Многопоточное программирование и Java

Корректность, методики, оптимизация

Евгений Кирпичев
Яндекс

26 мая 2009 г.

Цель доклада

Расширить кругозор в области многопоточного программирования вообще и на Java в частности.

- Корректность
  - Как сформулировать, проверить, повысить, гарантировать
- Методики и велосипеды
  - Мудрость предков
- Производительность
  - Что влияет на производительность и как повлиять самому

Проблемы многопоточного программирования

Писать многопоточные программы - трудно.

- Обычно слабая поддержка со стороны (особенно императивного) языка
- Трудно сделать корректными
- Адски трудно тестировать
- Трудно сделать быстрыми (не всё хорошо масштабируется)

Классификация многопоточных программ

Parallelism vs Concurrency

Parallel	Одна задача бьется на много частей
Concurrent	Много разных задач

Классификация многопоточных программ (contd.)

Программы бывают разные.

	Raytracer	Server	IDE
Сколько задач?	Много	Много	Мало
Зависимы?	Нет	Нет	Да
Общие ресурсы?	Нет	Да	Да
Трудность	CPU perf	IO perf	Корректность

Содержание

Причины багов

Корень всех зол - изменяемое состояние.

- N нитей, K состояний $\Rightarrow O(K^N)$ переплетений
- Промежуточные состояния *видимы*
- Даже метод - это больше не черный ящик
  - Его тело больше не атомарно
- Корректные многопоточные программы не модульны
  - Sequentialy correct + Sequentially correct $\neq$ Concurrently correct

Доказуемая корректность

Face it:

- Либо программа *доказуемо* корректна
- Либо она, почти наверняка, некорректна
  - Полностью оттестировать невозможно
  - Вероятность бага 10^{-8}, 10^9 вызовов за неделю - и вероятность уже $1 - e^{-10} \approx 1$
- Работает *на бумаге* или не работает вообще

Доказуемая корректность

Формальные методы рассуждения и доказательства корректности *необходимы*.
Это не роскошь для фриков.

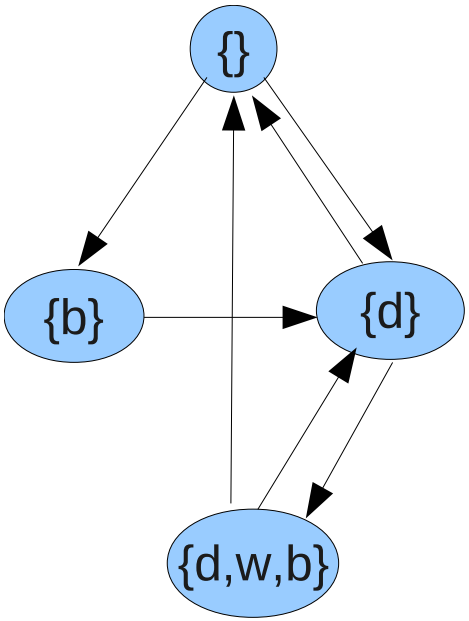
Структура Крипке

Структура Крипке $\approx$ граф состояний.

- Множество состояний
- Некоторые *начальные*
- Некоторые соединены *переходом*
  - Переход *безусловный*
- Множество *названий фактов*
- В каждом состоянии верны некоторые факты.

Структура Крипке

Микроволновка.



d=”дверь закрыта”
w=”включено излучение”
b=”зажата кнопка Пуск”

Структура Крипке

Переходы не подписаны, потому что:

- Номер состояния *полностью* описывает состояние программы
- Если переход условный - надо поделить состояние на два
  - То, в котором условие верно (переход *может* произойти)
  - То, в котором условие неверно (переход *не может* произойти)
- Программу *можно* транслировать в структуру Крипке
- Структуры Крипке прекрасно поддаются верификации

LTL

Линейная темпоральная логика (Linear Temporal Logic).
Рассуждает о последовательностях состояний мира во времени.

- Есть запрос $\Rightarrow$ когда-нибудь будет ответ
- Блокировка запрошена $\Rightarrow$ клиент будет обслужен сколь угодно большое число раз, пока не отпустит ее
- ...

Прелести:

- Краткая, понятная, мощная и однозначная
- LTL-свойства легко верифицируются даже вручную
  - Относительно структур Крипке
- Много мощных инструментов

