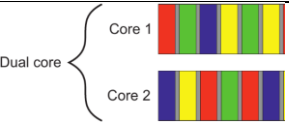


<p>Область распределенных вычислений — раздел теории вычислительных систем, изучающий теоретические вопросы организации распределенных систем.</p>	<p>Распределенная система — это такая система, в которой взаимодействие и синхронизация программных компонентов, выполняемых на независимых сетевых компьютерах, осуществляется посредством передачи сообщений.</p>	<p>Параллельная система — это система, в которой вычисления могут выполняться без ожидания завершения других вычислений. Современные ОС поддерживают параллелизм как минимум на уровне запускаемых программ.</p>
<p>Параллельные вычисления — форма вычислений, при которой несколько вычислений выполняются одновременно (в течение перекрывающихся периодов времени), вместо последовательных (с завершением одного до начала следующего). Являются частью распределённых вычислений.</p>	<p>Процесс (операционной системы) — это выполнение программных инструкций в рамках независимого виртуального адресного пространства, объединяющего код и данные компьютерной программы.</p>	<p>Поток (выполнения) — программный механизм, предоставляемый ОС для параллельного выполнения программных инструкций в адресном пространстве компьютерной программы.</p>
<p>Адресное пространство пользователя — объединение кода и данных выполняемой программы в рамках запущенного в ОС процесса, защищённые от области ядра ОС и других процессов.</p>	<p>Параллельная машина с произвольным доступом к памяти PRAM (наиболее популярная) — объединены p процессоров, общая память и устройство управления, которое передает команды программы P процессорам.</p>	<p>Устройство называется многопроцессорным, если в его составе используется два или более физических процессора. Операционная система или программа называется многопроцессорной, если способна распределять задачи между процессорами</p>
<p>Классификация Флинна поток определяется как последовательность команд или данных, выполняемых или обрабатываемых процессором. В этой модели программа может предоставлять процессору не только потоки инструкций для исполнения, но и поток данных для обработки. Потоки команд и потоки данных предполагаются независимыми.</p>	<p>Классификация Флинна 1. Один поток команд, один поток данных. 2. Один поток команд, несколько потоков данных — одна и та же операция выполняется одновременно над различными данными. 3. Несколько потоков команд, один поток данных — полезны в некоторых узкоспециальных задачах. 4. Несколько потоков команд, несколько потоков данных — большинство систем.</p>	<p>Среди MIMD-систем выделяют: 1. Мультипроцессоры — машины с общей памятью. 2. Мультикомпьютеры — машины, не обладающие удаленным доступом к памяти. Взаимодействие между процессорами с помощью механизма передачи сообщений.</p>
<p>В системах с распределенной памятью каждый вычислительный узел имеет доступ только к принадлежащему ему участку локальной памяти. Для межпроцессорного обмена данными предусматривается возможность отправки и приема сообщений по коммуникационной сети, объединяющей вычислительную систему. Эта модель программирования — модель передачи сообщений.</p>	<p>Средства параллельного программирования 1. Автоматическое распараллеливание последовательной версии программы средствами компилятора; 2. Использование специализированных языков для параллельного программирования (Erlang); 3. Использование библиотек, предоставляющих возможности параллельного исполнения кода; 4. Программирование с использованием особых расширений языка — средств распараллеливания.</p>	<p>Машина с произвольным доступом к памяти RAM: 1. Система, состоит из процессора, устройства доступа к памяти (системной шины) и памяти, состоящей из конечного числа ячеек. 2. Процессор последовательно выполняет команды, заложенные в программе; ему доступны основные арифметические и логические операции и чтение/запись данных в памяти. При этом постулируется, что каждая команда выполняется за фиксированное время.</p>
