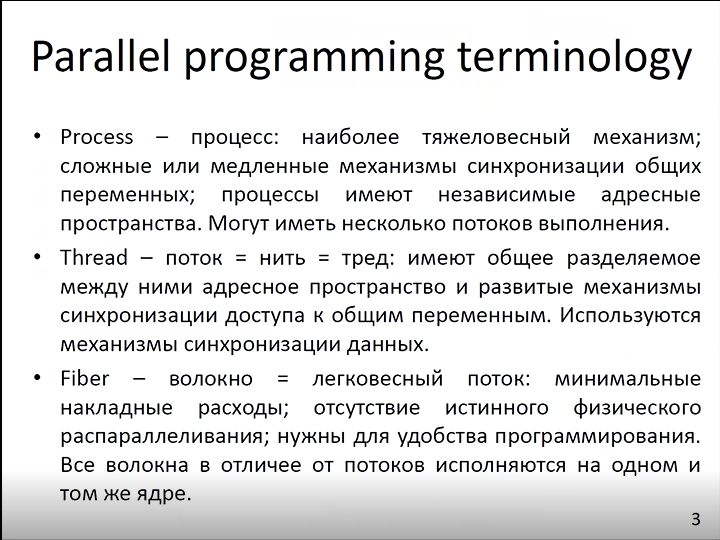


Типовой сценарий работы программиста примерно вот такой. Вы пишете однопоточную программу. Написали программу, запустили ее, посмотрели ее результаты, остановились (вот это самый важный шаг) и подумали. И смотрите – вас устраивает, как она работает, или нет? Представьте – ваша программа работает за одну секунду. Есть смысл ее распараллеливать? Вы потратите неделю, а она будет работать полсекунды. Нет в этом никакого смысла, поэтому очень внимательно думайте. Если же она работает много – пробуйте автоматическое распараллеливание. Запустили, посмотрели результат – если не работает, подумали опять, пробуем специальные параллельные библиотеки, специальные структуры данных, проверяем. Если не работает – дальше запускаем профилировщик, смотрим вообще максимальный достижимый параллельный эффект (это специальная программа для того, чтобы проверить распараллеливание), считаете теоретически по закону Амдала, закону Густавсона-Барсиса, параллельную эффективность, параллельное ускорение; посчитали и подумали – а вообще возможно здесь что-то получить или невозможно. То есть вам нужно подумать вообще в задаче, нужно ли вам параллельное программирование. И в конце, если есть какие-то узкие места, вот тогда вы там уже разбираетесь дальше, это следующий шаг. Вот видите, цель – избежать параллельного программирования, это шутка, но это очень важно.

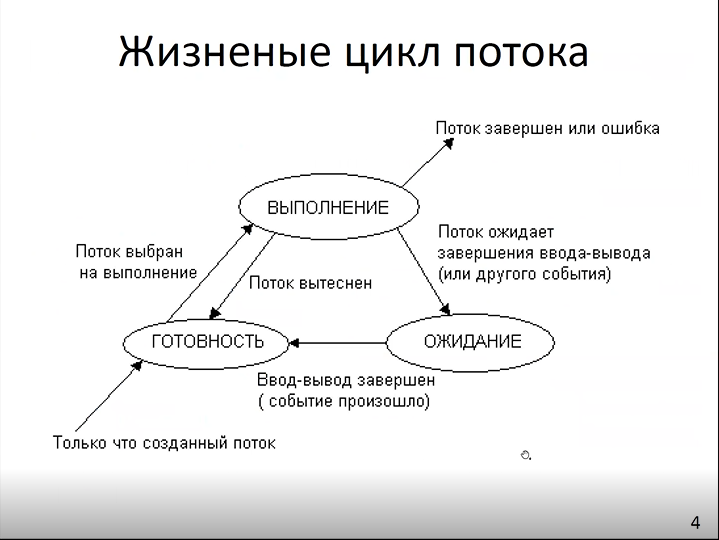


В чем разница между параллельным и многопоточным программированием? Чтобы ответить на этот вопрос, нужно разобрать некоторые вещи. У нас в терминологии параллельного программирования существует три определения, три сущности. Первое – это процесс. Это наиболее тяжеловесный механизм, имеет сложные и медленные механизмы синхронизации общих переменных, независимые адресные пространства; может быть несколько потоков выполнения и так далее. Все вы когда-то с виндой сталкивались – запускаете диспетчер задач, у вас там процессы. Вот вы видите процессы, которые происходят – калькулятор, браузер, почта, вот все это у вас процессы.

