Параллельные вычисления Оборудование параллельных вычислительных систем

Созыкин Андрей Владимирович

к.т.н.
Заведующий кафедрой высокопроизводительных компьютерных технологий
Институт математики и компьютерных наук

Базовые сведения об оборудовании

Разработка последовательных программ

- Об оборудовании можно не знать ничего

Разработка параллельных программ:

- Особенности устройства современных многоядерных процессоров существенно влияют на работу многопоточных и параллельных программ

Современные процессоры

Процессоры устроены очень сложно

- Несколько ядер
- Несколько аппаратных потоков
- Несколько исполнительных устройств
- Кэш нескольких уровней
- Конвейер

Программа часто выполняется не в том виде, как ее написал программист:

- Оптимизация компилятором
- Оптимизация процессором

Многие особенности процессоров противоречат интуиции программиста

Использовать ли сортировку?

```
long sum = 0;

if (sorting)
    // Sorting data
    std::sort(values.begin(), values.end());

// Processing data
for (int j = 0; j < REPEAT_NUM; ++ j)
    for (int i = 0; i < SIZE; ++i)
        if (values[i] >= MAX_VAL / 2)
            sum += values[i];

std::cout << "Sum = " << sum << std::endl;
```

Источник: http://stackoverflow.com/q/11227809/3595291
P.S. Пример программы тестовый

Использовать ли сортировку?

```
./branch_prediction
Generating random numbers

Processing unsorted data
Sum = 316652800000
Calculating time = 34093

Processing sorted data
Sum = 316652800000
Calculating time = 13724
```

Предсказание ветвлений

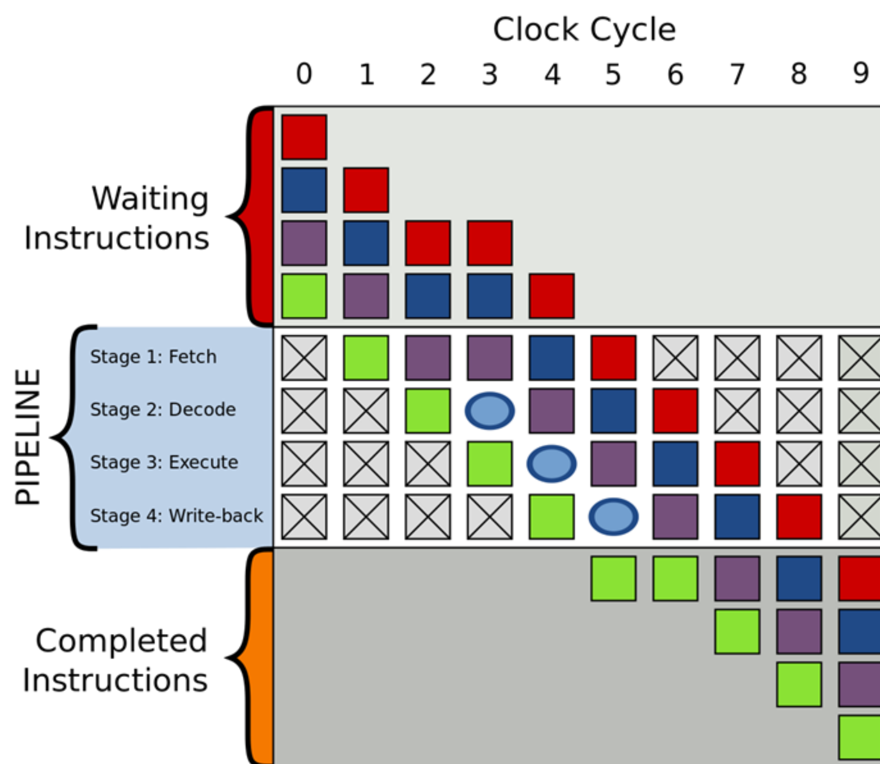
Использование сортировки ускоряет вычисления!

