времени и от этого статуса осталось лишь название. Другими словами, Real Time приоритеты на самом деле не являются таковыми. Они являются приоритетами с исключительно высоким значением приоритета. Т.е. если операционная система будет по какой-то причине повышать приоритет потока с приоритетом из динамической группы (об этом — позже, но, например, потому, что потоку освободили блокировку) и при этом значение до повышения было равно 15, то повысить приоритет операционная система не сможет: следующее значение равно 16, а оно — из диапазона реального времени. Туда повышать такими вот "твиками" нельзя.

Уровень приоритетов процессов с позиции Windows API.

Приоритеты — штука относительная. И чтобы нам всем было проще в них ориентироваться, были введены некие правила относительности расчетов: во-первых все потоки вообще (от всех приложений) равны для планировщика: планировщик не различает потоки это различных приложений или же одного и того же приложения. Далее, когда программист пишет свою программу, он задаёт приоритет для различных потоков, создавая тем самым модель многопоточности внутри своего приложения. Он прекрасно знает, почему там был выбран пониженный приоритет, а тут — обычный. Внутри приложения всё настроено. Далее, поскольку есть пользователь системы, он также может выстраивать приоритеты для приложений, которые запускаются на этой системе. Например, он может выбрать повышенный приоритет для какого-то расчетного сервиса, отдавая ему тем самым максимум ресурсов. Т.е. уровень приоритета можно задать и у процесса.

Однако, изменение уровня приоритета процесса не меняет *относительных приоритетов* внутри приложения: их значения сдвигаются, но не меняется внутренняя модель приоритетов: внутри по-прежнему будет поток с пониженным приоритетом и

поток — с обычным. Так, как этого хотел разработчик приложения. Как же это работает?

Существует 6 *классов приоритетов* процессов. Класс приоритетов процессов — это то, относительно чего будут создаваться приоритеты потоков. Все эти классы приоритетов можно увидеть в "Диспетчере задач", при изменении приоритета какого-либо процесса.

Название	Класс	Базовый приоритет
1. Real Time	4	24
2. High	3	13
3. Above Normal	6	10
4. Normal	2	8
5. Below Normal	5	6
6. Idle	1	4

Другими словами класс приоритета — это то, относительно чего будут задаваться приоритеты потоков внутри приложения. Чтобы задать точку отсчёта, было введено понятие *базового приоритета*. Базовый приоритет — это то **значение**, чем будет являться приоритет потока с типом приоритета **Normal**:

- Если процесс создаётся с *классом Normal* и внутри этого процесса создаётся поток с *приоритетом Normal*, то его **реальный приоритет Normal будет равен 8** (строка №4 в таблице);
- Если Вы создаёте процесс и у него *класс приоритета Above Normal*, то *базовый приоритет* будет равен 10. Это значит, что потоки внутри этого процесса будут создаваться с более повышенным приоритетом: **Normal будет равен 10**.

Для чего это необходимо? Вы как программисты знаете модель многопоточности, которая у вас присутствует. Потоков может быть много и вы решаете, что один поток должен быть фоновым, так как он производит вычисления и вам не столь важно, когда данные станут доступны: важно чтобы поток завершил вычисления (например поток обхода и анализа дерева). Поэтому, вы устанавливаете пониженный приоритет данного потока. Аналогично может сложиться ситуация когда необходимо запустить поток с повышенным приоритетом.

Представим, что ваше приложение запускает пользователь и он решает, что ваше приложение потребляет слишком много процессорных ресурсов. Пользователь считает, что ваше приложение не столь важное в системе, как какие-нибудь другие приложения и понижает приоритет вашего приложения до Below Normal. Это означает, что он задаёт базовый приоритет 6 относительно которого будут рассчитываться приоритеты потоков внутри вашего приложения. Но в системе общий приоритет упадёт. Как при этом меняются приоритеты потоков внутри приложения?