Второе – это поток (thread). Они имеют общее разделяемое адресное пространство и развитые механизмы синхронизации доступа, используют механизмы синхронизации данных. То есть это та штука, которая непосредственно обрабатывает данные, которая все вычисляет.

И третье – это волокна. Это легковесные потоки, у которых минимальные накладные расходы, они на самом деле не распараллеливаются (такое псевдораспараллеливание), в отличие от потоков – всегда на одном и том же ядре, только потоки могут скакать между ядрами. В рамках данного курса мы это рассматривать не будем. Вам нужно только понимать, что в процессах независимое адресное пространства, а в потоках общее адресное пространство.

Рассмотрим такой пример. Если у нас есть два каких-то процесса, то каждый из них имеет отдельный сегмент данных, сегмент кода, сегмент стека, вот они вообще никак не пересекаются, то есть в памяти это все отдельные сущности. Второй вариант – если у нас только один процесс работает. В рамках этого процесса можно создать несколько потоков, и у них сегмент кода, сегмент данных – они одни, а вот сегмент стека переменной – разные. И получается, что, если один поток изменил какую-нибудь переменную в рамках одного процесса, второй поток может автоматически использовать его значение. То есть потоки – это более глубокая вещь, нежели процессы. Переключение контекста между потоками в одном и том же процессе происходит обычно быстрее, чем переключение контекста между процессами, в качестве аналогии потоки выполнения можно сравнить с несколькими поварами. Представьте, повара работают вместе, все они готовят одно и то же блюдо, читают одну и ту же кулинарную книгу с одним и тем же рецептом и следуют ее инструкциям. Но не обязательно, что все они читают одну и ту же страницу. То есть взяли какое-то большое-большое, длинное блюдо, которое расписано на 3 страницы, и вот продукты из холодильника одни и те же, готовим одно и то же, рецепт один и тот же, просто один поток может быть на первой странице (повар чуть позже начал), а второй начал чуть раньше и вот между собой у них вот такая разница.



Важный нюанс про жизненный цикл потока, чтобы вы понимали и дальше с ними взаимодействовали нормально.

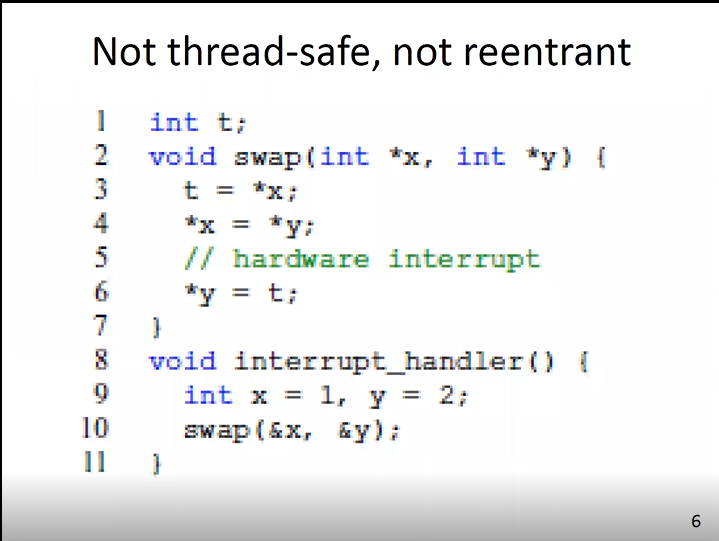
Вот когда создается поток (ОС создает), он переходит в состояние «Готовность», готовность к тому, чтобы быть выполненным, и дальше ОС может взять его на выполнение, допустим, он выполнился или завершился с ошибкой. Если же какой-то другой поток важнее, а этот еще особо ничего не сделал, то он может быть вытеснен. Так же есть состояние, когда поток ожидает завершения ввода-вывода или другого события, то есть поток зависит от другого, тогда у нас еще добавляется фаза «Ожидание». Основной переход – между стадиями «Готовность» и «Выполнение».