<div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <p>• Произвольное действие процессора состоит из трех этапов: 1. чтение данных из памяти в один из своих регистров r_i, где $1 \leq i \leq N$; 2. выполнение арифметической или логической операции над содержимым своих регистров; 3. запись данных из регистра r_j, где $1 \leq j \leq N$, в некоторую ячейку памяти. • Считается, что исполнение трех перечисленных шагов требует времени $\Theta(1)$.</p> </div> <div style="width: 45%;"> </div> </div>		
<p>Конфликты чтения. Если несколько процессоров пытаются прочесть данные из одной ячейки, то возможны два варианта дальнейших действий: 1. Исключающее чтение (ER). В данный момент времени чтение разрешено только одному процессору, в противном случае происходит ошибка выполнения программы. 2. Одновременное чтение (CR). Количество процессоров, получающих доступ к одной ячейке памяти, не ограничивается.</p>	<p>Конфликты записи. Если более одного процессора пытаются записать данные по одному адресу, разделяют два способа действия. 1. Исключающая запись (EW). Только одному процессору разрешено осуществлять запись в данную ячейку в конкретный момент времени. 2. Одновременная запись (CW). Несколько процессоров сразу получают доступ для записи к одной ячейке памяти.</p>	<p>Теорема об эмуляции (PRAM) Пусть алгоритм для CRCW-машины решает некоторую задачу с параметром размера N за время $T(N)$, используя p процессоров. Тогда существует алгоритм для той же задачи на EREW-системе с p процессорами, который может быть исполнен за время $O(T(N)\log_2 N)$. При этом объем памяти PRAM должен быть увеличен в $O(p)$ раз.</p>
<p>Ускорение вычислений (PRAM) с p процессорами $S_p(N) = \frac{T_1(N)}{T_p(N)}$ Это мера прироста производительности по сравнению с наилучшим последовательным алгоритмом.</p>	<p>Закон Амдала (PRAM). Пусть f — доля последовательных вычислений в алгоритме A. Тогда ускорение S_p при использовании A на системе из p процессоров удовлетворяет неравенству $S_p \leq \frac{1}{f + (1-f)/p}$</p>	<p>Закон Амдала (PRAM) — выводы. Существование последовательных вычислений, которые не могут быть распараллелены, накладывает ограничение на S_p: $S_p \leq (S_p)_{\max} \quad S_{\infty} = \frac{1}{f}$</p>

<p>Искусственный параллелизм — имитация параллельного выполнения задач на компьютерах с единственным ядром.</p> <p>Истинный параллелизм — возможность использовать несколько ядер для достижения параллельного выполнения задач.</p> <p>Многопоточность — искусственный или истинный параллелизм, реализованный в рамках одного процесса.</p>	<p>Задержки:</p> <ol style="list-style-type: none"> 1. Чтобы обеспечить чередование, система производит контекстное переключение при каждом переходе от одной задачи к другой. 2. Чтобы переключить контекст, ОС должна сохранить состояние процессора и счетчик команд для текущей задачи, определить, какая задача следующая, и загрузить в процессор состояние новой задачи. 3. Загрузка команды и данных новой задачи в кэш-память. 	