Причина

- Предсказание ветвлений
- Конвейер команд постоянно заполнен

Проблемы работы конвейера

- 10-20 ступеней
- Долго «нагревается» и «остывает»



Если переставить циклы?

```
long sum = 0;

if (sorting)
    // Sorting data
    std::sort(values.begin(), values.end());

// Processing data
for (int i = 0; i < SIZE; ++i)
    if (values[i] >= MAX_VAL / 2)
        for (int j = 0; j < REPEAT_NUM; ++ j)
            sum += values[i];

std::cout << "Sum = " << sum << std::endl;
```

Источник: http://stackoverflow.com/q/11227809/3595291
P.S. Пример программы тестовый

Если переставить циклы?

```
long sum = 0;

if (sorting)
    // Sorting data
    std::sort(values.begin(), values.end());

// Processing data
for (int i = 0; i < SIZE; ++i)
    if (values[i] >= MAX_VAL / 2)
        for (int j = 0; j < REPEAT_NUM; ++ j)
            sum += values[i];

std::cout << "Sum = " << sum << std::endl;
```

Компилятор может выполнить такую перестановку автоматически!

Источник: http://stackoverflow.com/q/11227809/3595291
P.S. Пример программы тестовый

Test and Set Lock

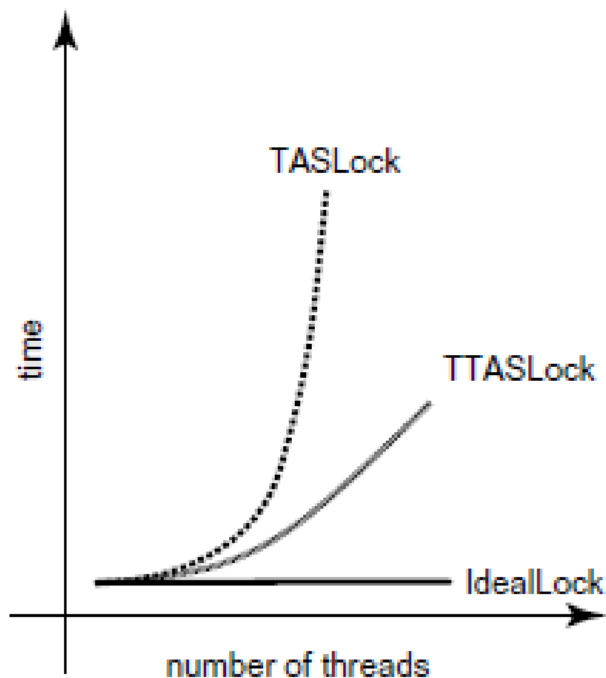
```
public class TASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);
  public void lock() {
    while (state.getAndSet(true)) {}
  }
  public void unlock() {
    state.set(false);
  }
}
```

Test and Test and Set Lock

```
public class TTASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);
  public void lock() {
    while (true) {
      while (state.get()) {};
      if (!state.getAndSet(true))
        return;
    }
  }
  public void unlock() {
    state.set(false);
  }
}
```