Рассмотрим пример. Предположим, что вы создаете программу, которая создает 10 потоков и запускает их на 10-ядерном компьютере. Позже вы должны выполнить эту программу на более плохой машине, в которой всего 2 ядра. ОС выставит на выполнение 2 потока, они будут выполняться, так как у нее доступно всего 2 ядра. Остальные 8 потоков будут переведены в состояние «Готовы» и будут ожидать смещения потоков, вытеснения. Вы не будете знать, какой поток будет выбран для исполнения, ОС делает это автоматически. Что не так с этим подходом, в чем здесь сложности? Только сама ОС в конкретный момент времени знает, какой поток она возьмет, а какой нет. Поскольку у нас было разбиение на 10 потоков, первые 2 выполнятся, а в какой последовательности будут выполняться остальные 8, мы не знаем, и бывает такая ситуация, от того, как написана программа, эти потоки могут друг другу мешать, иметь какую-то зависимость или еще что-то. То есть это получается проблема масштабируемости – мы взяли программу для 10-ядерного компьютера, перенесли ее на несколько 2хядерных и можем получить в итоге неправильный результат выполнения программы, и это очень плохо.



Рассмотрим следующий вопрос. Существуют 2 концепции, которые называются потокобезопасность и реинтерабельность (позволяет сделать повторный вход). Существует непонимание, поскольку разработчики разного рода придумывали разные функции (обозначали их), то по-разному это все называется. Нам надо понять, что мы будем использовать linux-base концепт, и вообще понимать, что у нас есть.

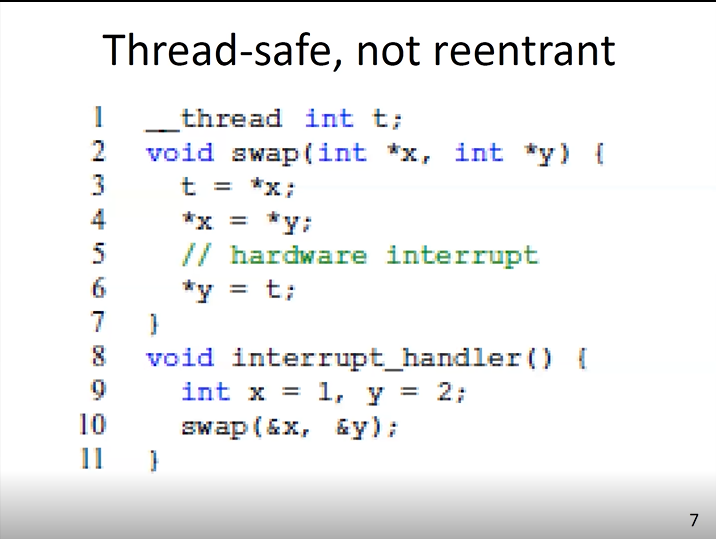
Итак, Thread-safe – функция показывает правильные результаты, даже если вызвана одновременно несколькими тредами. То есть такая функция, которая всегда все правильно выполняет. Функция реинтерабельная – функция показывает правильный результат, если она повторно вызвана внутри себя. Другие определения похожи, но есть некоторые нюансы, из-за которых можно не переживать.



Давайте с вами рассмотрим несколько примеров и поймем, когда у нас есть проблемы, когда у нас нет проблем.

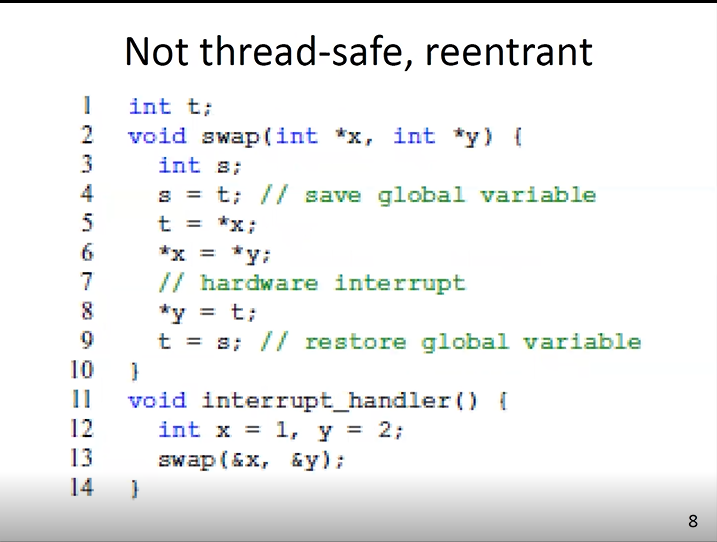
У нас есть некоторая программа, которая делает swap двух чисел x и y между собой. Поскольку мы не используем специальную функцию, если мы меняем две переменные местами, мы используем третью переменную – это стандартный подход. Таким образом, мы t присваиваем указатель на х, х меняем с y местами и возвращаем указатель на y, вроде бы у нас все в порядке. Но теперь у нас происходит следующая ситуация – где-то в строке 5 срабатывает обработчик прерываний, и он вызывает эту функцию swap еще раз. И обработчик прерываний говорит – у меня есть x1, y2, мне нужно их поменять местами. Вызывается еще раз эта функция swap, несмотря на то, что мы в ней до этого находились. Соответственно, глобальная переменная t, когда она будет вызвана во второй раз вторым swap-ом, она будет переписана новым значением х. И в результате, когда мы захотим вернуться из этой функции, начальная функция будет содержать в себе значение 1 вместо х, но это не то, что мы хотим.

