Высокопроизводительные приложения в средах с JIT и GC

Алексей Сорокин
undev.ru

JIT

- JIT (just-in-time compilation) - компиляция «на лету» или динамическая компиляция
- Что-то среднее между интерпретацией и полной статической (ahead-of-time) компиляцией
- Компилирует код во время выполнения программы.

JIT

Варианты:

1. JIT-компиляция каждого метода перед первым его исполнением (CLR)
2. Начинаем исполнять программу как интерпретатор, обнаруживаем “горячие” участки и только их JIT-компилируем (Сановская JVM, поэтому и название HotSpot)

JIT

- Весь код пишем на Java или C# не прибегая к C.
- В результате процессор исполняет Вашу программу, скомпилированную в машинный код, а не управляющий цикл интерпретатора.

No JIT

Python, Ruby...

- Значительно бОльшая часть ядра написана на C
- Прикладные библиотеки требующие высокой производительнсти (парсинг и генерация XML и JSON, математические вычисления, работа с бинарными данными и т.п.) также написаны на C
- придется как минимум это читать, а может и отлаживать

Описание примера

- Из области биржевой торговли
- Система сбора оперативных данных о торгах на бирже (Market Data)

Описание примера

- На входе поток сообщений о добавлении и исполнении ордеров. Ордер это заявка на покупку или продажу ценной бумаги.
- На выходе сообщения о лучшей цене на продажу и на покупку (Top of Book)

Описание примера

Почему выбран этот пример

- Для получения выходных данных в памяти надо хранить состояние по каждой ценной бумаге
- Миллионы ордеров и сотни тысяч ценных бумаг
- Много поисков и вставок в структуры данных
- Вывод: **особое внимание к организации доступа к данным**

Тест

Замеряется кол-во обработанных сообщений в секунду

- При обработке каждого входящего сообщения увеличивется counter
- Раз в секунду (почти) срабатывает timer, выводит и сбрасывает counter
- Для компенсации пауз GC замеряется кол-во прошедших реальных секунд

Результаты

501833 per second (time lag: 0)
204548 per second (time lag: 2)
389246 per second (time lag: 0)
619864 per second (time lag: 0)
277394 per second (time lag: 0)
65473 per second (time lag: 2)
593106 per second (time lag: 0)
185395 per second (time lag: 0)
158793 per second (time lag: 1)
554909 per second (time lag: 0)
101293 per second (time lag: 0)
560602 per second (time lag: 0)
544216 per second (time lag: 0)

493897 per second (time lag: 0)
492882 per second (time lag: 0)
230510 per second (time lag: 1)
490209 per second (time lag: 0)
391658 per second (time lag: 0)
32412 per second (time lag: 6)
497220 per second (time lag: 0)
125709 per second (time lag: 1)
488258 per second (time lag: 0)
479322 per second (time lag: 0)
476025 per second (time lag: 0)
25008 per second (time lag: 7)
470262 per second (time lag: 0)

Организация кода

На входе сообщения вида:

A0000000888123ABCS000034.5000000100

A0000000999456QWEB000974.2500007894

E0000000888

E0000000999

Организация кода

MessageReader парсит в объект:

```
public class Order {
  public String symbol;
  public int id;
  public Side side;
  public int price;
  public int qty;
}
```

И передает в TopOfBook

```
tob.addOrder(order);
```

Организация кода

TopOfBook содержит основные структуры в памяти:

```
tops = new HashMap<String, Top>();

orders = new HashMap<Integer, Order>();

symbol2sell = new HashMap<String,
NavigableMap<Integer, Integer>>();

symbol2buy = new HashMap<String,
NavigableMap<Integer, Integer>>();
```

Память и кэши

Random Access Memory

- Запоминающее устройство с произвольным доступом
- На столько ли **произвольным**?
- Какова **цена** произвольного доступа?

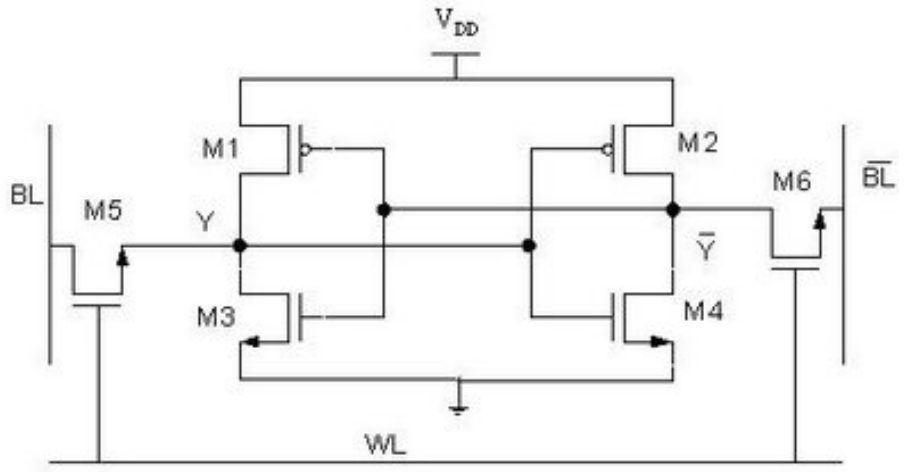
Память и кэши

"memory is the new disk"

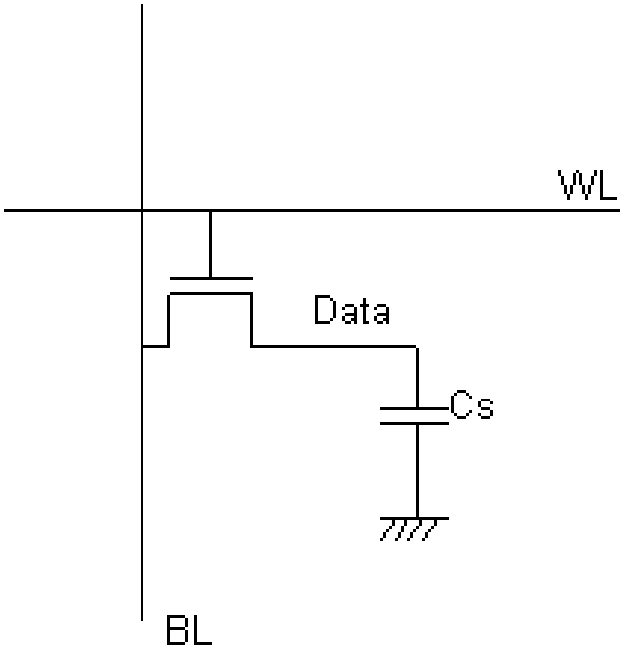
- Скорость работы процессора стала значительно быстрее пропускной способности памяти
- Доступ к памяти – очень дорогая операция в сравнении с вычислительными операциями.

Два типа памяти

Static RAM (SRAM)



Dynamic RAM (DRAM)



Память и кэши

- Доступ к памяти всегда через кэш
- Чтение и запись идет блоками, называемыми cache line. Размер 32 или скорее 64 байта в современных процессорах

Память и кэши

Факторы влияющие на эффективность кэша

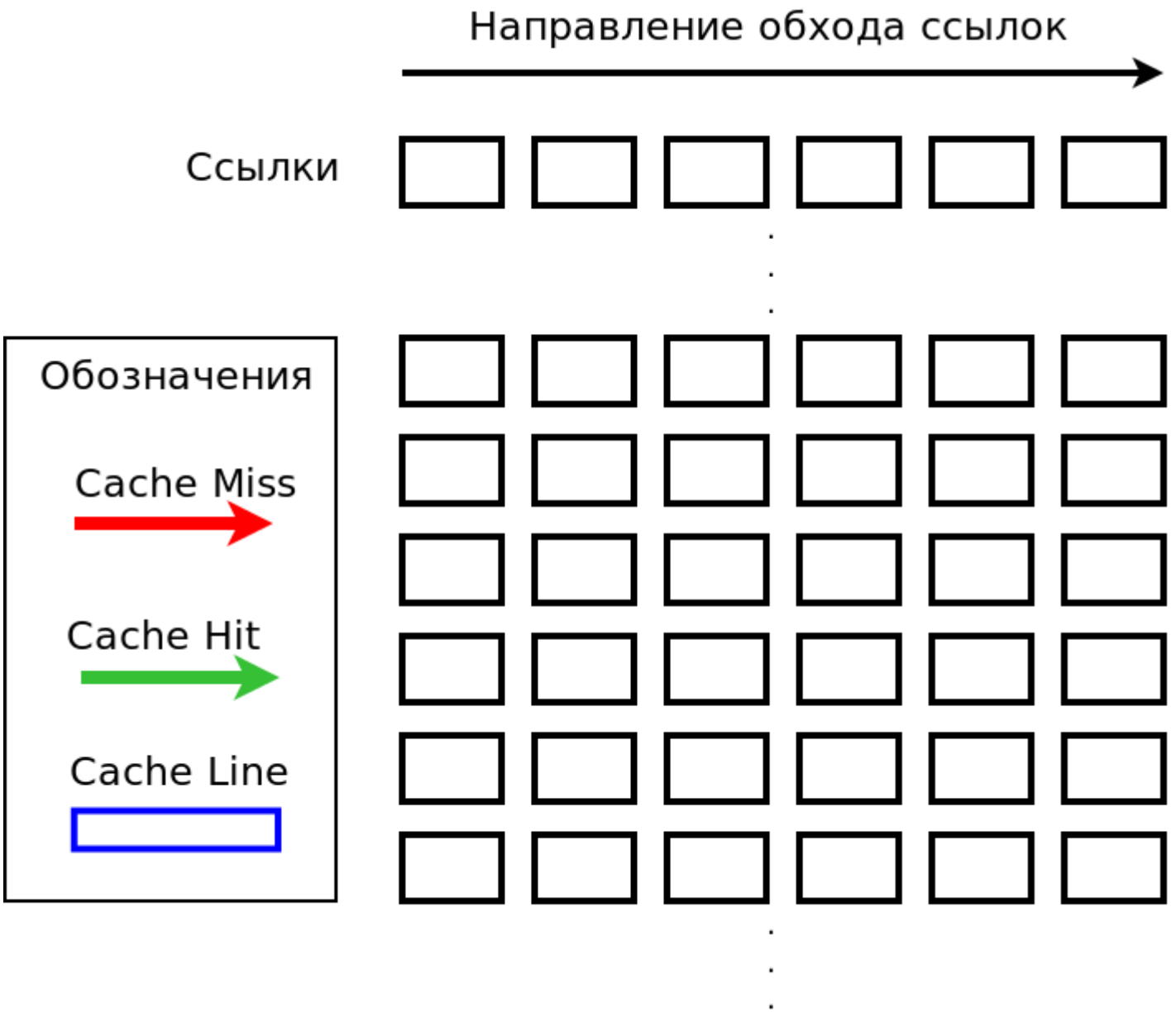
- Размер данных
- Лоакльность данных
- Вид доступа к памяти (случайный или последовательный)

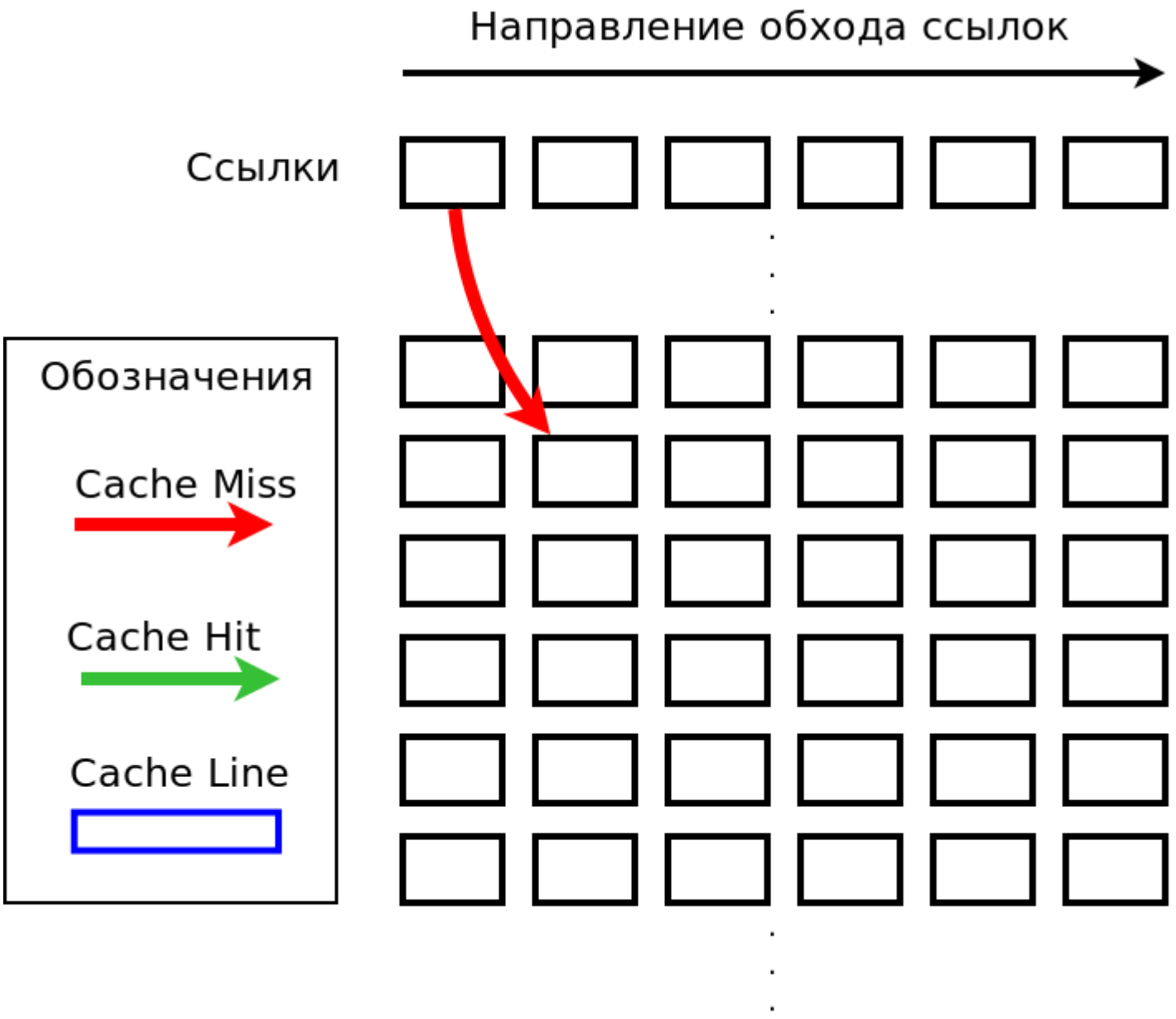
Локальность

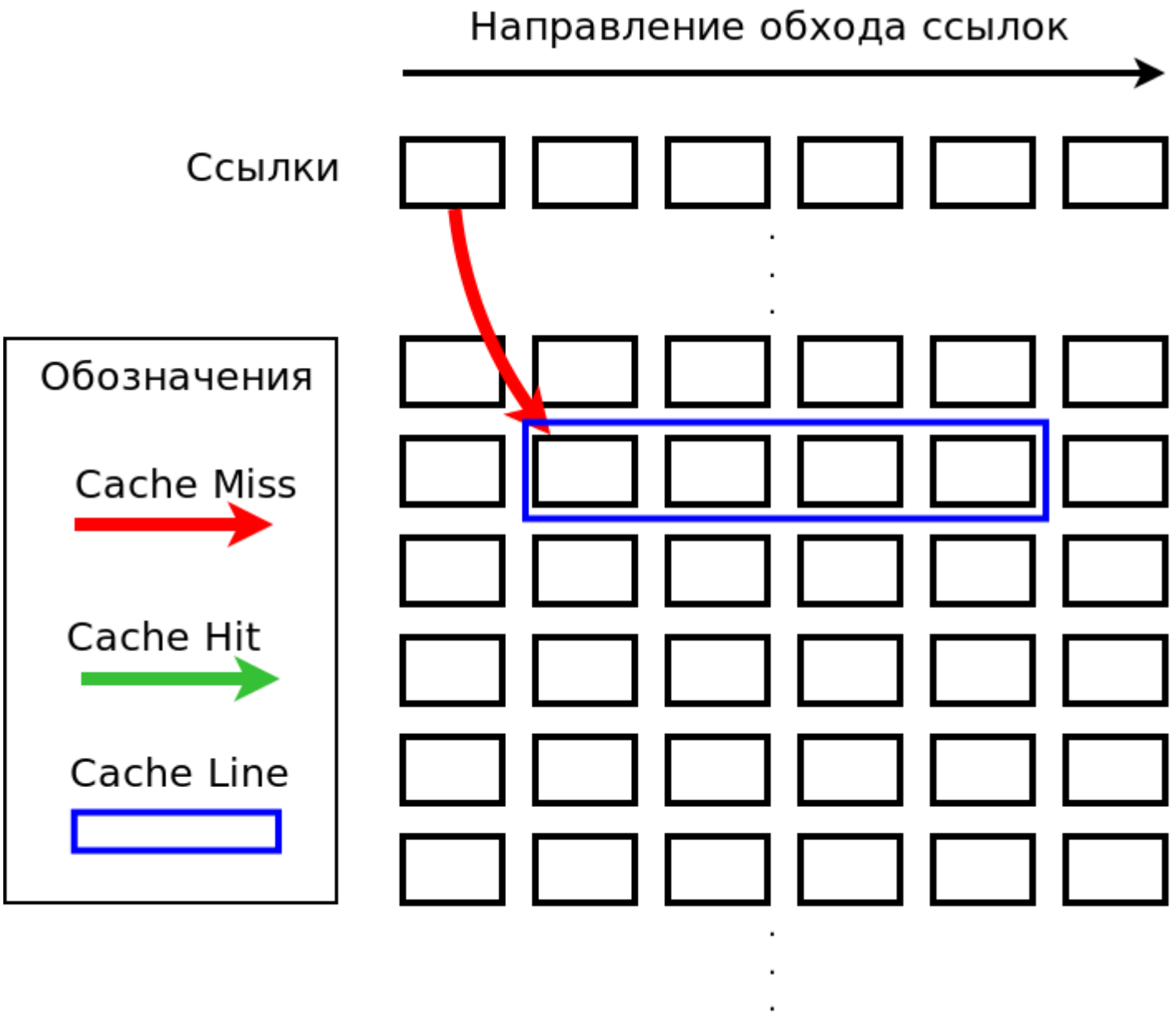
Виды локальности (locality of reference)