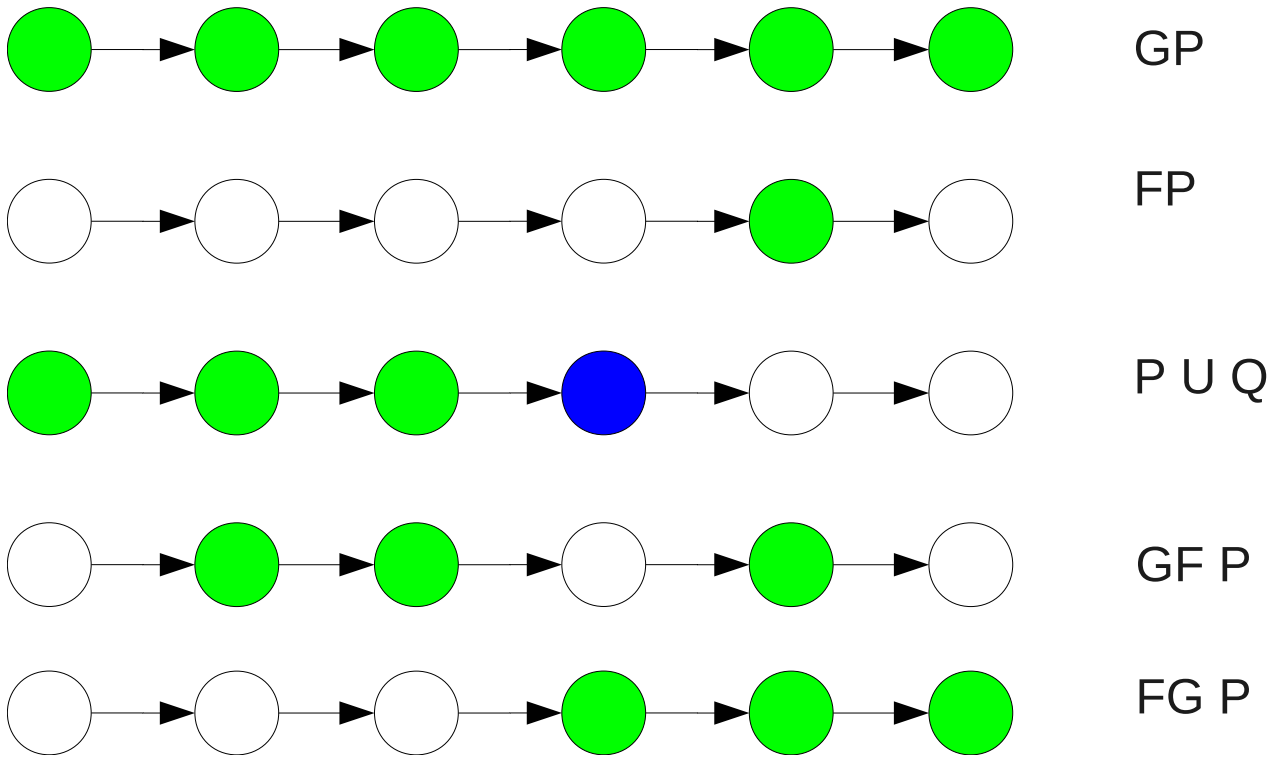
Формулы LTL

$p_1, p_2, \ldots$	Переменные
$A \wedge B, \neg A \ldots$	Логические связки
XA	В следующий момент A (neXt)
$GA \equiv \Box A$	Всегда A (Globally)
$FA \equiv \Diamond A$	Когда-нибудь A (Future)
$A \mathcal{U} B$	A, по крайней мере до наступления B

Бесконечности

- Начиная с некоторого момента, навсегда: $FG\ A \equiv \Diamond\Box A$
- Бесконечно часто: $GF\ A \equiv \Box\Diamond A$

Иллюстрация формул LTL



Свойства

- Живость (liveness)
  - Что-то хорошее когда-нибудь произойдет: $\Diamond Alive$
- Безопасность (safety)
  - Что-то плохое никогда не произойдет: $\Box \neg Dead$
- Справедливость (fairness)
  - То, что может произойти - произойдет
  - Слабая: $\Diamond \Box Req \rightarrow \Box \Diamond Ack$
  - Сильная: $\Box \Diamond Req \rightarrow \Box \Diamond Ack$
  - Антоним - starvation (голодание)

Примеры LTL-свойств

Пример 1

$\Box$ (state==OFF) $\rightarrow$
(state==OFF $\mathcal{U}$ turnOnIsCalled)

Пример 2

$\Box$ (req_interrupt $\rightarrow$
$\Diamond \Box$ (state==COMPLETED $\vee$ state==INTERRUPTED))

Model checking

Наука о проверке свойств программ (например, LTL) на основе их моделей (например, структур Крипке) - *Model checking*
Премия Тьюринга 2007 - за достижения в области model checking, сделавшие его применимым для проектов реального размера.

Инструмент: SPIN

Язык + верификатор LTL-свойств для параллельных программ. Лауреат ACM Software System award (2001) (другие лауреаты: Unix, TeX, Java, Apache, TCP/IP, . . .).

- Си-подобный синтаксис
- Для нарушенных свойств генерируется нарушающая их трасса
- (уродливый, но терпимый) GUI
- Я проверял: Пригодно для применения на практике.

http://spinroot.com/spin/whatispin.html - официальный сайт.

Инструмент: J-LO

Задаем LTL-свойства, они проверяются в рантайме.

- Аннотация `@LTL("...")`
- Инструментирование байткода

Пример

(!call(void Foo.doWork())) U (call(Foo.init()))

Пример пожестче

Lock reversal.

```
pointcut lock(Thread t, Lock l):
  call(Lock.lock(Thread)) && args(t) && target(l);
pointcut unlock(Thread t, Lock l):
  call(Lock.unlock(Thread)) && args(t) && target(l);
Thread i,j; Lock x,y;
!lock(i,y) U (lock(i,x) /\ (!unlock(i,x) U (lock(i,y) /\ !target(x))))
-> G !(!lock(j,x) U (lock(j,y) /\ !args(i) /\ (!unlock(j,y) U lock(j,x) )))
```

AspectJ+LTL

http://www.sable.mcgill.ca/~ebodde/rv/JLO/ - сайт тулы.

Кстати

The work is based on a Haskell-prototype developed at our department.

http://abc.comlab.ox.ac.uk/extensions - Tracecheck: похожая тула от тех же авторов: проверяет регулярные выражения над трассами программы.

Model checking:further reading

http://www.inf.unibz.it/~artale/FM/slide3.pdf
туториал по LTL.
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.3062
статья про J-LO
http://spinroot.com/spin/Doc/course/mc-tutorial.pdf
Model Checking: A tutorial overview (много ссылок)

Ближе к делу

Что брать за аксиомы в реальных программах?

- Thread safety
- Java memory model

Thread safety

Понимайте, формулируйте и документируйте ее, иначе не из чего будет делать выводы о корректности.

- Immutable. У объекта всего 1 состояние.
- Thread-safe. Каждая операция сама по себе безопасна/атомарна. Вместе - нет.
  - Последовательности операций нужно координировать извне
  - Документируйте, какие *именно* последовательности все-таки безопасны
  - Пример: Банковский счет

Thread safety

- Conditionally thread-safe. Почти как Thread-safe.
  - Но есть хотя бы один метод, бесполезный “в одиночку”
- Thread-compatible. Можно использовать многопоточно, если не вызывать операции одновременно.
  - Для всего нужна внешняя синхронизация, но с ней будет работать
- Thread-hostile. Вообще нельзя использовать многопоточно.
- Thread-confined. Можно использовать из одной *конкретной* нити.
  - Пример: Swing/AWT

Java Memory Model

Грубо: Набор аксиом о том, какие происшествия в памяти кому и когда видимы.

