# Архитектурные стили

- Именованные концепции архитектурных решений
- Менее узкоспециализированные, чем архитектурные шаблоны

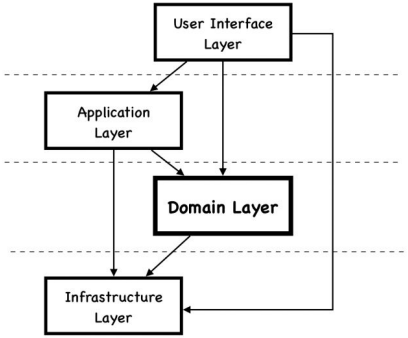
- Переиспользование архитектуры
  - Хорошо известные и изученные решения для разных задач
- Переиспользование кода
- Упрощение взаимодействия разработчиков
- Упрощение интеграции приложений
- Специфичные для стиля методы анализа
  - Возможны благодаря ограничениям на структуру системы
- Специфичные для стиля методы визуализации

# Монолитный стиль

- Каждый с каждым
- Теоретически, мы достигаем максимальной производительности в рамках одной машины

# Слоистый стиль

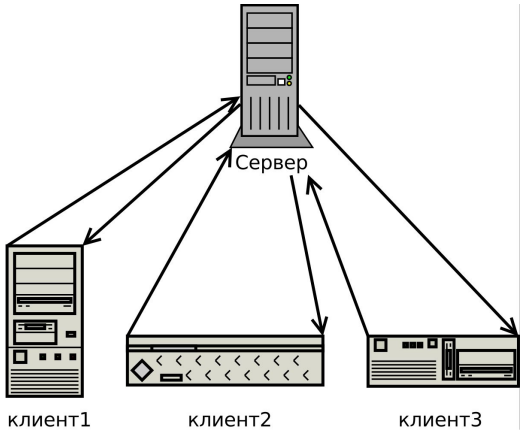
- Система делится на **слои**
  - **Слой взаимодействует с** соседними, либо в обе стороны, либо только в 1 сторону




\+ Повышение уровня абстракции
\+ Легкость расширения
\+ Изменения на каждом уровне затрагивают максимум 2 соседних
\+ Возможность последовательной разработки и отладки
\+ Возможны различные реализации уровня, совместимые с интерфейсами
\- Не всегда применим
\- Проблемы с производительностью

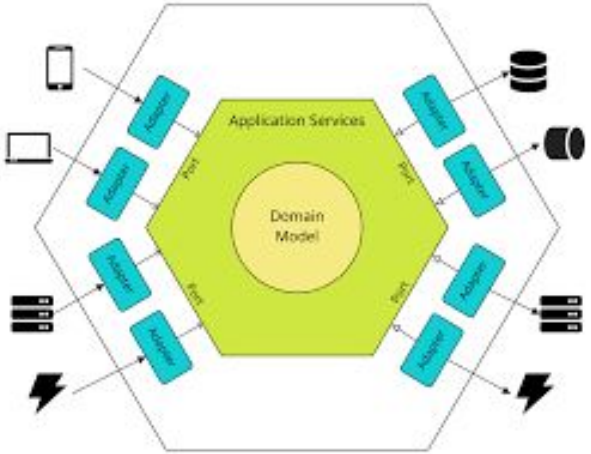
# Клиент-сервер

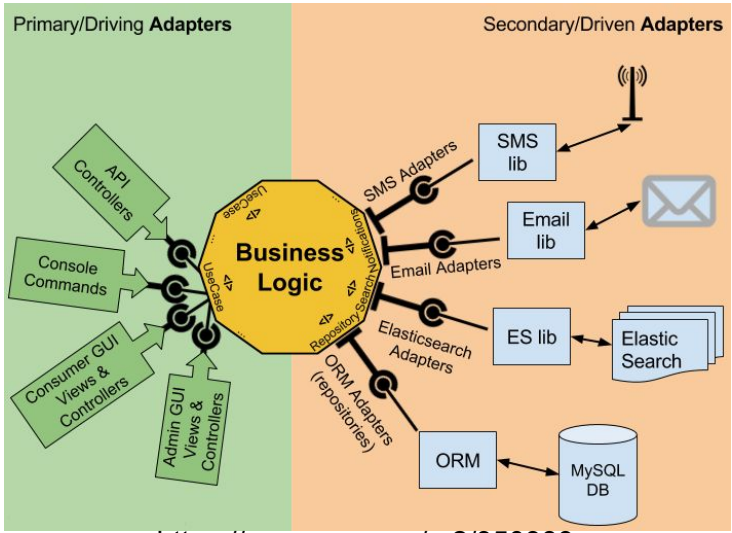
Компоненты -- клиенты и серверы
Серверы не знают о ещё не подключенных клиентах (как правило)
Клиенты не знают друг о друге
Соединители -- сетевые протоколы или просто IPC



# Гексагональная архитектура (“порты и адаптеры”)

+ Изоляция механизмов доставки
+ Изоляция вспомогательных механизмов
  - Библиотеки, и др.
+ Легкость тестирования, mocks
+ “Чистая” бизнес логика и модель предметной области
  + Максимальная простота, возможность валидации и преобразования данных
- Тяжеловесна (от необходимости отделять средства доставки от бизнес-логики)
- Неподробна
- Что делать с фреймворками?



