МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

СЕРГИЕНКО ЛЕВ ЭДУАРДОВИЧ

Отчет по
Лабораторная работа 8
Параллельные вычисления в Java
Алгоритмы синхронизации потоков

Преподаватель

Кондратьева О.М.

2025

Maurice Herlihy, Nir Shavit, Victor Luchangco, Michael Spear – The Art of Multiprocessor Programming-Elsevier Inc. (2021)
Chapter 9 Linked lists: The role of locking - 214

Алгоритмы синхронизации потоков

- Грубая (Coarse-grained) синхронизация;
- Тонкая (Fine-grained) синхронизация;
- Оптимистичная (Optimistic) синхронизация;
- Ленивая (Lazy) синхронизация;
- Неблокирующая (Nonblocking) синхронизация

Задание 1.

Определить проблему

Ответ

В параллельных и многопоточных приложениях простая стратегия «один замок на всю структуру» (coarse-grained synchronization) порождает серьёзный методологический и практический вопрос: как обеспечить масштабируемость и эффективность при высоком уровне конкуренции между потоками?

- При низкой конкуренции единичный блокировочный механизм работает вполне приемлемо, но когда число потоков растёт, все они вынуждены выстраиваться в очередь, ожидая освобождения замка.
- В результате объект становится «узким местом» (bottleneck), и даже сверхэффективный lock не спасает: производительность падает почти до уровня последовательной реализации.

«Класс, использующий один-единственный lock для всех своих операций, не масштабируется, когда слишком много потоков пытаются обращаться к объекту одновременно» (Goetz et al., *The Art of Multiprocessor Programming*, гл. 9.1).

Задание 2.

Описать суть каждого алгоритма

Ответ

1. Coarse-grained synchronization
    - Идея: одна глобальная блокировка на всю структуру.
    - Плюсы: простота и очевидная корректность (всё выполняется под lock’ом).
    - Минусы: полное отрезание параллелизма → при высокой нагрузке очереди потоков → низкая пропускная способность.
2. Fine-grained synchronization
    - Идея: каждому «кусочку» (узлу, сегменту) структуры соответствует свой lock.
    - Плюсы: потоки могут одновременно работать с разными частями структуры.
    - Минусы: накладные расходы на частые lock()/unlock(), всё ещё возможна конкуренция при смежных узлах.
3. Optimistic synchronization
    - Идея: сначала идти без блокировок (оптимистично), найти «место» в структуре, затем взять минимальный набор lock’ов и валидировать (проверить, что ничего не изменилось).
    - При неудаче (валидация провалилась) — откат и повторный проход.
    - Плюсы: снижение стоимости обхода при редких изменениях; изолирует дорогостоящую часть (lock) на минимальный участок.
    - Минусы: возможны «ретраи» — повторные попытки, если структура активно меняется.
4. Lazy synchronization
    - Идея: логическое и физическое удаление разделены.
        1. Сначала помечаем узел (например, флагом marked = true), но не убираем из цепочки.
        2. Позже (lazy), когда приходится брать lock на предшественника, «выдергиваем» физически.
    - contains() при этом становится wait-free: он идёт по цепочке без блокировок и смотрит только на marked.
    - Плюсы: максимально лёгкие и быстрые запросы чтения; возможность группировать физические удаления.
    - Минусы: добавление и удаление всё ещё блокирующие.
5. Nonblocking (lock-free / wait-free) synchronization
    - Идея: совсем без lock’ов, синхронизация через атомарные CAS-операции (compareAndSet).
    - Поток, не найдя ожидаемых значений, просто повторяет свою операцию.

- Lock-free: гарантируется, что «кто-то» всегда делает прогресс;
  Wait-free: каждый поток завершается за конечное число шагов.
- Плюсы: высочайший уровень параллелизма, нет дедлоков и приоритетных инверсий.
- Минусы: реализация и доказательства корректности значительно сложнее.

Задание 3

Описать эксперимент для демонстрации работы алгоритмов.
Подсказка:

- много элементов;
- много потоков;
- в типичных приложениях, использующих наборы, вызовов contains() значительно больше, чем вызовов add() или remove().

Ответ (скелет программы)

Цели эксперимента
Сравнить пропускную способность и время отклика разных алгоритмов под нагрузкой. Проверить, как меняется производительность при росте числа потоков и соотношения операций.
Сценарий

- Количество элементов в структуре.
- Число потоков.
- Соотношение операций.

