Программирование распределенных систем

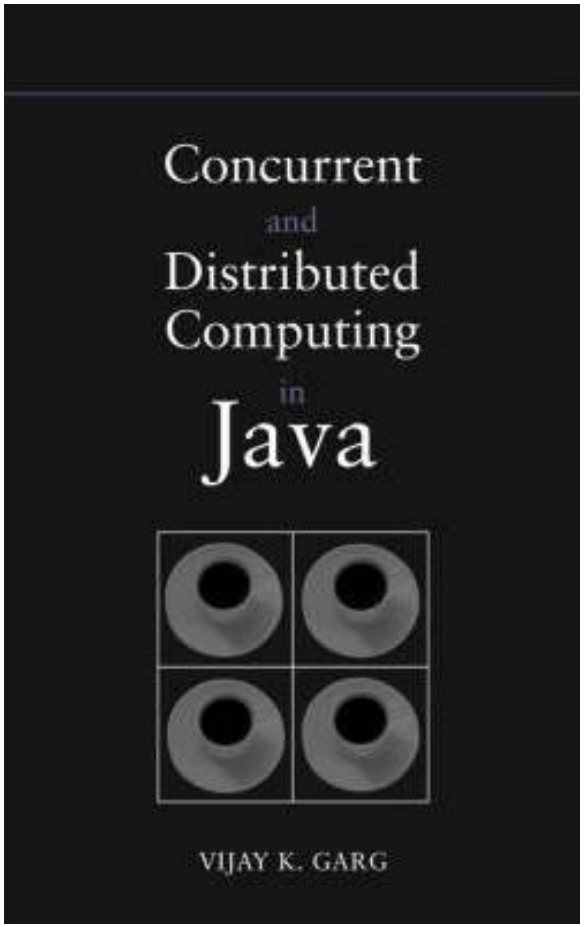
Роман Елизаров, 2022

elizarov@gmail.com

Лекция 1

ВВЕДЕНИЕ, ЛОГИЧЕСКОЕ ВРЕМЯ И ВЗАИМНОЕ ИСКЛЮЧЕНИЕ

Литература



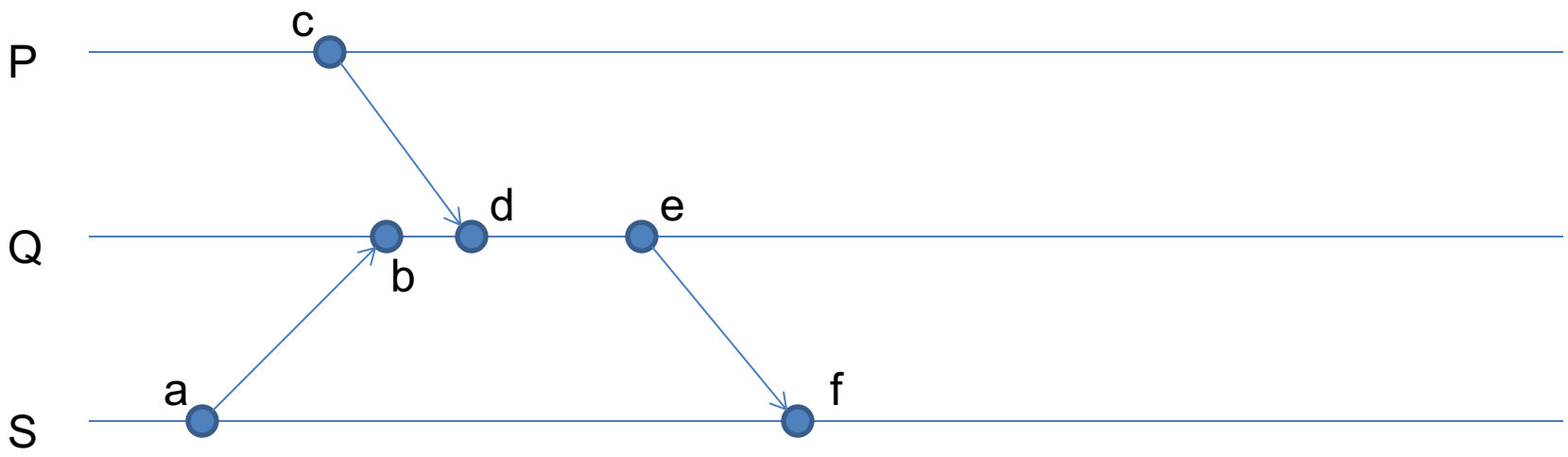
Сравнение: Concurrent Vs Distributed

Параллельная система	Распределенная система
SMP/NUMA система с общей памятью	Кластер из машин в сети
Вертикальная масштабируемость	Горизонтальная масштабируемость
Чтение/запись общей памяти	Посылка/получение сообщений
Централизованная	Возможно географически распределена
Обычно однородная	Часто гетерогенная
Взаимодействие всех со всеми	Возможно неполная *топология* связей
Определено общее время	Нет понятия общего времени
Определено состояние системы	Нет понятия общего состояния системы
Отказ всей системы в целом	Частичный отказ системы
Меньше надежность	Больше надежность
Проще программировать	Сложней программировать
Больше стоимость оборудования	Меньше стоимость оборудования

Модель

- Процессы $P, Q, R, \ldots \in \mathbf{P}$
- События $a, b, c, d, e, f, g, \ldots \in \mathbf{E}$, в процессах $proc(e) \in \mathbf{P}$
- Сообщения $m \in \mathbf{M}$, события посылки/приема $snd(m), rcv(m) \in \mathbf{E}$
