# История развития параллельного программирования

Как и другие прикладные области компьютерных наук, параллельное программирование прошло несколько стадий. Оно возникло благодаря новым возможностям, предоставленным развитием аппаратного обеспечения, и развилось в соответствии с технологическими изменениями. Через некоторое время специализированные методы были объединены в набор основных принципов и общих методов программирования.

Параллельное программирование возникло в 1960-е годы в сфере [операционных систем](http://wiki.mvtom.ru/index.php/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). Причиной стало изобретение аппаратных модулей, названных каналами, или контроллерами устройств. Они работают независимо от управляющего процессора и позволяют выполнять операции ввода-вывода параллельно с инструкциями центрального процессора. Канал взаимодействует с процессором с помощью [прерывания](http://wiki.mvtom.ru/index.php?title=%D0%9F%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5&action=edit&redlink=1) - аппаратного сигнала, который говорит: "Останови свою работу и начни выполнять другую последовательность инструкций".

Результатом появления каналов стала проблема программирования (настоящая интеллектуальная проблема) - теперь части программы могли быть выполнены в непредсказуемом порядке. Следовательно, пока одна часть программы обновляет значение некоторой переменной, может возникнуть прерывание, приводящее к выполнению другой части программы, которая тоже попытается изменить значение этой переменной. Это специфическая проблема (задача критической секции).

Вскоре после изобретения каналов началась разработка многопроцессорных машин, хотя в течение двух десятилетий они были слишком дороги для широкого использования. Однако сейчас все крупные машины являются многопроцессорными, а самые большие имеют сотни процессоров и часто называются машинами с массовым параллелизмом (massively parallel processors).

Многопроцессорные машины позволяют разным прикладным программам выполняться одновременно на разных процессорах. Они также ускоряют выполнение приложения, если оно написано (или переписано) для многопроцессорной машины. Но как синхронизировать работу параллельных процессов? Как использовать многопроцессорные системы для ускорения выполнения программ?

Итак, при использовании каналов и многопроцессорных систем возникают и возможности, и трудности. При написании параллельной программы необходимо решать, сколько процессов и какого типа нужно использовать, и как они должны взаимодействовать. Эти решения зависят как от конкретного приложения, так и от аппаратного обеспечения, на котором будет выполняться программа. В любом случае ключом к созданию корректной программы является правильная синхронизация взаимодействия процессов.

# Опишите что такое SIMD, SSE? Чем версии SSE отличаются между собой? Примеры SIMD и SSE

The Wikipedia page (http://en.m.wikipedia.org/wiki/SIMD) does a good job of explaining SIMD, and the instruction sets that implement it.

Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously.

SIMD is the 'concept', SSE/AVX are implementations of the concept. All SIMD instruction sets are just that, a set of instructions that the CPU can execute on multiple data points. As long as the CPU supports executing the instructions, then it is feasible for multiple SIMD instruction sets to coexist, regardless of data size.

Команды мб?

SSE used only a single data type for XMM registers:

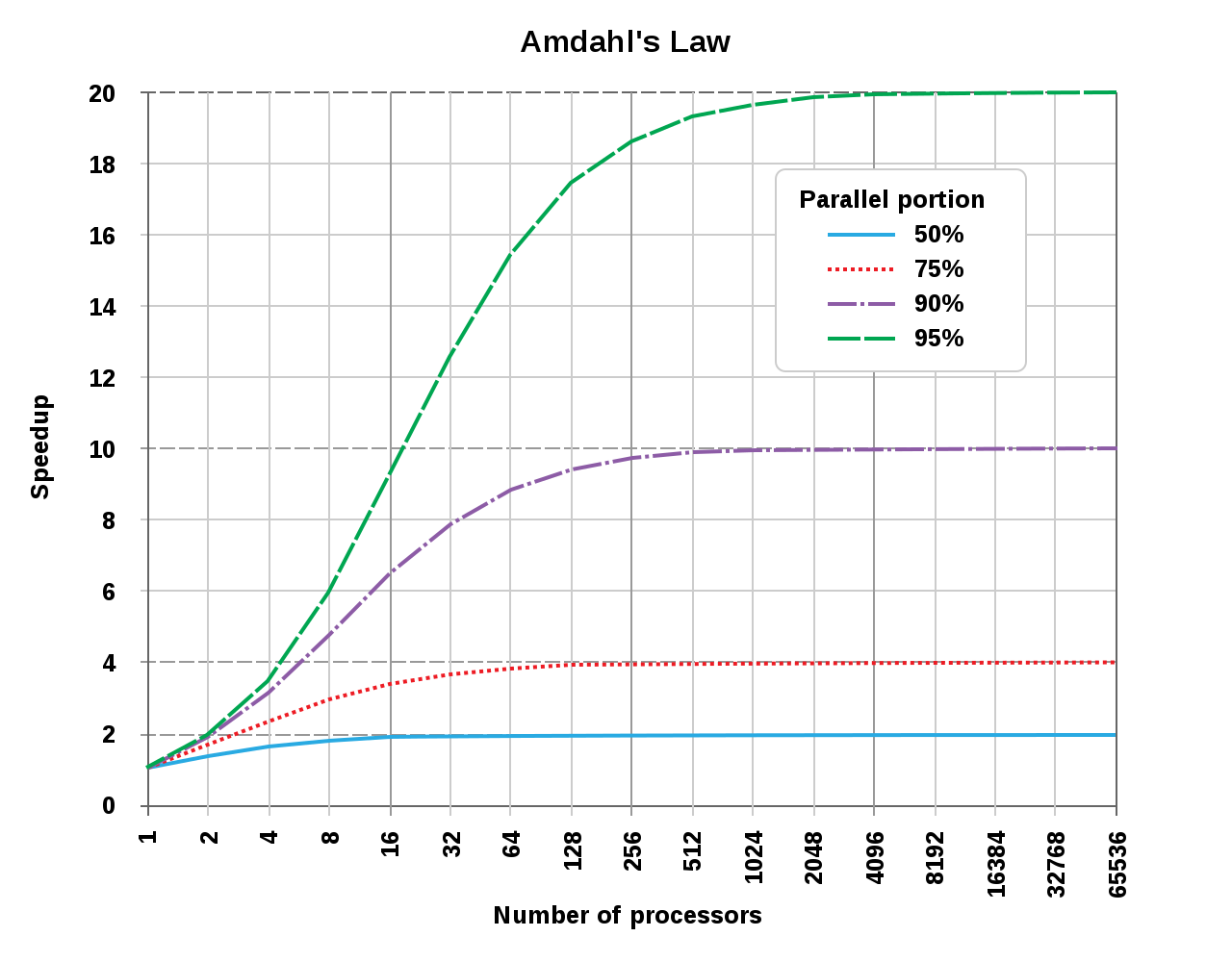
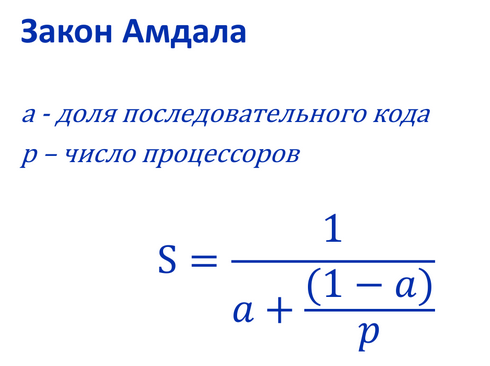
* four 32-bit [single-precision](https://en.wikipedia.org/wiki/Single-precision) floating-point numbers

[SSE2](https://en.wikipedia.org/wiki/SSE2) would later expand the usage of the XMM registers to include:

* two 64-bit [double-precision](https://en.wikipedia.org/wiki/Double-precision) floating-point numbers or
* two 64-bit integers or
* four 32-bit integers or
* eight 16-bit short integers or
* sixteen 8-bit bytes or characters.

# Что такое закон Амдала? Что этот закон описывает?

Закон Амдала иллюстрирует ограничение роста производительности вычислительной системы с увеличением количества вычислителей.



# Опишите архитектуру современных процессоров (branch prediction, CPU prefetch, ...)

Предсказание ветки которая будет выполнена в результате работы if else. Повышает производительность, ему пойдет такой ответ <https://en.wikipedia.org/wiki/Branch_predictor>

CPU prefetch это взятие данных из основной памяти в кэш-память процика с такой же целью увеличить производительность работы программы <https://en.wikipedia.org/wiki/Cache_prefetching>

# Что такое классификация Флинна? Приведите примеры отличий

Таксономия (Классификация) Флинна — общая классификация архитектур ЭВМ по признакам наличия параллелизма в потоках команд и данных.

Всё разнообразие архитектур ЭВМ в этой таксономии Флинна сводится к четырём классам:

* ОКОД — Вычислительная система с одиночным потоком команд и одиночным потоком данных (SISD, single instruction stream over a single data stream).

Архитектура SISD — это традиционный компьютер фон-Неймановской архитектуры с одним процессором, который выполняет последовательно одну инструкцию за другой, работая с одним потоком данных. В данном классе не используется параллелизм ни данных, ни инструкций, и, следовательно, SISD-машина не является параллельной. К этому классу также принято относить конвейерные, суперскалярные и VLIW-процессоры.