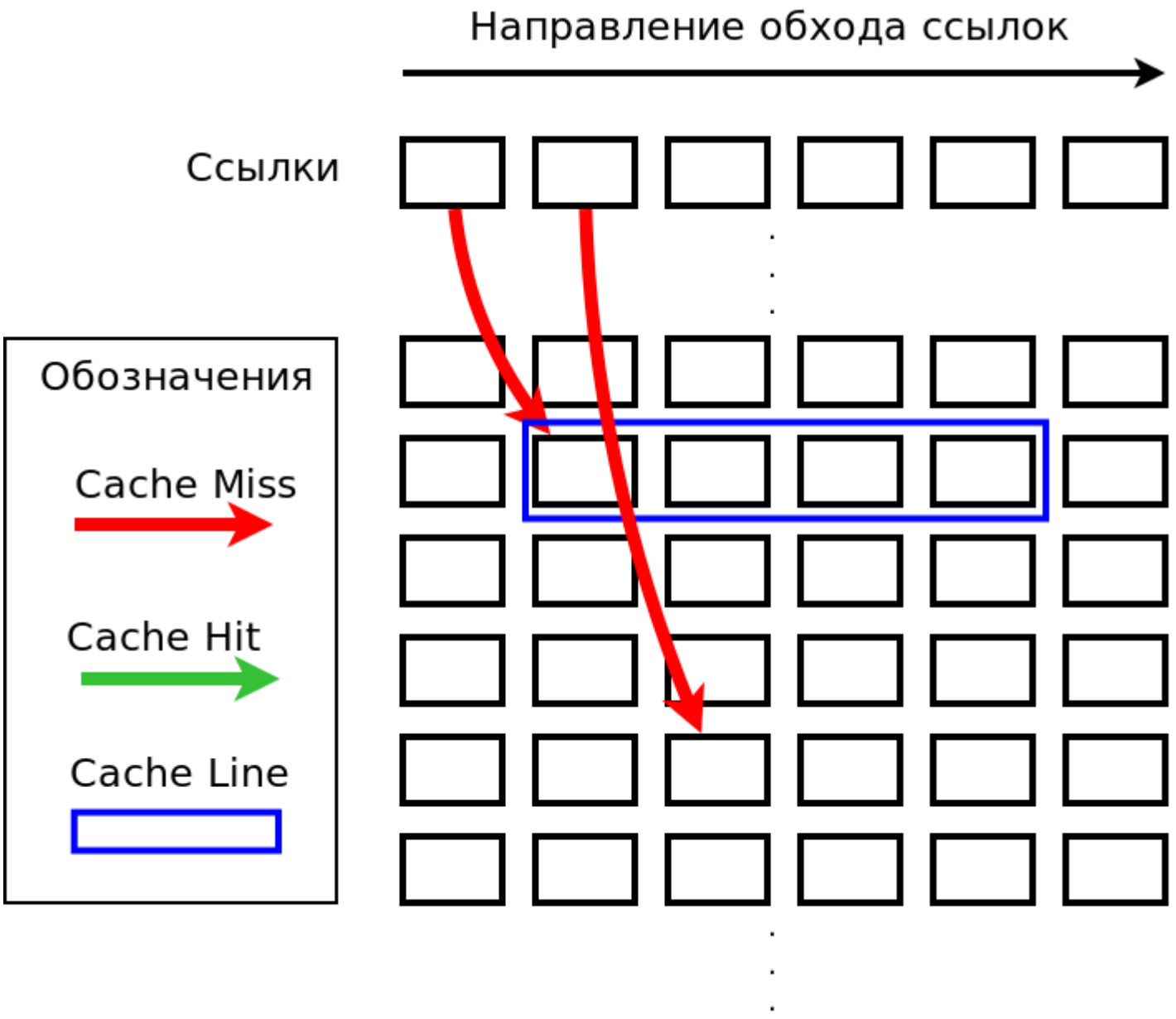
- ВременнАя (Temporal locality)
- Пространственная (Spatial locality)
- Равноудаленная (Equidistant locality)