- DEF: Отношение произошло-до между событиями ($\rightarrow$)
  - Такой минимальный *строгий частичный порядок* (транзитивное, антирефлексивное, антисимметричное отношение), что:
    - Если e и f произошли в одном процессе и e шло перед f (обозначаем как e<f), то $e \rightarrow f$
    - Если m сообщение, то $snd(m) \rightarrow rcv(m)$
  - То есть, это транзитивное замыкание порядка событий на процессе и посылки/приема сообщений

Графическая нотация



- Здесь стрелки задают посылки сообщений (рисуем направо!)
- a и c не связаны отношением произошло-до
  - Такие события называются *параллельными*
- c произошло до f (из транзитивности)

Логические часы

- Для каждого события e определим число C(e) так, что:

$$\forall e, f \in \mathbf{E} : e \to f \Rightarrow C(e) < C(f)$$

- DEF: Такую функцию C будем называть логическими часами
- В обратную сторону отношение не верно и не может быть верно, ибо отношение порядка на числах задает полный порядок, а отношение произошло-до на событиях - частичный

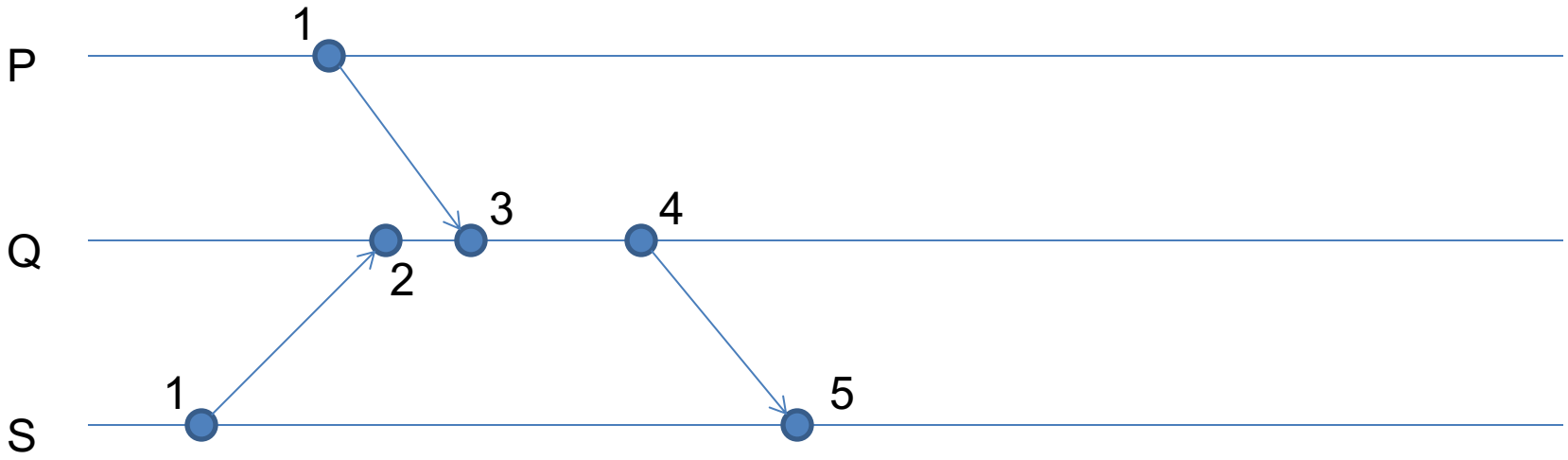
Логические часы Лампорта

- Время это целое число C в каждом процессе
- Алгоритм
  - Перед каждой посылкой сообщения процесс увеличивает часы на единицу

