Архитектура компьютера

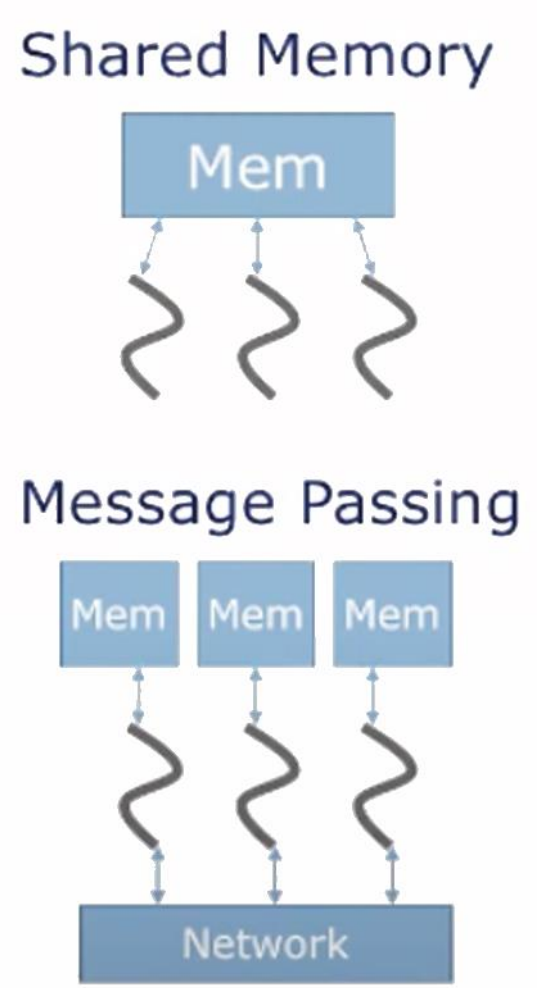
Синхронизация. Когерентность кэш

План лекции

- Параллелизм уровня потоков
- Потоково-безопасное программирование
- Мультипроцессорность
- Когерентность кэш
  - VI, MSI, MESI

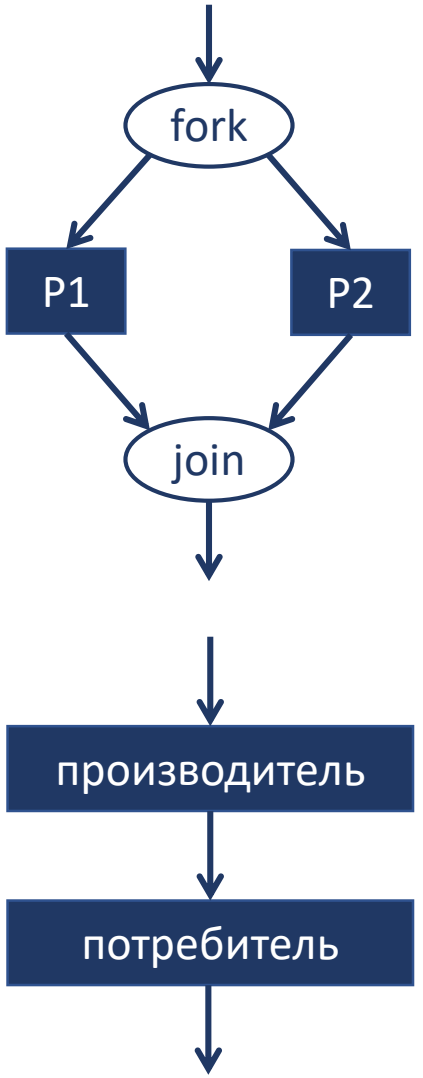
Параллелизм уровня потоков

- Разделение вычислений между несколькими исполнительными потоками
  - Несколько независимых последовательных потоков, которые конкурируют за общие ресурсы, такие как память, устройство ввода/вывода
  - Несколько взаимодействующих последовательных потоков, которые взаимодействуют друг с другом
- Коммуникационная модель
  - Общая память
    - Единое адресное пространство
    - Неявная связь с помощью загрузки и сохранения в память
  - Обмен сообщениями
    - Разделенное адресное пространство
    - Явная связь путем отправки и получения сообщений



Синхронизация

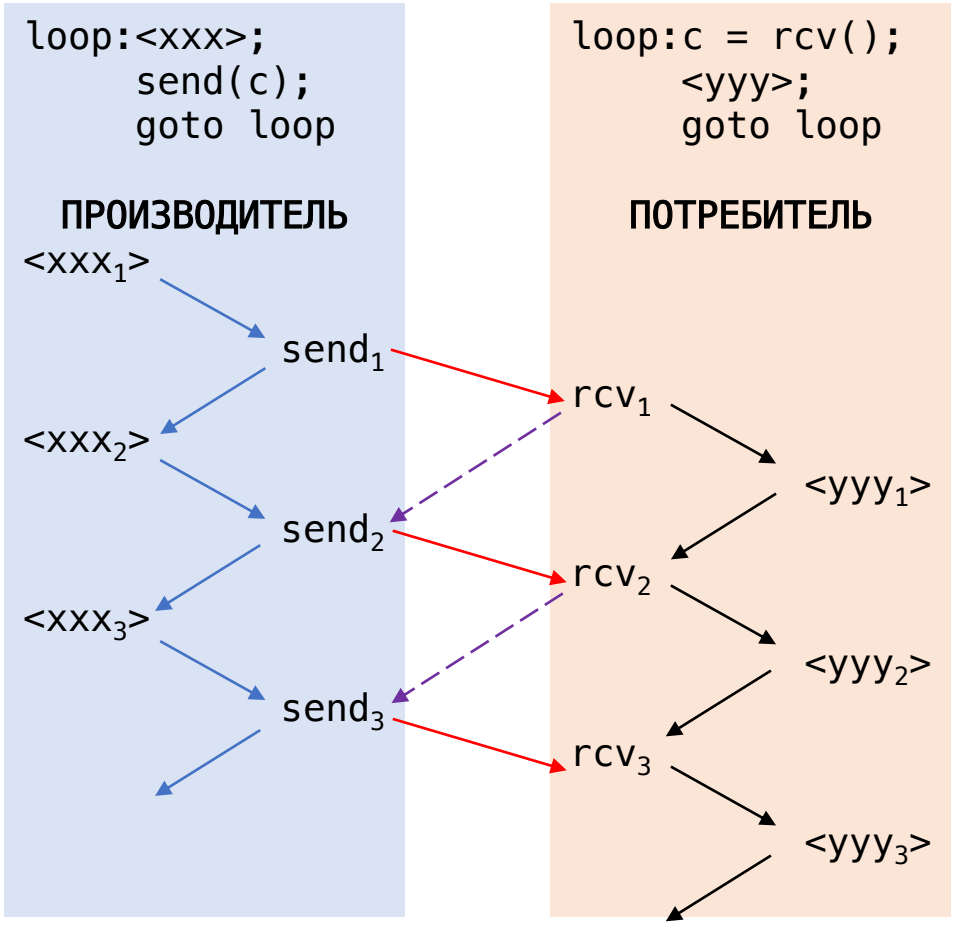
- Необходимость в синхронизации возникает каждый раз, когда в системе существуют параллельные процессы
  - Вилки и соединения (join and fork): параллельный процесс может подождать, пока не произойдет несколько событий
  - Производитель-потребитель (producer-consumer): потребительский процесс должен ждать, пока процесс производителя не произведет данные
  - Взаимное исключение: операционная система должна гарантировать, что ресурс используется только одним процессом в данный момент времени