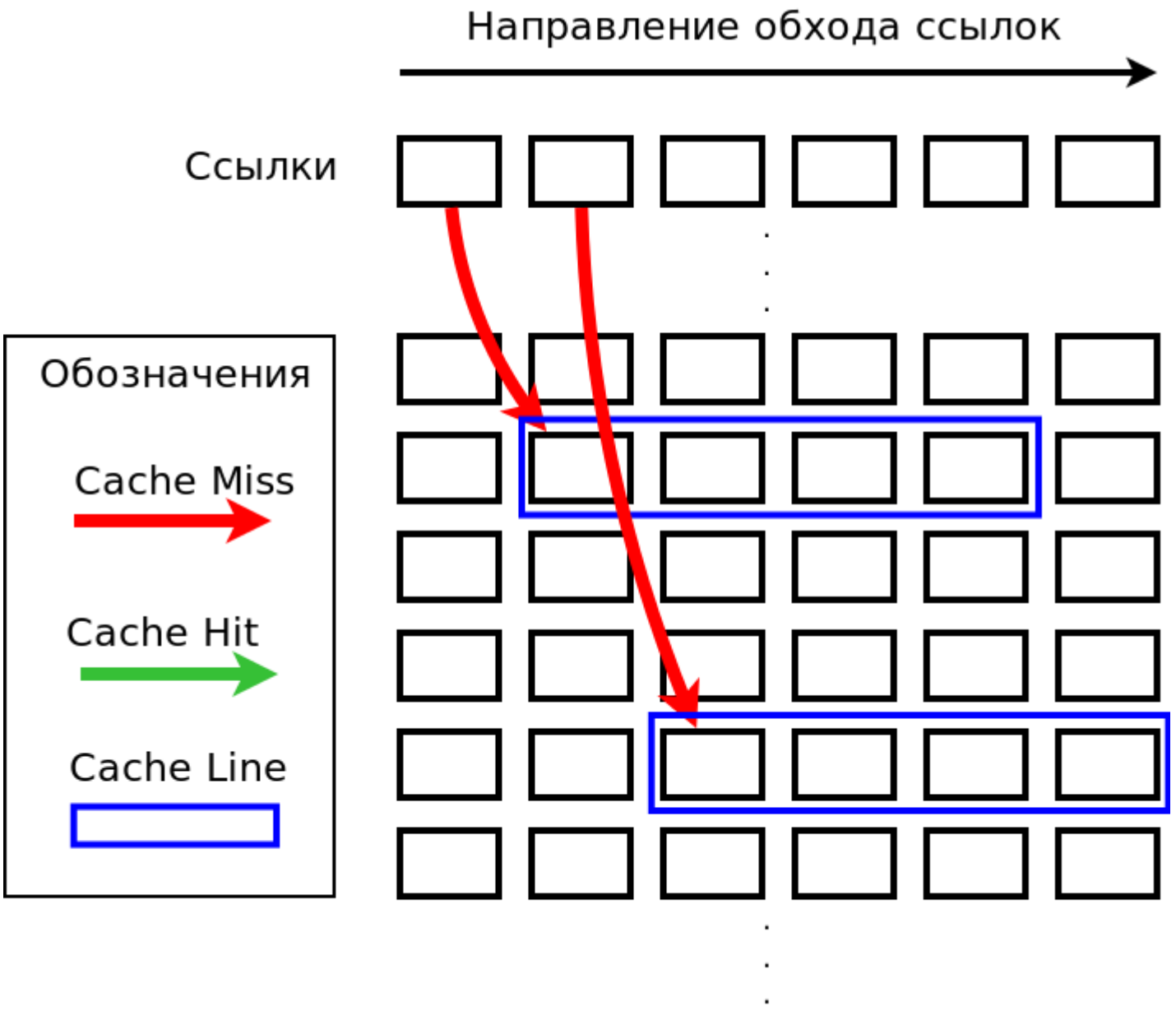
Произвольный доступ

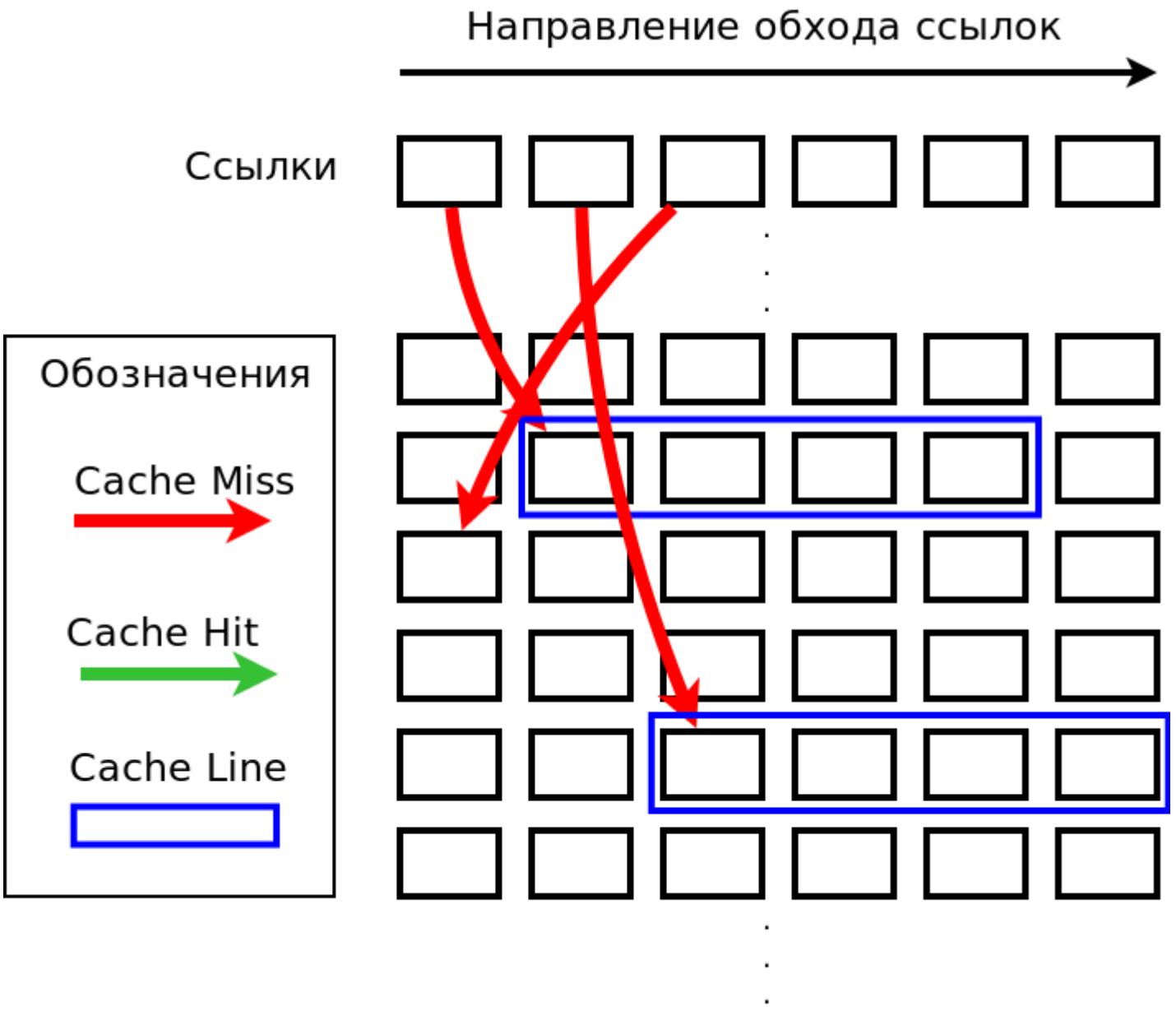

Направление обхода ссылок
Ссылки
Обозначения
Cache Miss
Cache Hit
Cache Line

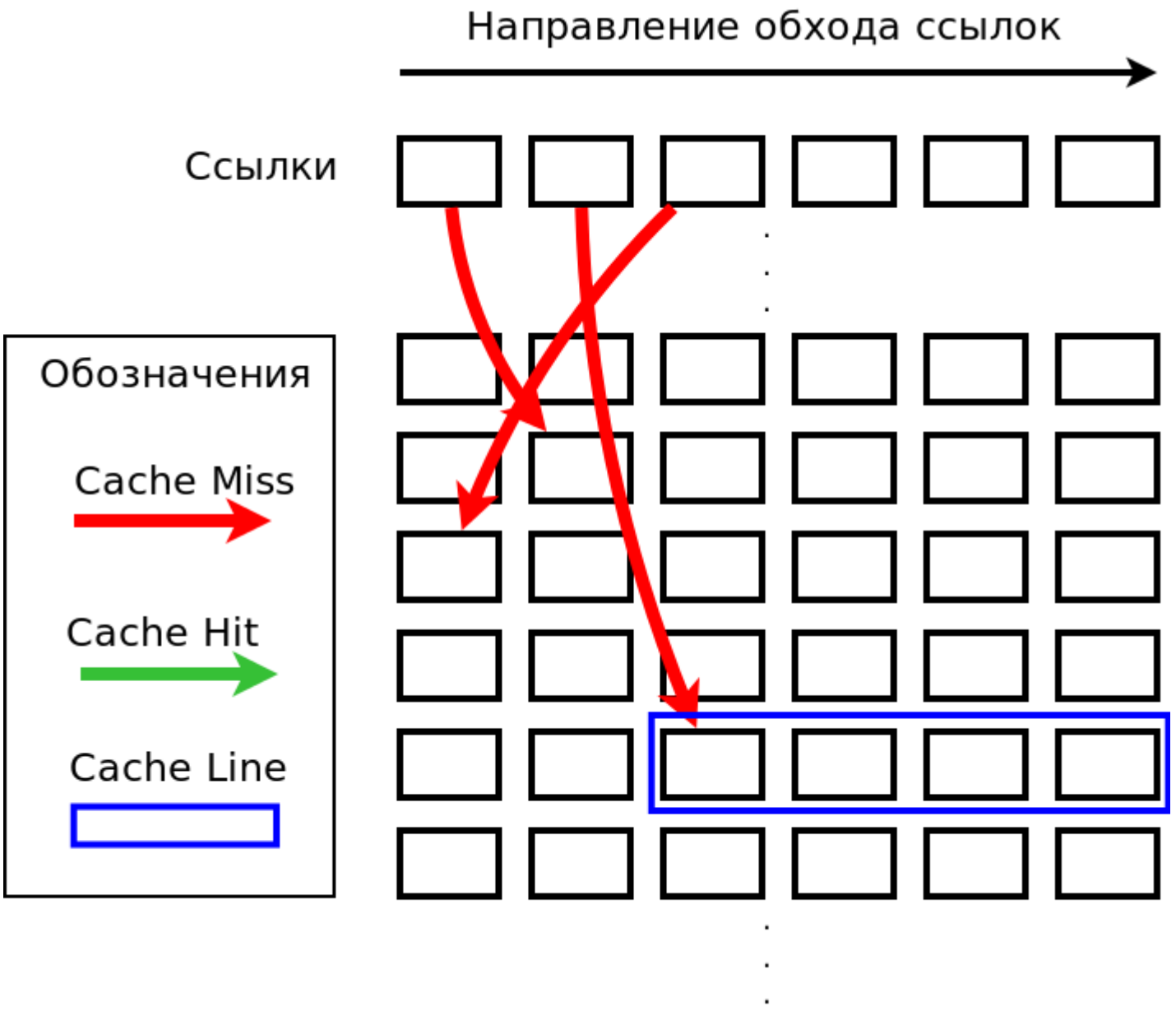

Направление обхода ссылок
Ссылки
Обозначения
Cache Miss
Cache Hit
Cache Line

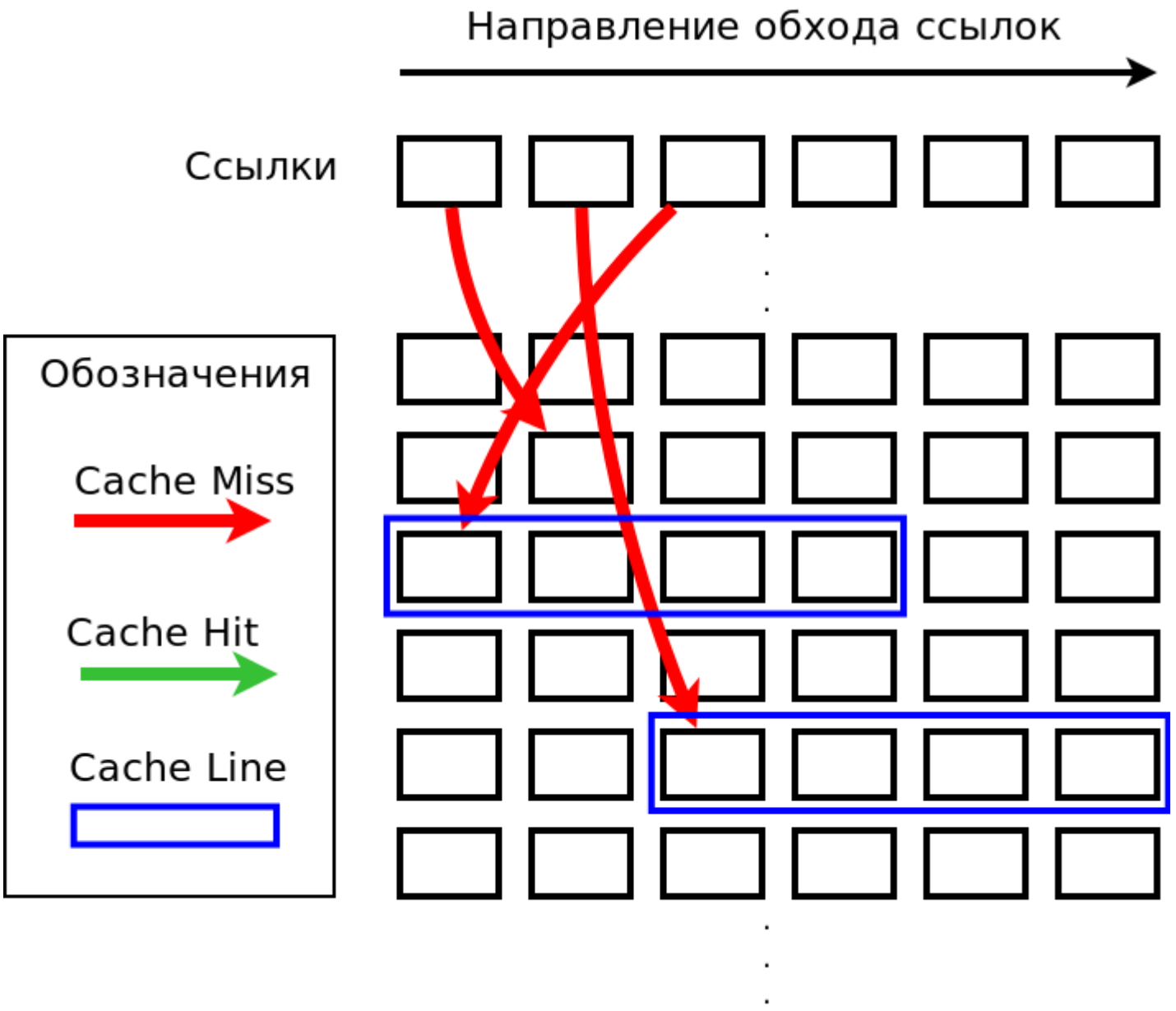

Направление обхода ссылок
Ссылки
Обозначения
Cache Miss
Cache Hit
Cache Line


Направление обхода ссылок
Ссылки
Обозначения
Cache Miss
Cache Hit
Cache Line

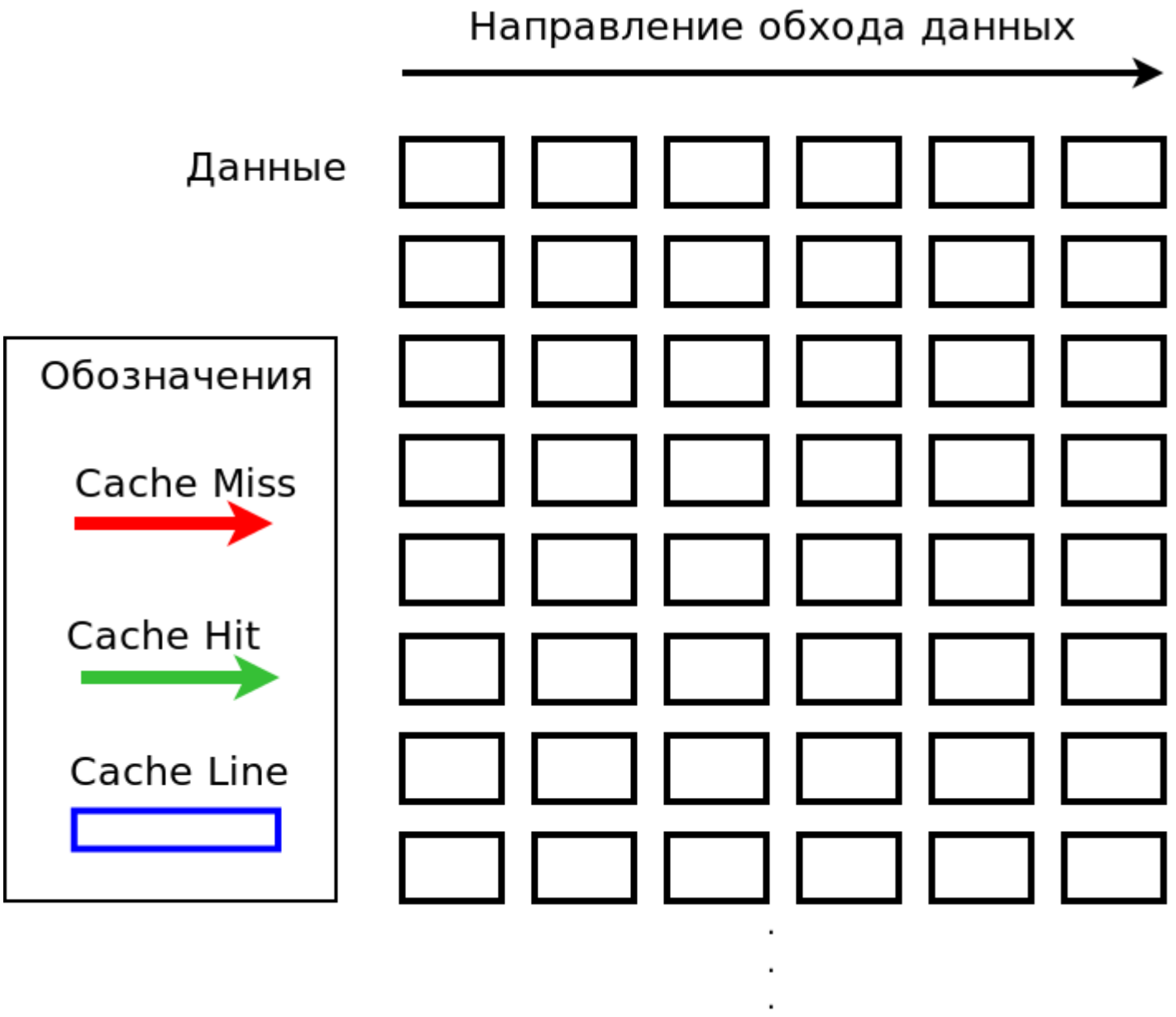

Направление обхода ссылок
Ссылки
Обозначения
Cache Miss
Cache Hit
Cache Line

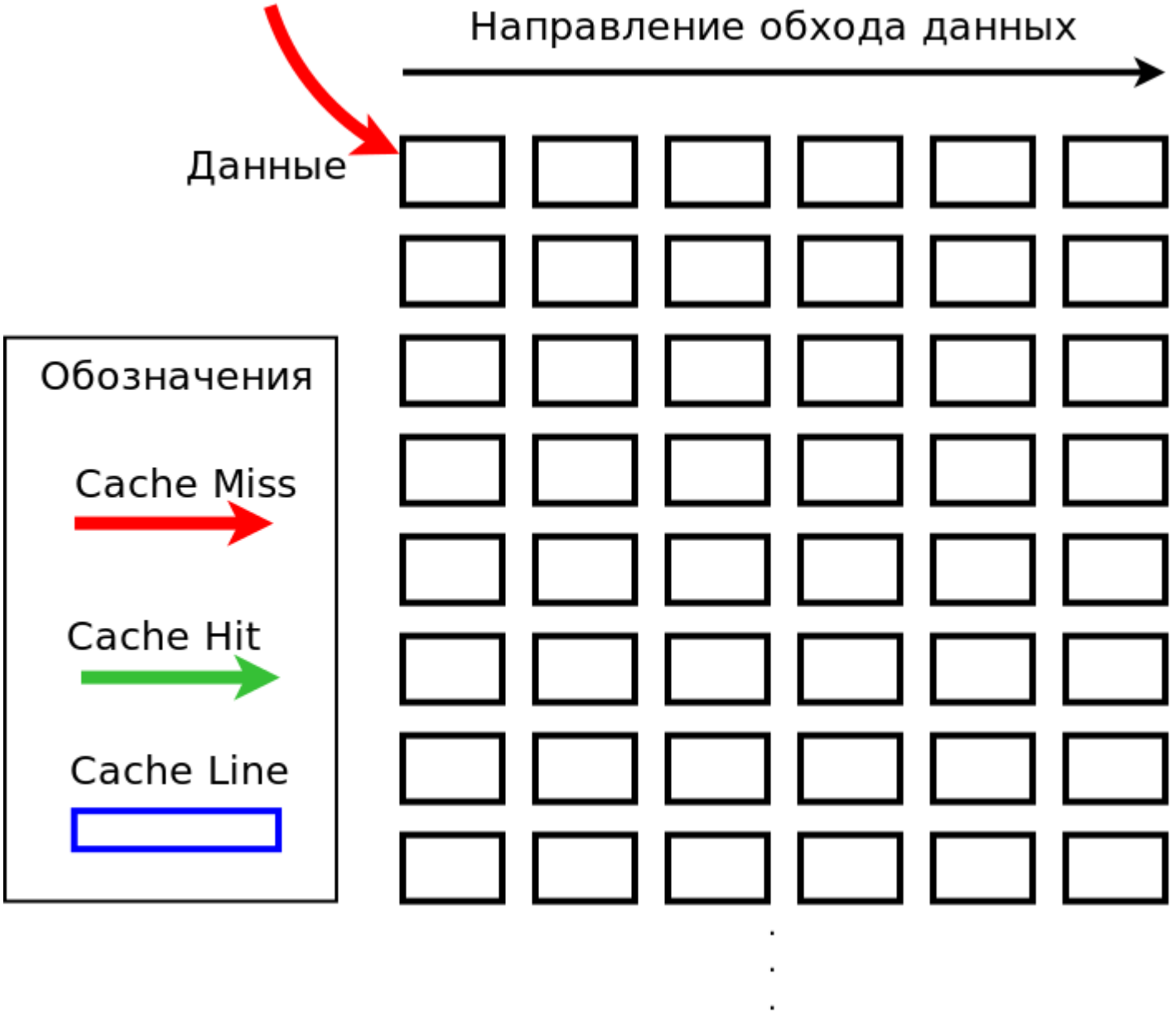

Направление обхода ссылок
Ссылки
Обозначения
Cache Miss
Cache Hit
Cache Line

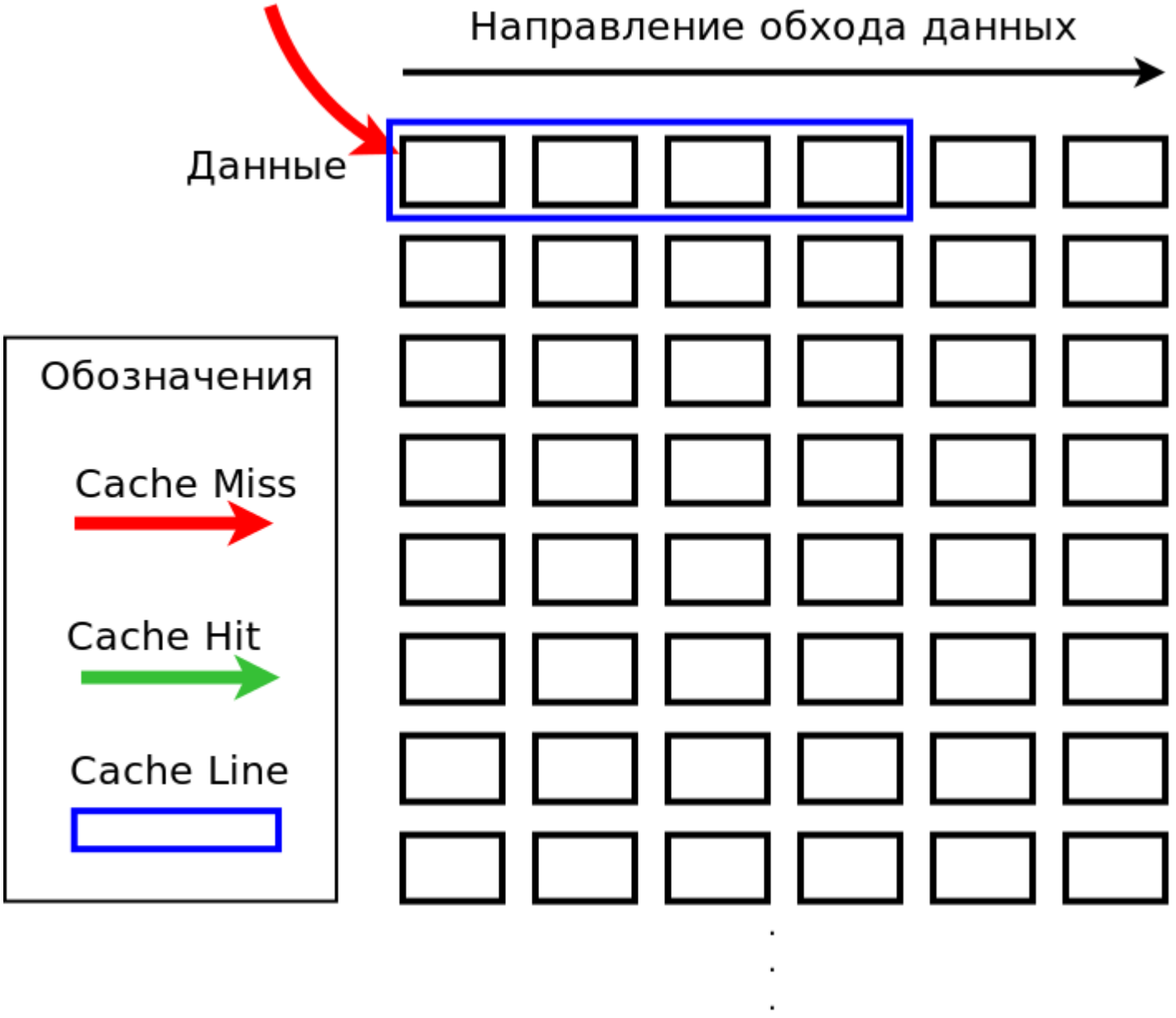

Направление обхода ссылок
Ссылки
Обозначения
Cache Miss
Cache Hit
Cache Line

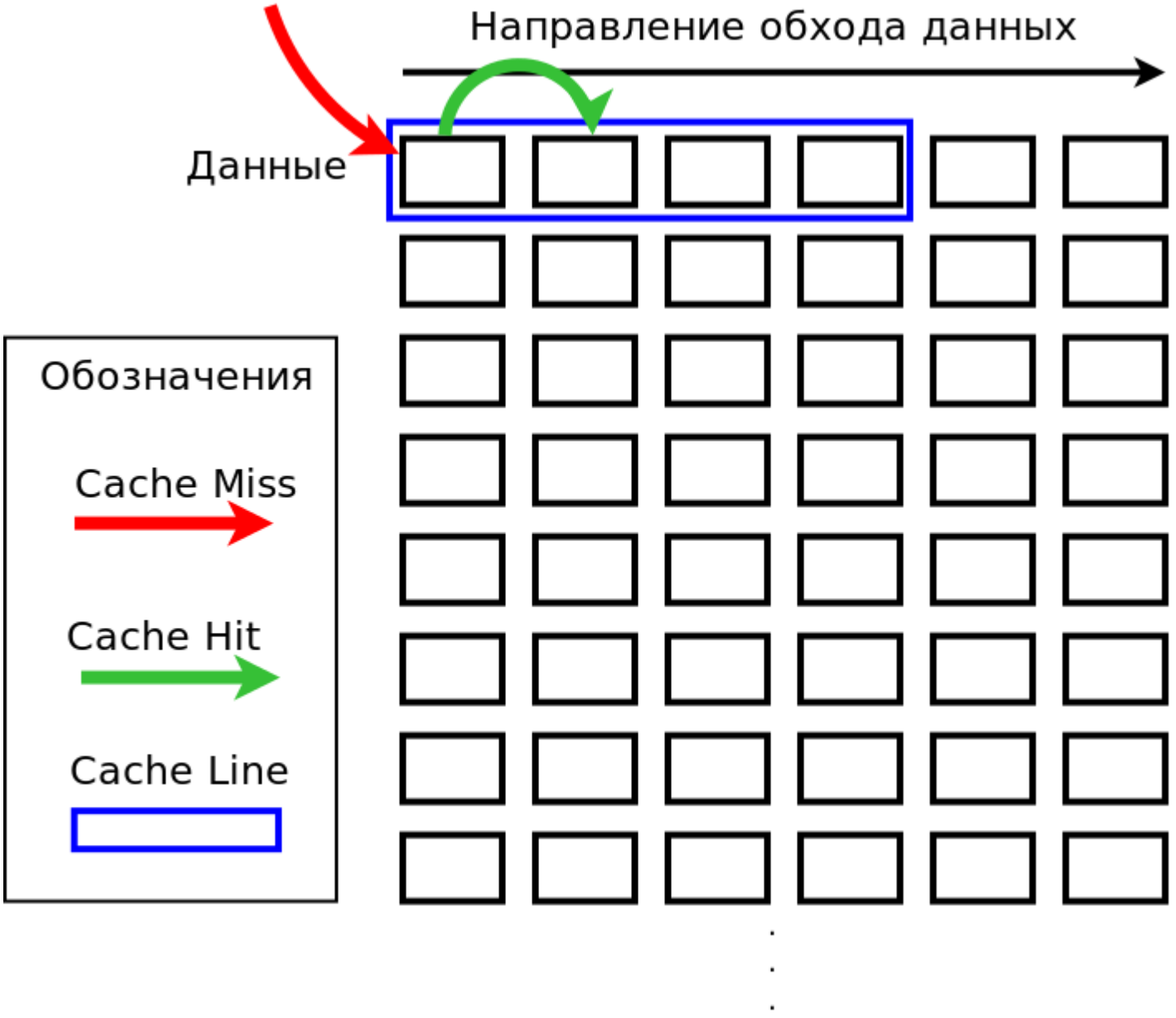

Направление обхода ссылок
Ссылки
Обозначения
Cache Miss
Cache Hit
Cache Line

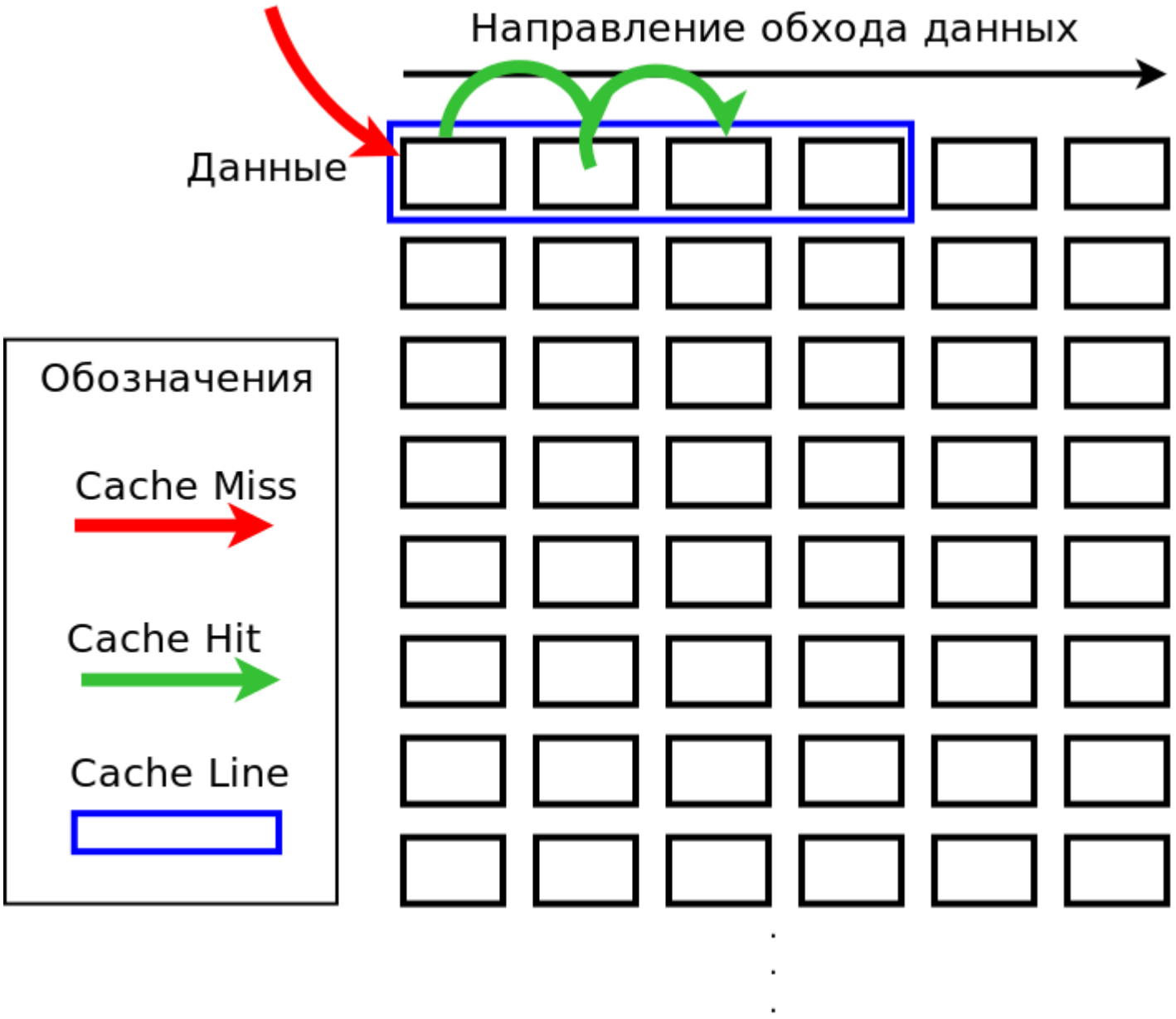
Последовательный доступ

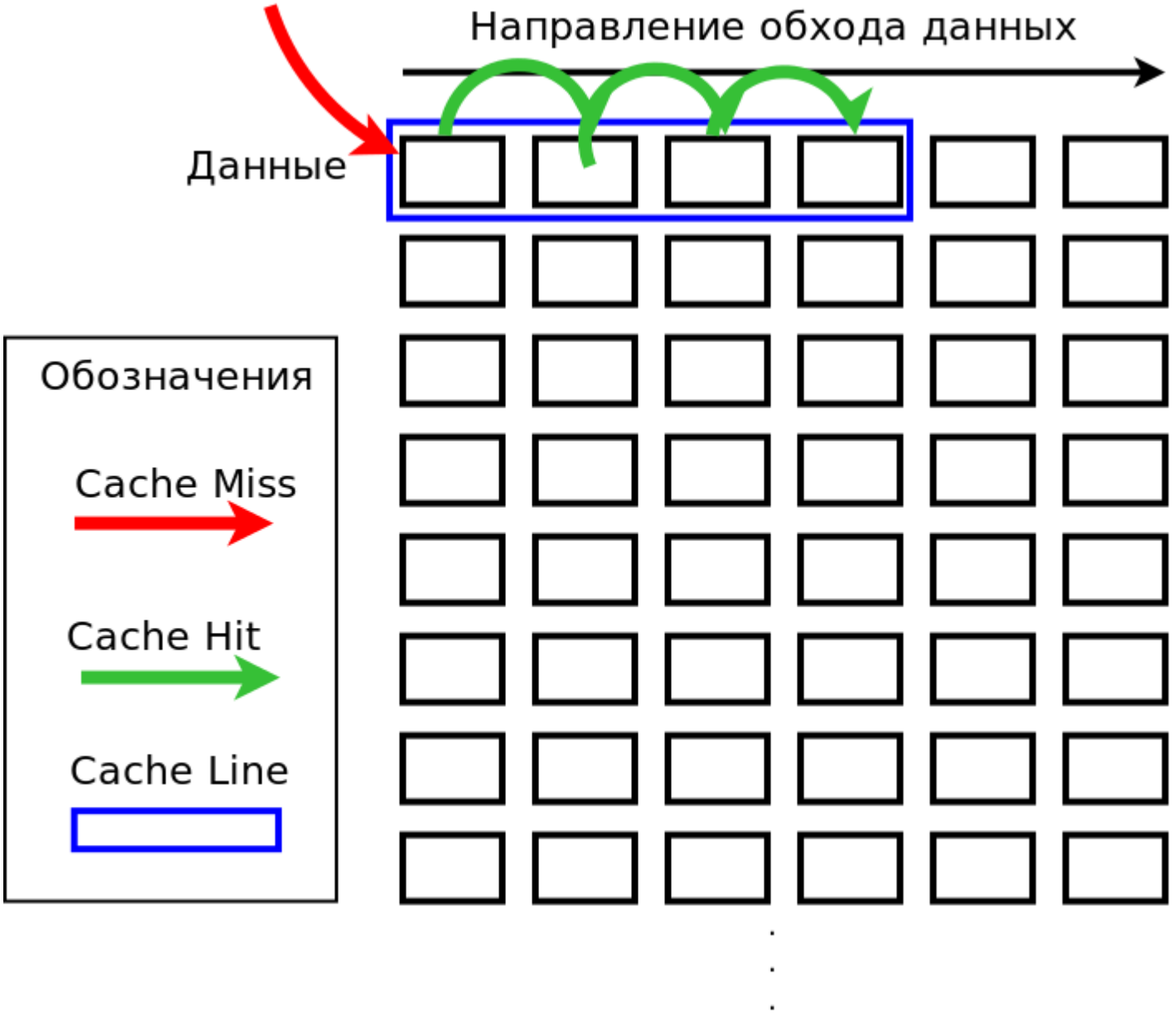

Направление обхода данных
Данные
Обозначения
Cache Miss
Cache Hit
Cache Line

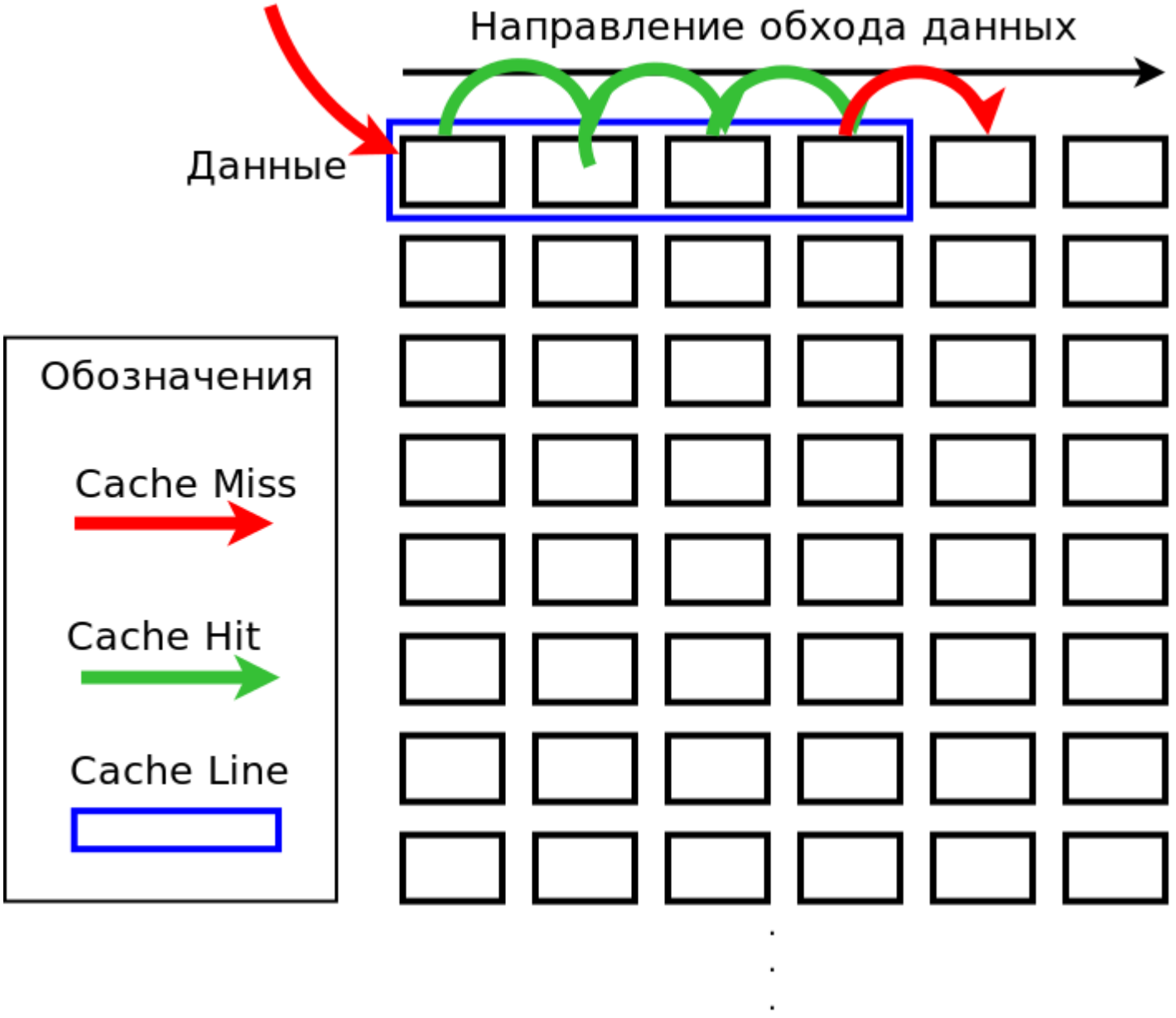

Направление обхода данных
Данные
Обозначения
Cache Miss
Cache Hit
Cache Line

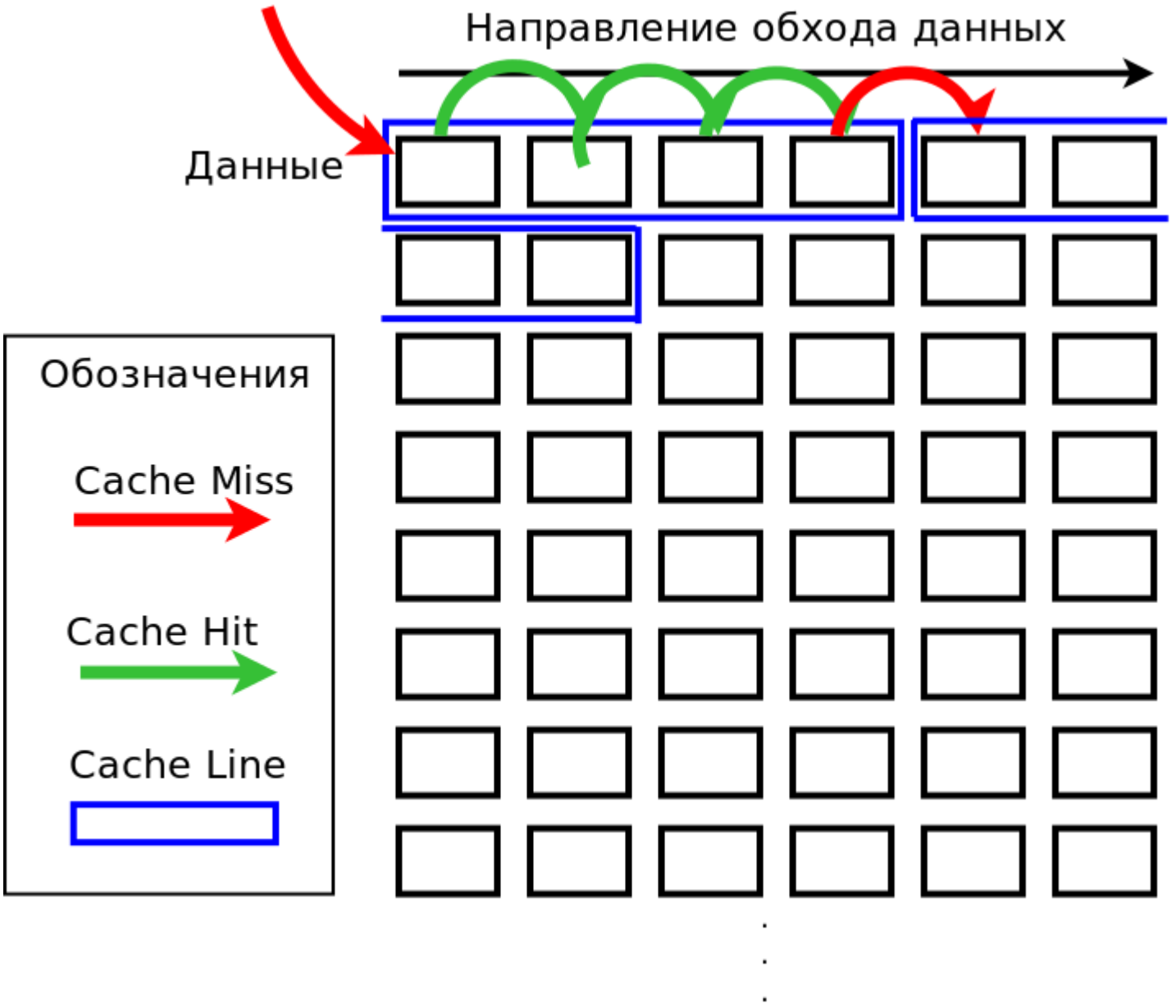

Направление обхода данных
Данные
Обозначения
Cache Miss
Cache Hit
Cache Line

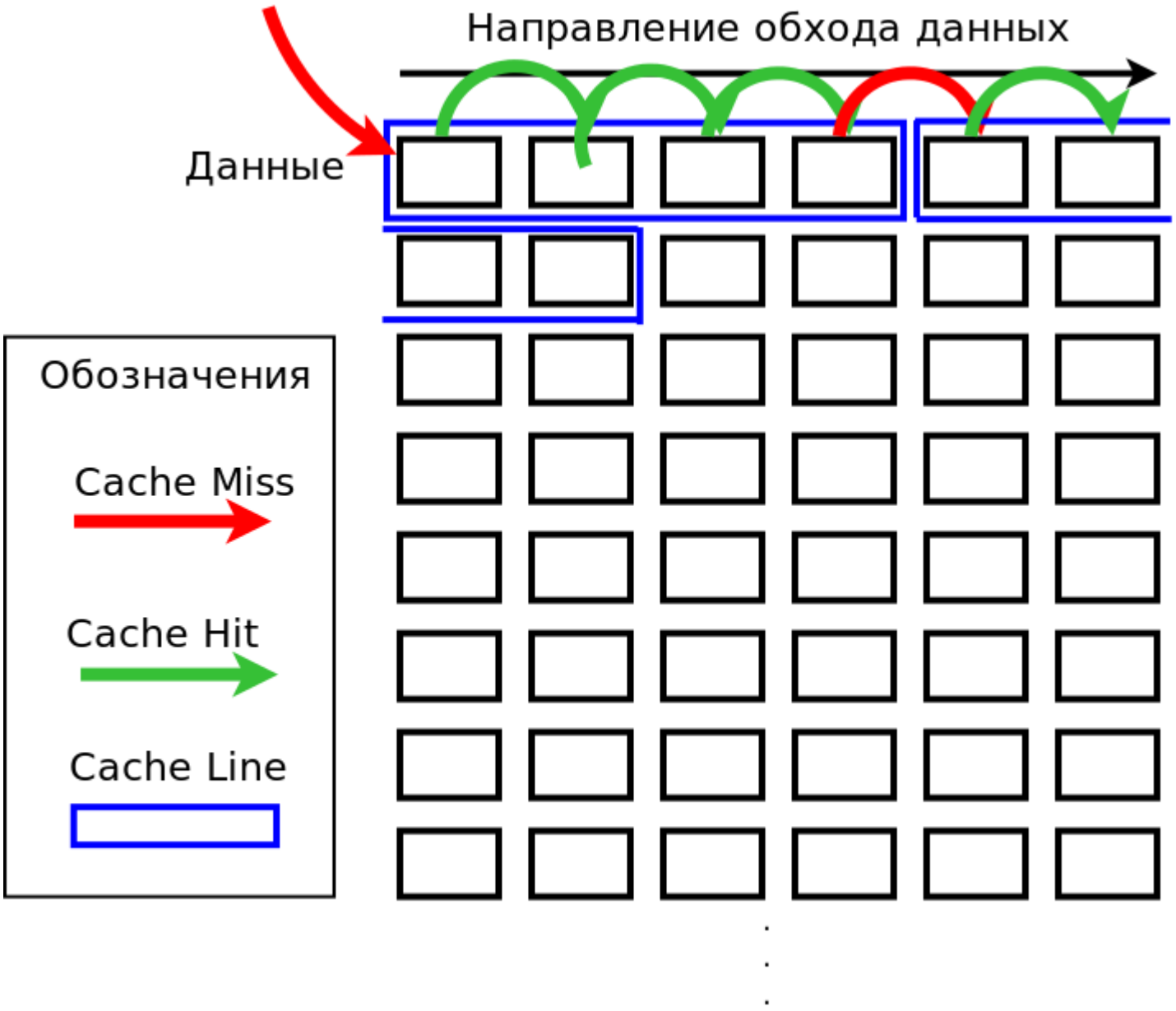

Направление обхода данных
Данные
Обозначения
Cache Miss
Cache Hit
Cache Line


Направление обхода данных
Данные
Обозначения
Cache Miss
Cache Hit
Cache Line


Направление обхода данных
Данные
Обозначения
Cache Miss
Cache Hit
Cache Line