    C:= C+ 1
  - При посылке сообщение процесс посылает свое время C вместе с сообщением
  - При приеме сообщения делаем

    C := max(received_C, C) + 1

Логические часы Лампорта (2)



- Здесь указано время события (после увеличения переменной время и после операции max при приеме)
  - Время события не уникально!
  - Выполняется определение логических часов!

Векторные часы

- Для каждого события e определим вектор VC(e) так, что:

$$\forall e, f \in \mathbf{E} : e \to f \Leftrightarrow VC(e) < VC(f)$$

- Сравнение векторов происходит покомпонентно
- DEF: Такую функцию VC будем называть векторными часами

Алгоритм векторного времени

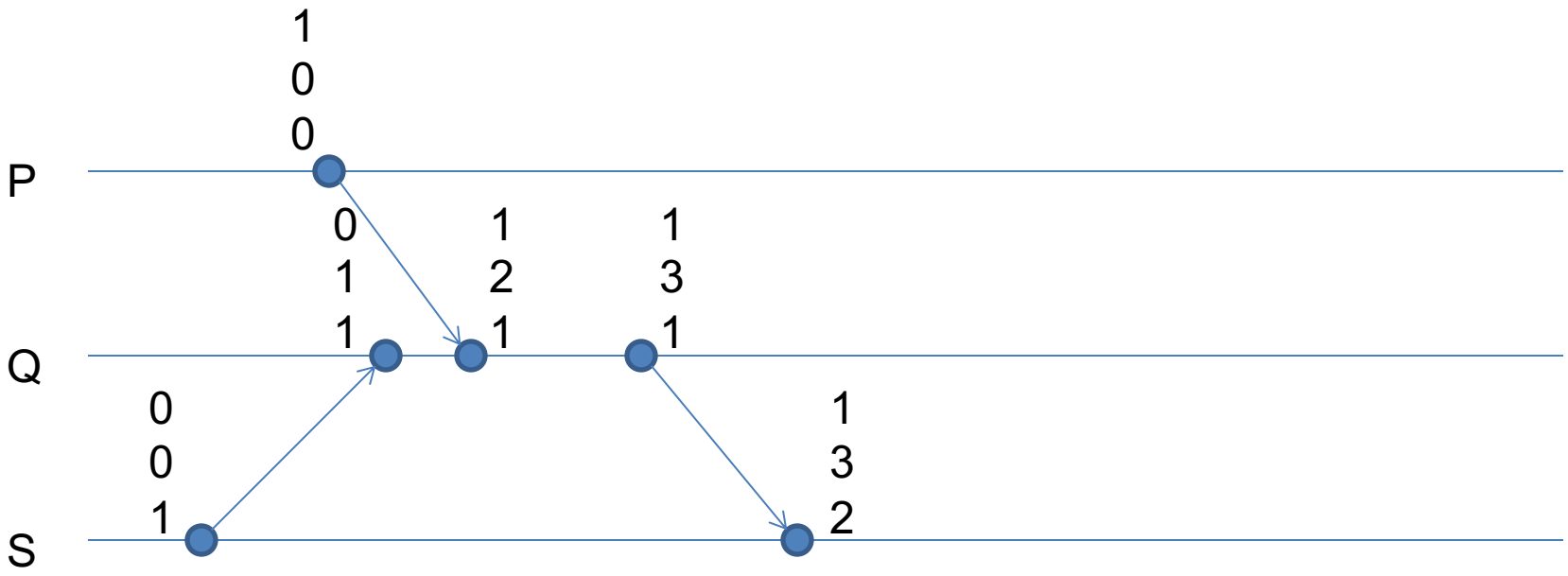
- Время это целое вектор VC в каждом процессе
  - Размер вектора это количество процессов
- Алгоритм
  - Перед каждой посылкой или приемом сообщения процесс увеличивает свой компонент в векторе времени на единицу