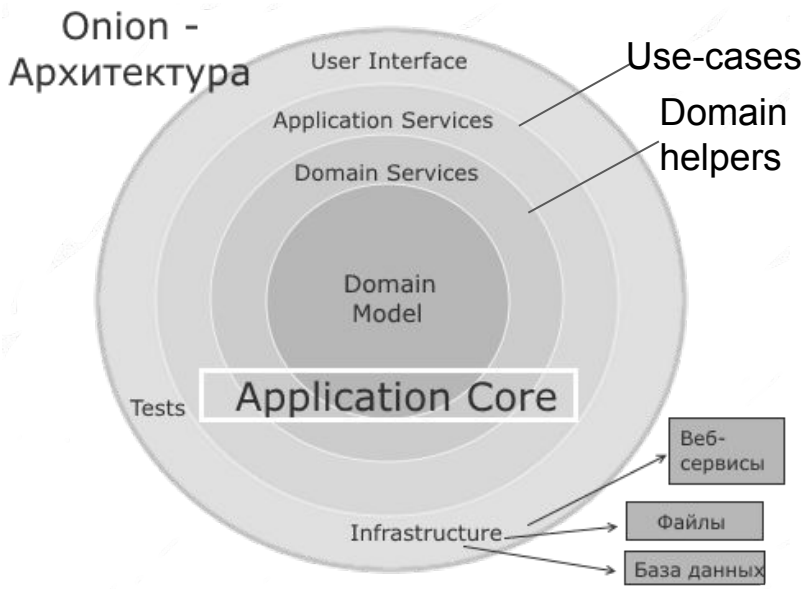




https://www.pvsm.ru/c-2/350389

# Луковая архитектура (уточнение гексагональной)

Определяет (уточняет) внутреннюю структуру ядра
Внутренние слои не знают о внешних, модель предметной области не знает ни о ком
Внутренние слои определяют интерфейсы, внешние -- их реализуют
Уровневость нестрогая -- слой может использовать все слои под ним
Репозитории, БД -- внешняя зависимость (в самом начале было не так)

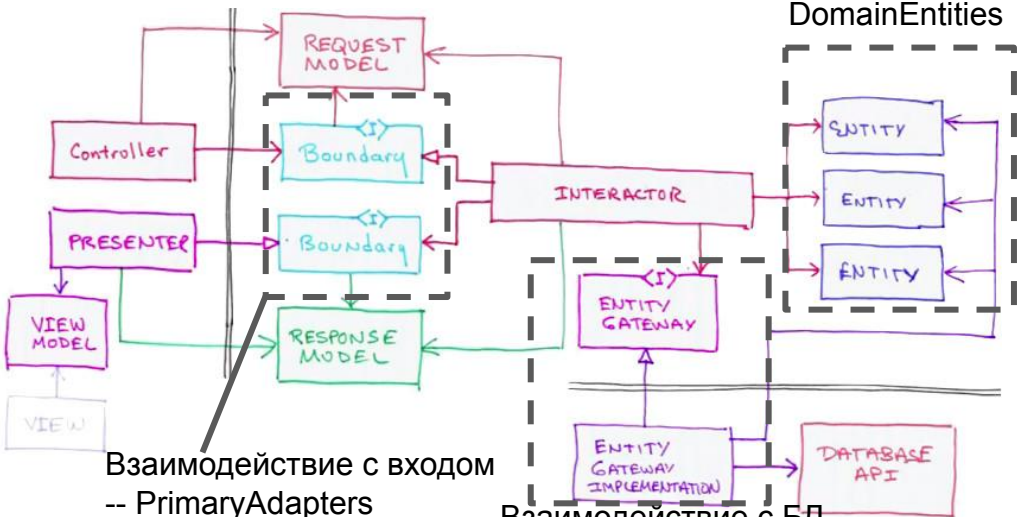
\+ Уточнение, большая ясность
\- Почти те же, что и у гексагональной



# “Чистая” архитектура (уточнение луковой)

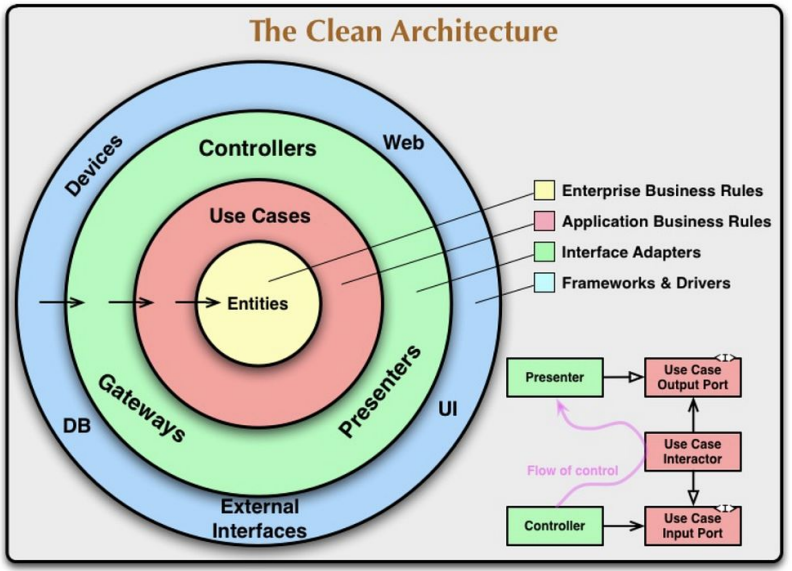
Определяет (уточняет) потоки управления
Добавляет описание набора граничных интерфейсов, которые описывают взаимодействие с внешним “не чистым” миром.