Потоково-безопасное программирование

- Многопоточные программы могут выполняться на одном процессоре с помощью таймшеринга
  - Каждый поток выполняется некоторое время (прерывание по таймеру), а затем ОС переключается на другой поток, неоднократно
- Потоково-безопасные многопоточные программы ведут себя одинаково независимо от того, выполняются ли они на нескольких процессорах или на одном процессоре
  - Мы будем предполагать, что каждый поток имеет свой собственный процессор для запуска

Синхронная связь



Приоритет очередности

$a \le b$

a предшествует **b**

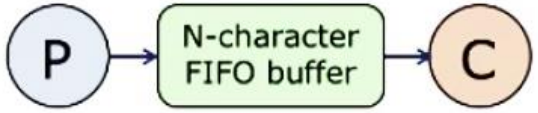
- Потребитель не может использовать данные до их получения

$$send_i \le rcv_i$$

- Производитель не может перезаписать данные до их использования потребителем

$$rcv_i \le send_{i+1}$$

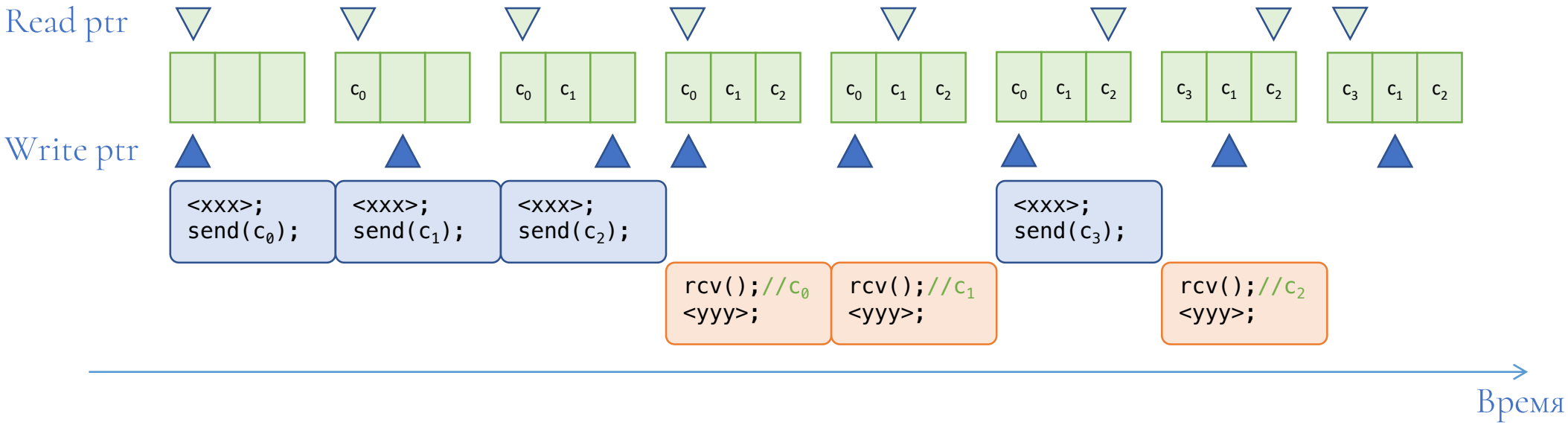
Буфер FIFO



- Буфер FIFO ослабляет ограничения, связанные с синхронизацией. Производитель может опередить потребителя на N значений

$$rcv_i \le send_{i+1}$$

Обычно реализуется как кольцевой буфер в общей памяти



Буфер FIFO с общей памятью

ОБЩАЯ ПАМЯТЬ:

```
char buf[N];          // кольцевой буфер
int  in = 0, out = 0;
```

ПРОИЗВОДИТЕЛЬ:

```
void send(char c) {
    buf[in] = c;
    in = (in + 1) % N;
}
```

ПОТРЕБИТЕЛЬ:

```
char rcv() {
    char c;
    c = buf[out];
    out = (out + 1) % N;
    return c;
}
```

- Работает неправильно. Почему?
- Не применяет никаких ограничений приоритета (например, rcv() может быть вызван до любой отправки)

Семафоры

- Программная конструкция для синхронизации
  - Новый тип данных: semaphore, число `≥ 0`
  - `semaphore s = K; // инициализируем s значением K`
- Новые операции (определены для семафоров)
  - `wait(semaphore s)`

    ждет если s == 0, после чего s = s – 1
  - `signal(semaphore s)`

    s = s + 1 (один ожидающий поток теперь может быть в состоянии продолжить)
- Семантические гарантии: семафор s инициализированный как K, применяет ограничение приоритета
  - $\texttt{signal(s)}_{i} < \texttt{wait(s)}_{i+K}$

i-ый вызов signal(s) должен завершиться до завершения (i + K) вызова wait(s)

Семафоры для приоритета

```
semaphore s = 0;

Thread A                Thread B

A1;                     B1;

A2;                     B2;
signal(s);
A3;                     B3;
                        wait(s);
A4;                     B4;

A5;                     B5;
```

Цель: необходимо, чтобы оператор A2 в потоке A был завершен до того, как начнется оператор B4 в потоке B

$$A2 < B4$$

Решение

- Объявить semaphore = 0
- signal(s) в начале стрелки
- wait(s) в конце стрелки

Семафоры для распределения ресурсов

Описание проблемы

- Существует K ресурсов
- Существует множество потоков, каждый из которых нуждается в ресурсах в случайные моменты времени
- Необходимо гарантировать, что в любой момент времени используется не более K ресурсов

Решение с использованием семафоров:

- В основной памяти:

```
semaphore s = K;   // K ресурсов
```

- Используемые ресурсы

```
wait(s);           // выделение ресурса
...                // использование его некоторое время
signal(s);         // возвращение ресурса
```