Если мы вызовем ее из разных потоков, у нас будут разные значения, и поэтому она у нас не потокобезопасная. Второе – мы не можем вызвать нормальный swap из себя, потому что тоже везде будут кривые результаты, поэтому этот пример непотокобезопасный и не реинтерабельный.

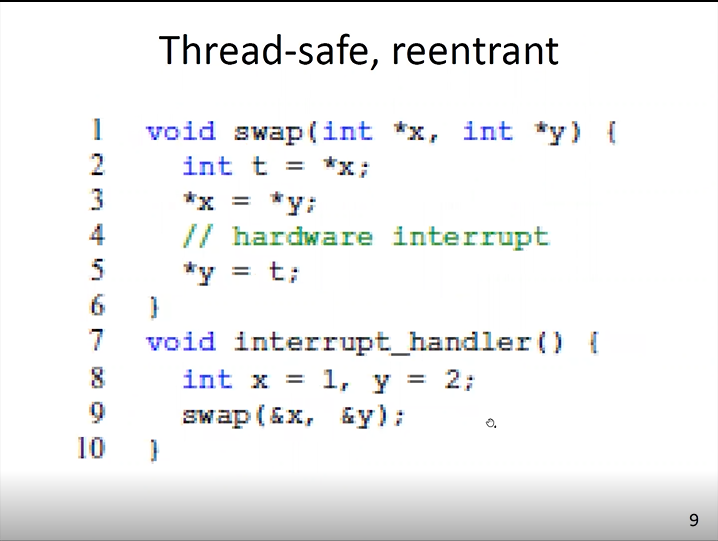


Для того, чтобы в gcc-компиляторе нам было легче, мы помечаем переменную, что она потокобезопасна (\_\_thread), и мы говорим, что она потокобезопасна. В других языках и компиляторах есть похожие решения. И теперь, если в каждом потоке мы вызываем t, то у нас в этом плане все будет в порядке. Но при этом у нас по-прежнему существует рабочее прерывание и swap у нас будет все портить, то есть в рамках одного потока мы все испортим. То есть она не реинтерабельна, но между собой потки не будут мешаться.

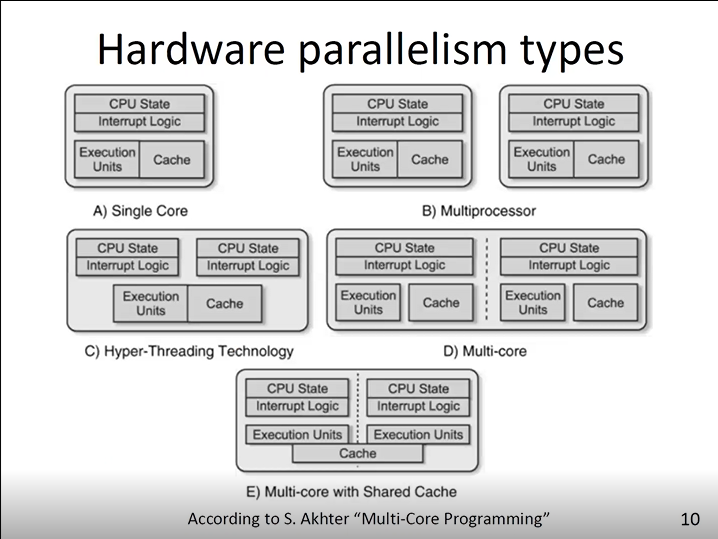
Итак, представьте, что у нас есть 5ядерный компьютер и мы запускаем эту программу. Соответственно, у нас создастся 5 копий переменной t, потому что потокобезопасно. Ну и вопрос – сколько потоков создаст компьютер в результате выполнения этой программы, если у нас 16ядерный компьютер? Тоже 16. То есть никаких сюрпризов здесь нет.



