# ТЕОРИЯ И ПРАКТИКА МНОГОПОТОЧНОГО ПРОГРАММИРОВАНИЯ

## Тема 14

Использование конечных автоматов для разработки и анализа неблокирующих алгоритмов.

Д.ф.-м.н., профессор А.Г. Тормасов
Базовая кафедра «Теоретическая и Прикладная Информатика», МФТИ

# Тема

- Использвание конечных автоматов для разработки и анализа неблокирующих алгоритмов.
- Пример конечного автомата с «всегда корректными» состояниями для создания неблокирующей хеш таблицы.
- Доказательства корректности предложенной хеш таблицы.

# Хеш таблицы и MP

- Хеш таблицы – высокоэффективные структуры данных, котоые могут дать время операций O(1)
  - При эффективной реализации
- Для корректной поддержки работы в условиях SMP требуется много усилий
  - Один или несколько синхронизационных примитивов для обслуживания обращений
  - Может плохо работать в условиях интенсивной нагрузки (lock contention/…)
  - Каждый из них обычно требует, как минимум, 1 операции с блокировкой шины типа LOCK CMPXCHG
- Хорошая цель для создания неблокирующих алгоритмов – см предыдущую тему
  - Создание корретных алгоритмов непросто, требует доказательства

# Общие требования

- Хеш таблица есть коллекция пар (ключ, значение)
- Должна быть быстрой
  - при сложности доступа O(1), накладные расходы превысят затраты на собственно работу основных алгоритмов
- Должна принимать во внимание кеш
  - Ключ и значение на одной линии
- Автоматическое изменение размера по мере надобности
- Не должна делать аллокаций памяти на add/del

# Варианты реализации

```
// Пример кода для get(key)

idx = hash = key.hashCode();

// цикл по коллизиям
while( true )
{ // ограничим индекс размером
  idx &= (size-1);
  // обратимся к кешу заранее
  k = get_key(idx);
  // запомним значение
  h = get_hash(idx);
  if( k == key ||
      (h == hash && key.equals(k)) )
      // нашли, возвращаем
      return get_val(idx);
  if( k == null )
      // нет, возвращаем 0
      return null;
  // следующая коллизия
  idx++;
}
```

- Можно использовать, например, размер таблицы как простое число, и использовать операцию MOD
  - Правда, MOD примерно в 30 раз медленне чем AND
- Можно использовать открытые таблицы (с односвязными списками), но...
  - Требует аллокацию на add()
  - Идем по ссылкам типа next
    - Каждый next == промах кеша!

# Варианты реализации

- Как доказывать корректность работы?
  - Модель упорядоченности памяти, барьеры, «событие а было до события б» и тд
  - Много функций (add, del, addIfno, change, lookup и тд)
- Используем конечный автомат

# Конечный автомат

- Определим все возможные значения для пар {Key, Value} состояния
- Определим переходы из состояния в состояние
- Покажем, что ВСЕ состояния легальны