	Уровни насыщения
1. Time Critical	(+15)
2. Highest	(+2)
3. Above normal	(+1)
4. Normal	(+0)
5. Below normal	(-1)
6. Lowest	(-2)
7. Idle	(-15)

Таблица 3

Normal остаётся на уровне +0 относительно уровня базового приоритета процесса. Below normal — это (-1) относительно уровня базового. Т.е. в нашем примере с понижением уровня приоритета процесса до класса Below Normal приоритет потока 'Below Normal' пересчитывается и будет не $8 - 1 = 7$ (каким он был при классе Normal), а $6 - 1 = 5$. Lowest (-2) станет равным 4.

Idle и Time Critical — это уровни насыщения (-15 и +15). Почему Normal — это 0 и относительно него всего два шага: -2, -1, +1 и +2? Легко провести параллель с обучением. Мы ходим в школу, получаем оценки наших знаний (5,4,3,2,1) и нам понятно, что это за оценки: 5 — молодец, 4 — хорошо, 3 — вообще не постарался, 2 — это не делал ни чего, а 1 — это то, что можно исправить потом на 4. Но если у нас вводится 10-ти балльная система оценок (или что вообще ужас — 100-балльная), то возникает неясность: что такое 9 баллов или 7? Как понять, что вам поставили 3 или 4?

То же самое и с приоритетами. У нас есть Normal. Дальше, относительно Normal у нас есть чуть повыше Normal (Normal above), чуть пониже Normal (Normal below). Также есть шаг на два вверх или на два вниз (Highest и Lowest). Нам, поверьте, нет никакой необходимости в более подробной градации. Единственное, очень редко, может раз в жизни, нам понадобится сказать: выше чем любой приоритет в системе. Тогда мы выставяем уровень Time Critical. Либо наоборот: это надо делать, когда во всей системе делать нечего. Тогда мы выставяем уровень Idle. Это значения — так называемые уровни насыщения.

Как рассчитываются уровни приоритета?

Уровни приоритета

1. Берётся класс приоритета процесса
2. Класс преобразуется в базовый приоритет
3. Уровень приоритета потока = установленный уровень + базовый приоритет процесса

Класс приоритета / относительный приоритет	Real-Time	High	Above-Normal	Normal	Below-Normal	Idle
Time Critical (+насыщение)	31	15	15	15	15	15
Highest (+2)	26	15	12	10	8	6
Above Normal (+1)	25	14	11	9	7	5
Normal (0)	24	13	10	8	6	4
Below Normal (-1)	23	12	9	7	5	3
Lowest (-2)	22	11	8	6	4	2
Idle (-насыщение)	16	1	1	1	1	1

19

 CLRium #6

У нас бал класс приоритета процесса Normal (Таблица 3) и приоритет потоков Normal — это 8. Если процесс Above Normal то поток Normal получается равен 9. Если же процесс выставлен в Highest, то поток Normal получается равен 10.

Поскольку для планировщика потоков Windows все потоки процессов равнозначны, то:

- Для процесса класса Normal и потока Above-Normal
- Для процесса класса Highest и потока Normal
конечные приоритеты будут одинаковыми и равны 10.