\+ Уточнение, большая ясность
\- Схема очень сложна




Взаимодействие с входом
-- PrimaryAdapters

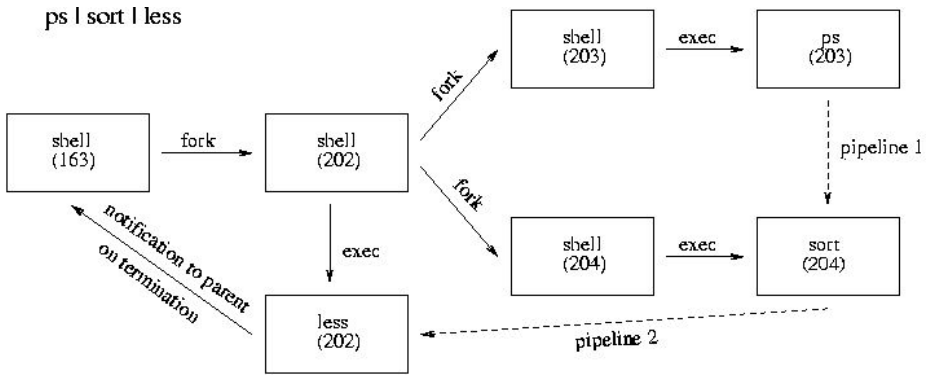
Взаимодействие с БД
-- SecondaryAdapters




https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
https://softwareengineering.stackexchange.com/questions/388578/clean-architecture-is-the-input-boundary-necessary

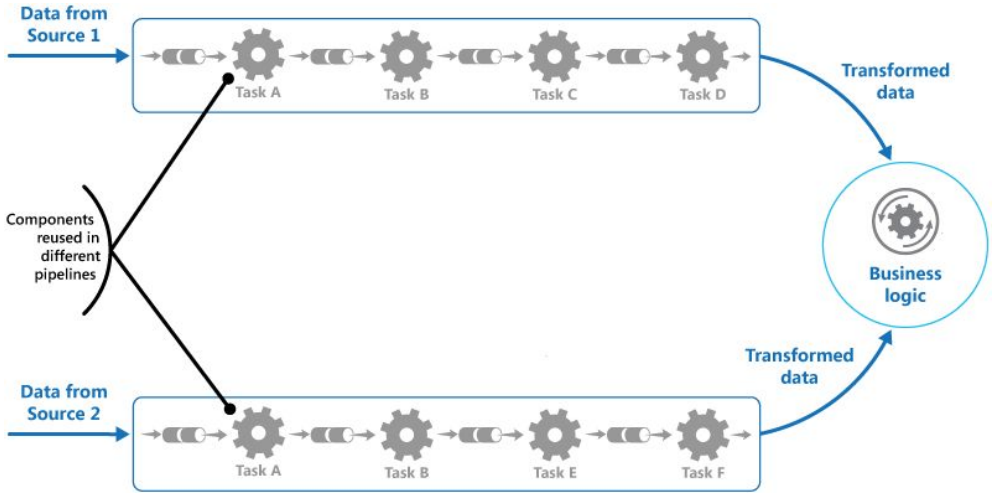
# Пакетная обработка, конвейер

- Вспомним конвейеры с прошлого семестра…



# Каналы и фильтры

Улучшение концепции пакетной обработки
Фильтры (как правило) не пересоздаются, а существуют
Фильтры (как правило) типизируемые, а каналы -- ограничены по пропускной способности