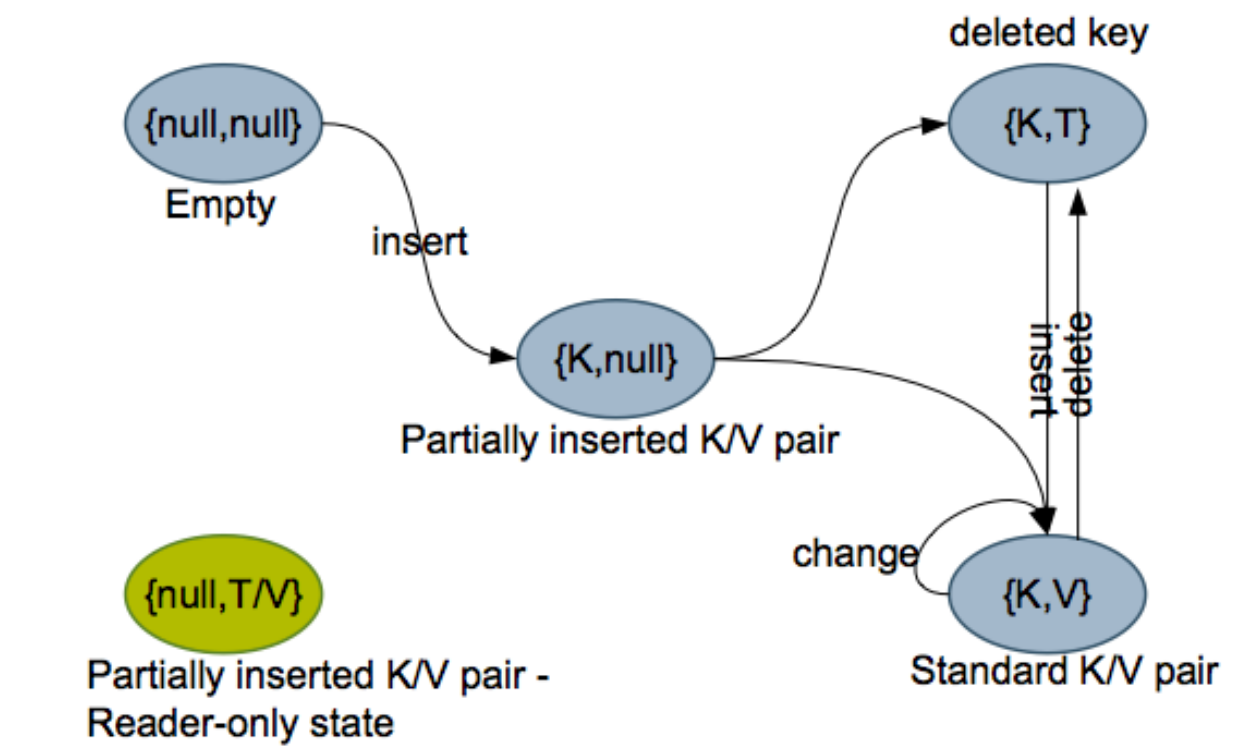
# Состояния и переходы

- Последовательность чтения-записи в память не важна!
  - lookup может читать key и value в любом порядке
  - add может менять key и value в любом порядке
  - put должно использовать CAS для изменения key и value (не одновременно!)
- Для корректности не нужен барьер памяти
  - Правда, иногда может потребоваться более сильная гарантия – тогда может быть потребуется барьер
- Доказательство корректности алгоритма в целом просто

# Состояния автомата

- Key
  - null - пусто
  - K – некий ключ (не может быть изменен!)
    - ! это – существенное ограничение
- Value
  - null - пусто
  - T - удалено
  - V – некое значение
- Состояние есть пара {Key, Value}
- Переход – успешная CAS операция

# Диаграмма переходов



# Замечания

- После добавления ключа изменение его невозможно
  - Нельзя вернуть неправильный ключ
  - Увы, но таблица будет заполняться мертвыми ключами со временем
  - Как бороться с этим покажем чуть позже (через resize)
- Нет гарантий упорядоченности операций
  - Если надо – «сделай сам»
- {null,V} – разрешено, но бессмысленно
  - Прочлось пустое значение ключа – «промазали» (порядок чтения-записи?)

# Замечания

- Нет единого согласованного состояния всего автомата
- Никто не гарантирован в возможности считывания того же самого состояния
  - Разве что на том же CPU без других писателей
- Да это и не нужно!
- По сути, предоставляются ТЕ ЖЕ гарантии, что и для одноячечной глобальной разделяемой переменной
  - Много читателей и  писателей без взаимной синхронизации

# Больше гарантий

- Возможно, надо иметь гарантии порядка для Value
  - Примерно то же самое, что объявить volatile нашу «разделяемую переменную»
- Если чтото записали до (add), то оно должно быть видно после (lookup)
- Надо: барьер до CAS на запись (бесплатно для x86)
- Надо: барьер после чтения Value на чтение данных