- Значение семафора == количество оставшихся ресурсов

Буфер FIFO с семафорами

ОБЩАЯ ПАМЯТЬ:

```
char buf[N];          // кольцевой буфер
int  in = 0, out = 0;
semaphore chars = 0;
```

ПРОИЗВОДИТЕЛЬ:

```
void send(char c) {
    buf[in] = c;
    in = (in + 1) % N;
    signal(chars);
}
```

ПОТРЕБИТЕЛЬ:

```
char rcv() {
    char c;
    wait(chars);
    c = buf[out];
    out = (out + 1) % N;
    return c;
}
```

- Приоритет, управляемый семафором: $send_i \le rcv_i$
- Ресурс, управляемый семафором: количество символов в буфере
- Все еще некорректная ситуация, так как производитель может переполнить буфер
- Должны применять требование $rcv_i \le send_{i+1}$

Буфер FIFO с семафорами

ОБЩАЯ ПАМЯТЬ:

```
char buf[N];          // кольцевой буфер
int  in = 0, out = 0;
semaphore chars = 0, spaces = N;
```

ПРОИЗВОДИТЕЛЬ:

```
void send(char c) {
    wait(spaces);
    buf[in] = c;
    in = (in + 1) % N;
    signal(chars);
}
```

ПОТРЕБИТЕЛЬ:

```
char rcv() {
    char c;
    wait(chars);
    c = buf[out];
    out = (out + 1) % N;
    signal(spaces);
    return c;
}
```

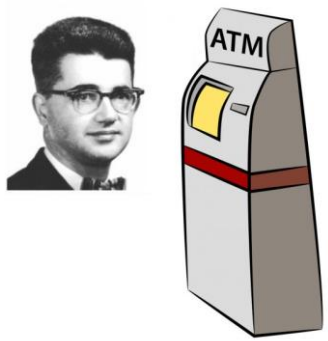
- Ресурсы, управляемые семафорами: символы в FIFO, свободное место в FIFO
- Работает с одним производителем и потребителем
- Что будет, если потребителей и производителей будет больше?

Одновременные транзакции

Предположим, что вы и ваш друг посещаете банкомат в одно и то же время и снимаете 100 рублей со своего счета.

Что происходит?

debit(0903, 100)



debit(0903, 100)

```
void debit(int account, int amount) {
    t = balance[account];
    balance[account] = t – amount;
}
```

Что должно произойти?

```
// предположим, что t0 содержит адрес balance[account]
```

Thread #1	Thread #2
`lw t1, 0(t0)`	
`sub t1, t1, a1`	
`sw t1, 0(t0)`	
…	…
	`lw t1, 0(t0)`
	`sub t1, t1, a1`
	`sw t1, 0(t0)`

Результат: у вас есть 200 рублей и баланс счета уменьшился на 200 рублей

Одновременные транзакции

```
// предположим, в t0 адрес balance[account]
```

Thread #1	Thread #2
`lw t1, 0(t0)`	
	`lw t1, 0(t0)`
	`sub t1, t1, a1`
	`sw t1, 0(t0)`
`sub t1, t1, a1`	
`sw t1, 0(t0)`	
…	…

Результат: у вас есть 200 рублей, а баланс счета уменьшился только на 100 рублей

Необходимо быть осторожным при написании параллельных программ. В частности, при изменении общих данных.

Для некоторых сегментов кода, называемых критическими секциями (critical sections) мы хотели бы убедиться, что никакие два выполнения не перекрываются.

Это ограничение называется взаимным исключением (mutual exclusion).

Решение: внедрить в критические секции в обертки, которые гарантируют их атомарность, то есть делают их похожими на отдельные мгновенные операции.

Семафоры для взаимных исключений

```
semaphore lock = 1;


void debit(int account, int amount) {
    wait(lock);    // ждем эксклюзивного доступа
    t = balance[account]
    balance[account] = t – amount;
    signal(lock); // конец блокировки
}
```

Блокировка управления доступом к критической секции

a <> **b**

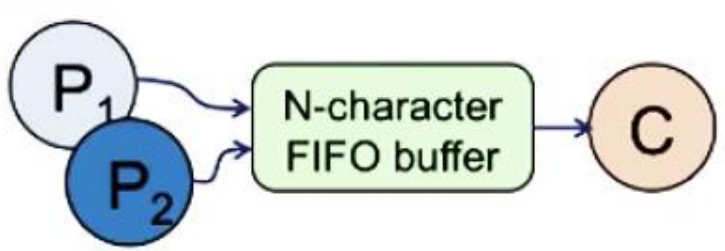
a предшествует **b** или

b предшествует **a**

(то есть они не пересекаются)

Проблемы атомарности

Рассмотрим несколько потоков производителей:



P_1 **P_2**

```
…                          …
buf[in] = c;               buf[in] = c;
in = (in+1) % N;           in = (in+1) % N;
…                          …
```

Проблема: производители мешают друг другу

Буфер FIFO с семафорами

ОБЩАЯ ПАМЯТЬ:

```
char buf[N];          // кольцевой буфер
int  in = 0, out = 0;
semaphore chars = 0, spaces = N;
semaphore lock = 1;
```

ПРОИЗВОДИТЕЛЬ:

```
void send(char c) {
    wait(spaces);
    wait(lock);
    buf[in] = c;
    in = (in + 1) % N;
    signal(lock);
    signal(chars);
}
```

ПОТРЕБИТЕЛЬ:

```
char rcv() {
    char c;
    wait(chars);
    wait(lock);
    c = buf[out];
    out = (out + 1) % N;
    signal(lock);
    signal(spaces);
    return c;
}
```

Мощность семафоров

ОБЩАЯ ПАМЯТЬ:

```
char buf[N];          // кольцевой буфер
int  in = 0, out = 0;
semaphore chars = 0, spaces = N;
semaphore lock = 1;
```

ПРОИЗВОДИТЕЛЬ:

```
void send(char c) {
    wait(spaces);
    wait(lock);
    buf[in] = c;
    in = (in + 1) % N;
    signal(lock);
    signal(chars);
}
```

ПОТРЕБИТЕЛЬ:

```
char rcv() {
    char c;
    wait(chars);
    wait(lock);
    c = buf[out];
    out = (out + 1) % N;
    signal(lock);
    signal(spaces);
    return c;
}
```

Единый примитив синхронизации, обеспечивающий

- Отношение приоритета
  - $send_i \le rcv_i$
  - $rcv_i \le send_{i+1}$
- Отношение взаимного исключения
  - Защита переменных in и out

Реализация семафоров

Семафоры сами по себе являются общими данными, и для реализации операций `wait` и `signal` требуются последовательности чтения / изменения / записи, которые должны выполняться как критические секции. Как мы можем гарантировать взаимное исключение в этих конкретных критических секциях без использования семафоров?