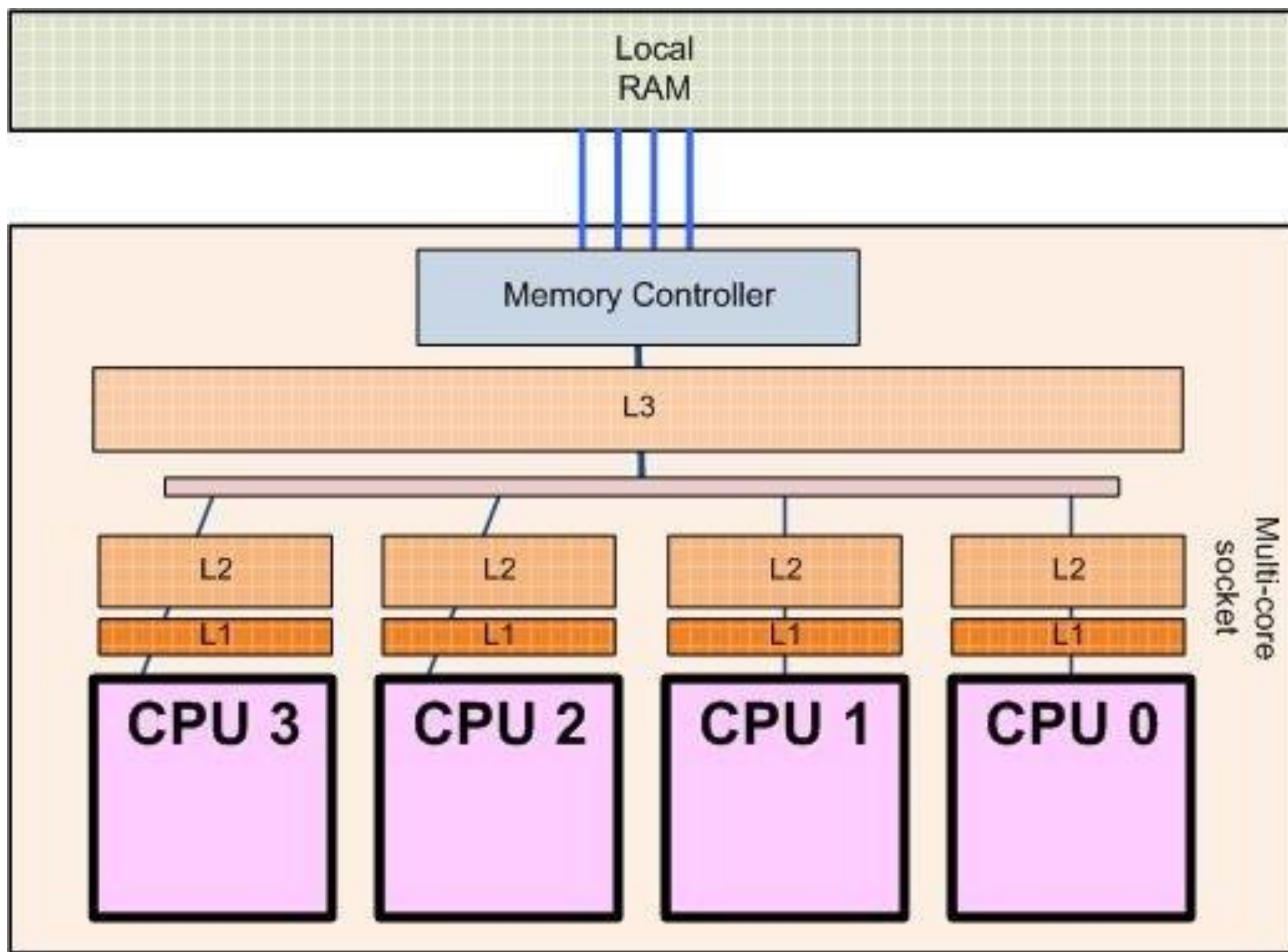
TASLock vs TTASLock

Какая блокировка сработает быстрее?



Maurice Herlihy. Nir Shavit. The Art of Multiprocessor Programming

Иерархия памяти



Ориентировочное время доступа

Кэш 1 уровня ~1-2 такта

Кэш 2 уровня ~10-40 тактов

Кэш 3 уровня ~50-100 тактов

Основная память – сотни тактов

Когерентность кэшей

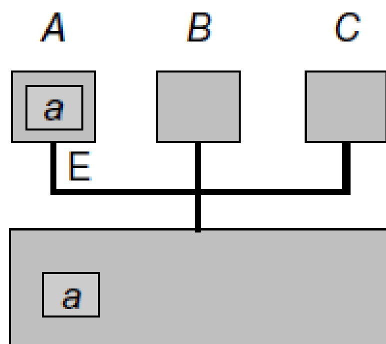
Если один из процессоров (ядер) поменял данные в своем КЭШе, другие процессоры должны узнать об этом