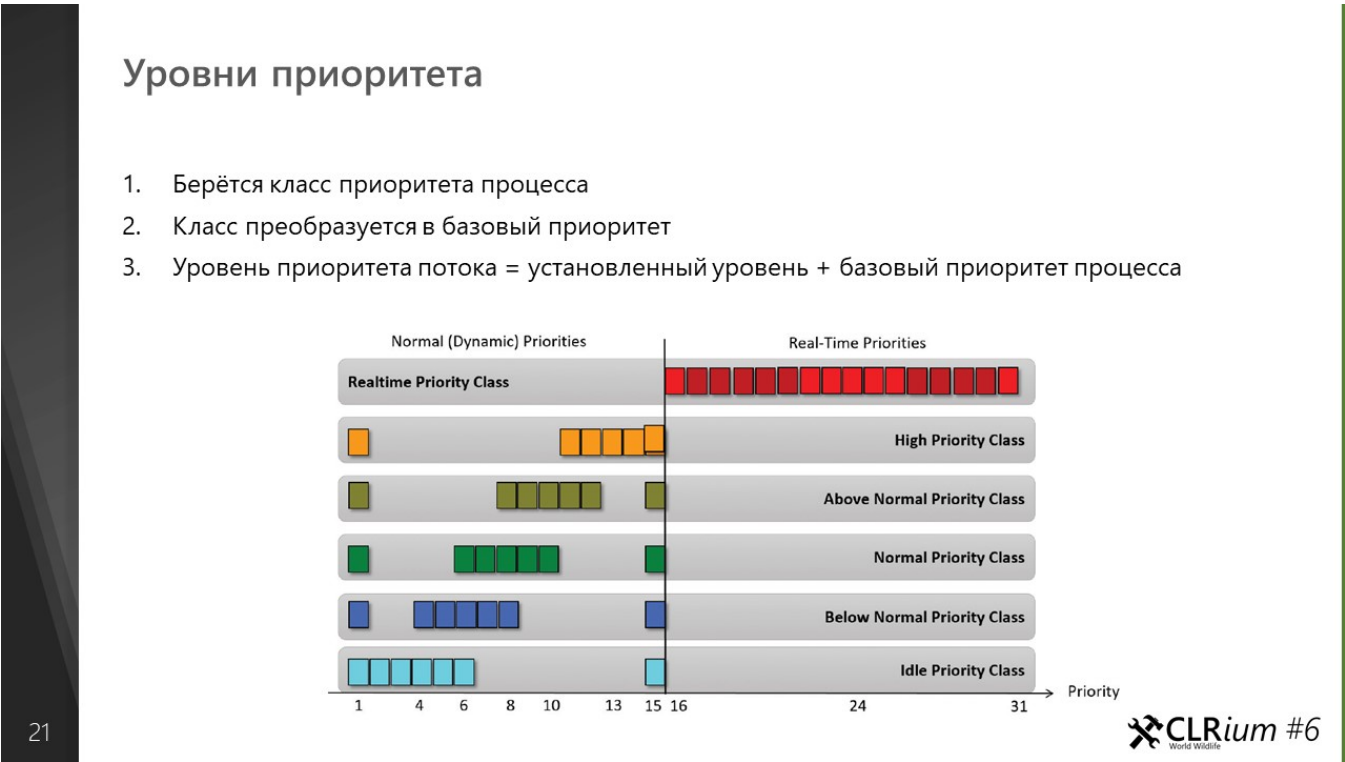
Если мы имеем два процесса: один с приоритетом Normal, а второй — с приоритетом Highest, но при этом первый имел поток Highest а второй Normal, то система их приоритеты будет рассматривать как одинаковые.

Как уже обсуждалось, группа приоритетов Real-Time на самом деле не является таковой, поскольку настоящий Real-Time — это гарантированная доставка сообщения за определённое время либо обработка его получения. Т.е., другими словами, если на конкретном ядре есть такой поток, других там быть не должно. Однако это ведь не так: система может решить, что низко приоритетный поток давно не работал и дать ему время, отключив real-time. Вернее его назвать классом приоритетов который работает над обычными приоритетами и куда обычные приоритеты не могут уйти, попав под ситуации, когда Windows временно повышает им приоритет.

Но так как поток повышенным приоритетом выполняется только один на группе ядер, то получается, что если у вас даже Real-Time потоки, не факт, что им будет выделено время.

Класс приоритета / относительный приоритет	Real-Time	High	Above-Normal	Normal	Below-Normal	Idle
Time Critical (+насыщение)	31	15	15	15	15	15
Highest (+2)	26	15	12	10	8	6
Above Normal (+1)	25	14	11	9	7	5
Normal (0)	24	13	10	8	6	4
Below Normal (-1)	23	12	9	7	5	3
Lowest (-2)	22	11	8	6	4	2
Idle (-насыщение)	16	1	1	1	1	1

Если перевести в графический вид, то можно заметить, что классы приоритетов пересекаются. Например, существует пересечение Above-Normal Normal Below-Normal (столбик с квадратиками):



Класс приоритета / относительный приоритет	Real-Time	High	Above-Normal	Normal	Below-Normal	Idle
Above Normal (+1)	25	14	11	9	7	5
Normal (0)	24	13	10	8	6	4
Below Normal (-1)	23	12	9	7	5	3
Lowest (-2)	22	11	8	6	4	2
Idle (-насыщение)	16	1	1	1	1	1

Кстати говоря, мы стартовали продажи на CLIRium #7, в котором мы с огромным удовольствием будем говорить про практику работы с многопоточным кодом. Будут и домашние задания и даже возможность работы с личным ментором.