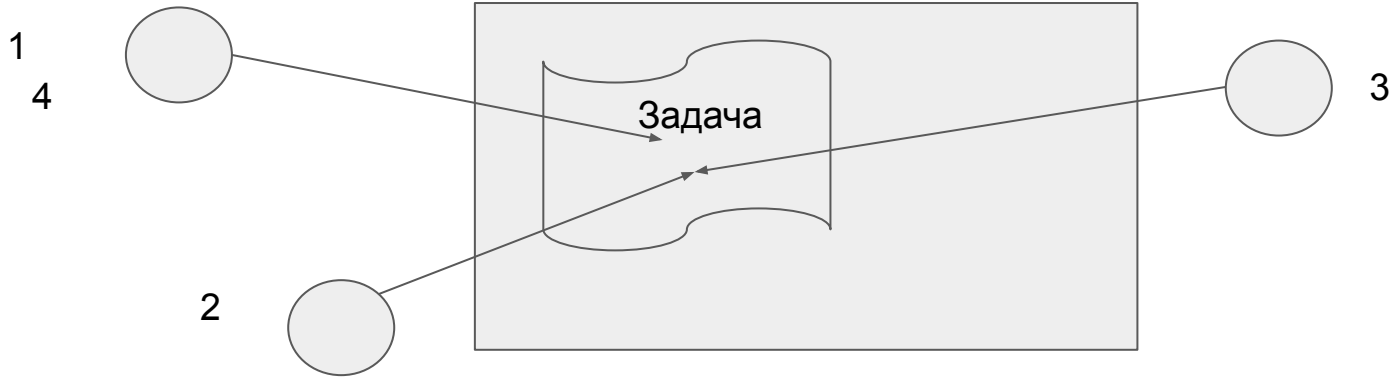



\+ Простота
\- Возможны сложности с откликом для пользователя
\- Узкие места определяют производительность всей системы

# Blackboard

Агенты без состояния обрабатывают центральную структуру данных, хранящую состояние, либо по порядку, либо пока есть что обрабатывать.
Используется в:

- Компиляторных оптимизаторах
- Графовых грамматиках
- Всяких решателях…



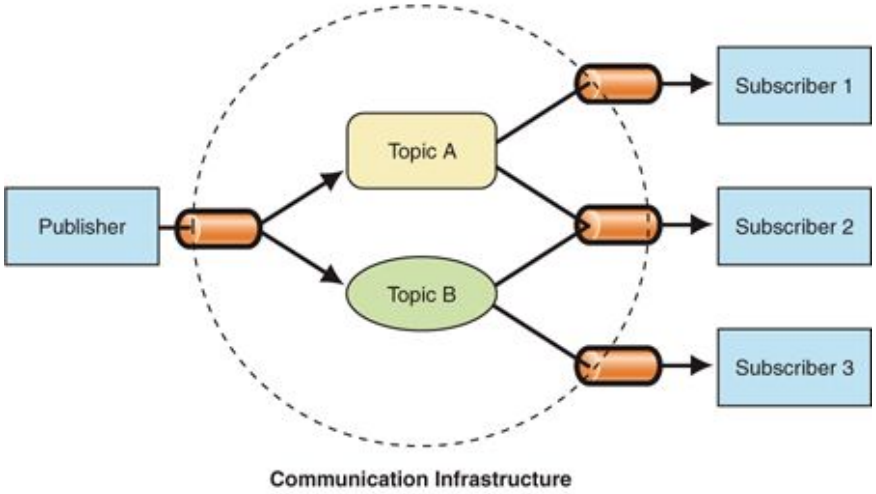
# Событийный стиль

- Оповещение о событии вместо явного вызова метода
  - Слушатети: подписываются
  - Система при наступлении события сама вызывает все зарегистрированные методы слушателей
- Компоненты имею 2 вида интерфейсов -- методы и события
- Соединения:
  - Явный вызов метода
  - Неявный вызов по наступлению события
- Инварианты:
  - Источники событий не знают, кто на них реагирует
  - Нет предположений о том, как обрабатывается событие (и будет ли обработано вообще)

\+ Переиспользуемость компонент через подписки
  \+ Очень низкая связность между компонентами
\- Неинтуитивная структура программы (для “сырого стиля”)
\- Компоненты не управляют последовательностью вычислений
\- Непонятно, кто реагирует на запрос
\- Тяжело отлаживать

# Событийный стиль: Push-subscribe

- Уточнение событийного стиля
  - Компоненты разделены на подписчиков и издателей




\+ Наводит порядок в событийном стиле
  \+ Очень низкая связность между компонентами
\- Вносит избыточность в случае, если напрашивается прямая взаимосвязь компонентов
\- Асинхронное поведение: всё равно тяжело отлаживать

# Событийная шина

Компоненты общаются между собой только по шине, по которой летят события. Это -- централизованное место (+ балансировщик нагрузки и маршрутизатор).

Компоненты могут не иметь своего состояния вообще, а строить его по сообщениями с шины.

Шина используется как единый источник истины для очень распределенной системы

# peer-to-peer

- Система из одинаковых компонентов, которые соединены сетевыми протоколами
- Соединения могут появляться и исчезать
  - Считается, что чем больше в сети динамики и хаоса, тем лучше …
- Множество одинаковых компонентов решает совместно одну задачу, кооперируясь между собой, например:
  - Раздают торрент (peer-2-peer -- раздача)
  - Детектируют лесной пожар (рой БПЛА)

# 3. Распределённые системы

# 3.1. Проблемы и нюансы разработки распределённых систем

# Проблемы и нюансы

- **Ни один из компонентов системы может не владеть общим знанием о состоянии системы в целом**
  - Алгоритмы функционирования распределённых систем сложны
  - Исключение -- системы / компоненты с выбором лидера
    - Лидер должен быть переизбираемым
    - На переизбирание лидера не должно уходить слишком много времени
- **Обмен данными**
  - Передача сообщений
  - Распределенная общая память, распределенные хранилища
- **При работе с распределёнными ресурсами необходима синхронизация**
  - Алгоритмы синхронизации должны учитывать распределённость
    - Либо выбор лидера
    - Либо token-based
  - Должны быть обеспечены гибкость, масштабируемость, безопасность, живучесть
- **Взаимодействие между компонентами -- асинхронное**, ответ может быть недетерминированным по времени
  - Как правило, всё же устанавливают верхнюю границу -- timeout
- **Как отмерять время?**
  - Сервера с физическим временем
  - Логические часы