- Чтобы компилятор мог выполнять оптимизации
  - Переупорядочение, удаление дохлого кода ...
- Чтобы можно было эффективно использовать кэши, особенно в многопроцессорных системах.

Пример

```
private int x, y;

void foo() {
    x = 5;
    y = 7;
}

void bar() {
    if(x==5) {
       // Неверно, что y==7
    }
}
```

final поля и JMM

Пример

```
public final class Foo {
    private int x;
    Foo(int x) {this.x = x;}
    int getX() {return x;  }
}
```

He thread-safe! x должен быть final.

Java Memory Model

Более точно: Набор аксиом о том, какие *наблюдения* результатов последовательности действий в памяти - возможны.

Цели:

- Безопасность
  - Безопасная инициализация
- Интуитивность
- Эффективная реализуемость

Java Memory Model

Влияющие факторы:

- `synchronized`
- `volatile`
- `final`

Java Memory Model

Основные понятия:

- Действие
  - Обращение к разделяемым переменным
  - Вход/выход из монитора
  - Запуск/ожидание нитей
- Видимость
- Отношение *happens-before*
  - Жизнь 1 нити упорядочена
  - Жизнь 1 монитора упорядочена
  - Жизнь 1 volatile переменной упорядочена

Что все-таки делает volatile?

Чтение *одной* volatile переменной - почти точка синхронизации.

```
class VolatileExample {
    int x = 0;
    volatile boolean v = false;
    public void writer() {
        x = 42; v = true;
    }

    public void reader() {
        if (v == true) { /* x==42 */ }
    }
}
```

Что все-таки делает final?

После конструирования `final` поля *видимы*.

- Видимы также объекты, доступные через них
- Ссылка на объект не должна утекать из конструктора
  - Регистрация листенером, присваивание статического поля...
  - Делайте это после конструирования.

CheckThread

Набор аннотаций для документирования thread safety.

Есть плагин к IDEA.

Очень полезно.

Пример

```
@ThreadSafe class SynchronizedInteger {
  @GuardedBy("this") private int value;
  synchronized int get()       { return value; }
  synchronized void set(int v) { value = v;    }
}
```

Java Path Finder

Виртуальная машина, умеющая символьно выполнять Java-код и верифицировать свойства *для всех возможных запусков с учетом многопоточности*. Сделано в NASA.
http://javapathfinder.sourceforge.net/
http://sourceforge.net/projects/visualjpf/ - GUI
http://www.visserhome.com/willem/presentations/ase06jpftut.ppt - подобие туториала

- Утверждается, что справляется с проектами до 10KLOC.
- Находил баги в NASA-овском софте (в марсоходе, например)
- Основан на SPIN
- Есть ant-task и плагин к eclipse; расширяем

Я написал маленький пример с дедлоком. Сработало.

Другие тулы

Статический анализ многопоточных ошибок есть и в:

- PMD
- FindBugs (есть плагин к IDEA)

Проверено: помогает (и не только от многопоточных).

Философское избавление от багов

Надо уменьшить $O(K^N)$.

- Уменьшить число наблюдаемых состояний (K)
  - Immutability
  - Инкапсуляция состояния
  - Синхронизация
  - Крупные операции (`incrementAndGet`, `putIfAbsent`)
- Навязать трассам эквивалентность
  - Идемпотентные операции
    - `foo();foo();` $\sim$ `foo();`
  - Коммутативные (независимые) операции
    - `foo();bar();` $\sim$ `bar();foo();`
    - В т.ч. инкапсуляция и независимые объекты

Immutability

Нельзя изменить *вообще* $\Rightarrow$ нельзя изменить *неправильно, не в том порядке* ...

- Все поля private, final, и их классы сами по себе immutable
  - Иначе поле может изменить кто-то другой
  - Иначе нарушится thread safety (см.сл.слайд)
  - Иначе изменение может произойти “тайком”
- Сам класс final
  - Иначе можно создать mutable потомка и передать его кому-нибудь в качестве объекта базового класса
- Базовый класс тоже immutable
  - Иначе можно изменить состояние при помощи операций базового класса

Immutability

Нельзя изменить *вообще* $\Rightarrow$ нельзя изменить *неправильно, не в том порядке* ...

- Все поля private, final, и их классы сами по себе immutable
  - Иначе поле может изменить кто-то другой
  - Иначе нарушится thread safety (см.сл.слайд)
  - Иначе изменение может произойти “тайком”
- Сам класс final
  - Иначе можно создать mutable потомка и передать его кому-нибудь в качестве объекта базового класса
- Базовый класс тоже immutable
  - Иначе можно изменить состояние при помощи операций базового класса

Immutability

Нельзя изменить *вообще* $\Rightarrow$ нельзя изменить *неправильно, не в том порядке* ...

- Все поля private, final, и их классы сами по себе immutable
  - Иначе поле может изменить кто-то другой
  - Иначе нарушится thread safety (см.сл.слайд)
  - Иначе изменение может произойти “тайком”
- Сам класс final
  - Иначе можно создать mutable потомка и передать его кому-нибудь в качестве объекта базового класса
- Базовый класс тоже immutable
  - Иначе можно изменить состояние при помощи операций базового класса

Immutability

Нельзя изменить *вообще* $\Rightarrow$ нельзя изменить *неправильно, не в том порядке* ...