Направление обхода данных
Данные
Обозначения
Cache Miss
Cache Hit
Cache Line


Направление обхода данных
Данные
Обозначения
Cache Miss
Cache Hit
Cache Line


Направление обхода данных
Данные
Обозначения
Cache Miss
Cache Hit
Cache Line

Prefetching

- В Intel Core реализован механизм Instruction Pointer-Based (IP) Prefetcher to Level 1 Data Cache (software.intel.com/file/18374/)
- Если данные проявляют свойство равноудаленной локальности, то у процессора есть возможность делать предварительную подкачку из памяти

Структуры данных в JVM (Java)

Коллекции в JDK

- Хранят **ссылки** на объекты, а не сами данные
- Не позволяют хранить примитивы без **боксинга**
- **Низкая** локальность данных
- Заметный **оверхэд** по используемой памяти

Fastutils (http://fastutil.dsi.unimi.it/)

- Набор коллекций для хранения **примитивов**
- Хранят **данные**, а не ссылки
- **Повышенная** локальность данных
- Эффективное использование памяти

+ мои доработки для хранения произвольных кусков данных фиксированной длины (data-collections)

Пытаемся решить отсутствие в Java структур (struct) аналогичных Си

Храним данные, а не ссылки

```
symbol2top =
        new FixedByteSlice2IntOpenHashMap(SYMBOL_LENGTH);
```

put(byte[] key, int val)

put(ByteBuffer src, int keyOffset, int val)

put(byte[] keySrc, int keyOffset, int val)

Внутри:

byte[] key

int[] value

Храним данные, а не ссылки

prices = new Int2IntLinkedOpenHashMap();

put(int k, int v)

int get(int k)

Структуры данных в CLR (C#)

- Примитывы в C# не имеют **дуализма** как в Java
- Лучше реализованы **Generics**, что позволяет хранить сами данные в коллекциях, в том числе примитивы без боксинга
- Ключевое слово struct!