# Знание о состоянии системы

- **Ни один из компонентов системы может не владеть общим знанием о состоянии системы в целом**
  - Алгоритмы функционирования распределённых систем сложны
  - Исключение -- системы / компоненты с выбором лидера -- компонента, который знает о системе больше, чем остальные, и занимается координацией её деятельности.
    - Лидер должен быть переизбираемым
    - На переизбирание лидера не должно уходить слишком много времени

**Общие соображения:**

Если все процессы / компоненты абсолютно идентичны, детерминированного способа выбрать лидера среди них нет. Поэтому:

1) Пусть процессы пронумерованы
2) Лидером нужно назначать процесс с наибольшим номером.
3) Каждый процесс знает номера всех остальных процессов
4) Неизвестно, какие процессы работают, а какие -- нет.

# Выбор лидера

**Ограничения**

- Работа алгоритмов голосования предполагает, что процессы, не являющиеся инициаторами, заняты своими задачами и принимать сообщения о голосовании -- не их основная задача.
- Если процессу приходит сообщение “голосование”, он должен быть оповещён каким-то сигналом, и начать участвовать в голосовании.
- Считаем, что каналы связи не рвутся, а вот процессы могут падать и запускаться
- Во время голосования процессы не падают.
- Задержки в каналах связи недетерминированы, передача сообщений асинхронная.
  - Но у времени ответа на запрос есть timeout.

**Простейшие подходы:**

1) Bully algorithm (Алгоритм задиры)
2) Кольцевой алгоритм (Чана и Робертса)
3) …

# Bully algorithm

**Описание**

- Процесс, обнаруживающий “смерть” лидера, инициирует голосование, рассылая сообщение процессам с большими номерами.
  - Если ответ не получен за timeout, сам становится лидером
  - Если получает в ответ “ОК” -- отходит в сторону. Процессы, ответившие “ОК” -- становятся задирами.

Следовательно, процесс с наибольшим номером, не получивший ответа за timeout становится лидером.

**Достоинства / недостатки**

+ Гарантия выбора лидера за счёт свойств порядка на номерах процессов и верхнего ограничения времени доставки.
- Слишком много сообщений гуляет по сети (верхняя оценка -- “каждый-к-каждому”)

# Кольцевой алгоритм (Чана и Робертса)

**Описание**

Процессы пронумерованы и соединены в кольцо по возрастанию номеров.

Пусть процесс P обнаружил, что лидер не отвечает.

1. Процесс P посылает сообщение “голосование” следующему по кольцу, добавив в него свой номер
2. Если ответа “ОК” нет, P посылает сообщение “голосование” с добавленным своим номером следующему по кругу процессу.
3. Если процесс получил сообщение “голосования”, он отвечает “ОК”.
   Если его нет в списке процессов сообщения, то переход на п.1.
4. Если есть, то он определяет координатора и посылает по кругу сообщение “координатор”, указав номер координатора и свой номер.
5. При завершении круга “координатор” дальше не передаётся.

**Достоинства / недостатки**

+ Гарантия выбора лидера за счёт свойств топологии и верхнего ограничения времени доставки.
+ Задержка на выбор может быть большей, чем для bully algorithm

# Взаимоисключение

- **Классические алгоритмы для защиты критических секций, рассмотренные в прошлом семестре, опирались на концепции разделяемой памяти и атомарных инструкций.**

- **Отсутствие общей памяти в распределенных системах не позволяет использовать классические способы для решения задач обеспечения взаимоисключения**
  - Распределенные алгоритмы должны опираться только на обмен сообщениями между процессами / компонентами распределенного приложения

- **Разработка алгоритмов осложняется тем, что приходится иметь дело с:**
  - Произвольными задержками передачи сообщений
  - Отсутствием информации о состоянии всей системы

**Требования к алгоритмам обеспечения взаимоисключения:**

- Процессы выполняются и взаимодействуют асинхронно, ни один из процессов и каналов связи не выходит из строя, все процессы связаны между собою по сети
- Нет предлопожений о времени выполнения процессов и их количестве
- Выполняется свойство безопасности*
- Выполняется свойство живучести**

***Свойства безопасности (safety properties)** утверждают, что нечто нежелательное никогда не случится в ходе работы ПО.
****Свойства живучести (liveness properties)** утверждают, наоборот, что нечто желательное при любом развитии событий произойдет в ходе его работы.