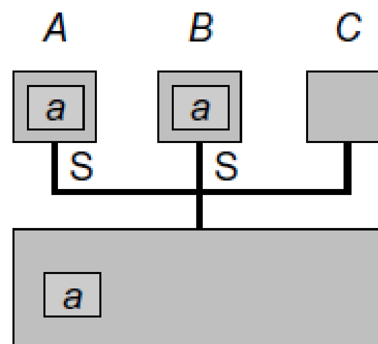
Поддержка когерентности кэшей требует высоких накладных расходов

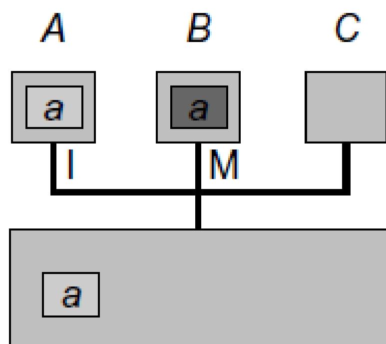
Протокол MESI:

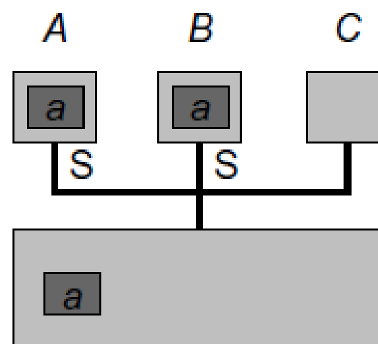
- **M**odified – данные в кэше были изменены
- **E**xclusive – данные загружены в кэш только одного процессора
- **S**hared – данные загружены в кэш разных процессоров, но не изменены
- **I**nvalid – кэш содержит неправильные данные

Протокол MESI









Maurice Herlihy. Nir Shavit. The Art of Multiprocessor Programming

Когерентность кэшей. Выводы

Требуется разное время на:

- Чтение данных (самое быстрое)
- Запись данных (среднее)
- Уверенность в том, что данные записались в общую память (самое медленное)

Test and Set Lock

```
public class TASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);
  public void lock() {
    // Каждый раз пытаемся ЗАПИСАТЬ данные
    // Высокий трафик для когерентности кэшей
    while (state.getAndSet(true)) {}
  }
  public void unlock() {
    state.set(false);
  }
}
```

Test and Test and Set Lock

```
public class TTASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);
  public void lock() {
    while (true) {
      // Загружаем данные в кэш и читаем из кэша
      // Если состояние поменялось, читаем из общей памяти или
      // кэша другого процессора
      while (state.get()) {};
        // Записываем только когда есть реальная возможность
        if (!state.getAndSet(true))
          return;
    }
  }
  public void unlock() {
    state.set(false);
  }
}
```

Ложное разделение данных

Данные в кэш записываются «строками»
- Размер строки зависит от архитектуры
- Типичный размер 32 или 64 байта

Локальность
- Временная
- Пространственная

Подсчет количества нечетных чисел

```
std::vector<int> values;
int odds = 0;

void sequential_odd_counting() {
    odds = 0;
    for (int i = 0; i < SIZE; ++i)
        if ( values[i] % 2 != 0 )
            ++odds;
    std::cout << “Количество нечетных чисел " << odds
        << std::endl;
}
```

Herb Sutter. Eliminate False Sharing
http://www.drdobbs.com/parallel/eliminate-false-sharing/217500206

Как сделать параллельную версию?

```
std::vector<int> values;
int odds = 0;

void sequential_odd_counting() {
    odds = 0;
    for (int i = 0; i < SIZE; ++i)
        if ( values[i] % 2 != 0 )
            ++odds;
    std::cout << “Количество нечетных чисел " << odds
        << std::endl;
}
```

Как сделать параллельную версию?