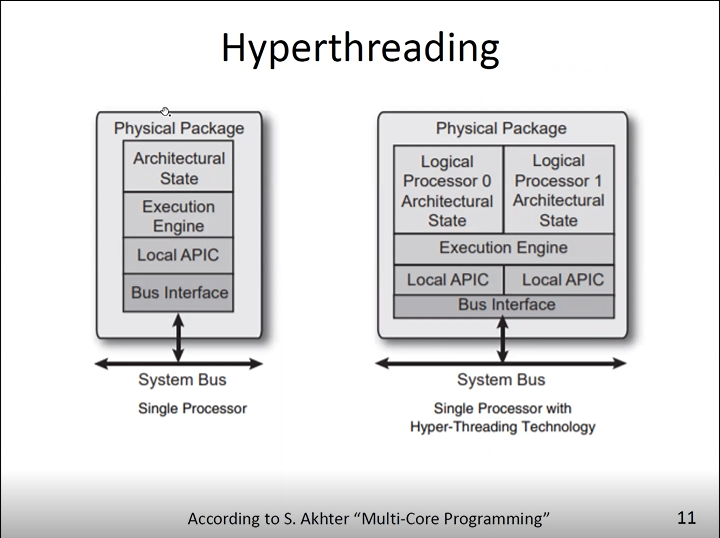
Следующий пример, который не потокобезопасный, но реинтерабельный. Внутри функции swap мы делаем локальную переменную s, она сохраняет значение глобальной переменной t при перезапуске. Когда мы сделаем еще один вызов, у нас в 9-ой строчке есть возврат значения глобальной переменной. Эта функция у нас становится реинтерабельной, мы можем ее вызвать из самой себя, но она не потокобезопасна.



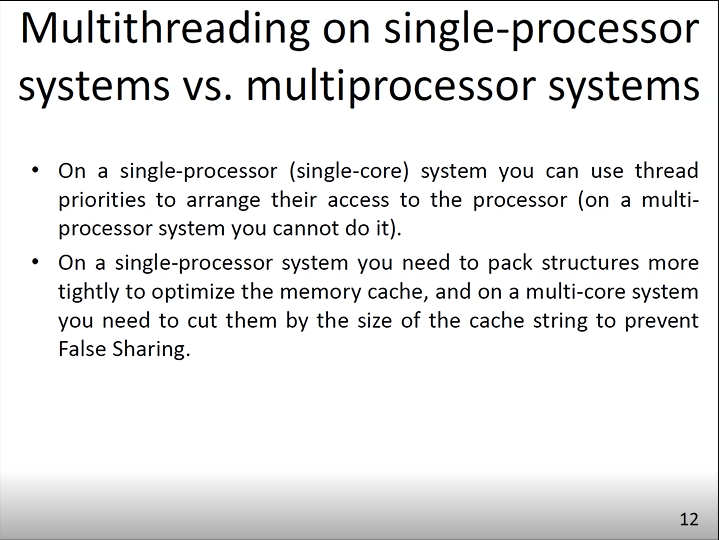
И если же мы хотим писать нормальный потокобезопасный код без глобальных переменных, мы сделаем переменную t локальной, объявляем ее внутри функции, и тем самым никаких глобальных переменных нет, только локальные переменные, и вот эта функция у нас и потокобесопасна, и реентерабельна. Этот пример функции может казаться немного странным, но ваша задача, чтобы, когда вы писали, вспоминали о вот этих слайдах, что нужно, чтобы у вас была потокобезопасная и реентерабельная функция.



На этом слайде изображены аппаратные особенности различных параллельных вычислителей. Единственный вычислитель – у него одно ядро, свой кэш, свои регистры. Если вы ставите несколько процессоров в одну и ту же машину, в материнскую плату в два разных сокета – у вас все дублируется. Есть гипертрединг, его отличие в том, что вся аппаратура у него общая. У многоядерного же вычислителя у каждого ядра свои состояния процессора, его прерывания, а также свой кэш. Есть разные уровни кэша, но третий уровень будет уже для всех общий, второй уровень зависит от реализации (AMD или Intel).