Загляните к нам на сайт: мы сильно постарались, чтобы его было интересно изучить.

Теги: windows, планировщик потоков, поток

Хабы: Блог компании Семинары Станислава Сидристого, Программирование, .NET, Параллельное программирование

↑

+19

↓

🔖

108

👁


11k

💬

20


➦

Поделиться



Семинары Станислава Сидристого

CLIRium #7: Parallel Computing Practice



142,5

0,0

Карма

Рейтинг

Stanislav Sidristij @sidristij

Семинары по платформе .NET CLIRium

ПОХОЖИЕ ПУБЛИКАЦИИ

22 января 2018 в 23:33

[DotNetBook] Стек потока. Его редактирование и клонирование потока

↑

+30

👁

12,3k

🔖

93

💬

22

2 октября 2014 в 12:12

Ручное клонирование потока. Когда Assembler + C# = Love

↑

+29

👁

18,7k


🔖

142

💬

32

Комментарии 20



AKudinov

17 февраля 2020 в 04:18

🔖

🔖

↑

+2

↓

Спешу возмутиться спорному утверждению автора, что поток — эмуляция параллелизма. Никакая он не эмуляция, и при наличии возможности (несколько ядер и/или процессоров) планировщик задач ОС запускает потоки действительно параллельно. В данном случае у нас есть не один, а несколько (по числу ядер) потоков команд, идущих параллельно, в них-то планировщик всё и раскладывает.



sidristij

17 февраля 2020 в 07:48

🔖

🔖

🔖

🔖

↑

0

↓

если у Вас на машине потоков больше чем ядер (а в современности это практически всегда), расскажите, как несколько потоков будут делить одно ядро без эмуляции параллелизма



AKudinov 17 февраля 2020 в 07:57



↑ 0 ↓

Давайте посмотрим внимательнее. Предположим, имеем на процессоре два ядра. Соответственно, планировщик ОС имеет две очереди на исполнение. И каждый тайм-слот на наших двух ядрах параллельно (совершенно честно!) выполняются две задачи. Предположим, что наше приложение создаёт два потока. Ничто не мешает планировщику запланировать выполнение этих двух потоков на двух ядрах совершенно честно параллельно.

А само приложение никогда не знает, исполняется ли оно честно параллельно, или параллельность «эмулируется». И не имеет права знать: сегодня его на одном ядре запустили, и потенциальные гонки, которые проглядел программист, просто не случились. А завтра запустили на 32 ядрах, все его потоки заработали действительно параллельно, и всё сломалось.



sidristij 17 февраля 2020 в 08:39



↑ 0 ↓

Я вас понял. Но в текущей ситуации все потоки эмулируют параллелизм относительно каких-то других. Нынче почти нет вероятности получения непрерывного отрезка времени, большего чем квант. Активных потоков на одном ядре всегда больше одного. Даже если поток_1 и поток_2 находятся на разных ядрах, вероятность их чисто параллельного исполнения крайне мала: помимо них на ядрах есть другие и попасть в один квант — скорее большая удача (особенно если учесть постоянное динамическое повышение приоритетов и динамическое изменение длины квантов на десктопе и как следствие — изменение этого порядка). От слов покаотказываться не буду: они покрывают 99.99% случаев.



AKudinov 17 февраля 2020 в 09:44



↑ 0 ↓

Зато создание потоков — отличный способ получить заметный прирост производительности на современном процессоре. Конечно, "заметность" зависит от свойств самой задачи, от того, сколько в ней параллелизма, вообще, есть.