Метрики

- Общая throughput.
- Latency.
- Число retries алгоритмах.

```
public class Benchmark {
    static final int INIT_SIZE = 100_000;
    static final int THREADS =
Runtime.getRuntime().availableProcessors();
    static final int OPS_PER_THREAD = 1_000_000;
    static final double CONTAINS_RATIO = 0.9;
    static final double ADD_RATIO = 0.05;
```

```
    static final double REMOVE_RATIO = 0.05;

    public static void main(String[] args) throws InterruptedException {
        Set<Integer> set = createConcurrentSet(); // CoarseList,
FineList, OptimisticList, LazyList, LockFreeList

        for (int i = 0; i < INIT_SIZE; i++) {
            set.add(ThreadLocalRandom.current().nextInt());
        }

        Thread[] threads = new Thread[THREADS];
        long start = System.nanoTime();

        for (int t = 0; t < THREADS; t++) {
            threads[t] = new Thread(() -> {
                ThreadLocalRandom rnd = ThreadLocalRandom.current();
                for (int i = 0; i < OPS_PER_THREAD; i++) {
                    int key = rnd.nextInt();
                    double op = rnd.nextDouble();
                    if (op < CONTAINS_RATIO) {
                        set.contains(key);
                    } else if (op < CONTAINS_RATIO + ADD_RATIO) {
                        set.add(key);
                    } else {
                        set.remove(key);
                    }
                }
            });
            threads[t].start();
        }

        for (Thread th : threads) {
            th.join();
        }
        long duration = System.nanoTime() - start;
        double seconds = duration / 1e9;
        long totalOps = (long) THREADS * OPS_PER_THREAD;

        System.out.printf("Threads=%d, Ops=%,d, Throughput=%,.0f
ops/sec%n",
                THREADS, totalOps, totalOps / seconds);
    }
}
```

Задание 4.

Выполнить эксперименты для сравнения двух (любых) алгоритмов.

Текст программы

Set.java

```java
package com;

public interface Set<T> {
    boolean add(T x);

    boolean remove(T x);

    boolean contains(T x);
}
```

CoarseList.java

```java
package com;

import java.util.concurrent.locks.ReentrantLock;

public class CoarseList<T> implements Set<T> {
    private static class Node<T> {
        final T item;
        final int key;
        Node<T> next;

        Node(int key) {
            this.item = null;
            this.key = key;
        }

        Node(T item) {
            this.item = item;
            this.key = item.hashCode();
        }
    }

    private final Node<T> head;
    private final ReentrantLock lock = new ReentrantLock();

    public CoarseList() {
        head = new Node<>(Integer.MIN_VALUE);
        head.next = new Node<>(Integer.MAX_VALUE);
    }

    @Override
    public boolean add(T item) {
        int key = item.hashCode();
```

```
        lock.lock();
        try {
            Node<T> pred = head;
            Node<T> curr = pred.next;
            while (curr.key < key) {
                pred = curr;
                curr = curr.next;
            }
            if (curr.key == key) {
                return false;
            } else {
                Node<T> node = new Node<>(item);
                node.next = curr;
                pred.next = node;
                return true;
            }
        } finally {
            lock.unlock();
        }
    }

    @Override
    public boolean remove(T item) {
        int key = item.hashCode();
        lock.lock();
        try {
            Node<T> pred = head;
            Node<T> curr = pred.next;
            while (curr.key < key) {
                pred = curr;
                curr = curr.next;
            }
            if (curr.key == key) {
                pred.next = curr.next;
                return true;
            } else {
                return false;
            }
        } finally {
            lock.unlock();
        }
    }

    @Override
    public boolean contains(T item) {
        int key = item.hashCode();
        lock.lock();
        try {
            Node<T> curr = head.next;
            while (curr.key < key) {
                curr = curr.next;
```

```
            }
            return curr.key == key;
        } finally {
            lock.unlock();
        }
    }
}
```