    $VC_i := VC_i + 1$
  - При посылке сообщение процесс посылает свой вектор времени вместе с сообщением
  - При приеме сообщение делаем покомпонентно

    VC:= max(received_VC, VC)
- ВАЖНО

$$\forall e, f \in E : proc(e) = P_i, proc(f) = P_j : e \to f \Leftrightarrow \begin{pmatrix} VC(e)_i \\ VC(e)_j \end{pmatrix} < \begin{pmatrix} VC(f)_i \\ VC(f)_j \end{pmatrix}$$

Алгоритм векторного времени (2)



- Здесь указано время события (после его обработки)
  - Векторное время уникально для каждого события
  - Полностью передает отношение произошло-до

Другие часы

- Часы с прямой зависимостью (direct dependency)

$$e \rightarrow_d f \overset{def}{\Leftrightarrow} e < f \vee \exists m \in \mathbf{M} : e \leq snd(m) \,\&\, rcv(m) \leq f$$

где

$$\forall e, f \in \mathbf{E} : e \rightarrow_d f \Leftrightarrow VC_d(e) < VC_d(f)$$

  - Алгоритм это комбинация Лампорта и Векторного (храним вектор, посылаем одно число)
- Матричные часы (храним матрицу, посылаем матрицу)

Взаимное исключение

Взаимное исключение в распределенных системах

- Критическая секция CS_i состоит из двух событий:
  - Enter(CS_i) вход в критическую секцию
  - Exit(CS_i) выход из критической секции
  - Здесь i это порядковый номер захода в критическую секцию
- Основное требование: ***взаимное исключение***
  - Два процесса не должны быть в критической сессии *одновременно*, то есть

$$Exit(CS_i) \rightarrow Enter(CS_{i+1})$$

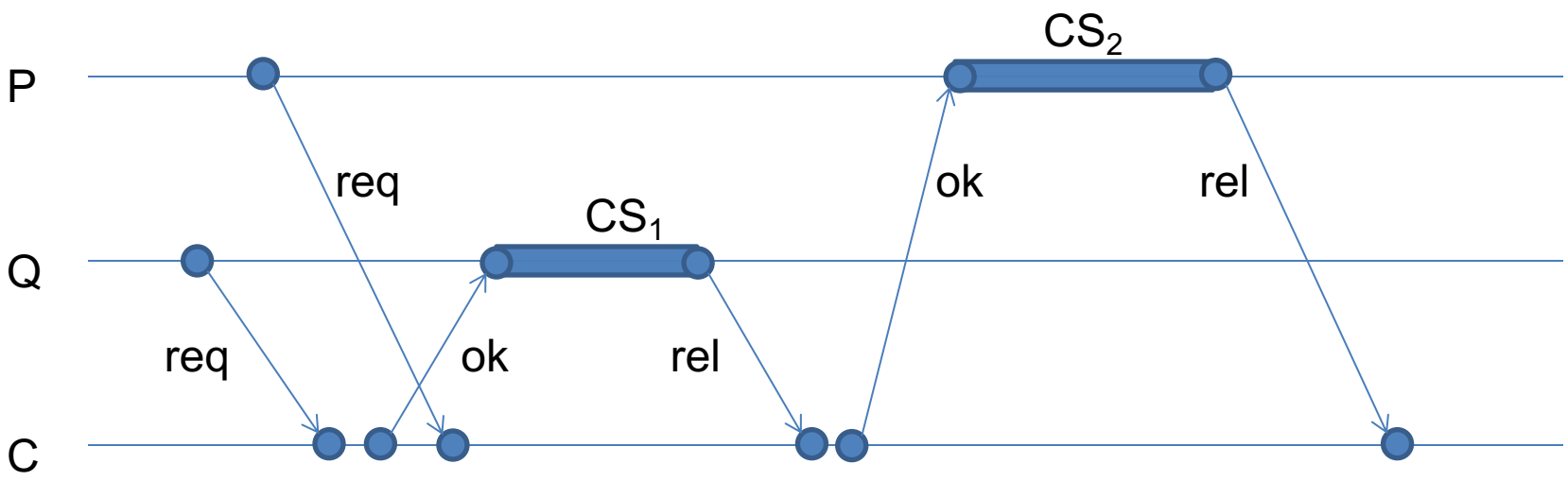
Взаимное исключение в распределенных системах (2)