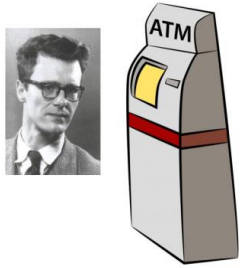
Подходы:

- Использовать специальную инструкцию (например, “`test and set`”), которая выполняет атомарное чтение–изменение–запись. Зависит от атомарности выполнения одной команды. Это самый распространенный подход
- Реализуется с помощью системных вызовов. Работает только в однопроцессорных системах, где ядро бесперебойно

Синхронизация: обратная сторона

Использование ограничений синхронизации может привести к возникновению собственного набора проблем, особенно когда потоку требуется доступ к нескольким защищенным ресурсам

```
void transfer(int account1, int account2, int amount) {
    wait(lock[account1]);
    wait(lock[account2]);
    balance[account1] = balance[account1] – amount;
    balance[account2] = balance[account2] + amount;
    signal(lock[account1]);
    signal(lock[account2]);
}
```



`transfer(0903, 1103, 100)`

Что может пойти не так?

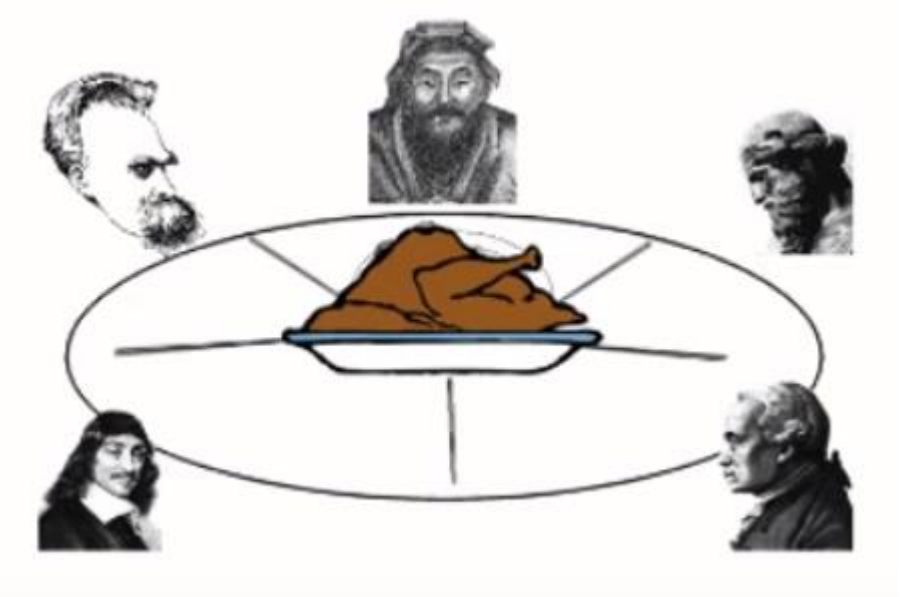
```
Thread 1: wait(lock[0903]);
Thread 2: wait(lock[1103]);
Thread 1: wait(lock[1103]);  // не завершиться, пока не произойдет signal 2 потока
Thread 2: wait(lock[0903]);  // не завершиться, пока не произойдет signal 1 потока
```

`transfer(1103, 0903, 100)`

Ни один поток не может добиться прогресса → Тупик (Deadlock)

Обедающие философы

Философы мыслят глубокими мыслями, но имеют простые мирские потребности. Когда вы голодны, группа из N философов будет сидеть вокруг стола с N палочками для еды, разбросанными между ними. Еда подается, и каждый философ наслаждается неторопливой едой, используя палочки для еды с обеих сторон.



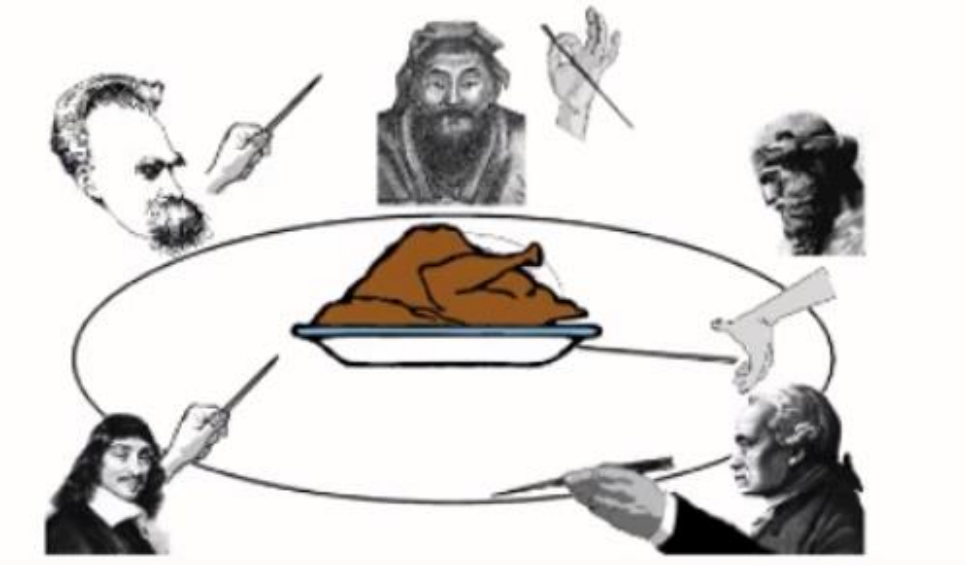
Они чрезвычайно вежливы и терпеливы, и каждый соблюдает обеденный протокол.

Алгоритм философа:

- Взять (дождаться) ЛЕВУЮ палочку
- Взять (дождаться) ПРАВУЮ палочку
- Кушать, пока не насытится
- Заменить обе палочки

Deadlock

Никто не может добиться прогресса, потому что все они ждут недоступного ресурса.