* ОКМД — Вычислительная система с одиночным потоком команд и множественным потоком данных (SIMD, single instruction, multiple data).

Типичными представителями SIMD являются векторные процессоры, обычные современные процессоры, когда работают в режиме выполнения команд векторных расширений, а также особый подвид с большим количеством процессоров — матричные процессоры. В SIMD-машинах один процессор загружает одну инструкцию, набор данных к ним и выполняет операцию, описанную в этой инструкции, над всем набором данных одновременно.

* МКОД — Вычислительная система со множественным потоком команд и одиночным потоком данных (MISD, multiple instruction, single data).

К классу MISD ряд исследователей относит конвейерные ЭВМ, однако это не нашло окончательного признания. Также, возможно считать MISD системами, системы с горячим резервированием. Помимо этого, к архитектуре MISD некоторые относят систолические массивы процессоров.

* МКМД — Вычислительная система со множественным потоком команд и множественным потоком данных (MIMD, multiple instruction, multiple data).

Класс MIMD включает в себя многопроцессорные системы, где процессоры обрабатывают множественные потоки данных. Сюда принято относить традиционные мультипроцессорные машины, многоядерные и многопоточные процессоры, а также компьютерные кластеры.

# Опишите взаимодействие с потоками на уровне POSIX (детали реализации и основные функции)

* errno – глобальная int переменная / макрос
* pthread\_t — идентификатор потока;
* pthread\_mutex\_t — мютекс;
* pthread\_create – создание потока
* завершение потока
  + потоковая функция выполняет return и возвращает результат произведенных вычислений;
  + в результате вызова завершения исполнения потока pthread\_exit();
  + в результате вызова отмены потока pthread\_cancel();
  + одна из нитей совершает вызов exit()
  + основная нить в функции main() выполняет return, и в таком случае все нити процесса резко сворачиваются.
* pthread\_join() – ожидание потока
* pthread\_cancel() – досрочное завершение
* pthread\_detach – отсоединение потока

# Опишите взаимодействие с потоками на уровне Java Threads (детали реализации основные методы)

Каждый процесс имеет хотя бы один выполняющийся поток. Тот поток, с которого начинается выполнение программы, называется главным. В языке Java, после создания процесса, выполнение главного потока начинается с метода main(). Затем, по мере необходимости, в заданных программистом местах, и при выполнении заданных им же условий, запускаются другие, побочные потоки.  
  
В языке Java поток представляется в виде объекта-потомка класса Thread. Этот класс инкапсулирует стандартные механизмы работы с потоком.

Создание потока

* Создать объект класса Thread, передав ему в конструкторе нечто, реализующее интерфейс Runnable. Этот интерфейс содержит метод run(), который будет выполняться в новом потоке. Поток закончит выполнение, когда завершится его метод run().
* Создать потомка класса Thread и переопределить его метод run():

В Java процесс завершается тогда, когда завершается последний его поток. Даже если метод main() уже завершился, но еще выполняются порожденные им потоки, система будет ждать их завершения.

Однако это правило не относится к особому виду потоков – демонам. Если завершился последний обычный поток процесса, и остались только потоки-демоны, то они будут принудительно завершены и выполнение процесса закончится.

* В Java существуют (существовали) средства для принудительного завершения потока. В частности метод Thread.stop() завершает поток незамедлительно после своего выполнения. Однако этот метод, а также Thread.suspend(), приостанавливающий поток, и Thread.resume(), продолжающий выполнение потока, были объявлены устаревшими и их использование отныне крайне нежелательно. Дело в том что поток может быть «убит» во время выполнения операции, обрыв которой на полуслове оставит некоторый объект в неправильном состоянии, что приведет к появлению трудноотлавливаемой и случайным образом возникающей ошибке.  
    
  Вместо принудительного завершения потока применяется схема, в которой каждый поток сам ответственен за своё завершение. Поток может остановиться либо тогда, когда он закончит выполнение метода run(), (main() — для главного потока) либо по сигналу из другого потока. Причем как реагировать на такой сигнал — дело, опять же, самого потока. Получив его, поток может выполнить некоторые операции и завершить выполнение, а может и вовсе его проигнорировать и продолжить выполняться. Описание реакции на сигнал завершения потока лежит на плечах программиста.