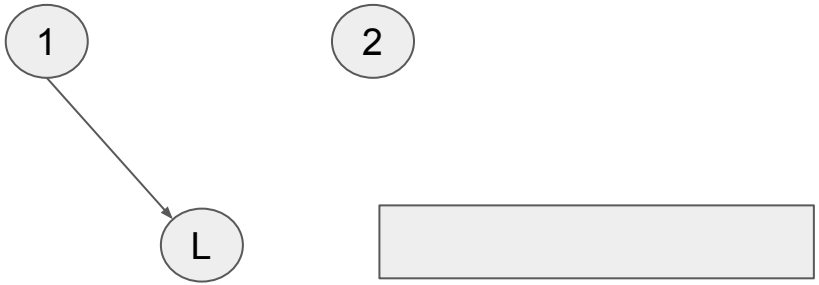
# Взаимоисключение: виды алгоритмов

- **Алгоритмы на основе получения разрешений (permission-based).**
  - Для входа в критическую секцию процессу необходимо собрать “достаточное количество разрешений” от других процессов.
  - Централизованный алгоритм, в котором право на вход выдаёт выделенный лидер -- также подвид такого вида алгоритмов

- **Алгоритмы на основе передачи маркера (token-based)**
  - Право на вход в критическую секцию -- у того, кто владеет маркером, который в каждый момент времени либо эксклюзивно принадлежит одному процессу, либо гуляет по сети.

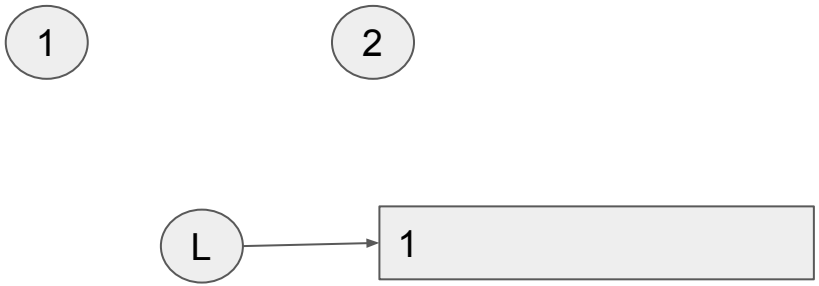
# Взаимоисключение на основе получения разрешений

- **Централизованный алгоритм.**
  - В системе есть переизбираемый лидер
  - Процесс 1, который хочет попасть в критическую секцию, обращается к лидеру “ask”, который вносит его в очередь на доступ и отвечает “access”, если в очереди только 1.
  - В конце деятельности процесс 1 отправляет “release” лидеру, лидер удаляет 1 из очереди.
  - Если очередь не пуста, лидер отправляет “access” процессу на вершине очереди.



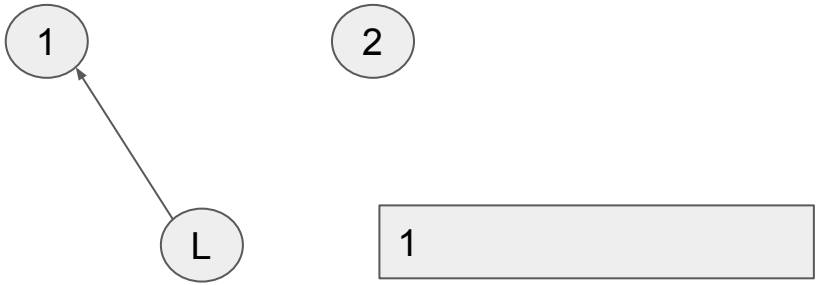
# Взаимоисключение на основе получения разрешений

- **Централизованный алгоритм.**
  - В системе есть переизбираемый лидер
  - Процесс 1, который хочет попасть в критическую секцию, обращается к лидеру “ask”, который вносит его в очередь на доступ и отвечает “access”, если в очереди только 1.
  - В конце деятельности процесс 1 отправляет “release” лидеру, лидер удаляет 1 из очереди.
  - Если очередь не пуста, лидер отправляет “access” процессу на вершине очереди.



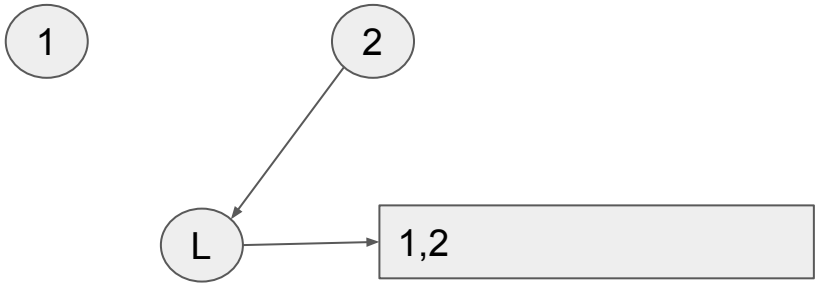
# Взаимоисключение на основе получения разрешений

- **Централизованный алгоритм.**
  - В системе есть переизбираемый лидер
  - Процесс 1, который хочет попасть в критическую секцию, обращается к лидеру “ask”, который вносит его в очередь на доступ и отвечает “access”, если в очереди только 1.
  - В конце деятельности процесс 1 отправляет “release” лидеру, лидер удаляет 1 из очереди.
  - Если очередь не пуста, лидер отправляет “access” процессу на вершине очереди.