- Все поля private, final, и их классы сами по себе immutable
  - Иначе поле может изменить кто-то другой
  - Иначе нарушится thread safety (см.сл.слайд)
  - Иначе изменение может произойти “тайком”
- Сам класс final
  - Иначе можно создать mutable потомка и передать его кому-нибудь в качестве объекта базового класса
- Базовый класс тоже immutable
  - Иначе можно изменить состояние при помощи операций базового класса

Immutability

Нельзя изменить *вообще* $\Rightarrow$ нельзя изменить *неправильно, не в том порядке* ...

- Все поля private, final, и их классы сами по себе immutable
  - Иначе поле может изменить кто-то другой
  - Иначе нарушится thread safety (см.сл.слайд)
  - Иначе изменение может произойти “тайком”
- Сам класс final
  - Иначе можно создать mutable потомка и передать его кому-нибудь в качестве объекта базового класса
- Базовый класс тоже immutable
  - Иначе можно изменить состояние при помощи операций базового класса

Инкапсуляция состояния

Программа корректна ровно настолько, насколько корректны и скоординированы все, кто *могут* повлиять на состояние объекта.

Мораль:

- Чем меньше способов повлиять на состояние, тем лучше.
- Независимые задачи - выносить в независимые объекты

Инкапсуляция состояния

```
class DataLoader {
    ClientToken beginLoad(Client client);
    void writeData(ClientToken token, Data data);
    void commit(ClientToken token);
    void rollback(ClientToken token);
}
```

Инкапсуляция состояния

Сложный код:

- Соответствие `ClientToken`/транзакция
- (многопоточные клиенты) `synchronized(locks.get(token))`

Инкапсуляция состояния

Разорвем лишнюю зависимость между независимыми задачами.

```
class DataLoader {
    PerClientLoader beginLoad(Client client);
}
class PerClientLoader {
    void writeData(Data data);
    void commit();
    void rollback();
}
```

Инкапсуляция состояния

- Простой и незагрязненный код
- Синхронизация простым `synchronized`
- Никаких таблиц
- Никаких проблем с охраной таблиц от многопоточного доступа

Синхронизация (сериализация доступа)

Это все знают: “Хочешь защитить объект - сделай на нем `synchronized`”.


11455

Синхронизация (сериализация доступа)

Если надо защитить доступ к комбинации объектов - надо синхронизироваться *в одинаковом порядке*.
Иначе - дедлоки.

Так делать не надо

```
void foo() {
  synchronized(x) {
    synchronized(y) {
      ...
    }
  }
}
void bar() {
  synchronized(y) {
    synchronized(x) {
      ...
    }
  }
}
```

Борьба с дедлоками

Нужен глобально фиксированный порядок блокировок.
“Экзотический” случай:

Так делать не надо

```
void transfer(Account from,Account to,int money) {
  synchronized(from) {
    synchronized(to) {
      ..
    }
  }
}
```

Борьба с дедлоками: Экзотический случай

Нужен глобально фиксированный порядок блокировок.

Надо делать вот так

```
void transfer(Account from,Account to,int money) {
  synchronized(from<to ? from : to) {
    synchronized(from?to ? to : from) {
      ..
    }
  }
}
```

Подробно - см. Java Concurrency in Practice.

Еще немного о дедлоках

Не стоит вызывать чужие методы (callbacks), держа блокировку (“alien call”). Такие вызовы лучше превратить в “open call” (вызов вне блокировки)

```
private void frobbleBar(Bar bar) {
    List<BarListener> copy;
    synchronized (this) {
       copy = new ArrayList<BarListener>(listeners);
    }
    for (Listener l : snapshot)
       l.barFrobbled(bar);
}
```

Более общий случай:

- Минимизируйте сложность и “неизвестность” кода внутри блокировки

Еще немного о дедлоках

Частая причина дедлоков - ресурсы и пулы.

- Доступ к ресурсам надо упорядочивать, как и блокировки
- Пулы + взаимозависимые задачи = проблема Thread starvation deadlock
  - Пул забит нитями, ждущими результата вычисления
  - Вычисление в очереди, но его некому начать - пул забит

Синхронизация и наследование

Очень трудно сделать эффективный thread-safe класс на основе non-thread-safe базового.

- Придется перегрузить и синхронизировать *все до единой* операции
- В т.ч. и длительные, тяжеловесные
- Не дай бог какие-то из операций `final`

Как с наследованием:

- “Design for inheritance or prohibit it”
- “Design for thread-safety”

Синхронизация и наследование

Паттерн “Thread-safe interface”:

- Поделите компонент на 2 части - методы интерфейса и методы реализации
- Методы интерфейса выполняют синхронизацию
- Методы реализации вызывают *только* другие методы реализации

Крупные операции

Идеальный способ избавить клиентов от необходимости координировать последовательность вызовов - сделать, чтобы им не надо было делать *последовательность* вызовов.

- `AtomicInteger.getAndIncrement`
- `ConcurrentMap.putIfAbsent`
- `Bank.transfer`

Достаточно правильно реализовать саму операцию.
Проектируйте API в таком стиле, если это возможно.

Крупные операции

Частный случай:

- У объекта в меру большое состояние
- Хочется атомарно поменять несколько его частей
- Запихните состояние в объект и поменяйте атомарно ссылку на этот объект
  - `AtomicReference`

http://tobega.blogspot.com/2008/03/java-thread-safe-state-design-pattern.html

Идемпотентные операции

`foo();foo();` $\sim$ `foo();`

Если это так, то можно не предпринимать мер по ограничению числа вызовов `foo()`.

Особенно важно в распределенных и веб-системах (с ненадежным подтверждением вызова `foo()`).

Частый пример - ленивая инициализация.

Идемпотентные операции

- `set.add(x)` - идемпотентна
- `x.setFoo(foo)` - идемпотентна
- `list.add(x)` - не идемпотентна
- `/buy.php?product_id=123` - не идемпотентна

Идемпотентные операции

Достижение идемпотентности:

- Присвоить каждому вызову операции уникальный `id`
- Если такой `id` уже был - это дубль
- Актуально в вебе.