GC

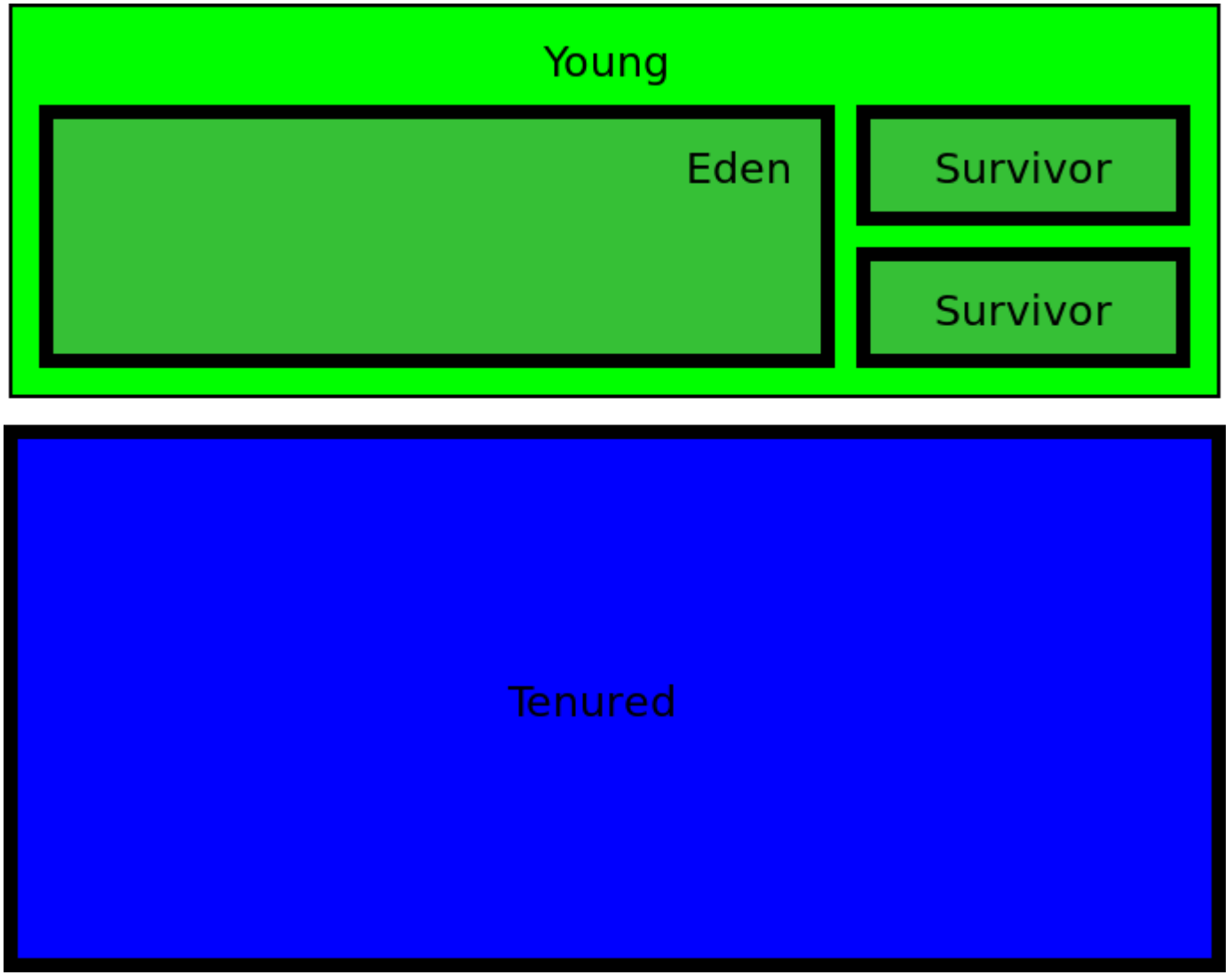
GC

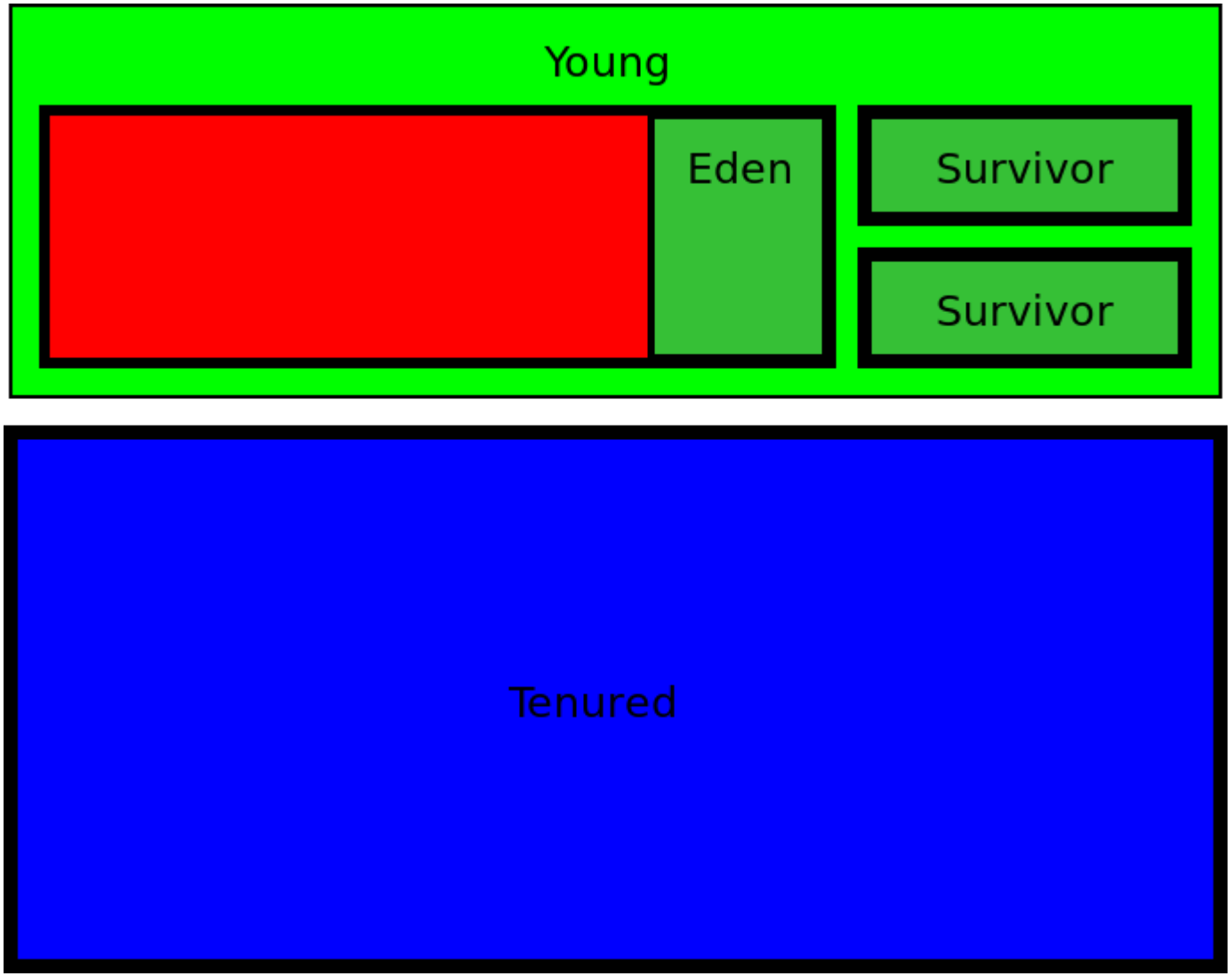
- GC (Garbage Collector) – Сборщик Мусора
- Целая подсистема управления памятью. Обычно отвечает так же и за аллокацию памяти и общую структуру “кучи”

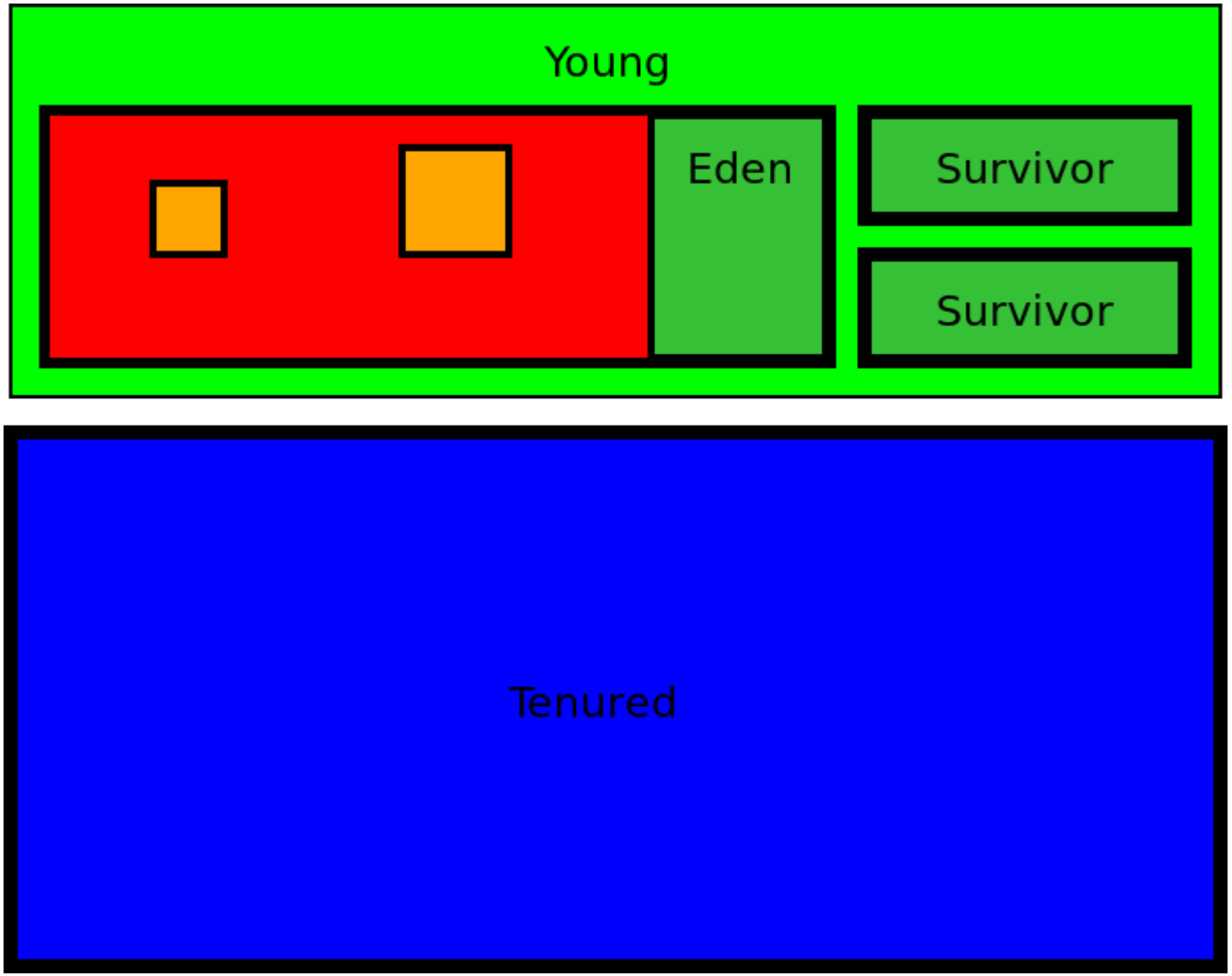
GC

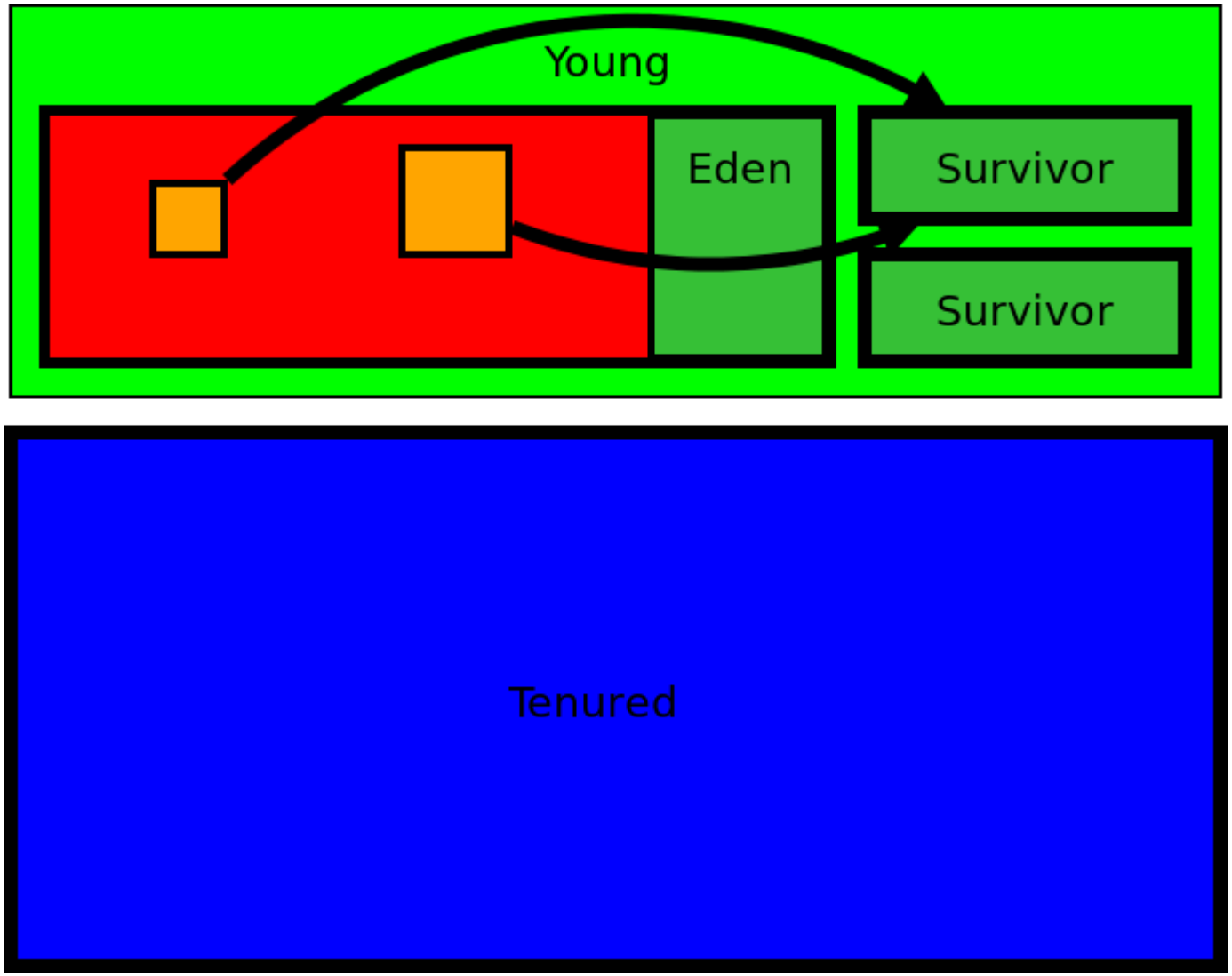
Поколения (на примере JVM)

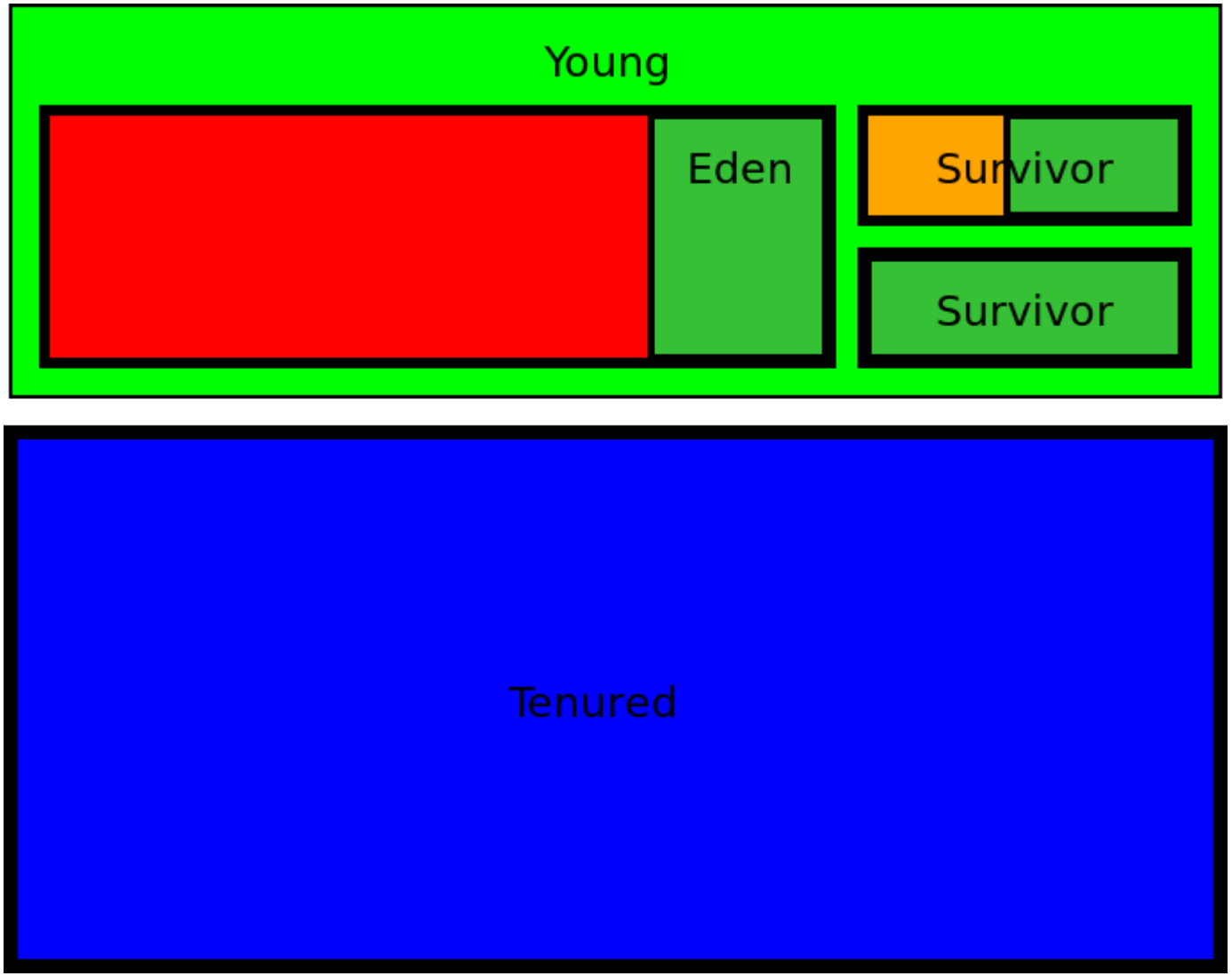
- Молодое поколение – GC предполагает что из этого поколения будет мало “выживших”
- Старое поколение – обратная картина, GC предполагает что большинство объектов в этом поколении будут жить долго
- Если в молодом поколении не достаточно места – объекты будут сразу отправлены в старое поколение! Используйте -XX:NewSize или -XX:NewRatio