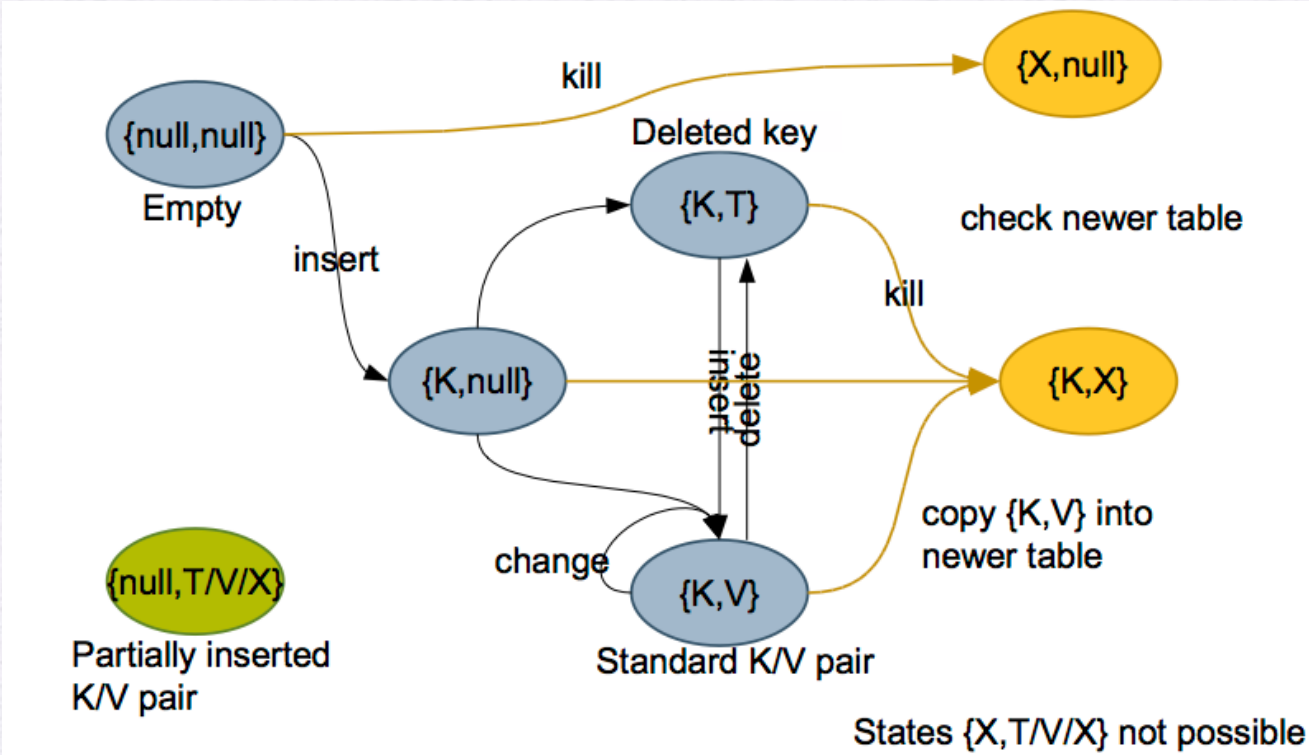
# Проблема размера

- Таблица переполняется или забивается удаленными элементами
- Решение:
  - автоматическая процедура переписи пар в новую таблицу (может быть, большего размера)
  - соисполнимая с add/del/lookup
- Много интересных проблем
  - барьер
  - упорядоченность относительно таблиц
  - не “забыть” последнюю запись в старую таблицу и т.д.

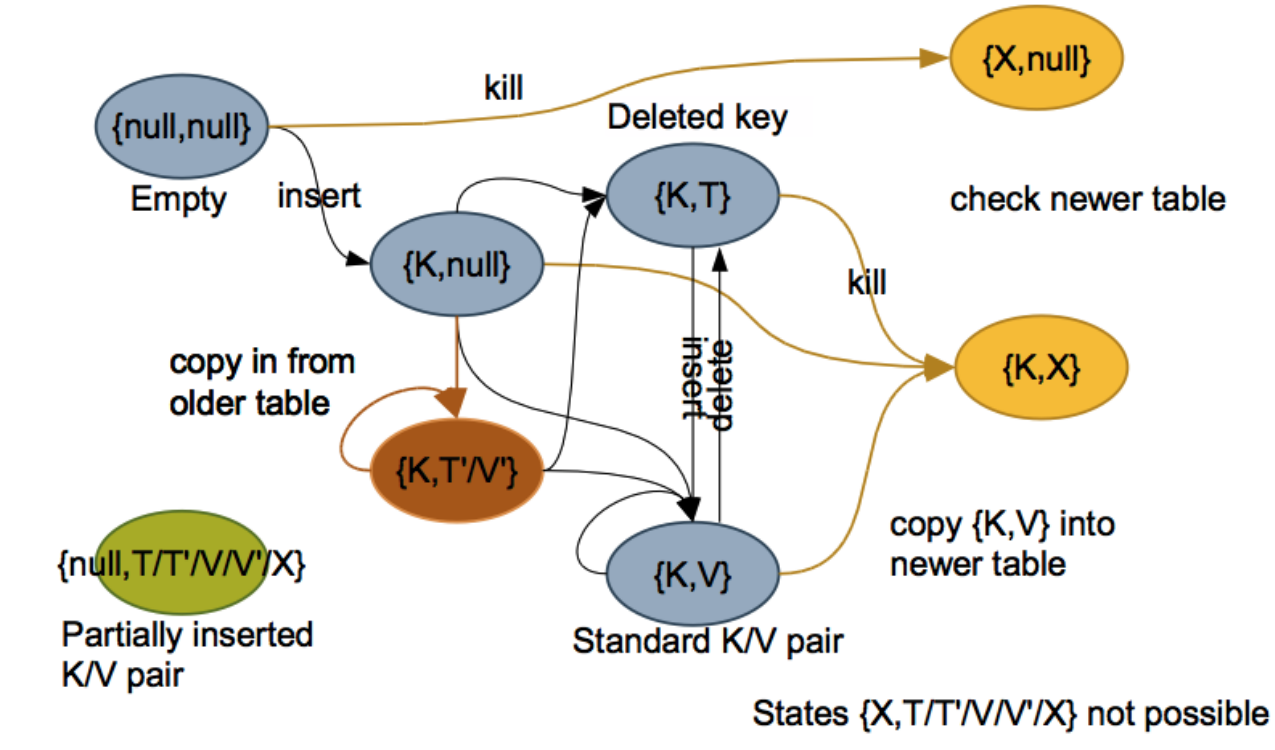
# Проблема размера

- Добавляем несколько новых состояний, причем получается что состояние “частично скопировано” должно быть нормальным
- lookup работает со старой таблицей пока не увидит специальной отметки
- add всегда работает с новой
- Постоянно надо проверять новую таблицу