Проблема

- Разделяемая переменная odds
- Возможны условия гонок

Пути решения

- Синхронизация (mutex)
- Атомарная переменная
- Заменить скалярную переменную на массив (promote scalar to array)

Параллельная версия

```
std::vector<int> values;
int odds = 0;
// Количество потоков
const int MAX_THREAD_NUM = 16;
// Массив с отдельными счетчиками нечетных чисел для
// каждого потока
int odds_threads[MAX_THREAD_NUM] = {};

void thread_odd_counter(int thread_id, int start_index, int
end_index){
    odds_threads[thread_id] = 0;
    for (int i = start_index; i < end_index; ++i)
        if ( values[i] % 2 != 0 )
            ++odds_threads[thread_id];
}
```

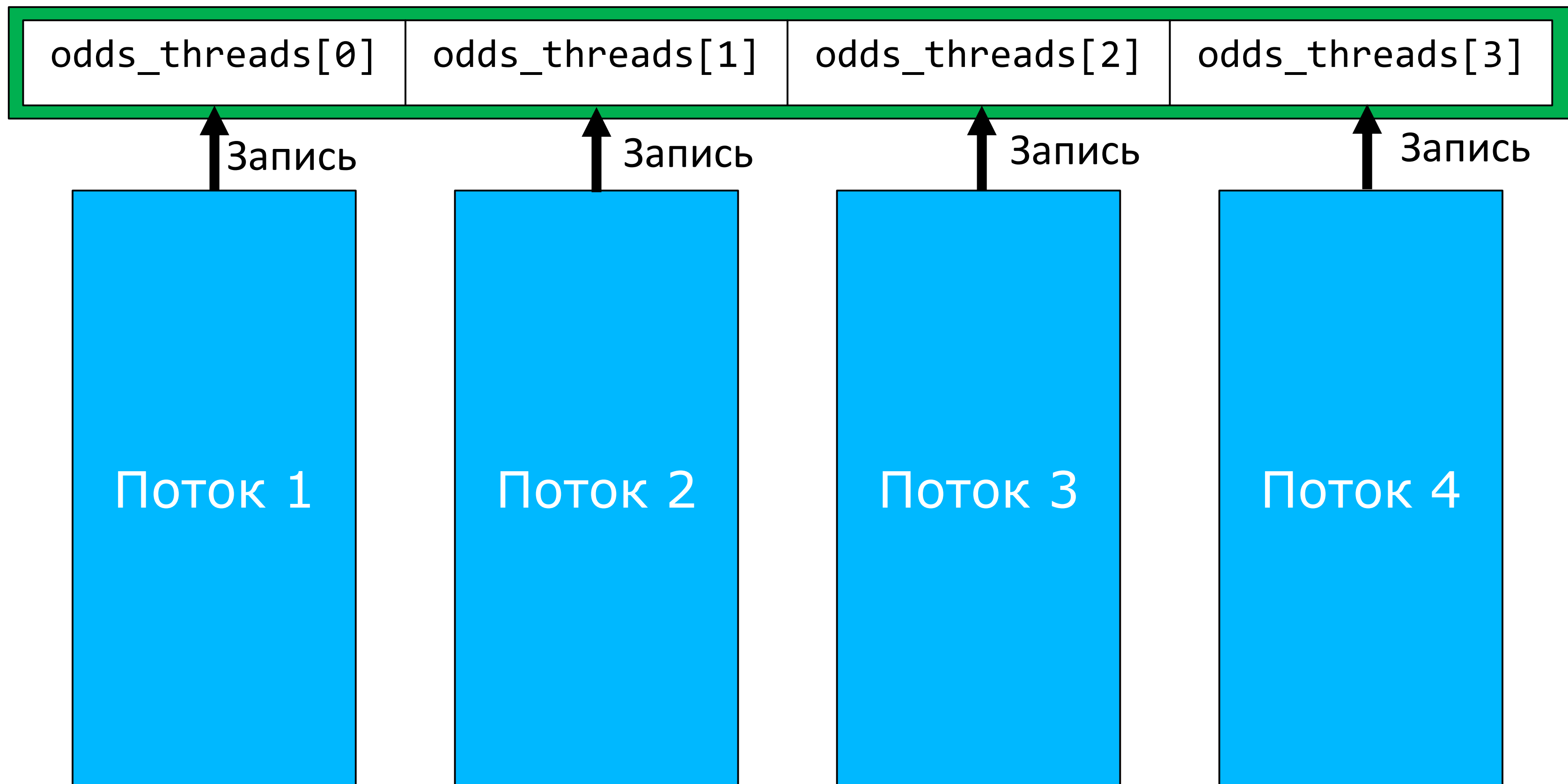
Параллельная версия

```
void concurrent_odd_counting(int thread_num) {
    std::vector<std::thread> threads;
    const int part_size = SIZE / thread_num;

    for (int i = 0; i < thread_num; ++i)
        threads.push_back(std::thread(thread_odd_counter, i,
            part_size * i, part_size * (i + 1)));

    for (std::thread& t : threads)
        t.join();

    odds = 0;
    for (int i = 0; i < thread_num; i++)
         odds += odds_threads[i];
    std::cout << «Количество нечетных чисел " << odds
        << std::endl;
}
```