Абстрагирование

Многопоточную программу намного легче отладить, если в ней нет ничего, кроме самой сути.
Баги вылезают на поверхность, нет соблазна сказать “В нашей задаче такая ситуация невозможна”.

- Не “блокировка индекса на чтение или запись”, а просто “блокировка на чтение или запись”
- Не “очередь e-mail’ов”, а “отказоустойчивая очередь задач”
- Не “параллельный подсчет слов”, а MapReduce
- Не “параллельная фильтрация массива”, а ParallelArray

Может оказаться, что такая штука уже есть.

Абстрагирование

А какие штуки уже есть?

Об этом - позже.

Содержание

Тестирование

Тестировать многопоточный код адски сложно.

- Протестировать правильность детерминированного кода - можно, хоть и трудно
- Протестировать отсутствие багов, связанных с недетерминизмом - совсем сложно
- Нет контроля над главным фактором трудноуловимых ошибок - недетерминизмом шедулера
- Необходимо статистическое тестирование
  - И даже оно *ничего* не гарантирует
  - Лучше, конечно, доказывать корректность

Тестирование

Что нужно оттестировать?

- Код последовательно корректен
- Код правильно работает без нагрузки
- Код не ломается под нагрузкой

Именно в таком порядке.

Наличие ожидаемого поведения

Как проверить, что многопоточный код реализует определенный протокол в “идеальных” условиях?

- Понаставить задержек, тем самым сделав порядок событий детерминированным
- Проверить, что происходят нужные события в нужном порядке
  - Проверять, что блокирующие вызовы - блокируются
  - Проверять, что неблокирующие вызовы - не блокируются
  - Mock objects, listeners во всех важных точках кода

Криво, длинно, уродливо, но работает.

Устойчивость к нагрузке

Как проверить, что многопоточный код не ломается под нагрузкой?

- Придумать инварианты
- Организовать нагрузку
- Постоянно проверять инварианты

Криво, длинно, уродливо, ничего не гарантирует, но может выявить ошибку.

- Повторить или отладить ее будет почти невозможно

Устойчивость к нагрузке

Пример: Тестируем `ConcurrentMap`.

- Инвариант: Каждый добавленный, но не удаленный элемент присутствует в мапе
- Инвариант: `size()` равно числу добавленных, но не удаленных элементов.

Устойчивость к нагрузке

Пример: Тестируем `ConcurrentMap`

- Повторить 100000000000000 раз
  - Сгенерировать 10000 случайных значений, разбить на 10 частей случайным образом
  - В 10 нитей:
    - добавлять в мапу значения из i-й части со случайными небольшими задержками
    - проверять, что все предыдущие значения из i-й части в мапе есть
  - По окончании проверить наличие каждого значения
  - Пригодится `CyclicBarrier`

Повторяемость

Как повторить готовую ошибку?

- Написать делегирующие mock-объекты, записывающие все действия
- Дождаться ошибки
- Написать mock-объекты, “проигрывающие” заданный сценарий

Поможет найти “гонки” в протоколе. Не поможет найти низкоуровневые “гонки” (между statement’ами).

- Без мозгов не обойтись!

Как спровоцировать ошибку

Как организовать максимум недетерминизма шедулера?

- Понаставить `Thread.yield()` или `Thread.sleep(1)` в разных местах кода
- Использовать $\approx 2 * NCPU$ нитей
  - Чтобы и все CPU были заняты, а нити и вытеснялись, и не простаивали слишком долго

Тулы

- TestNG имеет небольшие специальные средства
  - `@Test(threadPoolSize=5,invocationCount=10,threadMap=2,2,2,3,1)`
- MultithreadedTC (http://code.google.com/p/multithreadedtc/)
  - `assertTick(n)`, `waitForTick(n)`
  - Понаставить тиков можно только в тестовом коде
- Упомянутый J-LO, tracecheck ...

Содержание

Модели многопоточных вычислений

Параллелизм:

- MapReduce: Огромные объемы данных, распределенная обработка
- Fork/Join: Задачи с рекурсивной структурой
- Векторные модели: huge-scale SIMD
- Конвейер / data flow
- ...

Concurrency:

- Message-passing (actor model)
- Event loop (message-passing с одним / одинаковыми акторами)
  - Сервера, подавляющее большинство GUI-библиотек, в т.ч. Swing
- Software Transactional Memory

MapReduce

Слишком известно, чтобы на нем останавливаться

Реализация для Java: Hadoop

Fork/Join

```
if(isSmall(problem)) {
    return solveSequentially(problem);
} else {
    // Fork
    List<Problem> subproblems = split(problem);
    List<Result> res = parallelMap(solve, subproblems);
    // Join
    return combine(res);
}
```

Fork/Join

Пример: Шахматы. Предпросмотр на n шагов.

- n==0 $\Rightarrow$ оценить позицию
- Fork: По подзадаче на каждый ход
- Join: Выбрать ход с самой высокой оценкой

Fork/Join

Пример: Интегрирование.

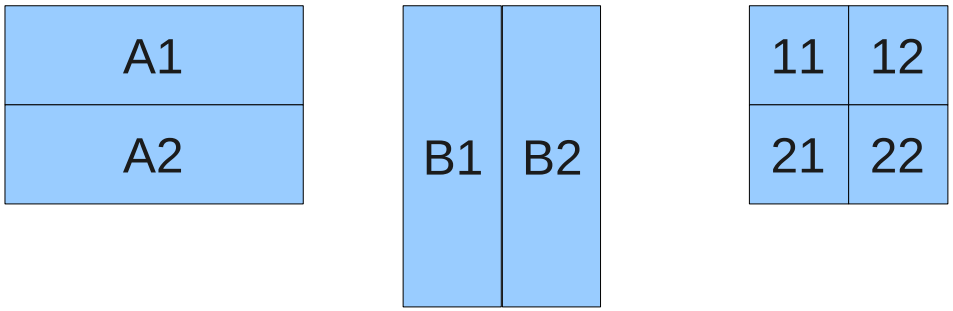
- Интервал мал $\Rightarrow$ проинтегрировать влоб
- Fork: по подзадаче на обе половины интервала
- Join: сложить результаты

Fork/Join

Пример: Умножение матриц: $C+ = AB$ (В т.ч. поиск путей в графах и т.п.)