 <p>С истинным параллелизмом количество задач может превышать число ядер, тогда будет применяться механизм переключения задач.</p>	<p>Межпроцессорный параллелизм</p> <p>Плюсы:</p> <ul style="list-style-type: none"> — разделение обязанностей; — высокая надёжность; — в шаге от распределённой системы. <p>Минусы:</p> <ul style="list-style-type: none"> — используется больше ресурсов; — взаимодействие занимает много времени. 	<p>Многопоточность. Плюсы:</p> <ul style="list-style-type: none"> — разделение обязанностей; — высокая производительность; — потребляется меньше ресурсов системы. <p>Минус: слабая надёжность.</p> <p>Используется для: разделения обязанностей, повышения производительности, экономии ресурсов, контроль надежности программы.</p>
<p>Проектирование многопоточности</p> <p>Распараллеливание по задачам — разделение программы на задачи и запуск их параллельно.</p> <p>Распараллеливание по данным — выполнение одинаковых операций с различными данными.</p> <p><i>Многопоточность требует изменения подхода к проектированию.</i></p>	<p>Когда вредна многопоточность</p> <p>Затраты на распараллеливание не компенсируют получаемый эффект: слишком мелкие и быстро выполняемые задачи (может помочь гетерогенный параллелизм, асинхронность); задач слишком много (может помочь пул потоков, асинхронность).</p>	<p>Гонка данных — это ситуация, когда по меньшей мере два потока имеют одновременный доступ к некоторой переменной и хотя бы один из потоков производит её запись.</p>
<p>Мьютекс — примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода.</p> <ol style="list-style-type: none"> 1. В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом. 2. Если другому потоку будет нужен доступ к данным, защищённым мьютексом, то этот поток блокируется до тех пор, пока мьютекс не будет освобождён. 	<p>Мёртвая блокировка, взаимоблокировка, дедлок — это состояние, в котором каждый из двух или более потоков заблокирован в ожидании ресурса, занятого другим потоком, и до своей разблокировки не может освободить ресурс, которого ожидает другой поток.</p> <p>Результат мёртвой блокировки — полная остановка работы потоков, а чаще всего и всей программы. Навечно.</p>	<p>Правила работы с мьютексами</p> <ol style="list-style-type: none"> 1. Если функция выбросит исключение, мьютекс никогда не будет освобождён. 2. Ни в коем случае нельзя вызывать из-под мьютекса функцию, внутреннее устройство которой неизвестно. В большинстве случаев неопределённое поведение выражается в мёртвой блокировке.
<p>Блокировщики управляют захватом и освобождением ресурса посредством идиомы RAII - захват ресурса есть инициализация. При объявлении объекта класса, работающего с файлами, на стеке происходит и его инициализация с вызовом конструктора, захватывающего ресурс. При выходе из области видимости объект выталкивается из стека, но перед этим вызывается деструктор объекта, который и освобождает захваченный ресурс.</p>	<p>Если значение переменной никогда не изменяется, нет нужды синхронизировать доступ к ней с помощью механизмов блокировки. Нужно лишь присвоить ей начальное значение потокобезопасным способом. В языке C++ есть три способа потокобезопасной инициализации:</p> <ul style="list-style-type: none"> — константные выражения (constexpr); — функция std::call_once вместе с флагом std::once_flag; — локальная статическая переменная. 	<p>Потокобезопасная инициализация</p> <p>Локальная статическая переменная</p> <p>Статические переменные, локальные для блока, инициализируются один раз ленивым образом.</p> <p>В стандарте C++ 11 для локальных статических переменных появилась новая гарантия: их инициализация потокобезопасна</p>