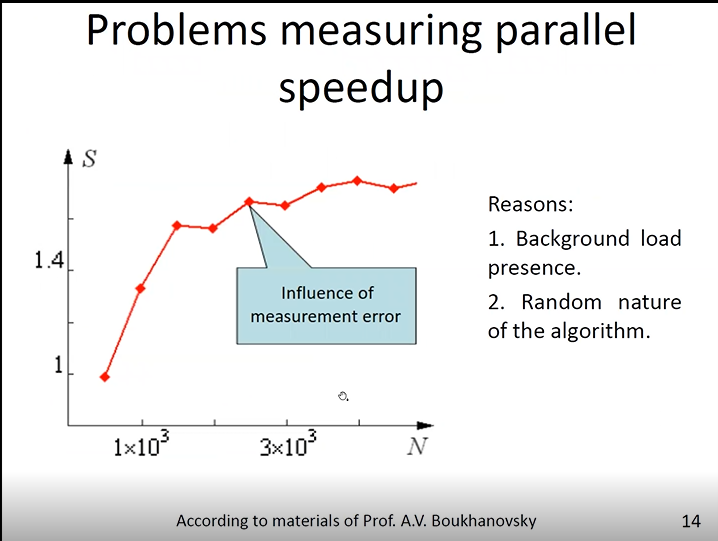


Немного подробнее про гиперпоточность. Даже по заявлениям Intel, которая придумала эту технологию, и которая ее всячески рекламирует, она дает прирост производительности не более 30%. Что это значит? У вас как будто 2 гиперядра, но в идеальном случае у вас всего 30% будет прирост, во всех остальных случаях может остаться, как было, а может ухудшиться из-за неверного представления о количестве вычислителей. Ваша программа запрашивает у ОС количество процессоров (ядер), и гипертрединговый процессор заявляет, что у него их в 2 раза больше, чем есть на самом деле. Вы послушно создаете нужное количество потоков и видите, что потоки простаивают, более того, тратится время на переключение между ними. Это все лишние накладные расходы, из-за которых ваша программа может даже начать работать хуже.



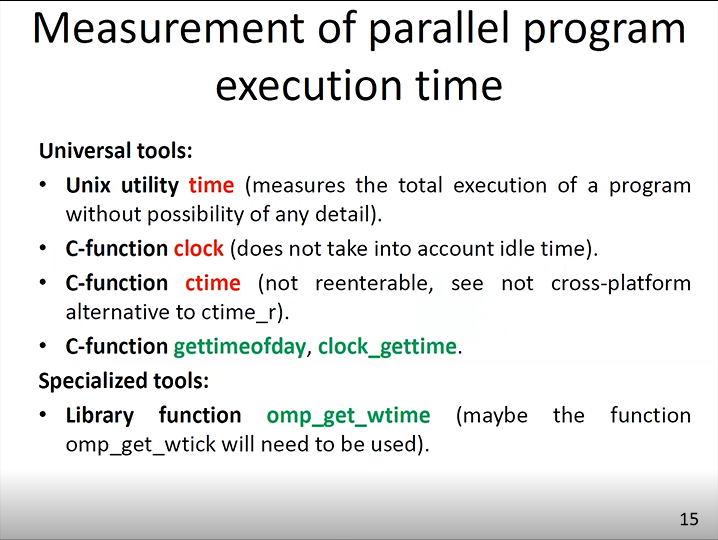
Отсюда важнее всего запомнить, что идея потоков не в том, чтобы упорядочить выполнение, а в том, чтобы делить исполняемый ресурс в необходимой пропорции, чтобы максимально использовать время, которое есть.

Раньше, когда использовались однопроцессорные машины, писались программы, которые concurrent, когда использовались несколько потоков с разделением времени, при этом использовалась приоритизация потоков, и, если в списке на выполнение имелся поток с более высоким приоритетом, он выполнялся раньше. ОС четко выполняла указания программиста, сначала выполняя высокоприоритетные задачи, а затем низкоприоритетные. Но если сейчас такую программу запустить на современном многоядерном вычислителе, у которого реально несколько процессоров, у вас все ломается. Почему? Потому что принцип приоритета сломался. ОС поставила высокоприоритетный тред сразу на одно из ядер, а дальше оказывается, что у нее еще, предположим, 8 ядер простаивает. Она возьмет низкоприоритетные треды и поставит на эти пустующие ядра, потому что немного поменялся смысл приоритетов. Приоритеты влияют теперь только на очередность выполнения процессов, но, если есть пустые ядра, которые не занимают высокоприоритетные процессы, почему бы не выполнить на них низкоприоритетные задачи? В итоге получается, что низкоприоритетный поток начал выполняться раньше, чем было запланировано, ведь изначально задумывалось, что сначала выполнится высокоприоритетный поток, и только потом, когда он сгенерирует нужные данные, выполнится уже низкоприоритетный. А тут низкоприоритетный влез раньше времени и начал использовать данные, которых еще нет, и все, ваша программа сломалась. Эта проблема до сих пор еще существует в старых программах.