- A и B малы $\Rightarrow$ влоб
- Fork: Поделить A на верх/низ и B на лево/право
- Join: Ничего, просто дождаться результатов Fork

Fork/Join



Cij += Ai Bj

Fork/Join in JDK7

Примерно так. Официального окончательного Javadoc до сих пор нет.

```
class Solver extends RecursiveAction {
    private final Problem problem;
    int result;
    protected void compute() {
        if (problem.size < THRESHOLD) {
            result = problem.solveSequentially();
        } else {
            int m = problem.size / 2;
            Solver left, right;
            left = new Solver(problem.subProblem(0, m));
            right = new Solver(problem.subProblem(m, problem.size));
            forkJoin(left, right);
            result = //get the result from left right
        }
    }
}
ForkJoinExecutor pool = new ForkJoinPool(
  Runtime.availableProcessors());
Solver solver = new Solver(new Problem());
pool.invoke(solver);
```

Векторные алгоритмы

Huge-scale SIMD: Примитивные операции оперируют сразу над большими массивами.

- Map: Применение функции к каждому элементу / склейка N массивов по N-арной операции
- Fold: Ассоциативная свертка в моноиде.

$$\star \text{ ассоциативна:} \quad a \star (b \star c) = (a \star b) \star c$$

$$u \text{ - единица } \star: \quad a \star u = u \star a = a$$

$$fold \ \star \ u \ [x_0, x_1, \ldots, x_n] = u \star x_0 \star x_1 \star \ldots \star x_n$$

- Scan (Prefix sum): Пробег (Префиксная сумма)

$$scan \ \star \ u \ [x_0, x_1, \ldots, x_n] = [u, u \star x_0, u \star x_0 \star x_1, \ldots, u \star \ldots \star x_n]$$

Векторные модели

Прелести:

- Свертка параллелизуется: Ускорение $O(P/logP)$ на P процессорах
- Пробег так же параллелизуется
- Удивительно мощные и универсальные операции

Префиксные суммы

Что можно сделать сверткой/пробегом (т.е. параллельно):

- sum,min,max,avg,...
- Radix sort, Quicksort
- Сложение длинных чисел
- Выпуклая оболочка
- Многие алгоритмы на графах
- Многие рекуррентные последовательности n-го порядка
- Трехдиагональные системы уравнений
- ...

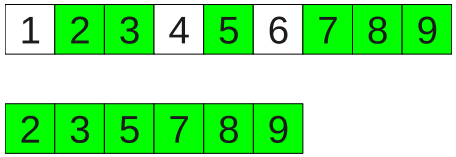
Префиксные суммы

Особенно интересно, *как* это делается. Базовые операции поверх пробега:

- Рекуррентная последовательность 1го порядка
- Упаковка массива
- Сортировка массива по 1-битному флагу (“разбиение”)
- Сегментированный пробег (пробег со сбросами)

Префиксные суммы

Упаковка массива (“pack”, “filter”).



Префиксные суммы

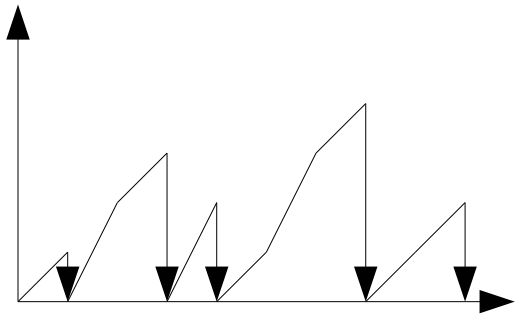
Сортировка по 1-битному флагу (“разбиение”, “partition”).

Radixsort тривиален.

Префиксные суммы

Сегментированный пробег (Segmented scan).

1	2	3	4	5	6	7	8	9
1	2	5	4	5	6	7	15	24



Позволяет реализовать рекурсию без рекурсии.

Префиксные суммы

В целом - красивый, мощный, простой в реализации и необычный векторный “язык”.
Часто используют на GPU.
http://sourceforge.net/projects/amino-cbbs/ - реализация на Java (содержит много других готовых параллельных алгоритмов).
http://www.cs.cmu.edu/~blelloch/papers/Ble90.pdf - прекрасная книга Vector models for data-parallel computing.
Строжайше рекомендую.

ParallelArray

New in JDK7. Реализация некоторых векторных примитивов поверх Fork/Join framework.

- Стандартные: Map, Zip, Filter, Fold (`reduce`), Scan (`cumulate` + `precumulate`), Sort
- Поинтереснее: Сечение по диапазону (`withBounds`), Перестановка (`replaceWithMappedIndex`)
- Функциональный до мозга костей API (подавляющее большинство функций - ФВП)
  - Правда, без фанатичного следования функциональной чистоте
- Ленивые операции
- Сплавка операций (*fusion*)

ParallelArray

Примерчик.

```
ParallelArray<Student> students = new ParallelArray<Student>(fjPool, data);
double bestGpa = students.withFilter(isSenior)
                         .withMapping(selectGpa)
                         .max();

public class Student {
    String name;
    int graduationYear;
    double gpa;
}

static final Ops.Predicate<Student> isSenior = new Ops.Predicate<Student>() {
    public boolean op(Student s) {
        return s.graduationYear == Student.THIS_YEAR;
    }
};
static final Ops.ObjectToDouble<Student> selectGpa = new Ops.ObjectToDouble<Student>() {
    public double op(Student student) {
        return student.gpa;
    }
};
```

Message-passing concurrency

Фактически единственно возможный подход в распределенных системах.