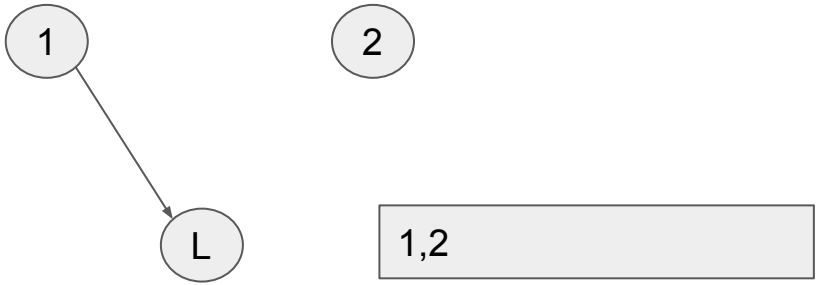
# Взаимоисключение на основе получения разрешений

- **Централизованный алгоритм.**
  - В системе есть переизбираемый лидер
  - Процесс 1, который хочет попасть в критическую секцию, обращается к лидеру “ask”, который вносит его в очередь на доступ и отвечает “access”, если в очереди только 1.
  - В конце деятельности процесс 1 отправляет “release” лидеру, лидер удаляет 1 из очереди.
  - Если очередь не пуста, лидер отправляет “access” процессу на вершине очереди.



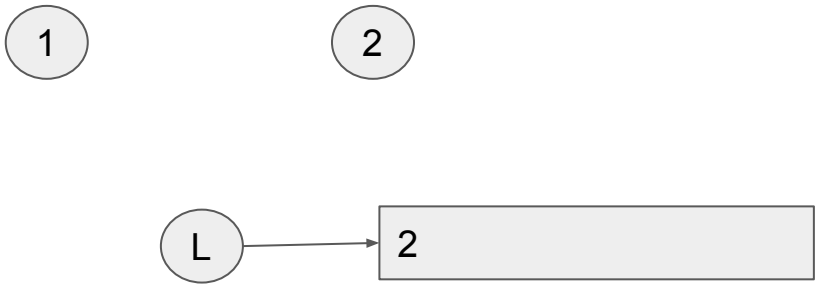
# Взаимоисключение на основе получения разрешений

- **Централизованный алгоритм.**
  - В системе есть переизбираемый лидер
  - Процесс 1, который хочет попасть в критическую секцию, обращается к лидеру “ask”, который вносит его в очередь на доступ и отвечает “access”, если в очереди только 1.
  - В конце деятельности процесс 1 отправляет “release” лидеру, лидер удаляет 1 из очереди.
  - Если очередь не пуста, лидер отправляет “access” процессу на вершине очереди.



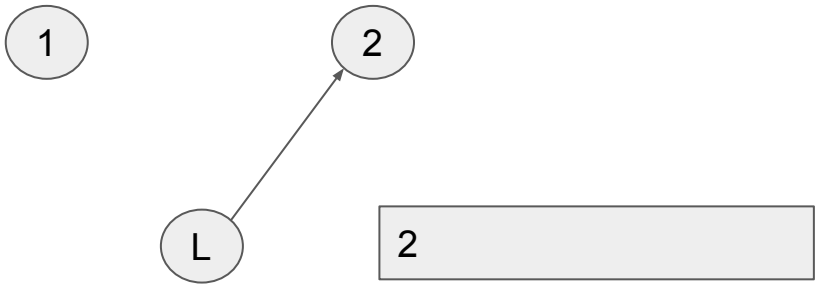
# Взаимоисключение на основе получения разрешений

- **Централизованный алгоритм.**
  - В системе есть переизбираемый лидер
  - Процесс 1, который хочет попасть в критическую секцию, обращается к лидеру “ask”, который вносит его в очередь на доступ и отвечает “access”, если в очереди только 1.
  - В конце деятельности процесс 1 отправляет “release” лидеру, лидер удаляет 1 из очереди.
  - Если очередь не пуста, лидер отправляет “access” процессу на вершине очереди.



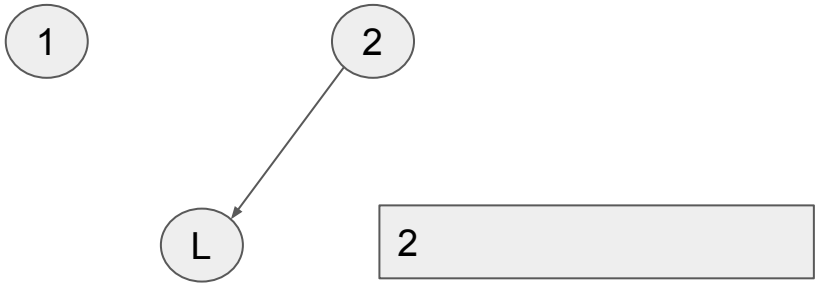
# Взаимоисключение на основе получения разрешений

- **Централизованный алгоритм.**
  - В системе есть переизбираемый лидер
  - Процесс 1, который хочет попасть в критическую секцию, обращается к лидеру “ask”, который вносит его в очередь на доступ и отвечает “access”, если в очереди только 1.
  - В конце деятельности процесс 1 отправляет “release” лидеру, лидер удаляет 1 из очереди.
  - Если очередь не пуста, лидер отправляет “access” процессу на вершине очереди.