<p>Переменные условия</p> <ol style="list-style-type: none"> 1. Позволяют синхронизировать потоки посредством обмена сообщениями 2. Один поток выступает отправителем сообщения, а другой поток (или несколько потоков) — получателем. 3. Получатель ждёт, пока не придёт сообщение. 4. Чаще всего применяются, когда нужно реализовать способ обработки данных по типу издателя и подписчика или производителя и потребителя. 5. Служит связующим звеном между отправителем и получателем сообщения. 	<p>Переменные условия. Методы класса.</p> <p>notify_one — оповестить один ожидающий поток о наступлении события;</p> <p>notify_all — оповестить все потоки;</p> <p>wait — ожидать сообщения, держа блокировщик открытым;</p> <p>wait_for — ожидать сообщения, держа блокировщик открытым, но не более заданного промежутка времени;</p> <p>wait_until — ожидать сообщения, держа блокировщик открытым, но не более, чем до заданного момента времени;</p> <p>native_handle — возвращает системный дескриптор переменной условия</p>	<p>Переменные условия. Значение предиката</p> <p>Предикат наделяет переменную условия состоянием. Функция ожидания всегда должна сначала проверить истинность предиката.</p> <p>Предикат, таким образом, помогает бороться с двумя известными слабыми местами переменных условия: утерянным пробуждением и ложным пробуждением.</p>
<p>Переменные условия. Утеранные и ложные пробуждения</p> <p>Утеранным пробуждением называется ситуация, когда поток-отправитель успевает послать оповещение до того, как получатель начинает его ожидать. Как следствие оповещение оказывается утерянным.</p> <p>Ложное пробуждение — это пробуждение ожидающего потока, когда отправители никаких оповещений не посылали. (одна из причин такого явления — похищенное пробуждение: перед тем как пробуждённый поток-адресат получает шанс запуститься, другой поток успевает вклиниться первым и начинает выполнение)</p>	<p>Семафор — примитив синхронизации работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счётчика является недопустимой.</p> <p>Служит для построения сложных механизмов синхронизации и используется для синхронизации параллельно работающих задач, для защиты передачи данных через разделяемую память, для защиты критических секций, а также для управления доступом к аппаратному обеспечению.</p>	<ol style="list-style-type: none"> 1. Семафор — это структура данных, содержащая очередь и счётчик. 2. Счётчик инициализируется значением, большим или равным нулю. 3. Операция wait захватывает семафор, уменьшая значение счётчика, если оно положительно, или блокирует поток в противном случае. 4. Операция signal освобождает семафор путём увеличения счётчика и, если очередь заблокированных потоков не пуста, пробуждает первый поток из неё. 5. Постановка заблокированных потоков в очередь необходима для предотвращения ресурсного голода.

<p>Критическая секция — участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком выполнения. При нахождении в критической секции двух (или более) потоков возникает состояние «гонки».</p>	<p>Ресурсный голод — ситуация, когда поток или процесс не может продолжить работу и проводит неопределённо долгое время в ожидании из-за того, что система каждый раз отказывает ему в предоставлении некоторого ресурса</p>	<p>Асинхронные задания (механизм будущих результатов) — позволяет выполнять две операции параллельно, пока результат выполнения одной операции не понадобится для другой.</p> <p>Задание характеризуется пакетом работы, которую предстоит выполнить, и обладает двумя частями: обещанием и будущим.</p>																					
<p>Асинхронные задания ведут себя подобно каналам, по которым данные проходят от входного разъёма до выходного. Один конец канала называется обещанием (помещает данные в канал), другой — будущим (в неопределённый момент в будущем извлекает из канала результат их преобразования). Этими разъёмами может управлять один и тот же канал, а могут — разные.</p>	<table border="1"> <thead> <tr> <th>Критерий</th><th>Потоки</th><th>Задания</th></tr> </thead> <tbody> <tr> <td>Основные сущности</td><td>Родительский и дочерний потоки</td><td>Обещание и фьючерс</td></tr> <tr> <td>Способ передачи данных</td><td>Общая переменная</td><td>Канал</td></tr> <tr> <td>Отдельный поток</td><td>Всегда</td><td>Иногда</td></tr> <tr> <td>Синхронизация</td><td>Функция join ожидает завершения потока</td><td>Функция get блокирует выполнение</td></tr> <tr> <td>Исключение в дочернем потоке (задании)</td><td>Оба потока завершаются вместе со всем процессом</td><td>Передаётся через обещание и фьючерс</td></tr> <tr> <td>Передаваемые данные</td><td>Значения</td><td>Значения, оповещения и исключения</td></tr> </tbody> </table>	Критерий	Потоки	Задания	Основные сущности	Родительский и дочерний потоки	Обещание и фьючерс	Способ передачи данных	Общая переменная	Канал	Отдельный поток	Всегда	Иногда	Синхронизация	Функция join ожидает завершения потока	Функция get блокирует выполнение	Исключение в дочернем потоке (задании)	Оба потока завершаются вместе со всем процессом	Передаётся через обещание и фьючерс	Передаваемые данные	Значения	Значения, оповещения и исключения	