- Вообще нету разделяемого состояния
- *Акторы* обмениваются *сообщениями*
- У каждого актора есть mailbox (FIFO-очередь сообщений)
- Отсылка сообщений асинхронная и неблокирующая

Message-passing concurrency

- Не панацея: дедлоки никуда не деваются, просто они другие
- Тем не менее, написать корректную программу - проще
- Есть формальные модели $\Rightarrow$ поддается верификации

Message-passing concurrency

Реализации:

- Куча языков: MPI
- Termite Scheme
- Clojure: agents
  - Очень интересная и мощная вариация. Definitely worth seeing. http://clojure.org/agents
  - “Clojure is to concurrent programming as Java was to OOP”
- Scala: actors
- Java: Kilim
- И, конечно, Erlang
  - Killer feature: Selective receive

Message-passing concurrency

Kilim:

- Легковесные нити (1млн в порядке вещей; 1000x faster)
- Очень быстрый message passing (утверждается, что 3x Erlang)
- Постпроцессит байт-код (CPS transform для методов, аннотированных `@pausable`).

Message-passing concurrency

http://kilim.malhar.net/ (в данный момент лежит . . .)
http://www.cl.cam.ac.uk/research/srg/opera/publications/papers/kilim_ecoop08.pdf - научная статья. Не для робких, но читать можно. Много интересных идей.

http://eprints.kfupm.edu.sa/30514/1/30514.pdf - оригинальная книга Тони Хоара про Communicating Sequential Processes - теоретическая основа message-passing

Event loop

Вариация на тему message passing, но - не много взаимодействующих акторов, а один (или несколько одинаковых) жирный и всемогущий.

- Серверы (и `accept`, и `select/poll/...` сюда относятся)
- GUI-фреймворки

Прелести:

- Это не фреймворк, но паттерн: легко реализовать и использовать даже на микроуровне
- К нему сводится куча задач
- Очень легко пишется
- Большинство многопоточных проблем отсутствуют

Event loop

```
private BlockingQueue<OurEvent> events =
  new LinkedBlockingQueue<OurEvent>();

void mainLoop() {
    while(true) {
        processEvent(events.poll());
    }
}

public class OurEvent {...}
```

Software Transactional Memory

Транзакции на уровне памяти; атомарные куски кода.

- Полноценный commit/rollback, полная атомарность
- Транзакции *повторяемы*
- Транзакционные программы комбинируемы!
  - В отличие от shared state (правильно+правильно≠правильно)
- Не должно быть *никаких* побочных эффектов, кроме обращений к транзакционной памяти

Software Transactional Memory

Псевдокод:

```
// Insert a node into a doubly-linked list atomically
atomic {
    newNode->prev = node;
    newNode->next = node->next;
    node->next->prev = newNode;
    node->next = newNode;
}
```

Software Transactional Memory

Псевдокод (впервые предложено в реализации в Haskell):

```
atomic {
    if (queueSize > 0) {
        remove item from queue and use it
    } else {
        retry
    }
}
```

`retry` перезапускает транзакцию в момент, когда изменится одна из *прочитанных* ею переменных.

Software Transactional Memory

Реализации:

- Haskell: полноценная и безопасная поддержка
- Clojure: полноценная и небезопасная поддержка
- Java: Deuce, JSTM, ...
- Много библиотек для C и C++, в т.ч. даже компиляторы с ключевым словом `atomic`
  - Не знаю, используют ли их

Software Transactional Memory

Further reading:

http://research.microsoft.com/Users/simonpj/papers/stm/stm.pdf - Composable Memory Transactions.

Содержание

Производительность

Что тормозит?

- Ждущие нити
- Захват (даже успешный) блокировок
- Переключение контекстов

Производительность

Что делать?

- Не ждать и не блокироваться
- Не переключаться

Уменьшение блокировок

Как не ждать?

- Использовать неблокирующие алгоритмы и операции
- Увеличить гранулярность блокировок
  - Антоним: Один монитор на всю программу
- Использовать алгоритмы, блокирующие не всех
  - ReaderWriterLock

Неблокирующие алгоритмы и операции

Базис:

- `get`, `set`, `getAndSet`
- `boolean compareAndSet(expect, update)` (CAS)
  - Поменять *с expect* на update
  - Если не expect - значит, мы прозевали, как кто-то поменял из другой нити
  - Надо перезапустить кусок алгоритма в надежде, что теперь обойдется без коллизий
- `addAndGet`, `getAndAdd`, ...

Неблокирующие алгоритмы и операции

Средства в Java:

- `java.util.concurrent.atomic`
- `AtomicInteger`, `AtomicLong`
- `AtomicReference`, `AtomicStampedReference`
- (малоизвестно) `AtomicIntegerArray`, Long, Reference

Атомарные операции

Пример алгоритма на CAS:

```
public class ConcurrentStack<E> {
    AtomicReference<Node<E>> head = new AtomicReference<Node<E>>();

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = head.get();
            newHead.next = oldHead;
        } while (!head.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        ...
    }
}
```

http://www.ibm.com/developerworks/java/library/j-jtp04186/index.html

Неблокирующие контейнеры

Готовые неблокирующие структуры данных (New in JDK6):

- ConcurrentSkipListMap
- ConcurrentSkipListSet

Прелести:

- Итераторы никогда не бросают CME

Уменьшение гранулярности блокировок

- Блокируйте не всё
- Блокируйте не всех

Уменьшение гранулярности блокировок

"Блокируйте не всё":
Если надо защитить большую структуру данных, но меняются только ее куски - защищайте куски или группы кусков.
Пример: `ConcurrentHashMap` (new in JDK6). Техника "lock striping".

- Чтение неблокирующее
- Запись блокирующая в $1/k$ случаев, где k настраивается
  - Рекомендуется $k \approx$ число нитей-писателей
- k блокировок защищают k кусков хэш-таблицы
- Итераторы также не бросают CME

Reader/Writer lock

“Блокируйте не всех”:
Чтение совместимо с чтением, но не с записью.
Бывают разные:

- Readers-preference
- Writers-preference
- Arrival-order
- ...