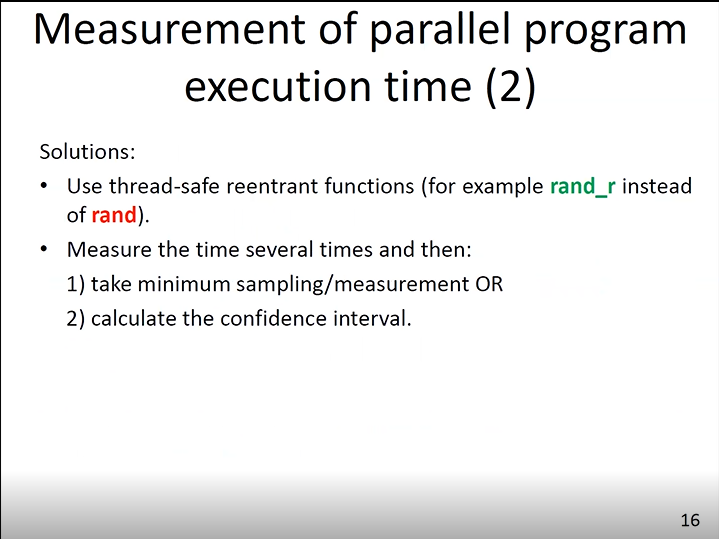
Помимо того, что вы делаете различные действия с оценкой параллельного ускорения, здесь есть еще две проблемы.

Измерение времени выполнения программы на Си не представляет собой сложной задачи само по себе, но в параллельном программировании есть некоторая специфика. Не все функции, которые подходят для измерения времени последовательной программы, работают для многопоточных программ (т.е. если больше, чем 1 поток; либо параллелизм на уровне процессов, либо параллелизм на уровне потоков). Если мы будем делать измерения, можно увидеть, что график у нас скачет. Понятно, что мы строим его по точкам, это не идеальная прямая. Мы меряем параллельное ускорение. Из-за чего может быть такой вид графика? Первое – это некоторая загрузка бэкграунда. Самый простой пример – обновление операционной системы, даже, может быть, не оно само, а таймер, который говорит: «О, мне нужно что-то сделать», например, проверить, нет ли там какого обновления или еще чего-то, то есть не само обновление, а проверка обновления. И некоторые алгоритмы, которые вообще в логике могут иметь случайное ветвление, и вы не можете нормально померить время.



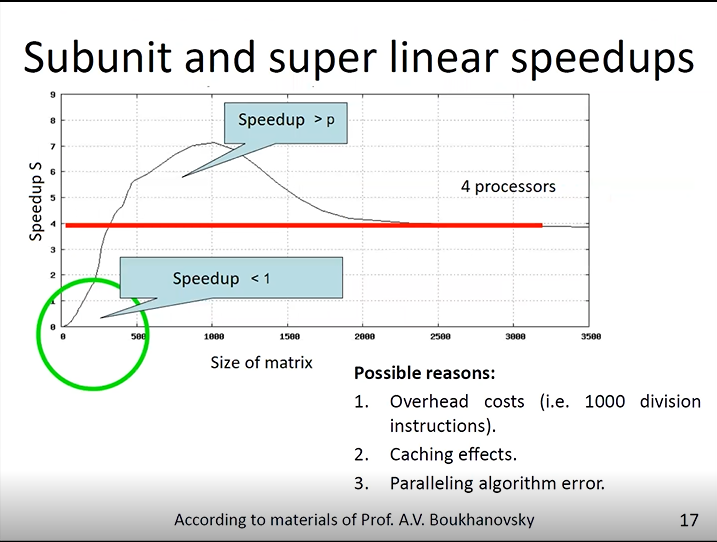
Давайте рассмотрим те функции, которые вы будете использовать для того, чтобы оценить время выполнения программы. Так получилось, что прежде всего в вычислительно технике есть такое понятие, как legacy. Наследственность, наследие программного обеспечения, которое является проблемой. Многие языки (в т.ч. Си) были созданы задолго до выполнения параллельных вычислений. Например, если вы пользуетесь функцией ctime (третья строка слайда), в однопоточной программе для измерения времени участка кода она успешно справляется с вашей программой. Однако после распараллеливания этой части кода, возможно, возникнут трудно выявляемые проблемы с некорректным измерением, так как эта функция имеет внутреннюю статистическую переменную, которая при попытке изменить ее одновременно несколькими потоками может занять какое-то время, а также вернет непредсказуемое значение. То есть данная функция не реэнтерабельна, не кроссплатформенная. Есть альтернатива ctime\_r, но она тоже не всегда удобна.