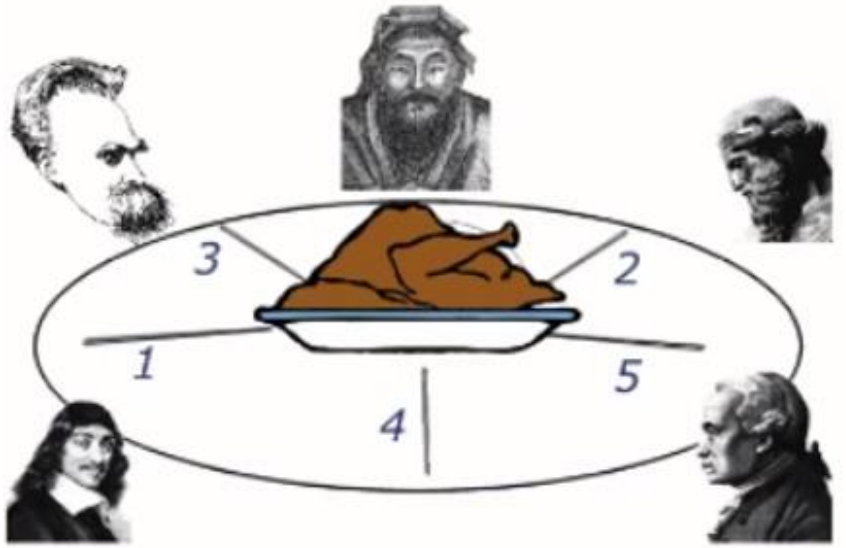


УСЛОВИЯ:

1) Взаимное исключение: только один поток может содержать ресурс в данный момент времени
2) Удержание и ожидание: поток удерживает выделенные ресурсы, ожидая других
3) Отсутствие вытеснения: ресурс не может быть удален из потока, удерживающего его
4) Круговое ожидание

Решение

Назначить уникальный номер каждой палочке для еды. Запрашивать ресурсы в последовательном порядке



Новый алгоритм:

- Взять палочку с МЕНЬШИМ номером
- Взять палочку с БОЛЬШИМ номером
- ЕСТЬ
- Заменить обе палочки

Простое доказательство

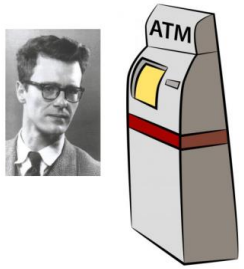
Deadlock означает, что каждый философ ждет ресурса, которым владеет какой-то другой философ.

Но философ, держащий самую большую палочку для еды не может ждать какого-либо другого философа

Пример

Можно ли исправить метод передачи, чтобы избежать тупика?

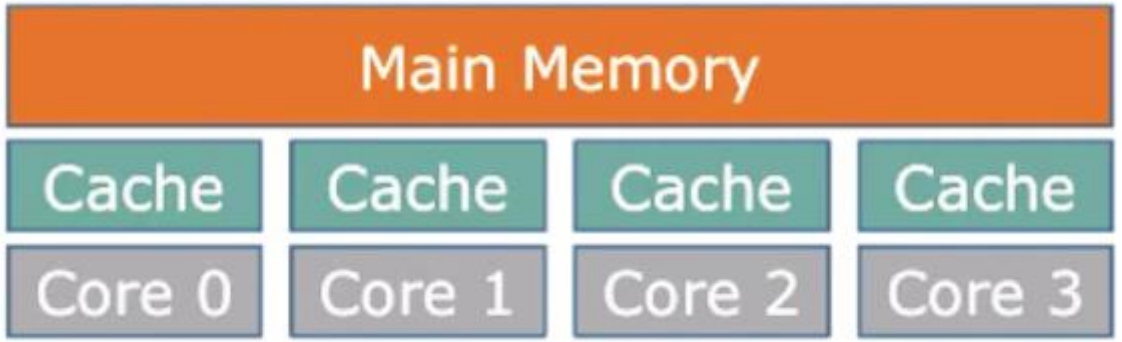
```
void transfer(int account1, int account2, int amount) {
    int a = min(account1, account2);
    int b = max(account1, account2);
    wait(lock[a]);
    wait(lock[b]);
    balance[account1] = balance[account1] – amount;
    balance[account2] = balance[account2] + amount;
    signal(lock[a]);
    signal(lock[b]);
}
```



transfer(0903, 1103, 100)

transfer(1103, 0903, 100)

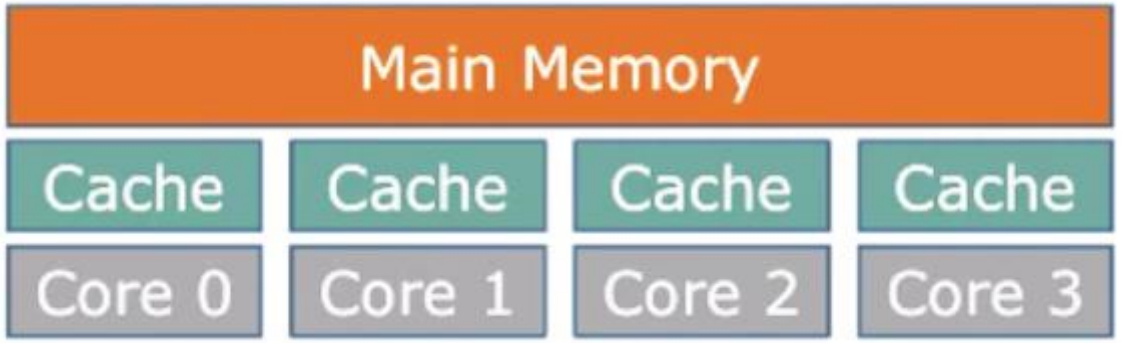
Мультиядерность



- Современные процессоры обычно имеют от 2 до 8 ядер, где каждое ядро имеет собственный кэш для повышения производительности
- Ядра могут использоваться совместно для ускорения работы приложения
- Ядра взаимодействуют друг с другом через память

Когерентность кэш

- Необходимо создать иллюзию единой общей памяти, даже если многоядерные системы имеют несколько приватных кэшей
- Проблема:



1. `LD 0xA → 2`
2. `ST 3 → 0xA`
3. `LD 0xA → 2 (stale!)`

- Решение: протокол когерентности кэша контролирует содержимое кэша, чтобы избежать устаревших строк
  - Например, сделать копию A ядра 0 недействительной, прежде чем позволить ядру 2 писать в него