Класс Thread содержит в себе скрытое булево поле, подобное полю mFinish в программе Incremenator, которое называется флагом прерывания. Установить этот флаг можно вызвав метод interrupt() потока. Проверить же, установлен ли этот флаг, можно двумя способами. Первый способ — вызвать метод bool isInterrupted() объекта потока, второй — вызвать статический метод bool Thread.interrupted(). Первый метод возвращает состояние флага прерывания и оставляет этот флаг нетронутым. Второй метод возвращает состояние флага и сбрасывает его. Заметьте что Thread.interrupted() — статический метод класса Thread, и его вызов возвращает значение флага прерывания того потока, из которого он был вызван. Поэтому этот метод вызывается только изнутри потока и позволяет потоку проверить своё состояние прерывания.

Каждый поток в системе имеет свой приоритет. Приоритет – это некоторое число в объекте потока, более высокое значение которого означает больший приоритет. Система в первую очередь выполняет потоки с большим приоритетом, а потоки с меньшим приоритетом получают процессорное время только тогда, когда их более привилегированные собратья простаивают.

# Что такое cancellation point? Опишите как можно обходить такие точки

Cencellation point это инструкции, во время выполнения которых поток может сейфово прерваться без возникновения блокировки ресурсов и т.п. Возможно имеется в виду и высокоуровневое понимание (скажем когда поток выставляет условный флаг cancelable на инстуркции Х)

# Что такое InterruptedException в Java? Опишите, когда такие исключения нужно игнорировать

https://www.yegor256.com/2015/10/20/interrupted-exception.html

# Как работает проброс исключений между потоками в Java? (нужны детали реализации)

# Опишите алгоритм булочника. Зачем этот алгоритм может быть нужен?

### Аналогия

Лампорт предлагает рассмотреть пекарню с устройством, выдающим номерки у входа. Каждому входящему покупателю выдаётся номерок на единицу больше предыдущего. Общий счётчик показывает номер обслуживаемого в данный момент клиента. Все остальные покупатели ждут, пока не закончат обслуживать текущего клиента и табло покажет следующий номер. После того, как клиент сделает покупку и сдаст свой номерок, служащий увеличивает на единицу допустимые для выдачи устройством у входа номера. Если совершивший покупку клиент захочет снова что-нибудь купить, он должен будет снова взять номерок у входа и встать в общую очередь.

Пусть покупатели - это потоки, получившие номера *i* от глобальной переменной.

Из-за ограничений компьютерной архитектуры момент выдачи номерков должен быть немного модифицирован, так как возникает ситуация неоднозначности в случае, если сразу два или несколько покупателей (потоков) захотели получить номерок с номером *n*. При наличии нескольких потоков, получивших номер *n* при входе в критическую секцию, поток с меньшим номером *i* будет иметь больший приоритет при обслуживании (входе в критическую секцию).

### Критическая секция

Критическая секция — это часть кода, которая требует исключительного доступа к ресурсам, и одновременно может выполняться только одним потоком.

Когда поток хочет войти в критическую секцию, он должен проверить, может ли он это сейчас сделать. Поток должен проверить номера *n*, полученные другими потоками, и убедиться, что у него меньший номер. В случае совпадения *n* у двух или нескольких потоков, в критическую секцию входит поток с наименьшим *i*.

На псевдокоде это сравнение для потоков *a* и *b* может быть записано как:

# Что такое гонка данных как ошибка многопоточного программирования? Приведите несколько примеров

Слишком изи

# Опишите виды мьютексов и чем они отличаются в примерах

Виды мьютексов в posix:

* PTHREAD\_MUTEX\_NORMAL — при повтроной попытке захвата мьютекса владеющим потоком происходит deadlock[[14]](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81#cite_note-:1-14);
* PTHREAD\_MUTEX\_RECURSIVE — повторные захваты тем же потоком допустимы, ведётся подсчёт таких захватов[[14]](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81#cite_note-:1-14);
* PTHREAD\_MUTEX\_ERRORCHECK — попытка повторного захвата тем же потоком возвращает ошибку[[14]](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81#cite_note-:1-14).

# Что такое CAS операции? Как можно на них построить Lock-Free Stack если считать, что при выделении памяти всегда выдаются разные адреса

# Что такое условные переменные? Приведите паттерны использования в POSIX и Java

# Что такое грубая синхронизация? Опишите как она устроена на примере списка

https://alexeykalina.github.io/technologies/synchronization.html

Один мьютекс на все

# Что такое тонкая синхронизация? Опишите как она устроена на примере списка

Мьютекс на каждый элемент

# Что такое оптимистичная синхронизация? Опишите как она устроена на примере списка В данном методе синхронизации мы хотим избавиться от промежуточных блокировок на пути к искомому элементу. Для этого мы бежим по списку вообще без блокировок, пока не доберемся до нужного места, и только тогда блокируем пару элементов. Однако, до того мгновения, как мы повесили блокировки на найденные элементы, они могли быть физически удалены из сета. В этом то и заключается оптимистичность этого метода. Мы заново пробегаемся по списку (снова без блокировок) и, если достигаем нашей пары элементов, то она по-прежнему в сете и с ней уже ничего не произойдет. Тогда то мы и можем выполнить необходимую операцию. Иначе, повторяем весь процесс заново.