Существуют некоторые другие стандартные функции измерения времени, которые также нельзя использовать, однако причина не в отсутствии возможностей повторного кода, а в особенностях того, как эта функция подсчитывает прошедшее время. Часы могут возвращать количество тактов процессора, которые были выполнены при выполнении программы в сумме со всеми потоками, и эта сумма практически неизменная при выполнении программы с другим количеством потоков, т.е. проблему исключительно функциями Си не решить. Здесь можно использовать (при использовании OpenMP) omp\_get\_wtime и omp\_get\_wtick, при этом нужно использовать заголовочный файл и учитывать некоторые особенности. Но большинство специальных библиотек для OpenMP имеют встроенные функции измерения времени. Так как с высокой долей вероятности вы не сможете точно сказать, какая доля программы у вас будет распараллелена, в рамках лабораторных работ вы будете делать измерения времени, на основании этого можете строить понимание того, как работает параллельное ускорение и пытаться подогнать его под теоретические значения, чтобы понять, что сходится, а что нет.

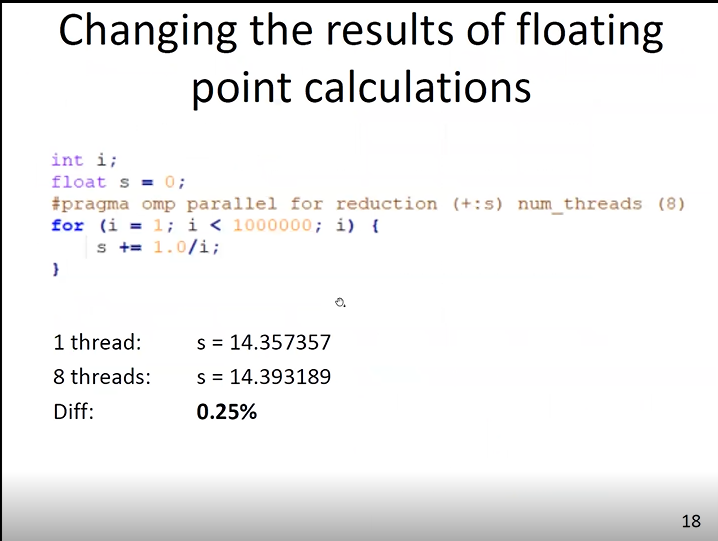


Второй момент, который важно здесь упомянуть для ЛР: у нас существуют всевозможные помехи, и для правильного подсчета времени стоит использовать функции (не только для измерения времени, но и для правильной работы остальных частей программы), которые также будут реентерабельны, например rand\_r вместо rand, которые генерируют произвольные числа. Это необходимо для того, чтобы каждый раз формировалась одинаковая последовательность чисел для того, чтобы сравнение результатов было корректным. Rand\_r позволяет использовать определенный генератор случайных чисел.

Какие еще варианты есть? Мы можем использовать для правильной оценки времени либо минимальное время, либо среднее время, либо можем считать доверительный интервал в идеале, чтобы понимать, как у нас все разбросано. То есть программу нужно будет запускать не просто один раз, а несколько раз для того, чтобы увидеть реальное время.



Следующий интересный момент. У нас есть параллельное ускорение (растет на графике). И в зависимости от размера матрицы оно сначала будет больше, чем число процессоров p, а может быть меньше единицы. График не совсем однозначный и не понятно, что здесь происходит. Возможные причины – потому что у нас есть накладные расходы, т.е. на маленьких-маленьких значениях матрицы, с которой мы что-то делаем, что может происходить? Первое – накладные расходы, т.е. если у нас больше тысячи инструкций, и пока это выполнится, с малым количеством данных будет большая проблема. Второе – элементы кэша – они могут быть правильными. Третье – ошибки распараллеливания, поэтому обязательно нужно сравнивать с нераспараллеленной программой.



И еще одна вещь, связанная с точностью выполнения операций. У нас есть некоторая программа, написанная на OpenMP. Основная особенность в том, что в цикле от 1 до миллиона мы считаем дробь: 1 разделить на i, то есть 1/1, ½, 1/3 и так далее. Если эта программа будет выполняться в одном потоке, то результат равен 14.357357, а если запустить на 8 потоках, то получается немного другое число, и разница между результатами в 0.25%. Это происходит из-за неправильного округления, мы не последовательно складываем числа, а разбиваем на итерации, которые выполняются разными потоками, и затем складываем между собой. Здесь возникает ошибка округления. То есть порядок выполнения операций очень важен и может влиять на получаемый результат.