Параллельная версия

```
./false_sharing
Generating random numbers
Counting odd numbers sequentially
Odd number count is 49998833
Counting time is 943
Counting odd numbers using 2 threads
Odd number count is 49998833
Counting time is 1112
Counting odd numbers using 4 threads
Odd number count is 49998833
Counting time is 1360
Counting odd numbers using 8 threads
Odd number count is 49998833
Counting time is 1168
```

Ложное разделение данных



Как решить проблему ложного разделения

Не использовать разделяемых данных вообще

- async/future

Заполнение (padding)

- Добавление в массив избыточных элементов
- Расстояние между используемыми элементами массива больше, чем длина строки кэша
- Зависит от платформы
- Необходимо знать размер строки кэша. В Linux

```
$ getconf LEVEL1_DCACHE_LINESIZE
64
```

Заполнение

```
std::vector<int> values;
int odds = 0;
// Заполнение на 16 байт (4 байта int * 16 = 64 байта)
const int PAD = 16;
// Массив счетчиков с заполнением
int odds_threads[MAX_THREAD_NUM * PAD] = {};

void thread_odd_counter(int thread_id, int start_index,
    int end_index){

    odds_threads[thread_id] = 0;
    for (int i = start_index; i < end_index; ++i)
        if ( values[i] % 2 != 0 )
            // Запись в массив счетчиков с учетом заполнения
            ++odds_threads[thread_id * PAD];
}
```

Заполнение

```
./false_sharing_padding
Generating random numbers
Counting odd numbers sequentially
Odd number count is 49998833
Counting time is 946
Counting odd numbers using 2 threads
Odd number count is 49998833
Counting time is 658
Counting odd numbers using 4 threads
Odd number count is 49998833
Counting time is 562
Counting odd numbers using 8 threads
Odd number count is 49998833
Counting time is 378
```