Критерий	Потоки	Задания																					
Основные сущности	Родительский и дочерний потоки	Обещание и фьючерс																					
Способ передачи данных	Общая переменная	Канал																					
Отдельный поток	Всегда	Иногда																					
Синхронизация	Функция join ожидает завершения потока	Функция get блокирует выполнение																					
Исключение в дочернем потоке (задании)	Оба потока завершаются вместе со всем процессом	Передаётся через обещание и фьючерс																					
Передаваемые данные	Значения	Значения, оповещения и исключения																					
<p>Всегда следует предпочитать асинхронные вызовы. C++ сам решает, выполнять асинхронный вызов в отдельном потоке или нет (зависит от доступных ядер процессора, загруженности системы и т.п.). Вызывая функцию std::async, программист лишь передаёт ей задание, которое должно быть выполнено (+ можно передать политику запуска). Вся работа по возможному созданию потока и управлению временем его жизни перекладывается на внутренние механизмы реализации.</p>	<p>Асинхронные задания. Политика запуска. Можно в явном виде указать, каким образом реализации следует выполнить асинхронный вызов: в том же потоке, который создал вызов, или в другом потоке.</p> <p>Обещание запускается только тогда, когда фьючерс в явном виде запрашивает его результат (ленивые вычисления).</p> <p>При строгой (жадной) стратегии выражение вычисляется немедленно, тогда как при ленивой (отложенной) стратегии вычисление откладывается до тех пор, пока значение не станет необходимо.</p>	<p>Особый случай составляют фьючерсы, о которых забывают сразу после создания. Они не сохраняются в каких-либо переменных и должны запускаться немедленно в момент создания. Существенно, что обещания таких фьючерсов должны выполняться в отдельном потоке, чтобы фьючерс мог начать работу немедленно.</p> <p>Короткоживущие фьючерсы выглядят удобными, но имеют существенный недостаток — они выполняются на самом деле последовательно.</p>																					
<p>Модель памяти — это договорённость о том, что: разработчик программы предотвращает гонку данных, а компилятор гарантирует строгость последовательности выполнения. Модель памяти C++ была создана таковой, чтобы обеспечивать оптимальный многопоточный код под множество архитектур процессоров.</p> <p>Строгость последовательности выполнения — это результат любого выполнения, как если бы: — операции всех потоков выполняются в некотором последовательном порядке; — операции каждого потока проявляются в этой последовательности в порядке, указанном их программой.</p>	<p>Механизмы синхронизации потоков std::atomic<>. Наиболее тонкий, сложный и низкоуровневый механизм синхронизации и реализации атомарных операций.</p> <p>Переменные, защищённые от гонки данных. Обеспечивает синхронизацию потоков.</p> <p>Атомарная операция — это операция, которую невозможно наблюдать в промежуточном состоянии, она либо выполнена, либо нет. Атомарные операции могут состоять из нескольких операций.</p>	<p>Нежелательное блокирование состояние потока, когда он не может продолжать выполнение, так как чего-то ждёт.</p> <p>Взаимоблокировка — когда один поток ждёт другого, а тот, в свою очередь, ждёт первого.</p> <p>Блокировка в ожидании завершения ввода/вывода или поступления данных из внешнего источника — если поток блокируется в ожидании данных из внешнего источника, то он не может продолжать работу, даже если данные так никогда и не поступят.</p>																					