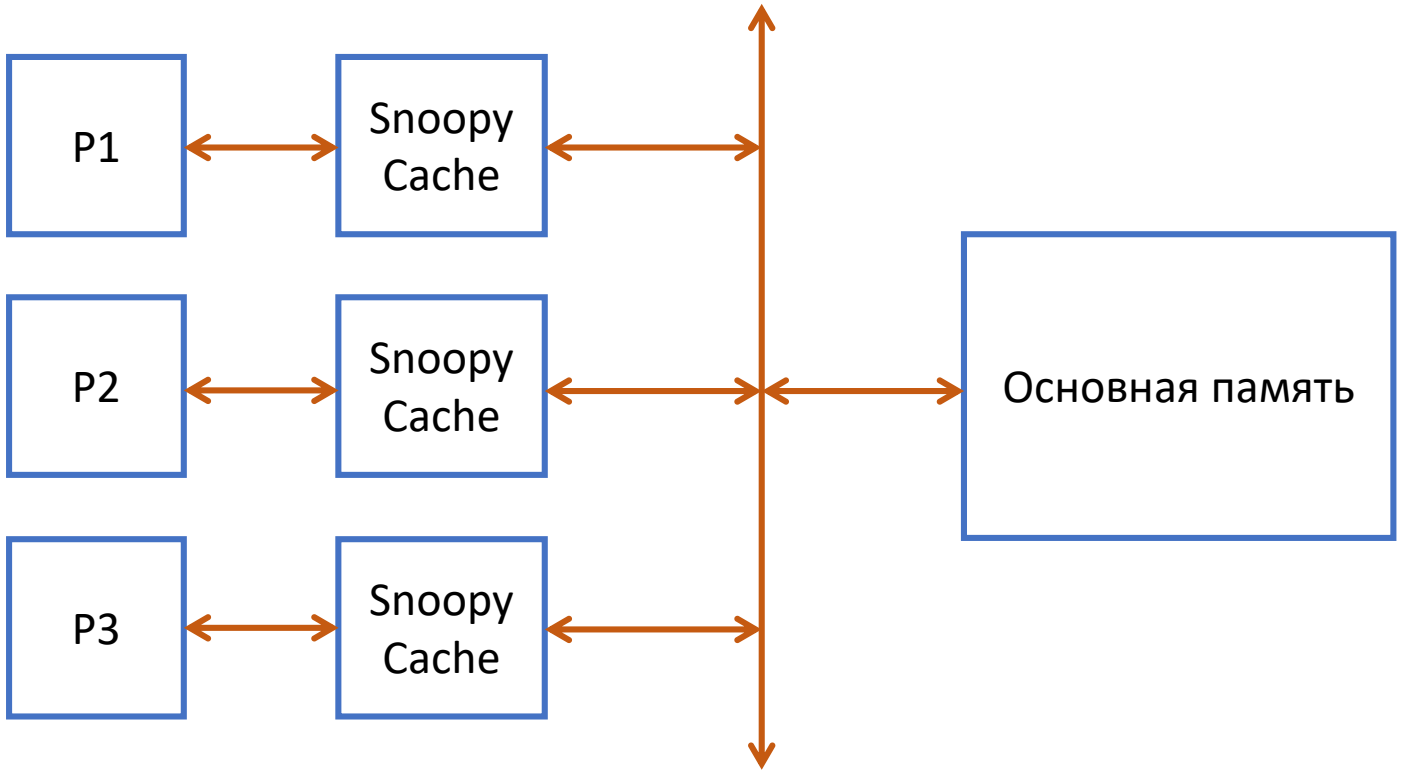
Поддержание когерентности

- В когерентной памяти все загрузки и сохранения размещаются в глобальном порядке
  - Несколько копий адреса в различных кэшах могут привести к нарушению этого свойства
- Это свойство может быть обеспечено, если:
  - Только один кэш одновременно имеет разрешение на запись
  - Никакой кэш не может иметь устаревшую копию данных после того, как была выполнена запись по адресу

Реализация когерентности кэш

- Протоколы когерентности должны обеспечивать соблюдение двух правил:
  - Распространяющаяся запись (Write propagation): записи в конечном итоге становится видимыми для всех процессоров
  - Сериализация записи (Write serialization): записи в одно и то же место сериализуются (все процессоры видят их в том же порядке)
- Как обеспечить распространение записи?
  - Write-invalidate protocols: аннулировать все другие кэшированные копии перед выполнением записи
  - Write-update protocols: обновить все другие кэшированные копии после выполнения записи
- Как обеспечить сериализацию записи?
  - Snooping-based protocols: все кэши наблюдают за действиями друг друга через общую шину
  - Directory-based protocols: каталог когерентности отслеживает содержимое частных кэшей и сериализует запросы

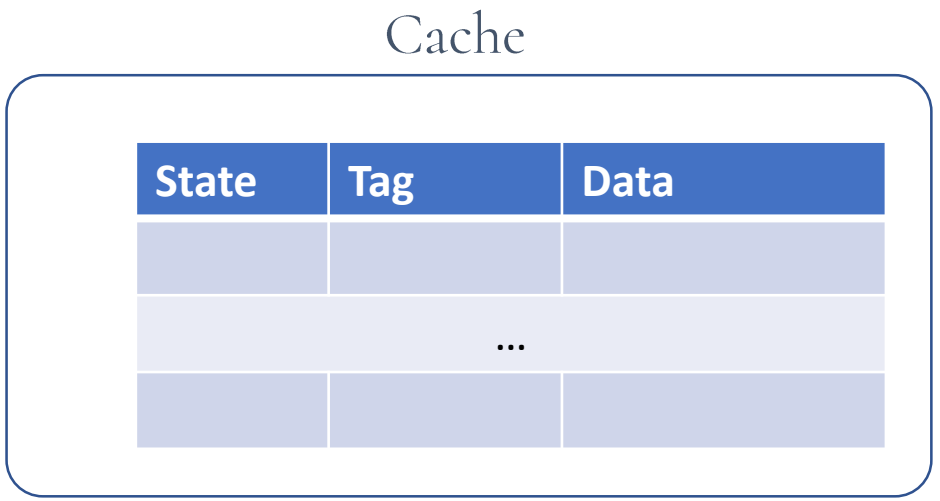
Snooping-Based Coherence



Кэширует слежение за шиной (отслеживание), чтобы все процессоры могли видеть память согласованной