- Так же как в системах с общей памятью нужны дополнительные требования ***прогресса***.
  - Минимальное требование прогресса заключается в том, что каждое желание процесса попасть в критическую секцию будет рано или поздно удовлетворено
  - Так же может быть гарантирован тот или иной уровень честности удовлетворения желания процессов о входе в критическую секцию

Централизованный алгоритм

- Выделенный координатор
- Три типа сообщений:
  - req[uest] (от запрашивающего процесса координатору)
  - ok (от координатора запрашивающему)
  - rel[ease] (после выхода из критической секции)
- Требует 3 сообщение на критическую секцию независимо от количества участвующих процессов
- Но не масштабируется из-за необходимости иметь выделенного координатора

Централизованный алгоритм (2)

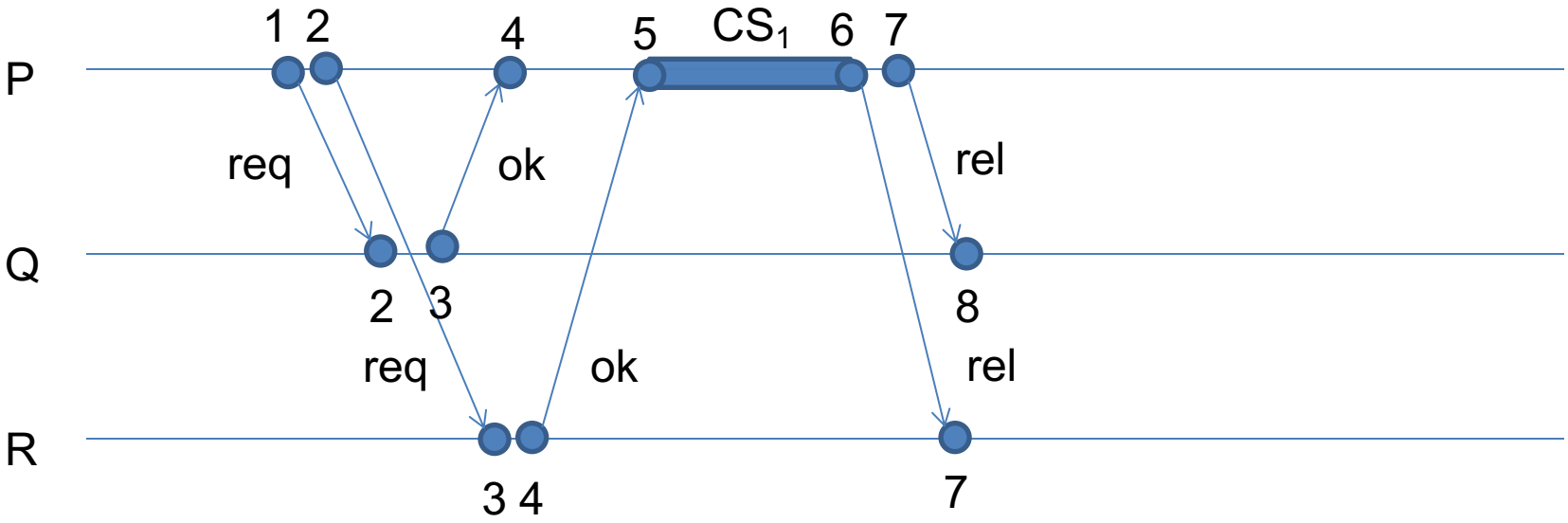


Алгоритм Лампорта

- Не нужен координатор – полностью распределенный
- Три типа сообщений (всего 3N-3 сообщения на CS):
  - req[uest] (от запрашивающего процесса всем другим)
  - ok (ответ-подтвержение – высылается сразу)
  - rel[ease] (после выхода из критической секции)
- Алгоритм использует логические часы Лампорта
  - И работает только если между процессами сообщения идут FIFO
- Все запросы хранятся в очередь, можно войти в CS если:
  - Мой запрос первый в очереди (при одинаковом времени упорядочиваем по id процесса)
  - Получено сообщение от каждого другого процесса с большим временем («заявка подтверждена»)