<p>Сопрограмма (корутина) — блок кода, который работает асинхронно с вызывающим кодом, то есть по очереди.</p> <p>Сопрограмма (корутина)— это функция, которая может приостановить выполнение, чтобы возобновить его позже.</p> <p>Какие бывают:</p> <p>Стековые сопрограммы — это сопрограммы, у которых есть свой стек (как у потоков).</p> <p>Безстековые сопрограммы — никак не зависят от операционной системы, и реализуются исключительно средствами компилятора.</p> <p>Где применяются:</p> <p>Конечные автоматы в рамках одной подпрограммы;</p> <p>Акторная модель параллелизма, например в видеоиграх;</p> <p>Генераторы;</p> <p>Передача последовательности процессов, где каждый подпроцесс является сопрограммой;</p> <p>Обратная связь, обычно используется в математическом ПО.</p>	<p>Условие (порядок) синхронизации std::memory_order_seq_cst</p> <p>Наиболее строгий порядок синхронизации.</p> <p>Свойства:</p> <ul style="list-style-type: none"> - порядок модификаций разных атомарных переменных в потоке thread1 сохранится в потоке thread2; - все потоки будут видеть один и тот же порядок модификации всех атомарных переменных (сами модификации могут происходить в разных потоках); - все модификации памяти (не только модификации над атомами) в потоке thread1, выполняющей store на атомарной переменной, будут видны - после выполнения load этой же переменной в потоке thread2. <p>Таким образом можно представить seq_cst операции, как барьеры памяти, в которых состояние памяти синхронизируется между всеми потоками программы.</p> <p>Этот флаг синхронизации памяти в C++ используется по умолчанию</p>	<p>Условие (порядок) синхронизации std::memory_order_relaxed</p> <p>Он гарантирует только свойство атомарности операций, при этом не может участвовать в процессе синхронизации данных между потоками.</p> <p>Свойства:</p> <ul style="list-style-type: none"> -модификация переменной "появится" в другом потоке не сразу; -поток thread2 "увидит" значения одной и той же переменной в том же порядке, в котором происходили её модификации в потоке thread1; -порядок модификаций разных переменных в потоке thread1 не сохранится в потоке thread2. 																					

<p>Гонка за данными — это особый тип гонки, который приводит к неопределённому поведению из-за несинхронизированного одновременного доступа к разделяемой ячейке памяти.</p> <p>— Обычно гонка за данными возникает вследствие неправильного использования атомарных операций для синхронизации потоков или в результате доступа к разделяемым данным, не защищённого подходящим мьютексом.</p>	<p>Нарушение инвариантов — такие гонки могут проявляться в форме висячих указателей (другой поток уже удалил данные, к которым мы пытаемся обратиться), случайного повреждения памяти (из-за того, что поток читает данные, оказавшиеся несогласованными в результате частичного обновления) и двойного освобождения (например, два потока извлекают из очереди одно и то же значение, и потом оба удаляют ассоциированные с ним данные).</p>	<p>Проблемы со временем жизни — такого рода проблемы можно было бы отнести к нарушению инвариантов, но на самом деле это отдельная категория. Основная проблема в том, что поток живет дольше, чем данные, к которым он обращается, поэтому может попытаться получить доступ к уже удалённым или разрушенным иным способом данным. Не исключено также, что когда-то отведённая под эти данные память уже занята другим объектом. Обычно такие ошибки возникают, когда поток хранит ссылки на локальные переменные, которые вышли из области видимости до завершения функции потока, но это не единственный сценарий.</p>
<p>Флаг синхронизации памяти acquire/release является более тонким способом синхронизировать данные между парой потоков (работают только в паре над одним атомарным объектом).</p> <p>Свойства:</p> <p>— модификация атомарной переменной с release будет видна в другом потоке, выполняющем чтение этой же атомарной переменной с acquire;</p> <p>— все модификации памяти в потоке thread1, выполняющей запись атомарной переменной с release, будут видны после выполнения чтения той же переменной с acquire в потоке thread2;</p> <p>- процессор и компилятор не могут перенести операции записи в память раньше release</p> <p>- операции в потоке thread1, и нельзя перемещать выше операции чтения из памяти позже acquire</p> <p>операции в потоке thread2.