Sm1le291 17 февраля 2020 в 17:29



↑ 0 ↓

опять таки относительно, если глобально говорить то да. Если в терминах дот нет, то лучше взять готовый поток из ThreadPool



AKudinov 17 февраля 2020 в 04:25



↑ +1 ↓

И далее про HyperThreading. "Распараллеливание" идёт за счёт того, что планировщик процессора (есть там такой блок) может запускать на исполнение две команды из соседних виртуальных "ядер", если эти команды задействуют разные исполнительные блоки процессора. Поэтому в данном случае, опять же, исполнение будет действительно параллельным. Но более медленным, чем на честных ядрах, как раз из-за блокировок. Но не доступа к памяти (тут-то вся ответственность как раз-таки ложится на программиста), а доступа к исполнительным устройствам процессора (АЛУ и т.п.)



sidristij 17 февраля 2020 в 07:59



↑ 0 ↓

Большинство команд, применяемых при компиляции задействуют 1-2 такта. Процессор не может отработать 1,5 такта. Поэтому даже если замедление и небольшое, для большинства команд оно будет "в 2 раза". Это — раз. Второе — повествование идёт для .NET разработчиков. У нас нет понимания, на каком ядре запущен код. Поэтому нет возможности убедиться, что код работает с одними данными, но, например, не на одном физическом ядре. Этого можно добиться, создав потоки друг за другом, да. Но уверенности это знание не даёт.



Temtaime 17 февраля 2020 в 09:09



↑ 0 ↓

Процессор давно стал суперскалярным.
Он может выполнить 2-3 команды за 1 такт.



sidristij 17 февраля 2020 в 09:14



↑ 0 ↓

Это не имеет отношения к замедлению отдельных команд.



sidristij 17 февраля 2020 в 09:16



↑ 0 ↓

Плюс не ясно, о чем спор: в тексте шла речь про наихудший сценарий

**hedger** 17 февраля 2020 в 10:02

↑ +2 ↓

Про Hyper-Threading абзац во многом неверный. HT подразумевает одно АЛУ на 2 виртуальных ядра, и в случае ошибок предсказания ветвления и прочего, что вызвало бы сброс конвейера и ожидание его заполнения, происходит переключение на второй набор регистров и исполнение инструкций с соседнего "ядра". Так что потоки на HT ядре работают по очереди не "в худшем случае", а всегда.

Далее,

Организация планирования в ... Windows является: приоритетной и вытесняющей.... А вытесняющей потому, что если возникает более приоритетный поток, он вытесняет тот, который сейчас исполнялся.

Это вообще неверно. Есть понятия вытесняющей и кооперативной многозадачности. В случае кооперативной, процессы (потоки) сами решают, когда отдать управление планировщику ОС для дальнейшей передачи другому процессу. (И если они плохо написаны или виснут, то могут не передать вовсе). А при вытесняющей многозадачности, планировщиком полностью рулит ОС, отбирая процессор у программ без их участия и согласия, "вытесняя" их. И возникновение новых потоков и их приоритеты с принципом работы такой многозадачности не связаны.

**sidristij** 17 февраля 2020 в 10:06

↑ 0 ↓

Благодарю за конструктивную критику, отправил Вам приглашение на Хабр :)

По первому вопросу — спасибо. Я не нашёл (может не там искал) хорошей статьи по HT. Тема, как говорится, не раскрыта. Дадите источник, с удовольствием дополню абзац.

По второму — по кооперативной и вытесняющей уточнил в статье: возможны оба варианта.

**maxbl4** 17 февраля 2020 в 14:01

↑ 0 ↓

Очень длинно и подробно описаны приоритеты потоков, но совершенно не раскрыто зачем это всё нужно знать программисту на C#. Точнее, я ещё хоть как-то могу понять зачем внутри своей программы создавать потоки с разными приоритетами, но зачем знать их приоритет с потоками других процессов ума не приложу. Более того, мне действительно сложно представить ситуацию где практически будет полезно создавать потоки и управлять их приоритетом в современном C# приложении. Было бы очень полезно, чтобы цикл статей начинался именно с обоснования зачем это нужно. Иначе не ясно, почему бы не использовать Task.Start везде и не думать о потоках

**sidristij** 17 февраля 2020 в 15:21

↑ 0 ↓

Task.Start() относится к параллельной вселенной. Вы можете создать свой пул потоков, объединить их в SynchronizationContext, на основе которого создать TaskScheduler и все задачи, зааасайненные на него начнут крутиться в вашем пуле. Но мысль понял, опишу :)

**maxbl4** 17 февраля 2020 в 15:54

↑ 0 ↓

Хочется увидеть описание реального проекта, где ручное создание потока с нестандартным приоритетом решило проблему, по сравнению с простым запуском того же потока через Task.Start

**sidristij** 17 февраля 2020 в 18:00

↑ 0 ↓

Ну, например, необходимо подгрузить и подготовить данные в кэше, пока приложение занимается чем-то другим. То, что приходит в голову на основе всем известных продуктов — есть вот IDE, и она предназначена для C# разработки. И если в ней есть JS код, IDE может также его анализировать, строить AST и прочие вопросы, но фоном: когда нет других задач. На пониженном приоритете.