Fine.kava

```
package com;

import java.util.concurrent.locks.ReentrantLock;

public class FineList<T> implements Set<T> {
    private static class Node<T> {
        final T item;
        final int key;
        Node<T> next;
        final ReentrantLock lock = new ReentrantLock();

        Node(int key) {
            this.item = null;
            this.key = key;
        }

        Node(T item) {
            this.item = item;
            this.key = item.hashCode();
        }
    }

    private final Node<T> head;

    public FineList() {
        head = new Node<>(Integer.MIN_VALUE);
        head.next = new Node<>(Integer.MAX_VALUE);
    }

    @Override
    public boolean add(T item) {
        int key = item.hashCode();
        head.lock.lock();
        Node<T> pred = head;
        try {
            Node<T> curr = pred.next;
            curr.lock.lock();
            try {
                while (curr.key < key) {
                    pred.lock.unlock();
                    pred = curr;
                    curr = curr.next;
```

```
                curr.lock.lock();
            }
            if (curr.key == key) {
                return false;
            }
            Node<T> node = new Node<>(item);
            node.next = curr;
            pred.next = node;
            return true;
        } finally {
            curr.lock.unlock();
        }
    } finally {
        pred.lock.unlock();
    }
}

@Override
public boolean remove(T item) {
    int key = item.hashCode();
    head.lock.lock();
    Node<T> pred = head;
    try {
        Node<T> curr = pred.next;
        curr.lock.lock();
        try {
            while (curr.key < key) {
                pred.lock.unlock();
                pred = curr;
                curr = curr.next;
                curr.lock.lock();
            }
            if (curr.key == key) {
                pred.next = curr.next;
                return true;
            }
            return false;
        } finally {
            curr.lock.unlock();
        }
    } finally {
        pred.lock.unlock();
    }
}

@Override
public boolean contains(T item) {
    int key = item.hashCode();
    head.lock.lock();
    Node<T> pred = head;
    try {
```

```
            Node<T> curr = pred.next;
            curr.lock.lock();
            try {
                while (curr.key < key) {
                    pred.lock.unlock();
                    pred = curr;
                    curr = curr.next;
                    curr.lock.lock();
                }
                return curr.key == key;
            } finally {
                curr.lock.unlock();
            }
        } finally {
            pred.lock.unlock();
        }
    }
}
```

Bench.java

```
package com;

import java.util.Random;
import java.util.concurrent.atomic.AtomicLong;

public class Benchmark {
    private static final int[] THREAD_COUNTS = { 1, 2, 4, 8 };
    private static final int[] DURATIONS_SEC = { 10 };
    private static final int[] INITIAL_SIZES = { 10000, 100000 };
    private static final int[] KEY_RANGES = { 1000, 10000 };
    private static final double[][] OP_PROBS = {
            { 0.8, 0.1, 0.1 },
            { 0.5, 0.25, 0.25 },
            { 0.9, 0.05, 0.05 },
    };

    public static void main(String[] args) throws InterruptedException {
        for (int threads : THREAD_COUNTS) {
            for (int duration : DURATIONS_SEC) {
                for (int initSize : INITIAL_SIZES) {
                    for (int keyRange : KEY_RANGES) {
                        for (double[] probs : OP_PROBS) {
                            double cProb = probs[0], aProb = probs[1], rProb =
probs[2];
                            System.out.printf(
                                    "Threads=%d, Duration=%ds, InitSize=%d,
KeyRange=%d, Ops(c/a/r)=%.2f/%.2f/%.2f\n",
                                    threads, duration, initSize, keyRange,
cProb, aProb, rProb);
                            runConfig(new com.CoarseList<Integer>(), threads,
duration, initSize, keyRange, cProb,
```

```
                                    aProb, rProb);
                            runConfig(new com.FineList<Integer>(), threads,
duration, initSize, keyRange, cProb, aProb,
                                    rProb);
                            System.out.println();
                        }
                    }
                }
            }
        }
    }

    private static void runConfig(Set<Integer> set,
            int numThreads,
            int durationSec,
            int initialSize,
            int keyRange,
            double containsProb,
            double addProb,
            double removeProb) throws InterruptedException {

        Random rnd = new Random();
        for (int i = 0; i < initialSize; i++) {
            set.add(rnd.nextInt(keyRange));
        }

        AtomicLong ops = new AtomicLong();
        Thread[] threads = new Thread[numThreads];
        long endTime = System.nanoTime() + durationSec * 1_000_000_000L;

        for (int t = 0; t < numThreads; t++) {
            threads[t] = new Thread(() -> {
                Random r = new Random();
                while (System.nanoTime() < endTime) {
                    int key = r.nextInt(keyRange);
                    double p = r.nextDouble();
                    if (p < containsProb) {
                        set.contains(key);
                    } else if (p < containsProb + addProb) {
                        set.add(key);
                    } else {
                        set.remove(key);
                    }
                    ops.incrementAndGet();
                }
            });
            threads[t].start();
        }

        for (Thread th : threads) {
            th.join();
```

```
        }

        long totalOps = ops.get();
        System.out.printf("  %-10s: total ops=%8d, ops/sec=%.2f\n",
                set.getClass().getSimpleName(), totalOps, totalOps / (double)
durationSec);
    }
}
```