Модель памяти

Необходима поддержка иерархии памяти в языке программирования

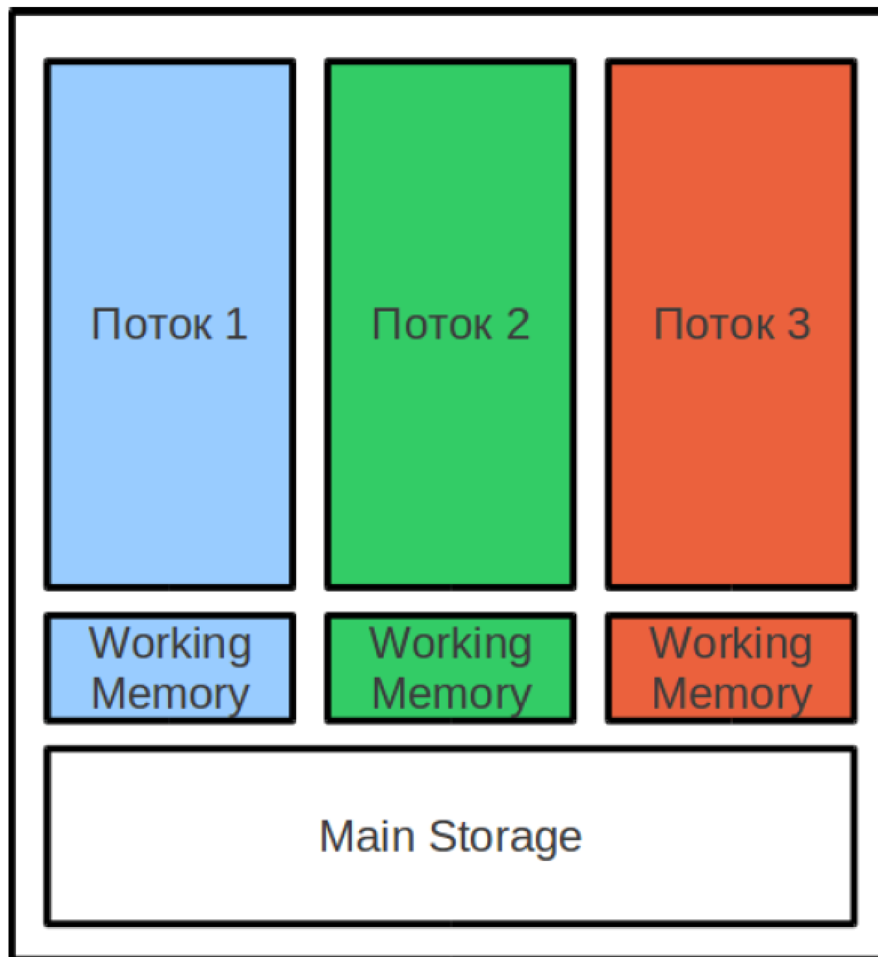
Модель памяти:

- Какие данные являются общими для всех потоков, а какие частными
- Как допустимо переставлять операции доступа в память

Языки программирования:

- Java
- C#
- C++11

Модель памяти Java



Пример использования модели памяти

```
public class TestMemoryModel {
  private static boolean ready;
  private static int number;

  private static class ReaderThread extends Thread {
    public void run () {
      while (!ready)
        Thread.yield();
      System.out.println( number );
    }
  }
  public static void main ( String [] args ) {
    new ReaderThread().start();
    number = 42;
    ready = true ;
  }
}
```

Ключевое слово volatile

Используется для объявления переменных «общими» для всех потоков

Чтение volatile переменной:

- Все копии в кэшах становятся инвалидными
- Данные читаются напрямую из памяти

Запись volatile переменной:

- Данные записываются напрямую в память

Пример использования volatile

```
public class TestMemoryModel {
  private static volatile boolean ready;
  private static int number;

  private static class ReaderThread extends Thread {
    public void run () {
      while (!ready)
        Thread.yield();
      System.out.println( number );
    }
  }
  public static void main ( String [] args ) {
    new ReaderThread().start();
    number = 42;
    ready = true;
  }
}
```

Перестановка операций доступа в память

```
public class TestMemoryModel {
  private static volatile boolean ready;
  private static int number;

  private static class ReaderThread extends Thread {
    public void run () {
      while (!ready)
        Thread.yield();
      System.out.println( number );
    }
  }
  public static void main ( String [] args ) {
    new ReaderThread().start();
    // Меняем местами флаг и установка значения number
    ready = true;
    number = 42;
  }
}
```

Везде volatile!

```
public class TestMemoryModel {
  private static volatile boolean ready;
  private static volatile int number;

  private static class ReaderThread extends Thread {
    public void run () {
      while (!ready)
        Thread.yield();
      System.out.println( number );
    }
  }
  public static void main ( String [] args ) {
    new ReaderThread().start();
    number = 42;
    ready = true;
  }
}
```

Модель памяти в C++

Появилась в C++11:

- До этого не было никаких гарантий порядка выполнения операций с памятью

Существенно сложнее модели памяти в Java:

- Для системного программирования (ОС и системы виртуализации) нужна максимальная производительность
- При разработке модели памяти в C++ выбрали возможность обеспечить производительность, а не упрощение языка

Проблемы C++

Многопоточность добавили в стандартную библиотеку C++ в стандарте C++11

- thread, mutex, condition_varaibles, future и т.п.