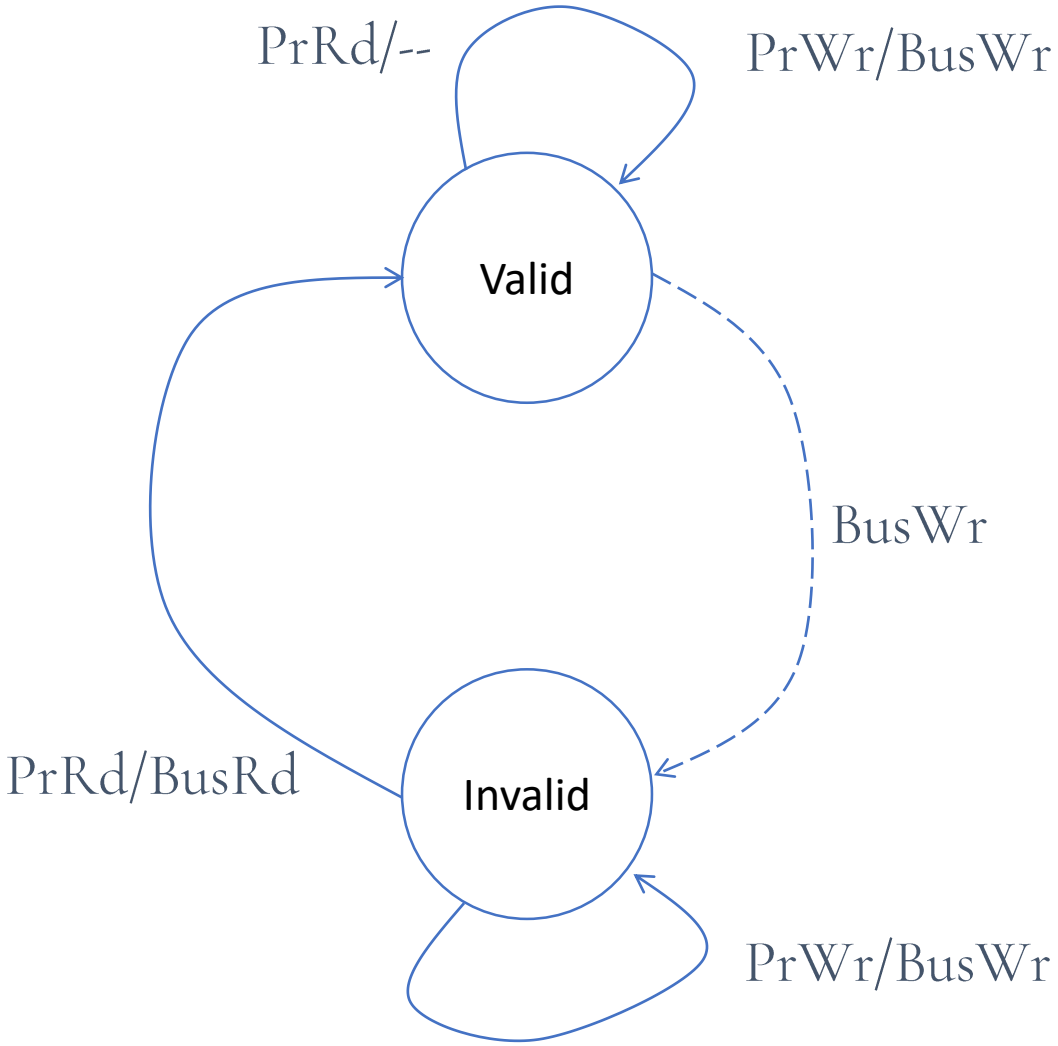
Snooping-Based Coherence

- Шина обеспечивает задачу сериализации
  - Широковещательный сигнал, полностью упорядоченный
  - Каждый кэш-контроллер «шпионит» за всеми транзакциями на шине
  - Контроллер обновляет состояние кэша в ответ на запросы процессора и snoop-события и генерирует транзакции на шине
- Snoopy-протокол (FSM)
  - Диаграмма состояний переходов
  - Действия



Snoop (обслуживание транзакций на шине)

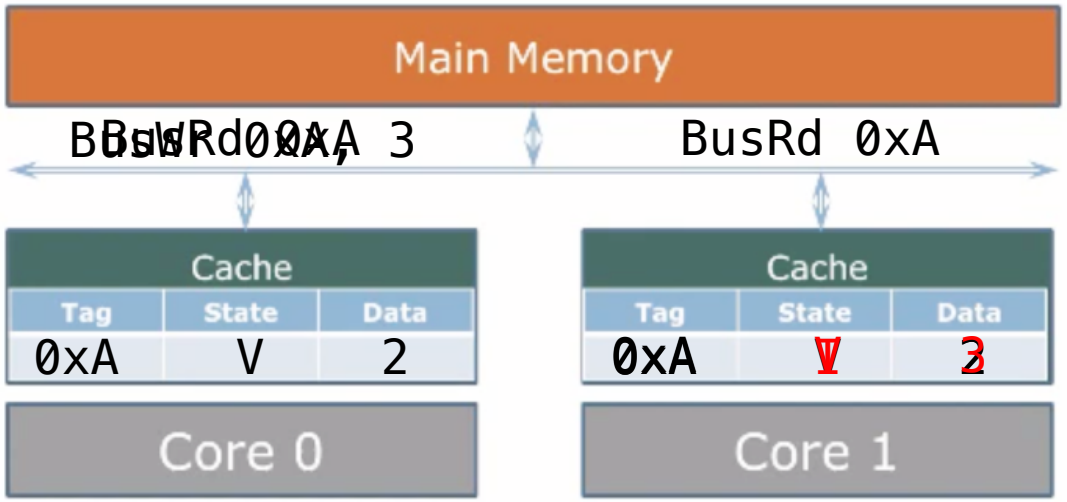
Протокол Valid/Invalid (VI)



Поддерживается только кэшем со сквозной записью

Действия
Processor Read (PrRd)
Processor Write (PrWr)
Bus Read (BusRd)
Bus Write (BusWr)

Пример Valid/Invalid



1. LD 0xA
2. LD 0xA
3. ST 0xA
4. LD 0xA

Проблемы VI? Каждая запись обновляет основную память
Каждая запись требует широковещательного передачи и слежки