Таблица с результатами экспериментов.

Threads = 1

InitSize	KeyRange	Ops (c/a/r)	CoarseList (Ops/sec)	FineList (Ops/sec)	Победитель	Разница (%)
10000	1000	0.80 / 0.10 / 0.10	1,524,933.70	609,556.10	Coarse	150.17%
10000	1000	0.50 / 0.25 / 0.25	1,614,495.90	612,823.60	Coarse	163.44%
10000	1000	0.90 / 0.05 / 0.05	1,459,990.90	602,416.40	Coarse	142.36%
10000	10000	0.80 / 0.10 / 0.10	73,165.20	50,420.90	Coarse	45.11%
10000	10000	0.50 / 0.25 / 0.25	76,765.10	50,384.10	Coarse	52.36%
10000	10000	0.90 / 0.05 / 0.05	66,421.20	49,617.40	Coarse	33.87%
100000	1000	0.80 / 0.10 / 0.10	1,656,902.90	608,564.40	Coarse	172.26%
100000	1000	0.50 / 0.25 / 0.25	1,709,602.10	592,157.30	Coarse	188.71%
100000	1000	0.90 / 0.05 / 0.05	1,499,867.40	590,152.70	Coarse	154.15%
100000	10000	0.80 / 0.10 / 0.10	42,596.20	27,033.10	Coarse	57.57%
100000	10000	0.50 / 0.25 / 0.25	54,385.00	51,479.60	Coarse	5.64%
100000	10000	0.90 / 0.05 / 0.05	62,141.60	43,234.80	Coarse	43.73%

Threads = 2

InitSize	KeyRange	Ops (c/a/r)	CoarseList (Ops/sec)	FineList (Ops/sec)	Победитель	Разница (%)
10000	1000	0.80 / 0.10 / 0.10	760,447.40	572,865.30	Coarse	32.74%
10000	1000	0.50 / 0.25 / 0.25	827,920.60	543,376.80	Coarse	52.37%
10000	1000	0.90 / 0.05 / 0.05	719,649.40	549,023.30	Coarse	31.08%
10000	10000	0.80 / 0.10 / 0.10	51,442.20	55,625.20	Fine	8.13%
10000	10000	0.50 / 0.25 / 0.25	59,747.90	57,199.00	Coarse	4.46%
10000	10000	0.90 / 0.05 / 0.05	50,849.70	50,064.50	Coarse	1.57%
100000	1000	0.80 / 0.10 / 0.10	776,178.30	556,188.20	Coarse	39.55%
100000	1000	0.50 / 0.25 / 0.25	833,372.60	559,584.30	Coarse	48.93%
100000	1000	0.90 / 0.05 / 0.05	724,486.80	579,955.60	Coarse	24.92%
100000	10000	0.80 / 0.10 / 0.10	54,497.00	51,902.30	Coarse	4.99%
100000	10000	0.50 / 0.25 / 0.25	61,834.00	57,663.00	Coarse	7.23%
100000	10000	0.90 / 0.05 / 0.05	50,681.60	44,192.60	Coarse	14.68%

Threads = 4

InitSize	KeyRange	Ops (c/a/r)	CoarseList (Ops/sec)	FineList (Ops/sec)	Победитель	Разница (%)
10000	1000	0.80 / 0.10 / 0.10	819,280.80	282,712.60	Coarse	189.80%
10000	1000	0.50 / 0.25 / 0.25	815,799.90	276,888.00	Coarse	194.63%
10000	1000	0.90 / 0.05 / 0.05	731,202.20	281,109.00	Coarse	160.12%
10000	10000	0.80 / 0.10 / 0.10	51,900.90	69,419.70	Fine	33.75%
10000	10000	0.50 / 0.25 / 0.25	55,242.40	72,411.00	Fine	31.10%