В стандарте C++11 нет понятия потоков и процессов

- Компилятор может оптимизировать код так, как будто он выполняется последовательно

Пример кода на C++

```
int data;
bool ready = false;

void foo(){          // Поток 1
    data = 42;
    ready = true;
}

void bar(){          // Поток 2
    if (ready)
        assert(data == 42)
}
```

https://events.yandex.ru/lib/talks/2698/

Модели памяти

Sequential consistency

- Никаких перестановок операций доступа к памяти
- Все процессоры видят изменения в одном и том же порядке

Strongly-ordered

- Возможны некоторые перестановки (acquire и release семантика)
- Процессоры Intel x86/64 (и аналоги)

Weakly-ordered

- Гарантируется упорядоченность для операций, зависимых по данным
- Процессоры ARMv7, PowerPC

Super-weak

- Никаких гарантий

Атомарные переменные и операции

Атомарные операции

- Может быть выполнена либо полностью, либо не выполнена вообще
- Другие потоки не могут прервать или «увидеть» операцию в процессе выполнения
- Выполняются над **атомарными** объектами

Атомарные объекты

- Стандартные атомарные типы данных (atomic_bool, atomic_char, atomic_int и т.д.)
- Пользовательские атомарные типы на основе шаблона std::atomic<UserDefinedType>

Операции с атомарными переменными

```
int data;
std::atomic<bool> ready = false;

void foo(){          // Поток 1
    data = 42;
    ready.store(true);
}

void bar(){          // Поток 2
    if (ready.load())
        assert(data == 42)
}
```

Барьер памяти

```
int data;
std::atomic<bool> ready = false;

void foo(){          // Поток 1
    data = 42;
    ____________________
    ready.store(true);
}

void bar(){          // Поток 2
    if (ready.load())
        assert(data == 42)
}
```

Барьеры памяти в C++

```
enum memory_order {
     memory_order_relaxed,
     memory_order_consume,
     memory_order_acquire,
     memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

Acquire – гарантирует, что любые операции после барьера будут выполнены после того, как будут выполнены все операции загрузки до барьера

Release – гарантирует, что любые операции до барьера будут выполнены до того, как начнут выполняться операции сохранения после барьера.

Барьеры памяти в C++

```
T load(memory_order = memory_order_seq_cst) const;

void store(T desired, memory_order = memory_order_seq_cst);

void atomic_thread_fence(memory_order order);
```

Барьер памяти

```
int data;
std::atomic<bool> ready = false;

void foo(){          // Поток 1
    data = 42;
    ready.store(true, std::memory_order_release);
}

void bar(){          // Поток 2
    if (ready.load(std::memory_order_acquire))
        assert(data == 42)
}
```

volatile в C++

volatile в стандарте C++

- Есть, но значение другое, чем в Java
- Говорит компилятору, что значение переменной может быть изменено за пределами программы:

```
volatile const long clock_register;
auto t1 {clock_register}
...
auto t2 {clock_register}
```

volatile в VisualStudio:

- Значение как в Java – запись данных только в память
- Не совместимо со стандартом

Почему все так сложно???

Понимание модели памяти C++ необходимо только если:

- Вы разрабатываете многопоточную программу
- Используется lock-free код (синхронизация без мьютексов)

Итоги

Для разработки многопоточных программ необходимо учитывать особенности современных процессоров

Кэш

- Многоуровневый
- Ложное разделение

Модель памяти:

- Гарантия порядка выполнения операций с памятью
- Необходимо использовать только при разработке Lock-Free структур данных

Вопросы?