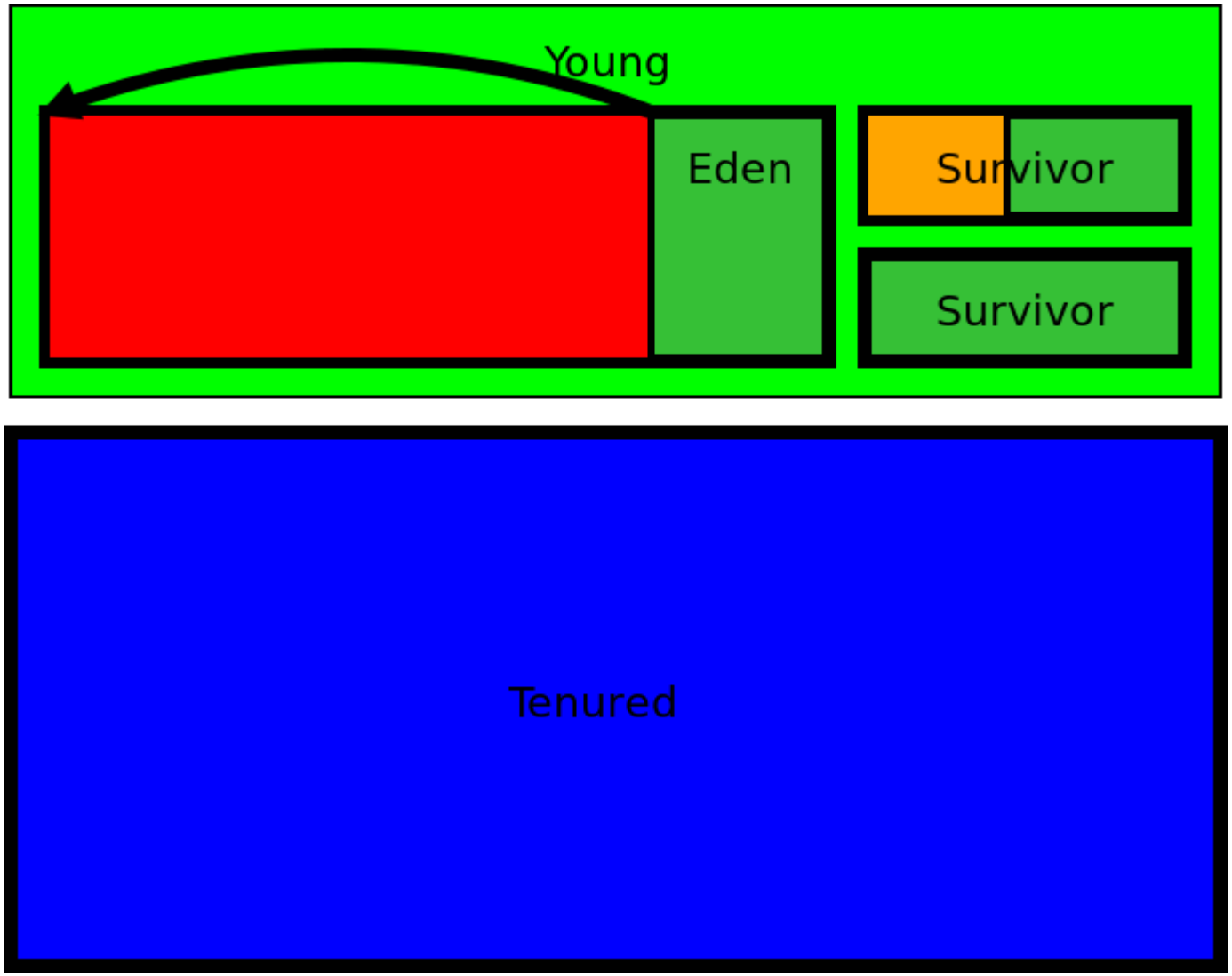

Young
Eden
Survivor
Survivor
Tenured

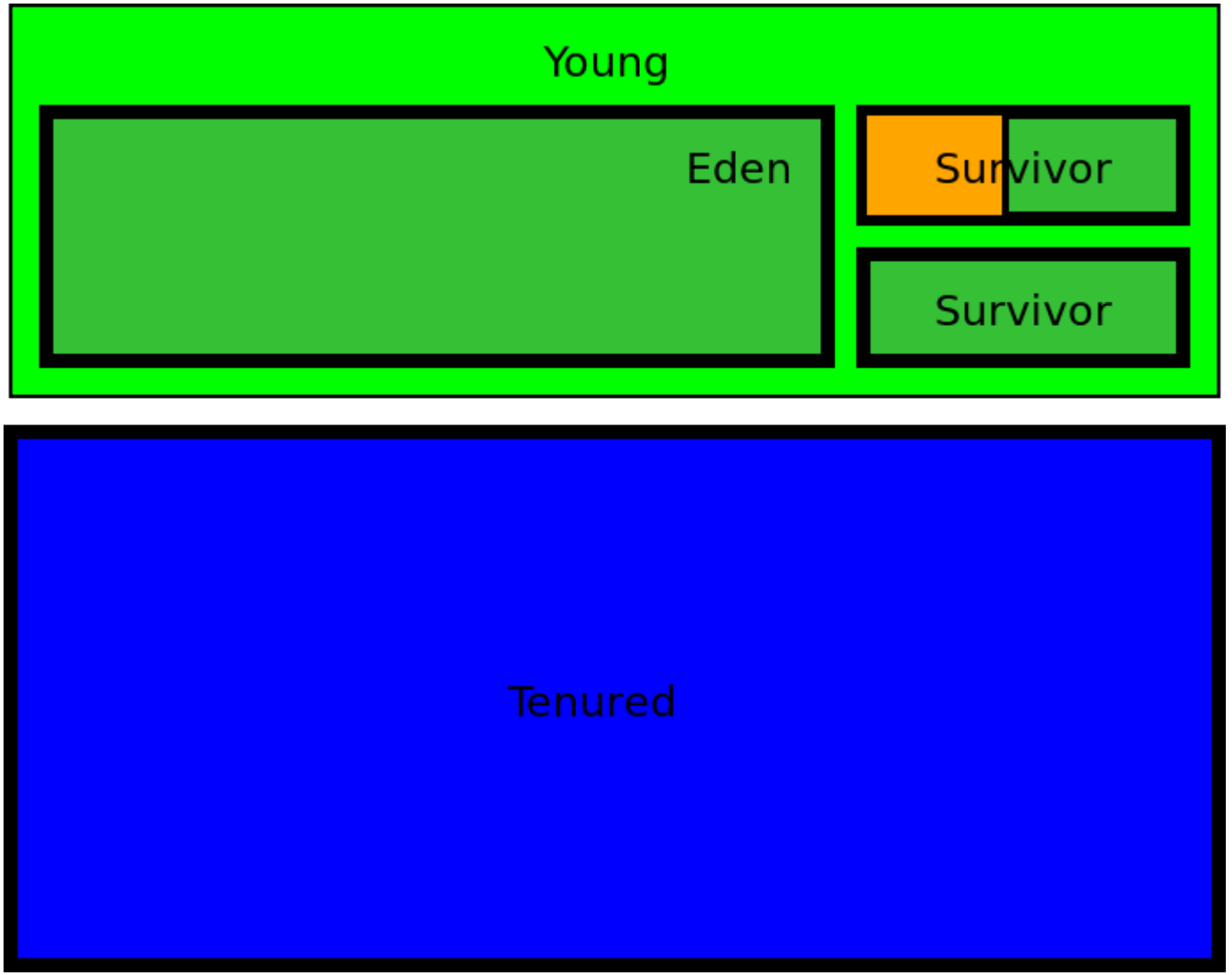

Young
Eden
Survivor
Survivor
Tenured

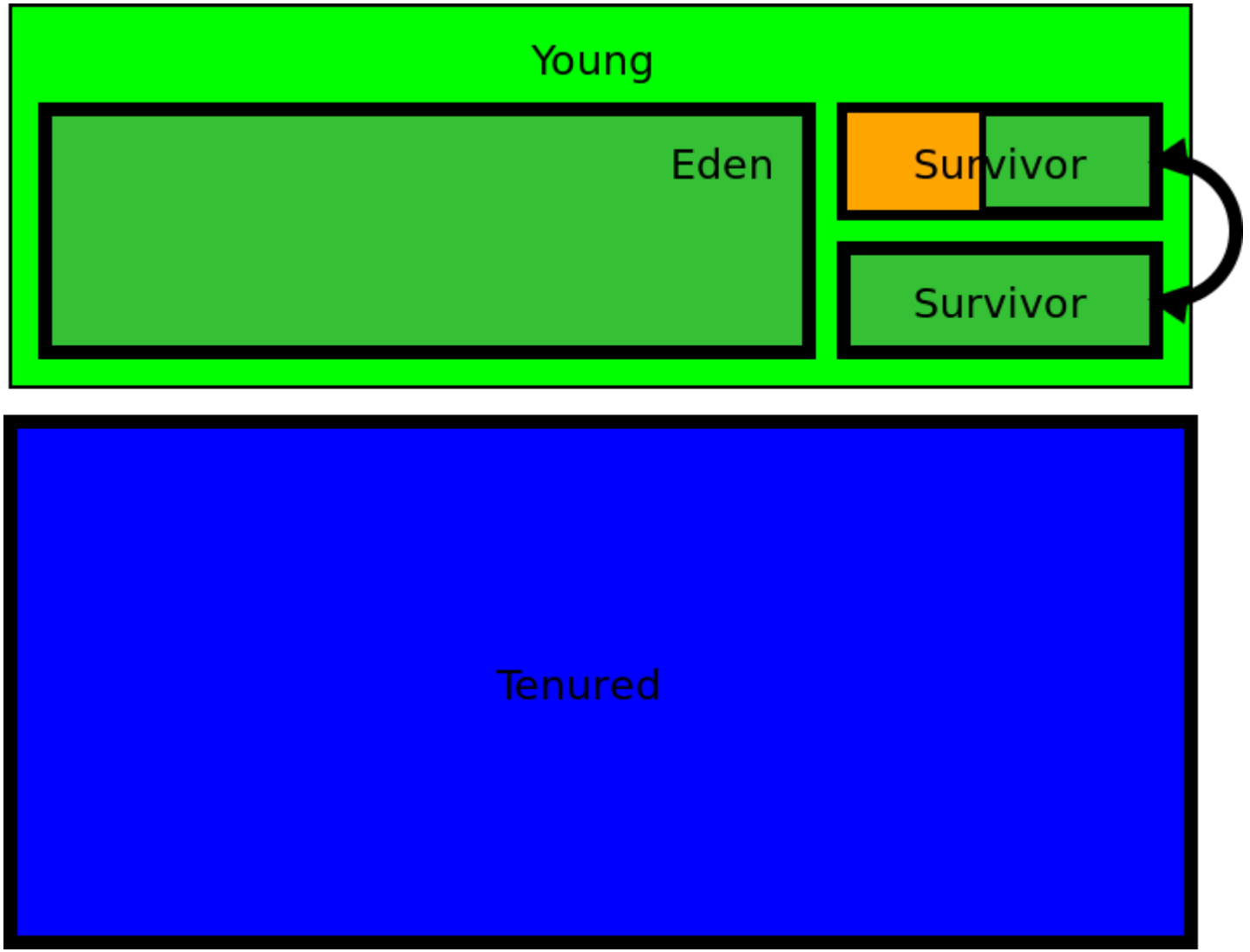

Young
Eden
Survivor
Survivor
Tenured

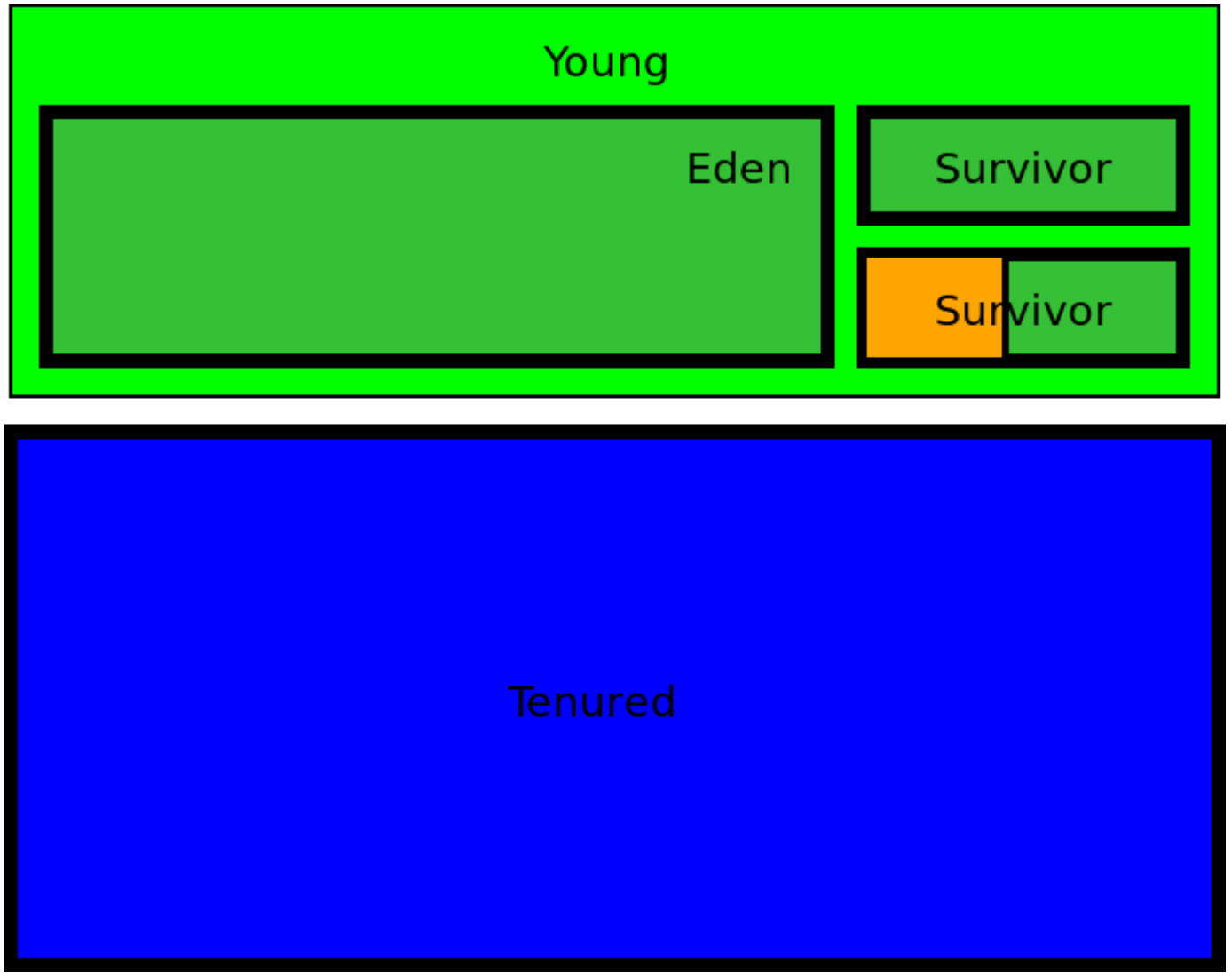

Young
Eden
Survivor
Survivor
Tenured

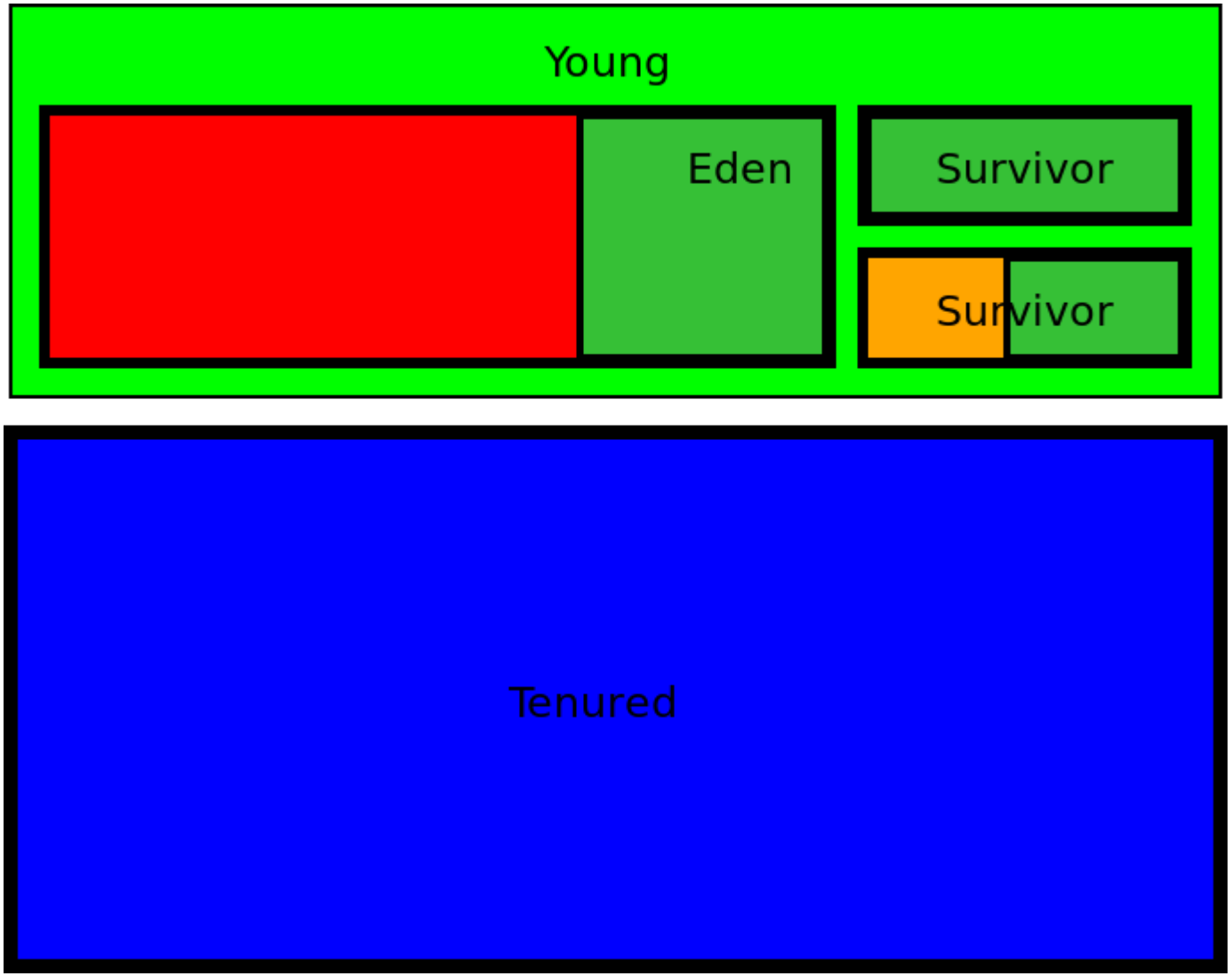

Young
Eden
Survivor
Survivor
Tenured

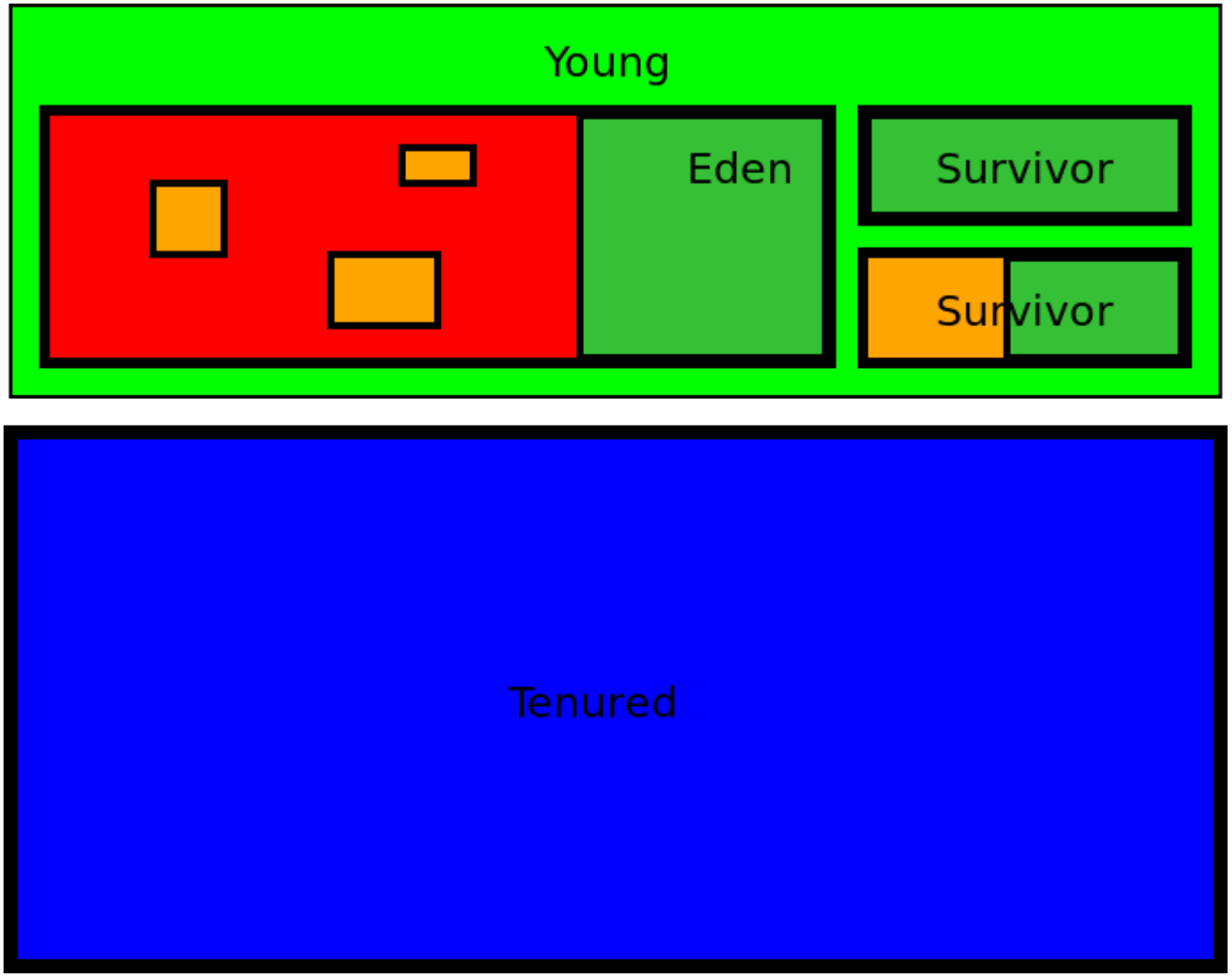

Young
Eden
Survivor
Survivor
Tenured

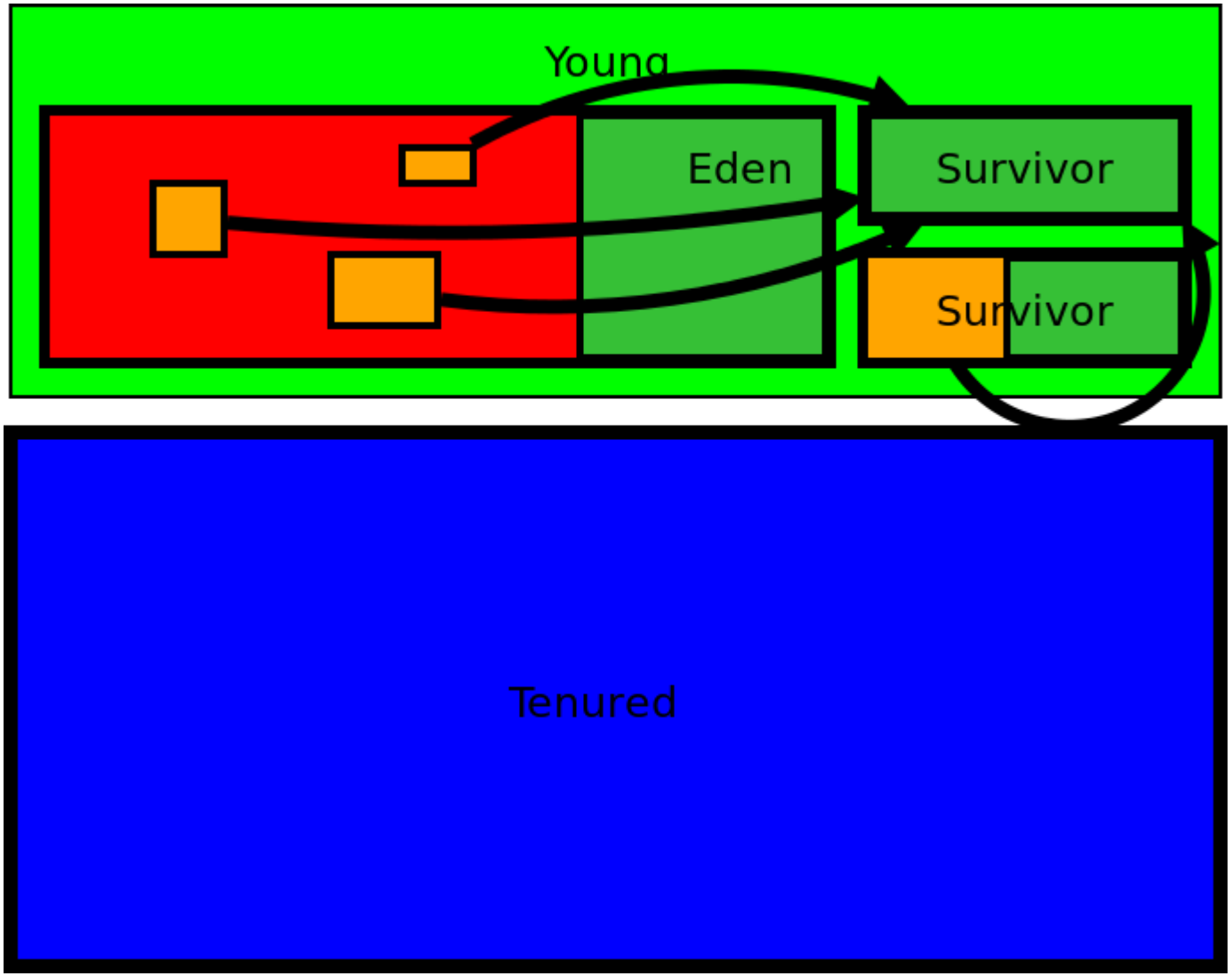

Young
Eden
Survivor
Survivor
Tenured

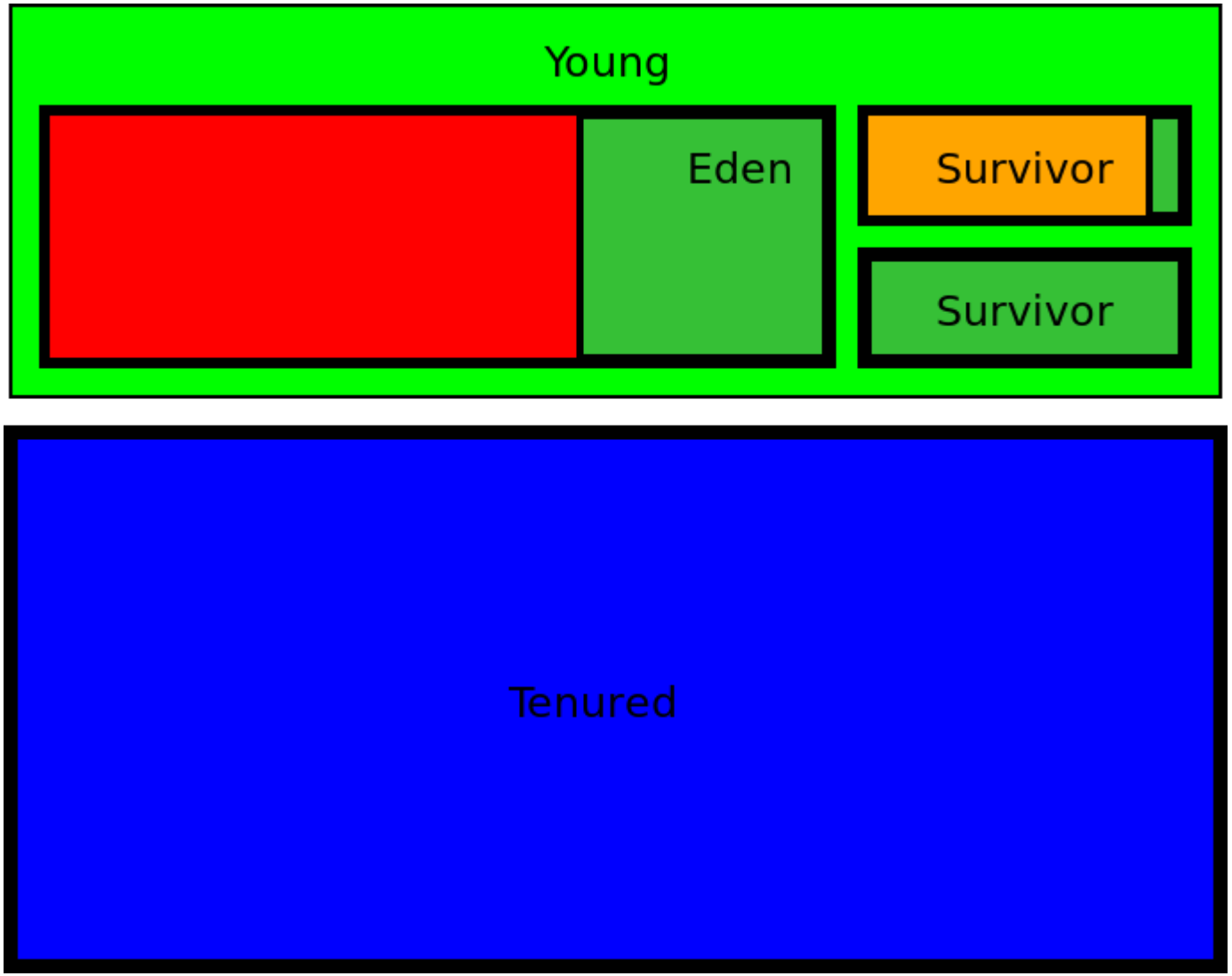

Young
Eden
Survivor
Survivor
Tenured

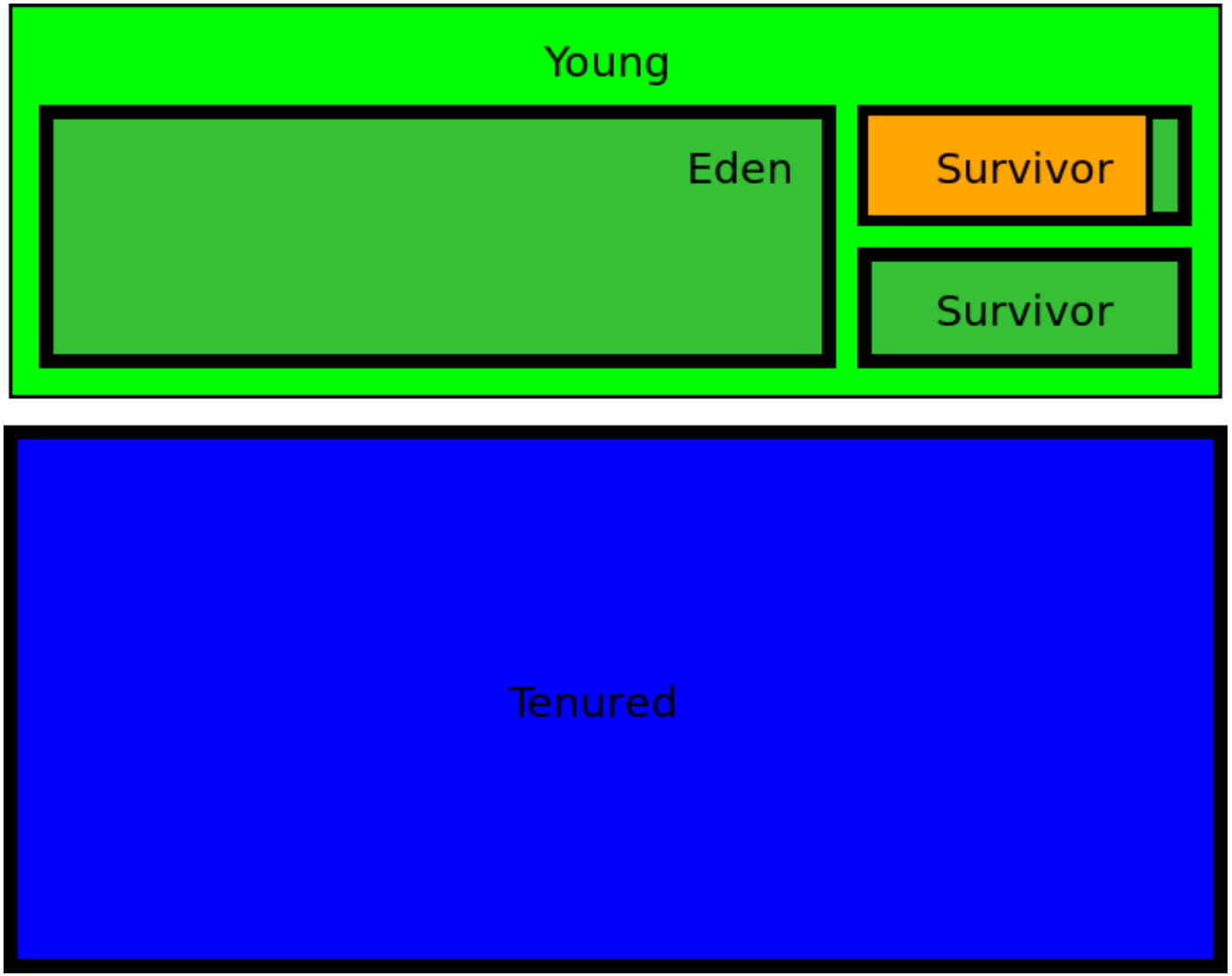

Young
Eden
Survivor
Survivor
Tenured

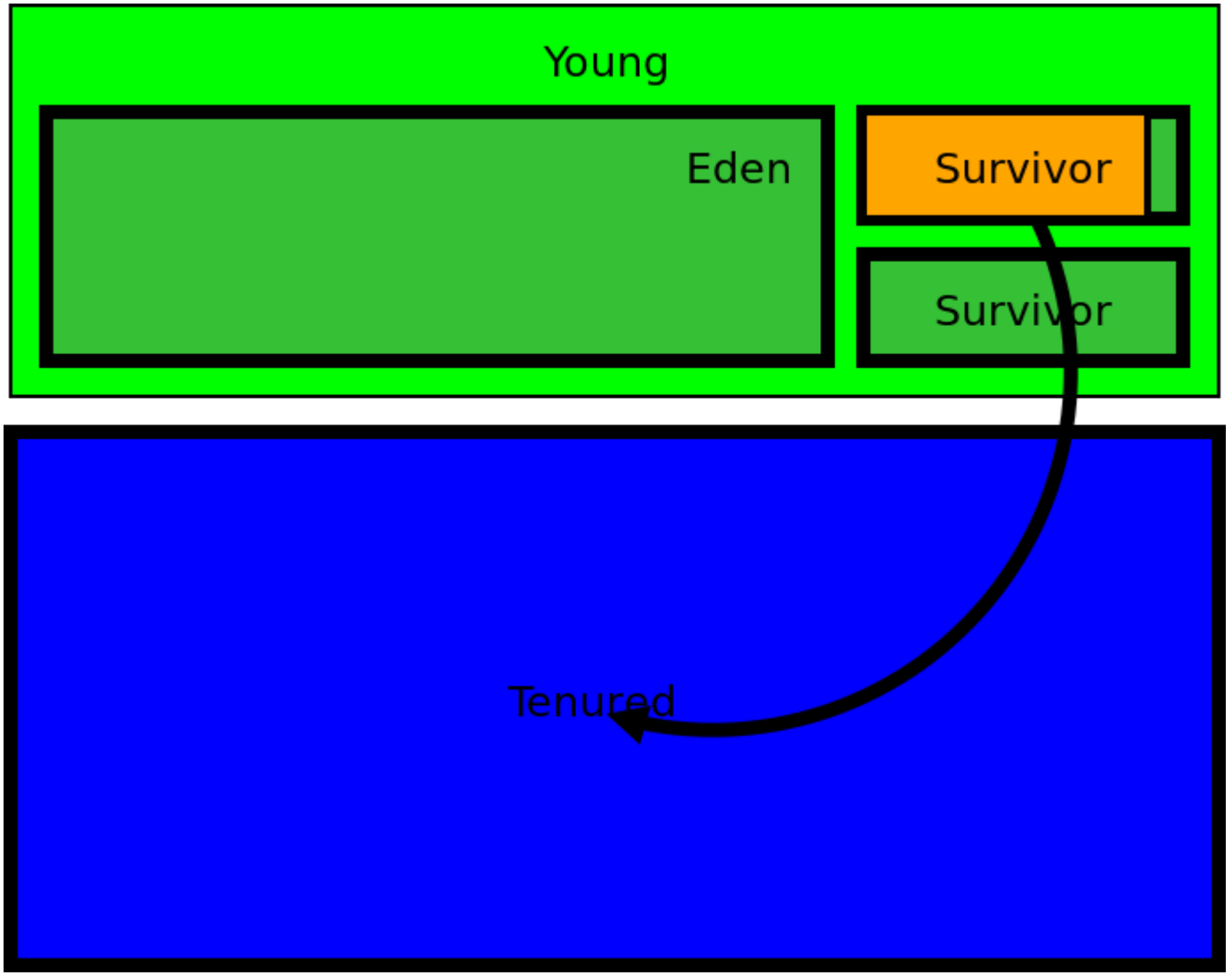

Young
Eden
Survivor
Survivor
Tenured

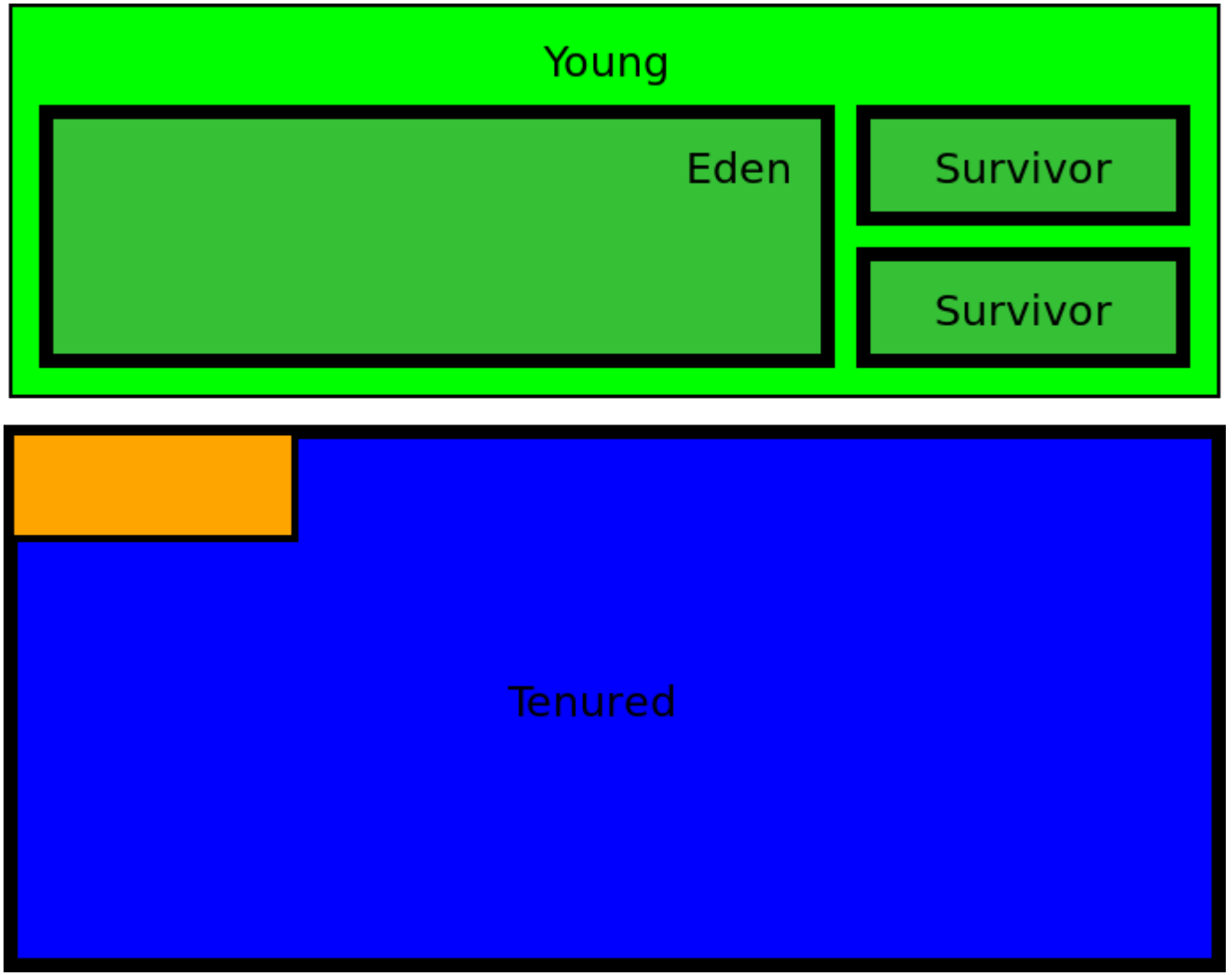

Young
Eden
Survivor
Survivor
Tenured


Young
Eden
Survivor
Survivor
Tenured


Young
Eden
Survivor
Survivor
Tenured


Young
Eden
Survivor
Survivor
Tenured


Young
Eden
Survivor
Survivor
Tenured


Young
Eden
Survivor
Survivor
Tenured

GC

Облегчить работу GC:

- Значительно уменьшить (в идеале избавиться) от порождения в куче объектов которые станут “мусором” сразу по выходу из метода
- Уменьшить общее число объектов в старом поколении

MessageReader

Парсим значения в примитивы...

```
int orderId =
AsciiByteUtils.parseInt(buffer,
buffer.position(), ID_LENGTH);
```

... и так их и передаем

```
tob.onAddOrder(buffer, orderId,
symbolOffset, side, price, qty);
```

А нужна ли нам строка?

- Если данные лежат в буфере и массиве, можно передать ссылку на него и оффсет:

```
onAddOrder(ByteBuffer src, int symbolOffset, …

Int2IntSortedMap prices = symbol2sell.get(src,
symbolOffset);
```

Храним Top в массивах

```
private int[] tops = new int[INITIAL_SIZE * 4];
private void setSellPrice(int pos, int price) {
    tops[pos * 4] = price;
}
...
private void setSellQty(int pos, int qty) {
    tops[pos * 4 + 1] = qty;
}
```

Храним ордера в коллекции как данные (не ссылки)

orders =

new Int2FixedByteSliceOpenHashMap(1 + 4 + 4 + 6)