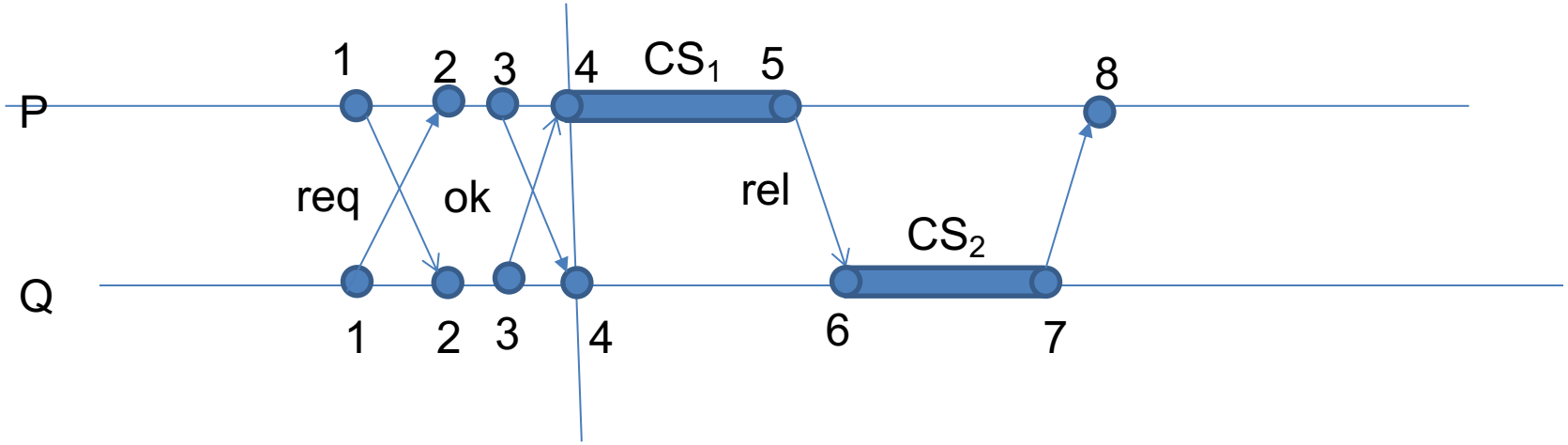
Алгоритм Лампорта (2)

Пример одной критической секции, три процесса



Алгоритм Лампорта (3)

Пример двух процессов, одновременно посылающих запрос



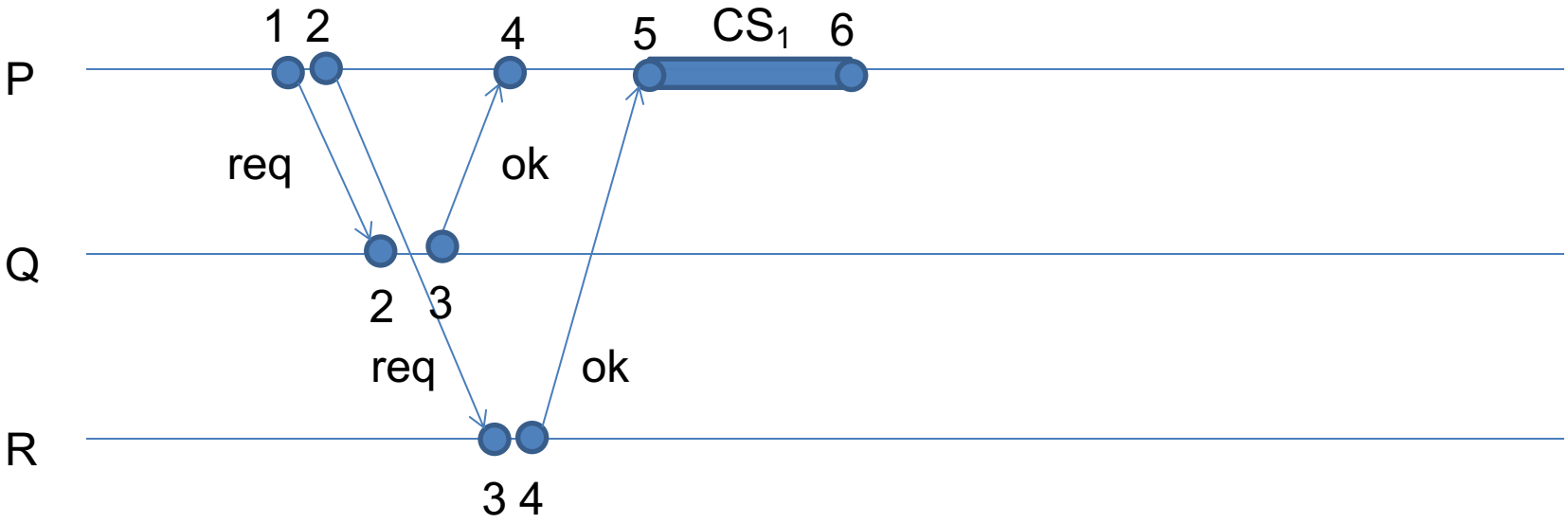
Здесь оба процесса имеют две подтвержденные заявки с временем “1” в очереди, но процесс P имеет более высокий приоритет