# Что такое ленивая синхронизация? Опишите как она устроена на примере списка

Заметим, что второго прохода по списку можно избежать. Для этого достаточно не удалять физически элементы сета, а помечать их как удаленные. Тогда после того, как блокировки будут установлены, останется проверить соответствующие флаги.

# Что такое неблокирующая синхронизация? Опишите как она устроена на примере списка?

Как ленивая, только юзаем CAS и удаляем элементы а не только помечаем

# Опишите что такое Lock-Free алгоритмы?

Грубо говоря без мьютексов.   
Вы работаете с несколькими потоками (прерываниями, обработчиками сигналов и т.д.)? — Да. — Потоки имеют доступ к разделяемой памяти? — Да. — Могут ли потоки блокировать друг друга (другими словами, можно ли выполнить планирование потоков так, чтобы они заблокировались на неопределенное время)? — Нет. — Это программирование без блокировок.

Если остановить любое количество потоков (не все), то система должна прогрессировать

Для алгоритмов без блокировок гарантируется системный прогресс по крайней мере одного потока. Например, поток, выполняющий операцию «сравнение с обменом» в цикле, теоретически может выполняться бесконечно, но каждая его итерация означает, что какой-то другой поток совершил прогресс, то есть система в целом совершает прогресс.

# Опишите что такое Wait-Free алгоритмы?

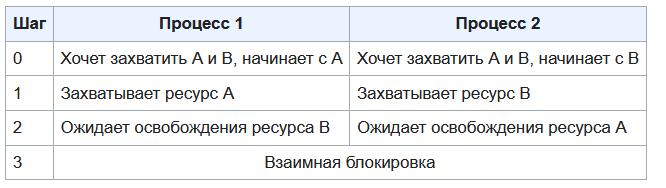
Если остановить любое количество потоков (не все), то система должна прогрессировать за определенное время.

Самая строгая гарантия прогресса. Алгоритм работает без ожиданий, если каждая операция выполняется за определённое количество шагов, не зависящее от других потоков.

# Опишите Lock-Free MRMW алгоритм снятие снепшота регистров

# Что такое deadlock? Приведите несколько примеров

Ситуация в многозадачной среде, при которой несколько процессов находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать свое выполнение.



# Что такое livelock? Приведите несколько примеров

Livelock означает такую ситуацию: система не «застревает» (как в обычной взаимной блокировке), а занимается бесполезной работой, её состояние постоянно меняется — но, тем не менее, она «зациклилась», не производит никакой полезной работы.

Жизненный пример такой ситуации: двое встречаются лицом к лицу. Каждый из них пытается посторониться, но они не расходятся, а несколько секунд сдвигаются в одну и ту же сторону.

# Что такое deadlock с использованием fork? Опишите этот случай

# Что такое ABA? Приведите пример, когда в stack с проблемой ABA могут исчезнуть вставленные элементы

В многозадачных вычислениях проблема ABA возникает при синхронизации, когда ячейка памяти читается дважды, оба раза прочитано одинаковое значение, и признак «значение одинаковое» трактуется как «ничего не менялось». Однако, другой поток может выполниться между этими двумя чтениями, поменять значение, сделать что-нибудь ещё и восстановить старое значение. Таким образом, первый поток обманется, считая, что не поменялось ничего, хотя второй поток уже разрушил это предположение.

# Опишите архитектуру инструментов поиска ошибок Valgrind, Google Sanitizers, Intel Parallel Studio

# Опишите алгоритм поиска ошибок в Google Thread Sanitizer. Приведите примеры в каких случая алгоритм справляется с поиском ошибок, а в каких нет

Вершинами графа являются мьютексы. Строится граф, в котором проводится направленное ребро в порядке захвата мьютексов. Если в графе существует цикл, то значит в коде возможен потенциальный deadlock.

Может не справиться если

# Что такое ORM?

ORM – это технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

# Что такое Future? Как они устроены изнутри

# Что такое CompleteableFuture? Как они устроены изнутри

# Что такое pull потоков? Какие пулы потоков бывают в Java?

# Примеры задач, решаемых с помощью Future