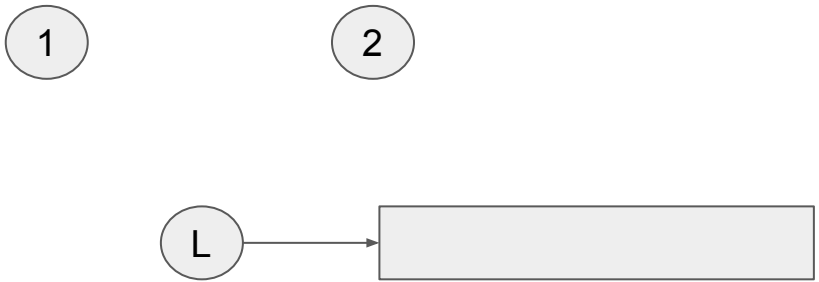
# Взаимоисключение на основе получения разрешений

- **Централизованный алгоритм.**
  - В системе есть переизбираемый лидер
  - Процесс 1, который хочет попасть в критическую секцию, обращается к лидеру “ask”, который вносит его в очередь на доступ и отвечает “access”, если в очереди только 1.
  - В конце деятельности процесс 1 отправляет “release” лидеру, лидер удаляет 1 из очереди.
  - Если очередь не пуста, лидер отправляет “access” процессу на вершине очереди.



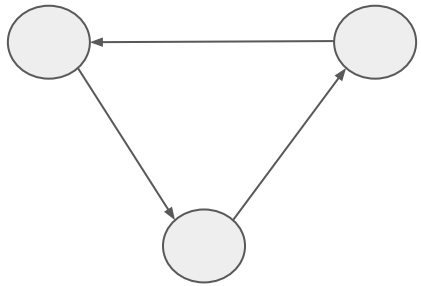
# Взаимоисключение на основе получения разрешений

- **Централизованный алгоритм.**
  - В системе есть переизбираемый лидер
  - Процесс 1, который хочет попасть в критическую секцию, обращается к лидеру “ask”, который вносит его в очередь на доступ и отвечает “access”, если в очереди только 1.
  - В конце деятельности процесс 1 отправляет “release” лидеру, лидер удаляет 1 из очереди.
  - Если очередь не пуста, лидер отправляет “access” процессу на вершине очереди.



# Взаимоисключение на основе получения маркера

- **Кольцевой алгоритм.**
  - В системе есть порядок процессов по кольцу
    - Так уменьшается количество передаваемых сообщений
  - Процесс, который получил маркер, должен отослать ОК источнику маркера и имеет право пройти в КС.
  - Если процесс не готов пройти в КС, но получил маркер, он передаёт его дальше по кольцу. В ответ он должен получить ОК, либо переслать следующему.
  - В предположении, что каналы не разрываются, до любого процесса в сети рано или поздно маркер будет доставлен.



# Время

- **Время на разных машинах разное и течёт по-разному относительно других машин (даже несмотря на UTS и всякие ухищрения)**
  - В особенности потому что для обмена информацией о времени между машинами нужно сходить в сеть
- **Для различных задач важно синхронизироваться по времени**
  - И даже вводятся сервера времени
- **Но для многих задач физическое время не принципиально**
  - И может использоваться абстракция логического времени -- временных меток
  - Либо векторного времени

# Логическое время (Л. Лэмпорт, 1979)

**Так как синхронизировать все узлы полностью невозможно, на множестве событий вводится отношение частичного порядка.** Часы Лэмпорта присваивают каждому событию единственное число, монотонно увеличивая счётчик каждого процесса согласно следующим правилам:

- счётчик увеличивается перед каждым внутренним событием процесса;
- при отправке сообщения значение счётчика прикрепляется к сообщению;
- при получении сообщения значение счётчика процесса-получателя выставляется в максимум (текущего и полученного) значения и увеличивается на 1.

**Доказывается (но не нами), что часы Лэмпорта непротиворечивы, то есть сохраняют инвариант частичного порядка множества событий в системе**

# Векторное время (К. Фридж, 1988)

**Векторные часы —** алгоритм получения частичного упорядочения событий в распределённой системе и обнаружения нарушений причинно-следственных связей:

- Таким же образом, как и во временных метках Лэмпорта, внутренние сообщения, передаваемые в системе, содержат состояние логических часов процесса.

Векторные часы в системе из N процессов — массив или вектор из N логических часов, одни часы на процесс. Локальный экземпляр вектора с наименьшими возможными значениями часов для каждого процесса строится следующим образом:

- изначально все значения часов равны 0;
- в случае внутреннего события счётчик текущего процесса увеличивается на 1;
- перед отправкой сообщения внутренний счётчик, соответствующий текущему процессу, увеличивается на 1, и вектор целиком прикрепляется к сообщению;
- при получении сообщения счётчик текущего процесса увеличивается на 1, далее значения в текущем векторе выставляются в максимум от текущего и полученного.

**Векторные часы также непротиворечивы.**