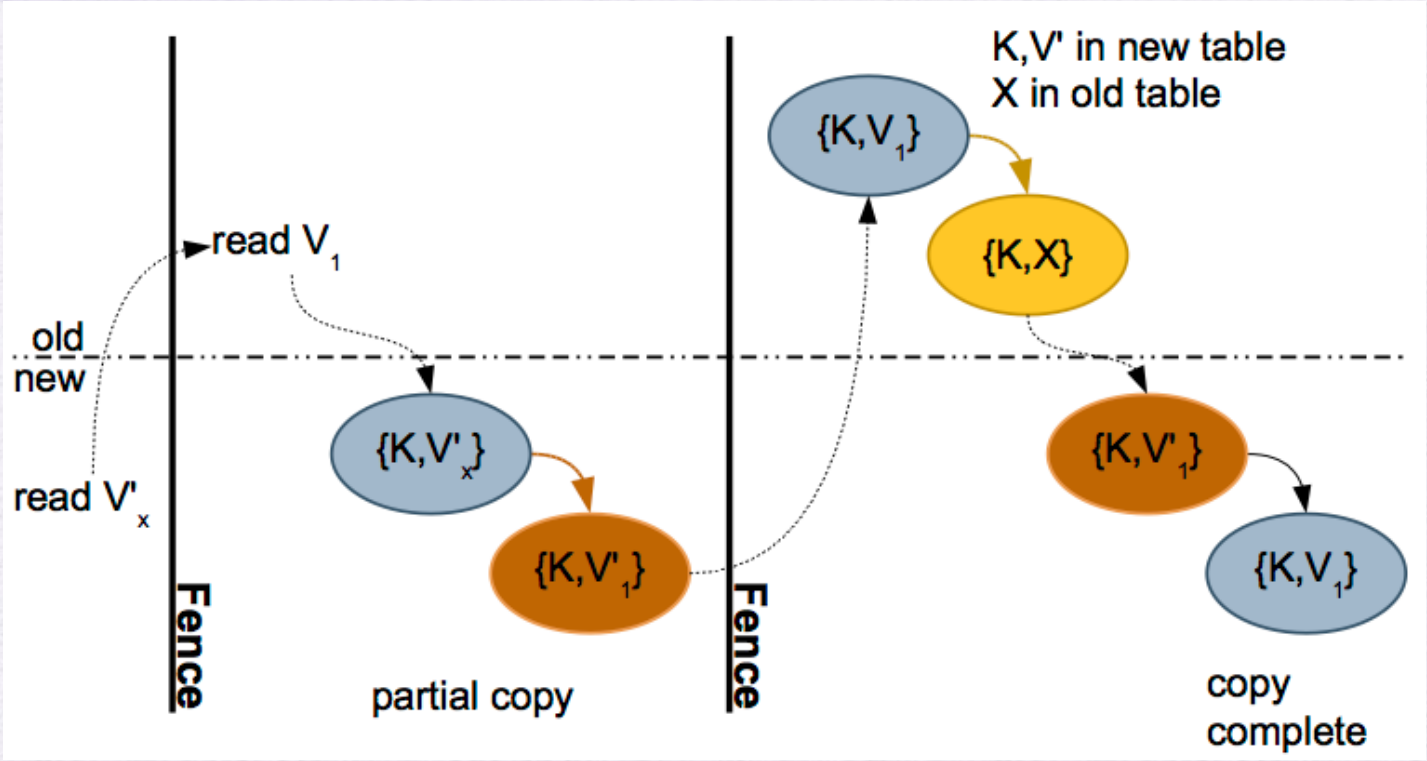
# Диаграмма переходов - старая



# Диаграмма переходов - новая



# Копирование пары



# Выводы

- Рассмотрена на методика создания алгоритмов, упрощающая доказательство их корректности и технику создания, путем создания машины состояний с атомарными переходами
- Если все состояния корректны, и переходы корректны (выполняются гарантированными примитивами), то и машина корректна!
- Требует аккуратного анализа барьеров и «относительности»
- Понятие корректности нетривиально, похоже на работу с одной переменной, чем со структурой данных со сложными состояниями – со всеми плюсами и минусами

# (c) А. Тормасов, 2010-11 г.

Базовая кафедра «Теоретическая и Прикладная Информатика» ФУПМ МФТИ
tor@ crec .mipt .ru_