Интерфейс позволяет копировать данные из одного большого буфера:

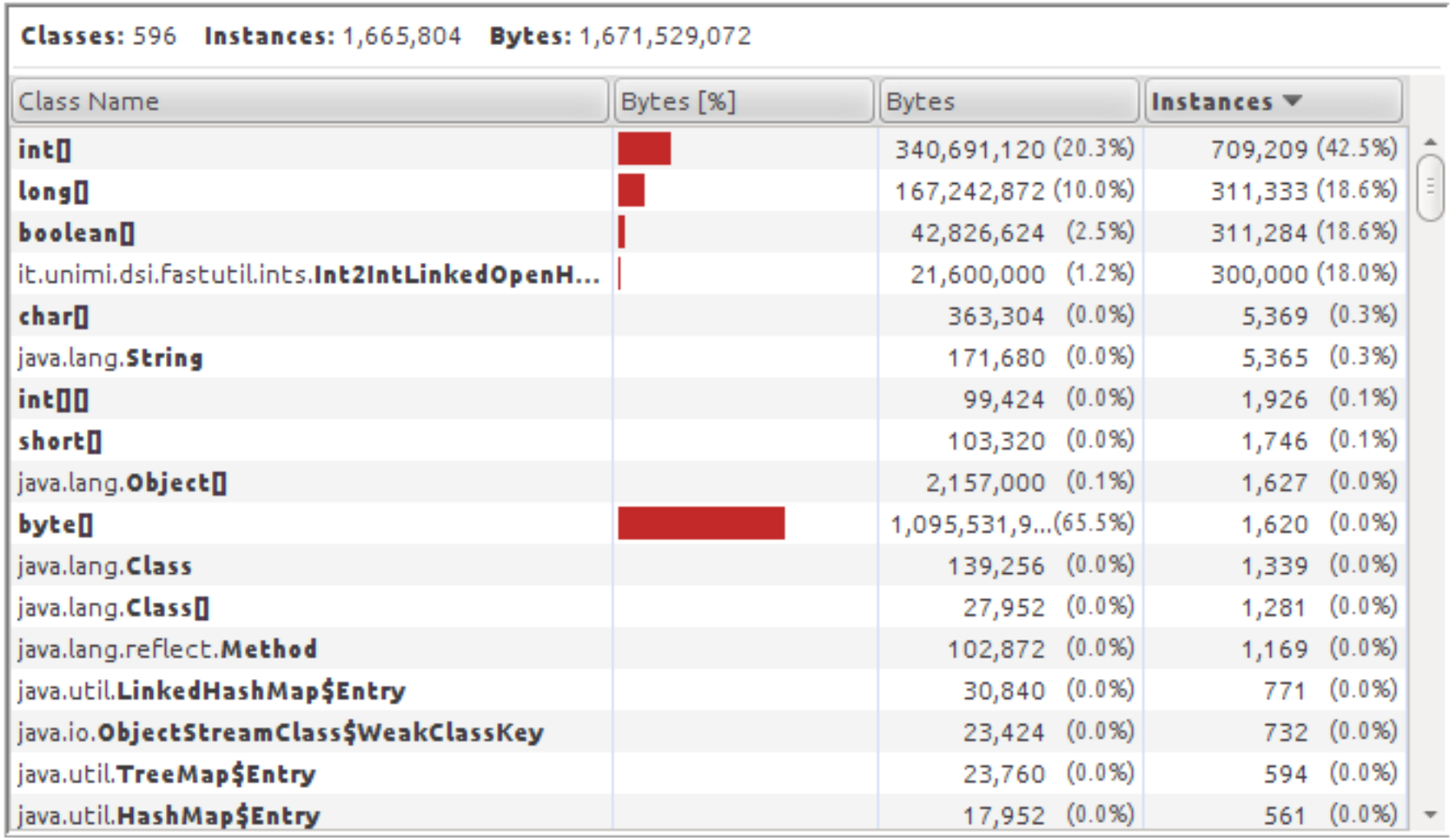
put(int k, byte[] valSrc, int valOffset);

Состояние “кучи” (до)

Classes: 597 **Instances:** 25,869,949 **Bytes:** 1,652,696,400

Class Name	Bytes [%]	Bytes	Instances ▼
java.lang.**Integer**		143,488,816 (8.6%)	8,968,051 (34.6%)
java.util.**HashMap$Entry**		109,988,896 (6.6%)	3,437,153 (13.2%)
char[]		102,914,696 (6.2%)	3,209,877 (12.4%)
java.lang.**String**		102,715,264 (6.2%)	3,209,852 (12.4%)
java.util.**TreeMap$Entry**		119,467,680 (7.2%)	2,986,692 (11.5%)
ru.secon.plain.**Order**		95,571,360 (5.7%)	2,986,605 (11.5%)
int[]		98,024,384 (5.9%)	571,099 (2.2%)
java.util.**TreeMap**		14,403,744 (0.8%)	300,078 (1.1%)
ru.secon.plain.**TopOfBook$Top**		4,800,000 (0.2%)	150,000 (0.5%)
java.nio.**HeapByteBuffer**		565,632 (0.0%)	11,784 (0.0%)
java.nio.**HeapCharBuffer**		565,632 (0.0%)	11,784 (0.0%)
java.util.**AbstractMap$SimpleIm...**		117,816 (0.0%)	4,909 (0.0%)
int[][]		96,432 (0.0%)	1,858 (0.0%)
java.lang.**Object[]**		68,936 (0.0%)	1,746 (0.0%)
short[]		101,048 (0.0%)	1,707 (0.0%)
byte[]		839,180,376 (50.7%)	1,614 (0.0%)
java.lang.**Class**		136,136 (0.0%)	1,309 (0.0%)

Состояние “кучи” (после)



Classes: 596 **Instances:** 1,665,804 **Bytes:** 1,671,529,072

Class Name	Bytes [%]	Bytes	Instances ▼
int[]		340,691,120 (20.3%)	709,209 (42.5%)
long[]		167,242,872 (10.0%)	311,333 (18.6%)
boolean[]		42,826,624 (2.5%)	311,284 (18.6%)
it.unimi.dsi.fastutil.ints.**Int2IntLinkedOpenH...**		21,600,000 (1.2%)	300,000 (18.0%)
char[]		363,304 (0.0%)	5,369 (0.3%)
java.lang.**String**		171,680 (0.0%)	5,365 (0.3%)
int[][]		99,424 (0.0%)	1,926 (0.1%)
short[]		103,320 (0.0%)	1,746 (0.1%)
java.lang.**Object[]**		2,157,000 (0.1%)	1,627 (0.0%)
byte[]		1,095,531,9...(65.5%)	1,620 (0.0%)
java.lang.**Class**		139,256 (0.0%)	1,339 (0.0%)
java.lang.**Class[]**		27,952 (0.0%)	1,281 (0.0%)
java.lang.reflect.**Method**		102,872 (0.0%)	1,169 (0.0%)
java.util.**LinkedHashMap$Entry**		30,840 (0.0%)	771 (0.0%)
java.io.**ObjectStreamClass$WeakClassKey**		23,424 (0.0%)	732 (0.0%)
java.util.**TreeMap$Entry**		23,760 (0.0%)	594 (0.0%)