</p>	<p>Пул потоков — поддерживает несколько потоков, ожидающих выделения задач для одновременного выполнения управляющей программой.</p> <p>Очередь задач — позволяет развязать основной алгоритм и другие шаблоны параллельного программирования (часто применяется с пулом потоков).</p> <p>Блокиратор — безопасное использование блокировки (как правило на основе мьютекса) с применением идиомы RAII в C++.</p> <p>Активный объект — шаблон проектирования, который отделяет поток выполнения метода от потока, в котором он был вызван (см. модель акторов).</p> <p>Блокировка с двойной проверкой — предназначен для уменьшения накладных расходов, связанных с получением блокировки.</p> <p>Монитор — объект, предназначенный для безопасного использования более чем одним потоком.</p> <p>Планировщик - обеспечивающий механизм реализации политики планирования, но при этом не зависящий ни от одной конкретной политики.</p>	<p>Пул потоков — шаблон параллельного программирования, который поддерживает несколько потоков, ожидающих выделения задач для одновременного выполнения управляющей программой.</p> <p>Преимущества:</p> <p>Накладные расходы на создание и уничтожение потоков ограничены -> растёт производительность;</p> <p>Обеспечивает управление параллелизмом и синхронизацией потоков на более высоком уровне абстракции</p> <p>Недостатки:</p> <p>При активном использовании синхронизации между задачами в пуле потоков понижается его эффективность.</p>
<p>Активная блокировка — взаимоблокировка при цикле активной проверки, например спинлоке. В одних случаях программа потребляет много процессорного времени, потому что потоки блокируют друг друга, продолжая работать. В других — блокировка рано или поздно может «рассасаться» из-за вероятностной природы планирования, но заблокированная задача испытывает ощутимые задержки, характеризующиеся высоким потреблением процессорного времени.</p>	<p>Использование доступных инструментов (желательно в CI/CD):</p> <ul style="list-style-type: none"> — «выкрутить» диагностику компилятора на максимум; — прогнать через статический анализатор; — прогнать через специальный профилировщик. <p>«Ручной» просмотр:</p> <ul style="list-style-type: none"> — посмотреть самому; — показать другу; — объяснить кому-то, как работает ваш код (можно плюшевому мишке или утёнку). <p>Поиск связанных с параллелизмом ошибок путем тестирования:</p> <ul style="list-style-type: none"> — проверить в однопоточном режиме; — протестировать в многопоточном. 	<p>Тестирование грубой силой — код прогоняется многократно на большом количестве потоков (например, стресс-тест). Позволяет «поймать» редкие ошибочные ситуации. Недостаток метода грубой силы в том, что он может вселять ложную уверенность.</p> <p>Комбинаторное имитационное тестирование</p> <p>- прогонять код под управлением специальной программы, которая имитирует реальную среду.</p> <p>Обнаружение возникающих во время тестирования проблем с помощью специальной библиотеки.</p>
<p>Нагрузочное тестирование</p> <p>Для исследования времени отклика системы на высоких или пиковых нагрузках. Моделирования ожидаемого использования приложения с помощью эмуляции работы нескольких пользователей одновременно. Таким образом, подобное тестирование больше всего подходит для многопользовательских систем, чаще — использующих клиент-серверную архитектуру (например, веб-серверов).</p> <p>Принципы:</p> <ul style="list-style-type: none"> — Уникальность запросов; — Время отклика системы; — Зависимость времени отклика системы от степени распределённости этой системы; — Разброс времени отклика системы; — Точность воспроизведения профилей нагрузки. 	<p>Регрессионное тестирование</p> <p>Направлены на обнаружение ошибок в уже протестированных участках исходного кода. Такие ошибки — когда после внесения изменений в программу, перестаёт работать то, что должно было продолжать работать, — называют регрессионными ошибками.</p> <p>Классификация:</p> <ul style="list-style-type: none"> - Минимизация набора тестов - Задача минимизации наборов - Задача с определением приоритетов - Задача выбора тестов 	