Алгоритм Рикарда и Агравалы

- Оптимизация алгоритма Лампорта
- Два типа сообщений (всего 2N-2 сообщения на CS):
  - req[uest] (от запрашивающего процесса всем другим)
  - ok (после выхода из критической секции)
- Здесь ok и rel объединены. Шлем ok только если не хотим сами войти в критическую секции или наш запрос имеет более поздний номер очереди

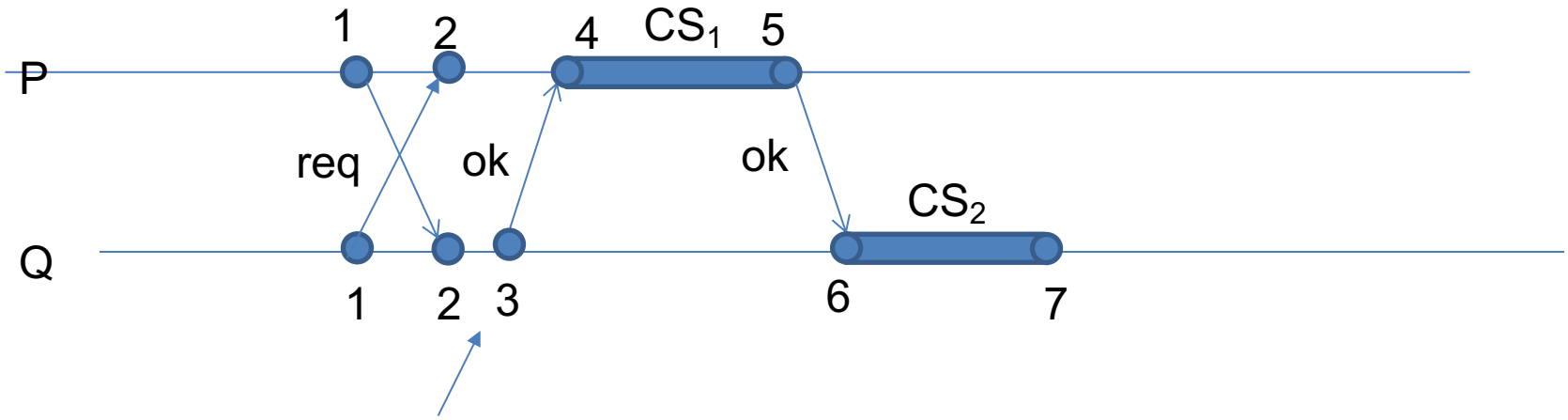
Алгоритм Рикарда и Агравалы (2)

Пример одной критической секции, три процесса



Алгоритм Рикарда и Агравалы (3)

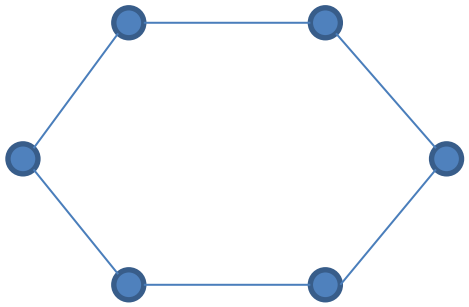
Пример двух процессов, одновременно посылающих запрос



Только это процесс посылает ok здесь, так как видит что его собственный запрос позже в очереди

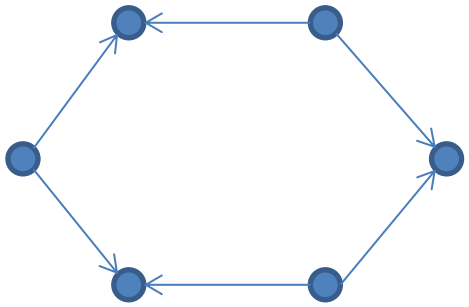
Алгоритм обедающих философов

- План:
  - Научимся решать задачу об обедающих философах (процессы – философы, ресурсы – вилки) в распределенной системе
  - Потом обобщим решение на произвольный *граф конфликтов*
- *Граф конфликтов* задачи об обедающих философах это цикл (пример для N=6):