Класс `ReentrantReadWriteLock`. Реализует примерно arrival-order. Если есть ждущий писатель - новые читатели только после него.

Переключение контекстов

Как поменьше переключать контексты?

- Иметь поменьше нитей
  - Например, одну (Event-driven)
- Побольше асихронности

Переключение контекстов

Как поменьше переключать контексты?

Пусть нити не отвлекаются.

- Одни пихают задания в очередь
- Другие вынимают и исполняют их

Цифры

Сколько реально стоят различные операции?

Атомарные операции и volatile	$\approx$ 0.1мкс; incAndGet 2-3x быстрее CAS
Переключение контекста	Несколько мкс (независимо от числа нитей)
Uncontended synchronized	$\approx$ 0.1-1мкс
Contented synchronized	$\approx$ 20мкс

http://mailinator.blogspot.com/2008/03/how-fast-is-java-volatile-or-atomic-or.html - сравнение скорости различных способов синхронизации

Что умеет JVM?

- Biased locking
- Lock coarsening
- Lock elision (escape analysis)

http://www.infoq.com/articles/java-threading-optimizations-p1 - Do Java threading optimizations actually work?

Что умеет JVM?

Biased locking (new in 1.5.0_6):

- Большинство объектов за всю жизнь блокирует только 1 нить
- Делается так, чтобы для этой нити синхронизация была *очень* быстрой (даже без CAS)
- `-XX:+UseBiasedLocking` (по умолчанию включено начиная с Java SE 6)

http://tinyurl.com/biasedlocking - от разработчиков JVM, по-русски

Что умеет JVM?

Lock coarsening:

```
public void setupFoo(Foo foo) {
    foo.setBarCount(1);
    foo.setQuxEnabled(true);
    foo.addGazonk(new Gazonk());
}
```

. . . и все 3 метода `synchronized`.

Тут незачем делать `synchronized` 3 раза. Можно запихнуть все в 1 `synchronized`.

Опция `-XX:+EliminateLocks` (по умолчанию включено).

http://citeseer.ist.psu.edu/459328.html - статья про технику.

Что умеет JVM?

Escape analysis: Если объект не убегает в другую нить, то нет смысла делать на нем `synchronized`.

```
DeepThought dt = new DeepThought();
dt.turnOnPower();
dt.enterData(data);
dt.wakeup();
return dt.computeUltimateAnswer();
```

Опция `-XX:+DoEscapeAnalysis`.

Содержание

Cutting edge

Что умеет Haskell:

- Все побочные эффекты контролируются
- Нет неконтролируемых проблем из-за их наличия
- Позволяет немыслимые в других языках вещи

Cutting edge

Что умеет Haskell:

- Самые быстрые в мире легковесные нити и примитивы синхронизации
  - 1 место на Language Shootout, 60x быстрее нативных Ubuntu, 5x быстрее Erlang!
- Параллельные стратегии
  - Простые *параллельные* вычисления: нет аналогов в других языках
- Data Parallel Haskell
  - Автоматически распараллеливаемые векторные вычисления
- Транзакционная память
- Работа в направлении GPU

Легковесные нити

Nothing special, просто очень быстрые и почти 0 синтаксического оверхеда.
C++0x futures/promises and more в 60 строк.

Параллельные стратегии

Ленивость позволяет отделить алгоритм и способ распараллеливания.
Примитив par: x `par` y - параллельно вычислить x, y.

```
results = map f args `using` parList rnf


-- | Applies a strategy to every
-- element of a list in parallel.
parList :: Strategy a -> Strategy [a]
parList st []     = ()
parList st (x:xs) = st x `par` (parList st xs)
```

Идея - ядерная. Это можно реализовать и на Java, если постараться.

Data Parallel Haskell

- Векторизатор (транслирует даже рекурсивные функции в векторные операции)
- Библиотека параллельных векторных операций

```
dotp_double :: [:Double:] -> [:Double:] -> Double
dotp_double xs ys = sumP [:x * y | x <- xs | y <- ys:]

smMul :: [:[:(Int,Float):]:] -> [:Float:] -> Float
smMul sm v = sumP [: svMul sv v | sv <- sm :]
```

http://research.microsoft.com/en-us/um/people/simonpj/papers/ndp/NdpSlides.pdf - слайды

Data Parallel Haskell

```
sort :: [:Float:] -> [:Float:]
sort a = if (length a <= 1) then a
       else sa!0 +++ eq +++ sa!1
 where
    m = a!0
    lt = [: f | f<-a, f<m :]
    eq = [: f | f<-a, f==m :]
    gr = [: f | f<-a, f>m :]
    sa = [: sort a | a <- [:lt,gr:] :]
```

Векторизуется и параллелизуется.

Транзакционная память

```
transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = atomically $ do
  deposit  to   amount
  withdraw from amount
```

Транзакционная память

```
check True = return ()
check False = retry

limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount = do
    bal <- readTVar acc
    check (amount <= 0 || amount <= bal)
    writeTVar acc (bal - amount)
```

Четкое разделение между:

- IO - код с произвольными побочными эффектами
- STM - код, единственный эффект которого - работа с транзакционной памятью
- `atomically :: STM t -> IO t`

Что было

- Классификация многопоточных программ
- Формальные методы (LTL) и тулы (SPIN,JPF,CheckThread)
- Java memory model
- Корректность: инкапсуляция состояния, дедлоки, атомарные и идемпотентные операции
- Тестирование: Фазы (последовательная работа, корректная работа, устойчивость к нагрузке), повторяемость, тулы
- Методики: fork/join, векторные алгоритмы, message-passing, event-driven, STM
- Производительность: влияющие факторы, гранулярность блокировок, цифры и конкретика JVM
- Haskell: легкие нити, параллельные стратегии, векторизация, STM

The end

Спасибо! Вопросы?