**alex_zzzz** 18 февраля 2020 в 13:18

↑ 0 ↓

Из того, что мы не имеем в .NET явно — это структура безопасности с атрибутами безопасности и размер стека.

Размер стека задаётся в параметрах конструктора потока.

**a-tk** 25 февраля 2020 в 08:17

↑ 0 ↓

Краткий (на самом деле не очень) пересказ главы из книги Windows Internal получился неплохим.



bvbr 18 марта 2020 в 00:01



0



> Именно поэтому раздача ядер идёт $+2$ в случае Hyper-Threading. Т.е. пропуская парные ядра.

Это уже давно немного не так.

> Технология эта — достаточно спорная

Это тоже давно не так, спорной она была в своих первых (не самых удачных) реализациях, сейчас же это отличный способ получить больше ядер и повысить общую производительность системы.

> которые при этом находятся на одном физическом ядре и работают с одной и той-же памятью, то вы будете постоянно попадать в ситуацию, когда второй логический процессор так же пытается обратиться к данной памяти, создавая блокировку либо попадая в блокировку, т.к. поток, находящийся на первом ядре работает с той же памятью.

Это тоже, проблема с доступом к шареной между потоками/процессами памяти, возникает независимо от того, является ядро логическим (от Hyper-Threading/SMT) или физически вторым (третьим, четвертым и т.д.), к разделению ресурсов между логическими ядрами на одном физическом это практически не имеет отношения.

Современные процессоры давно умеют выполнять множество операций за 1 такт, изменять порядок исполнения инструкций, предсказывать ветвления и пытается дальше исполнять код не дожидаясь реального срабатывания перехода (угадали – хорошо часть инструкций за переходом уже выполнена, не угадали – просто отбрасываем результаты)

Но хорошо это работает только на сильно оптимизированном специфическом коде, который равномерно нагружает все доступные блоки выполнения.

«Среднестатистический» код, даже после оптимизирующих компиляторов редко когда может равномерно загрузить все доступные ресурсы.

Если прибавить к этому огромные (в тактах) задержки если данные (даже не шаренные) лежат дальше, чем в L1 кэше то получается, что большинство блоков исполнения просто простаивают большую часть тактов.

Отсюда и родилась идея добавить еще немного элементов и показать для ОС второй процессор, которые может в это время исполнять код на свободных блоках. У процессоров SPARC емнип были вообще модели, где на одном физическом ядре было 4 или даже 8 логических.

Теоретически серьезная конкуренция между логическими процессорами за ресурсы, и просадка производительности может быть, когда оба логических процессора исполняют очень сильно оптимизированный код и все необходимые данные лежат в кэше, но на практике такое возникает достаточно редко.

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки

Неделя

Месяц

320 ГБ вместо 6 ТБ: как могут невинно облапошить в онлайн-магазинах

↑ +187 👁 57,2k 📖 60 💬 216

Установка Windows 98 на современный ПК

↑ +70 👁 27,7k 📖 82 💬 48

Индийский инженер в иске к Cisco утверждает, что 90% IT-работников из Индии в США принадлежат к высшей касте

↑ +65 👁 30,1k 📖 13 💬 192

«Некоторые более равны, чем другие». Удаленные сотрудники из Кремниевой долины сталкиваются с большим урезанием зарплаты

+45

31,5k

37

204

Фриланс с соцпакетом – дело недалёкого будущего. Или нет?

Мегапост

Ваш аккаунт	Разделы	Информация	Услуги
Войти	Публикации	Устройство сайта	Реклама
Регистрация	Новости	Для авторов	Тарифы
	Хабы	Для компаний	Контент
	Компании	Документы	Семинары
	Пользователи	Соглашение	Мегaproекты
	Песочница	Конфиденциальность	Мерч