Алгоритм обедающих философов (2)

- Ориентируем граф конфликтов так, чтобы в нем не было циклов
- **Теорема 1**: В ориентированном графе без циклов всегда есть исток
- **Теореме 2**: Если у истока перевернуть все ребра, то граф останется ациклическим



Алгоритм обедающих философов (3)

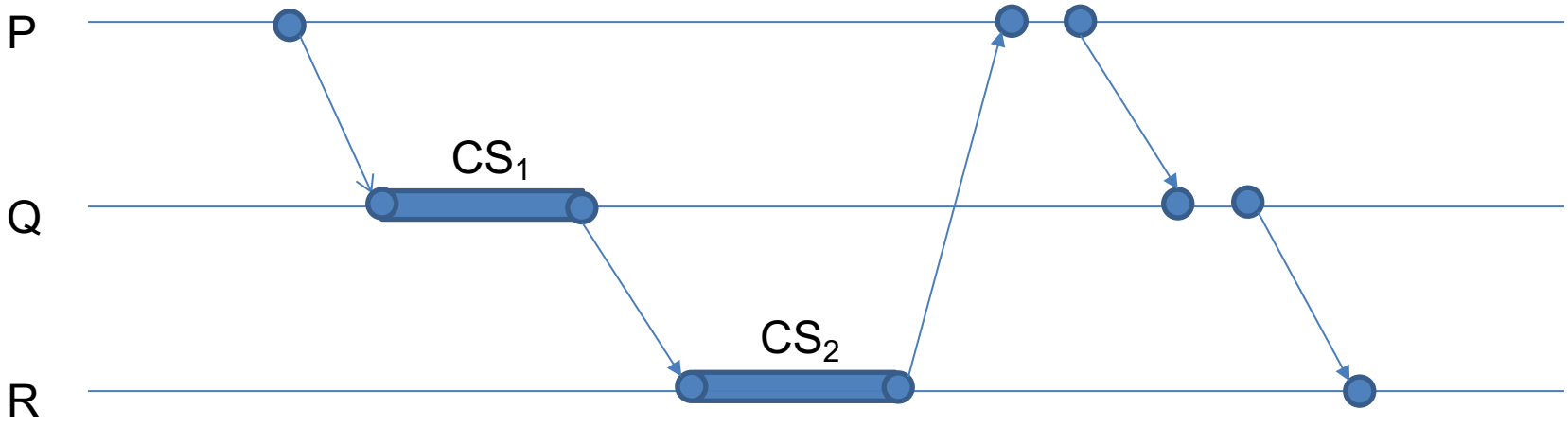
- У философа есть вилка == ребро в графе *направлено от него*
- Философ может есть, если он исток (у него есть обе вилки)
- После еды он должен отдать вилки, но мы не будет тратить сообщения на их передачу
  - После еды пометим вилки как грязные
  - Будет мыть вилки (делать чистыми) и отдавать их другому философу при запросе, даже если сами хотим есть
  - Но чистые вилки отдавать не будем, если хотим есть (ждем пока у нас обе вилки, едим, они грязные, после этого моем и отдает если был запрос)

Алгоритм обедающих философов - обобщение

- Обобщаем
  - Взаимное исключение == полный граф конфликтов (вилка для каждый пары процессов)
  - Вначале раздадим вилки например по результатам сравнения id процессов
- Получаем
  - 0 сообщений на повторный заход в CS одним процессом
  - 2N-2 сообщения в худшем случае
  - Количество сообщений пропорционально числу процессов, которые хотят попасть в критическую секцию

Алгоритм на основе токена

- Передает токен по кругу
- Входит в критическую секцию только процесс с токеном



Алгоритмы на основе кворума

- Определение кворума
  - Семейство подмножеств множества процессов $Q \subset 2^P$
  - Любые два кворума имеют непустое пересечение

$$\forall A, B \in Q : A \cap B \neq \emptyset$$

- Варианты кворума
  - Централизованный алгоритм как частный случай кворума
  - Простое большинство и взвешенное большинство
  - Рушащиеся стены

- Проблемы кворума
  - Потенциальный deadlock при пересечении кворумов
  - Решение – иерархическая блокировка