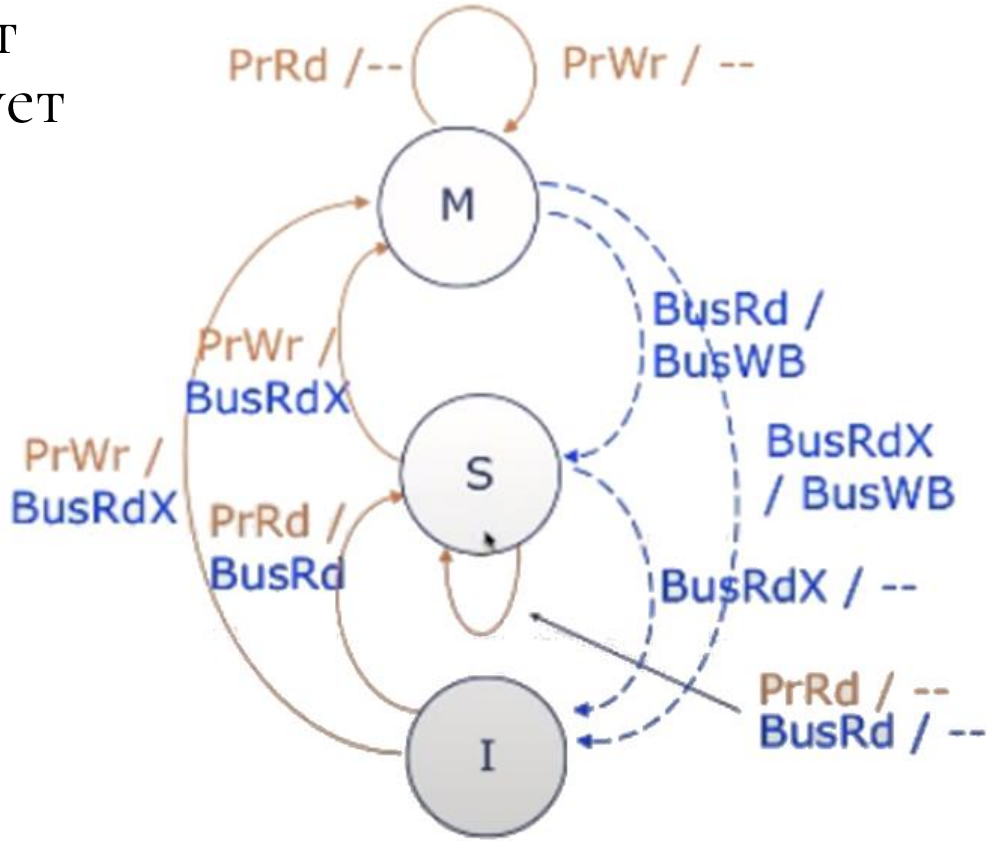
Modified/Shared/Invalid (MSI) протокол

- Каждая строка в каждом кэше поддерживает MSI состояния:
  - I – кэш не содержит адреса
  - S – кэш содержит адрес, но он так же может находится в других кэшах, следовательно он может быть только прочитан
  - M – только этот кэш содержит этот адрес, следовательно он может быть и прочитан и записан – любой другой кэш имевший этот адрес будет признан недействительным

MSI протокол FSM

- Недостатки VI: каждая запись обновляет основную память, и каждая запись требует широковещательной передачи и слежки
- MSI: возможность реализации кэша с обратной записью (writeback) + удовлетворяет локальную запись

Действия
Processor Read (PrRd)
Processor Write (PrWr)
Bus Read (BusRd)
Bus Read Exclusive (BusRdX)
Bus Write (BusWB)

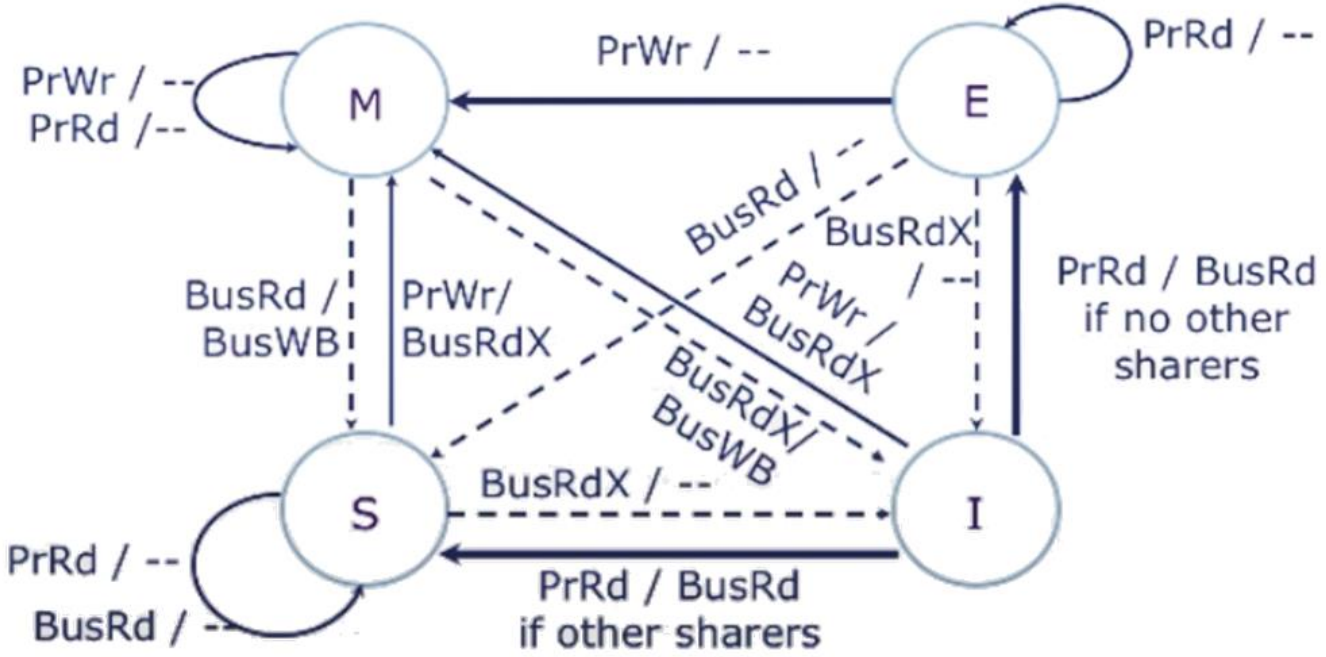


Оптимизация MSI: состояние E

- Наблюдение: выполнение последовательностей чтения-изменения-записи на частных данных является обычным делом
  - В чем проблема с MSI?
    - Две шинные транзакции для каждого чтения-изменения-записи частных данных
- Решение: E-состояние (Exclusive)
  - Если данные ни с кем не разделяются, то чтение переводит строку в состояние E вместо S
  - Запись не вещается на шину, потому что E → M (exclusive)

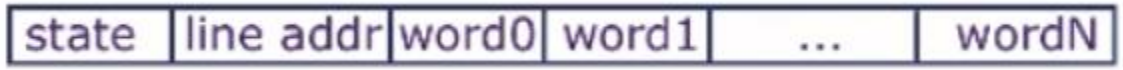
MESI: усовершенствованный MSI

- Каждая строка в кэше содержит тег и биты состояния
  - M: Modified Exclusive
  - E: Exclusive, unmodified
  - S: Shared
  - I: Invalid



Когерентность кэш и ложное совместное использование

- Строка кэша содержит более одного слова, и согласованность кэша выполняется на уровне детализации строки



- Предположим P1 записывает $word_i$ и P2 записывает $word_k$ и оба слова имеют один и тот же адрес строки
- Что может произойти?
  - Строка может быть недействительной (пинг-понг) много раз без необходимости, потому что адреса находятся в одной строке