10000	10000	0.90 / 0.05 / 0.05	54,487.00	68,179.50	Fine	25.13%
100000	1000	0.80 / 0.10 / 0.10	796,244.60	299,049.20	Coarse	166.26%
100000	1000	0.50 / 0.25 / 0.25	826,536.90	282,301.30	Coarse	192.80%
100000	1000	0.90 / 0.05 / 0.05	700,268.90	277,532.10	Coarse	152.32%
100000	10000	0.80 / 0.10 / 0.10	52,271.70	62,621.80	Fine	19.80%
100000	10000	0.50 / 0.25 / 0.25	59,250.20	70,613.50	Fine	19.18%
100000	10000	0.90 / 0.05 / 0.05	47,547.20	46,308.60	Coarse	2.67%

Threads = 8

InitSize	KeyRange	Ops (c/a/r)	CoarseList (Ops/sec)	FineList (Ops/sec)	Победитель	Разница (%)
10000	1000	0.80 / 0.10 / 0.10	401,821.90	135,347.60	Coarse	196.88%
10000	1000	0.50 / 0.25 / 0.25	415,008.20	130,072.20	Coarse	219.07%
10000	1000	0.90 / 0.05 / 0.05	462,138.00	237,258.90	Coarse	94.78%
10000	10000	0.80 / 0.10 / 0.10	46,361.80	98,919.50	Fine	113.36%
10000	10000	0.50 / 0.25 / 0.25	55,241.10	104,953.00	Fine	89.99%
10000	10000	0.90 / 0.05 / 0.05	53,019.50	84,641.90	Fine	59.64%
100000	1000	0.80 / 0.10 / 0.10	796,841.60	270,020.90	Coarse	195.10%
100000	1000	0.50 / 0.25 / 0.25	819,838.30	273,367.90	Coarse	199.89%
100000	1000	0.90 / 0.05 / 0.05	710,847.10	262,755.40	Coarse	170.50%
100000	10000	0.80 / 0.10 / 0.10	41,571.70	96,925.90	Fine	133.15%
100000	10000	0.50 / 0.25 / 0.25	57,838.10	106,085.20	Fine	83.42%
100000	10000	0.90 / 0.05 / 0.05	47,800.60	86,190.30	Fine	80.31%

Выводы

1. Влияние количества потоков:

CoarseList: Производительность резко падает с увеличением числа потоков. Это связано с тем, что одна блокировка на весь список создает состязание между потоками. При 1 потоке она показывает лучшую производительность, но плохо масштабируется.

FineList: Производительность либо остается стабильной, либо улучшается с увеличением числа потоков. Тонкая блокировка позволяет потокам работать параллельно над разными частями списка, снижая состязание, лучше масштабируется.

2. Влияние KeyRange:

Увеличение KeyRange с 1000 до 10000 значительно снижает производительность обоих типов списков. Операции поиска/вставки/удаления занимают больше времени в большем диапазоне.

Падение производительности при увеличении KeyRange гораздо более выражено для CoarseList, особенно при нескольких потоках.

3. Влияние InitSize:

Увеличение InitSize с 10000 до 100000 также снижает общую производительность, но влияние менее драматично, чем у KeyRange.

4. Влияние Соотношения Операций (Ops c/a/r):

Разное соотношение операций чтения (contains), добавления (add) и удаления (remove) влияет на абсолютные значения производительности. Сценарии с большим количеством модификаций (add/remove) могут сильнее выявлять преимущества FineList при многопоточности.

5. Точка Пересечения:

При 1 потоке CoarseList почти всегда быстрее FineList.

При увеличении числа потоков и большом KeyRange (10000), FineList начинает превосходить CoarseList (начиная с 2-4 потоков).

При малом KeyRange (1000), CoarseList остается быстрее даже при 8 потоках, хотя разрыв сокращается по сравнению с 1 потоком.

Итог:

CoarseList эффективна в сценариях с низкой конкуренцией (мало потоков) и когда операции выполняются быстро.

FineList гораздо лучше масштабируется с увеличением числа потоков и предпочтительнее в средах с высокой конкуренцией или когда операции могут занимать больше времени (например, при большом KeyRange).