java.util.**HashMap$Entry**		17,952 (0.0%)	561 (0.0%)

Где порождаются объекты

Способ замера:

- Инструментируем код
- Запускаем интересующий процесс
- Отмечаем начало замера
- Отмечаем конец замера
- Смотрим стэк-трейсы

Модифицированный HeapTracker

1: new: 184790 live: 184790 total: 184790
Ljava/lang/Object;.<init>@1[Object.java:20]
Lru/secon/plain/Order;.<init>@1[Order.java:6]
Lru/secon/plain/MessageReader;.processBuffer@53[MessageReader.java:46]
Lru/secon/plain/PerfTest;.main@147[PerfTest.java:65]
2: new: 184789 live: 184789 total: 184789
Ljava/lang/Object;.<init>@1[Object.java:20]
Ljava/nio/Buffer;.<init>@1[Buffer.java:170]
Ljava/nio/ByteBuffer;.<init>@6[ByteBuffer.java:259]
Ljava/nio/HeapByteBuffer;.<init>@10[HeapByteBuffer.java:52]
Ljava/nio/ByteBuffer;.wrap@7[ByteBuffer.java:350]
Ljava/lang/StringCoding$StringDecoder;.decode@41[StringCoding.java:137]
Ljava/lang/StringCoding;.decode@116[StringCoding.java:173]
Ljava/lang/StringCoding;.decode@11[StringCoding.java:185]
Ljava/lang/String;.<init>@13[String.java:570]
Lru/secon/plain/MessageReader;.processBuffer@96[MessageReader.java:52]
Lru/secon/plain/PerfTest;.main@147[PerfTest.java:65]

-verbose:gc

- Сборка мусора в Young Generation (Minor collections)

[GC 881024K->879266K(938624K), 0.0239800 secs]

- Сборка мусора в Old Generation (Major collections)

[Full GC 1153154K->1148852K(1546560K), 0.3228770 secs]

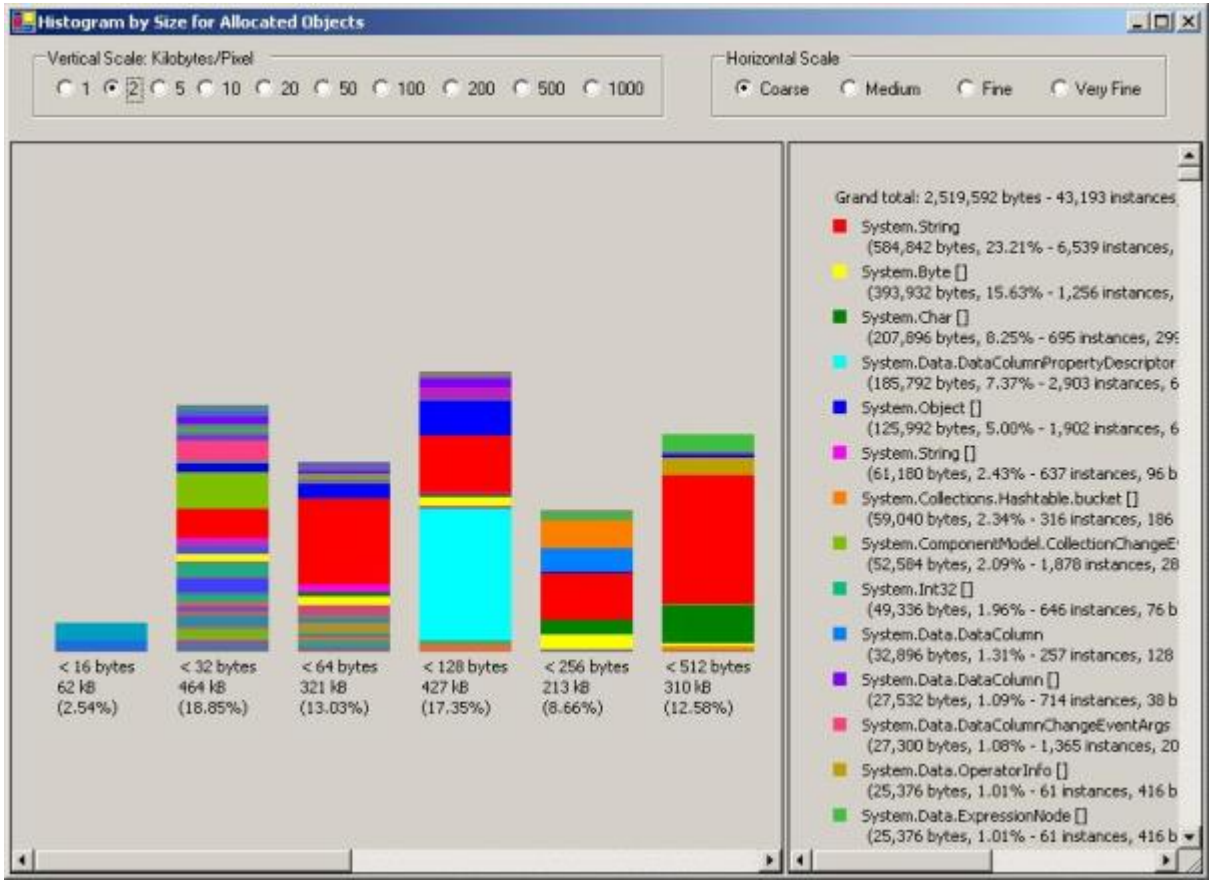
До

[GC 1270684K->1210948K(1499264K), 0.4404990 secs]
[GC 1319548K->1260924K(1528960K), 0.4733980 secs]
[Full GC 1260924K->1260869K(1786496K), 1.3706780 secs]
[GC 1377029K->1334149K(1803712K), 0.2321740 secs]
[GC 1485381K->1410677K(1810304K), 0.3162720 secs]
[GC 1561909K->1484965K(1855936K), 0.5098820 secs]
[Full GC 1484965K->1433957K(2069504K), 2.1848030 secs]
[GC 1630117K->1541605K(2069632K), 0.3652750 secs]
[GC 1737765K->1635349K(2071872K), 0.6371300 secs]
[GC 1825493K->1743829K(2112832K), 0.5383750 secs]
[Full GC 1743829K->1652420K(2382400K), 3.0437580 secs]
[GC 1842564K->1757700K(2406400K), 0.4218340 secs]
[GC 1979588K->1864964K(2414272K), 0.6487500 secs]

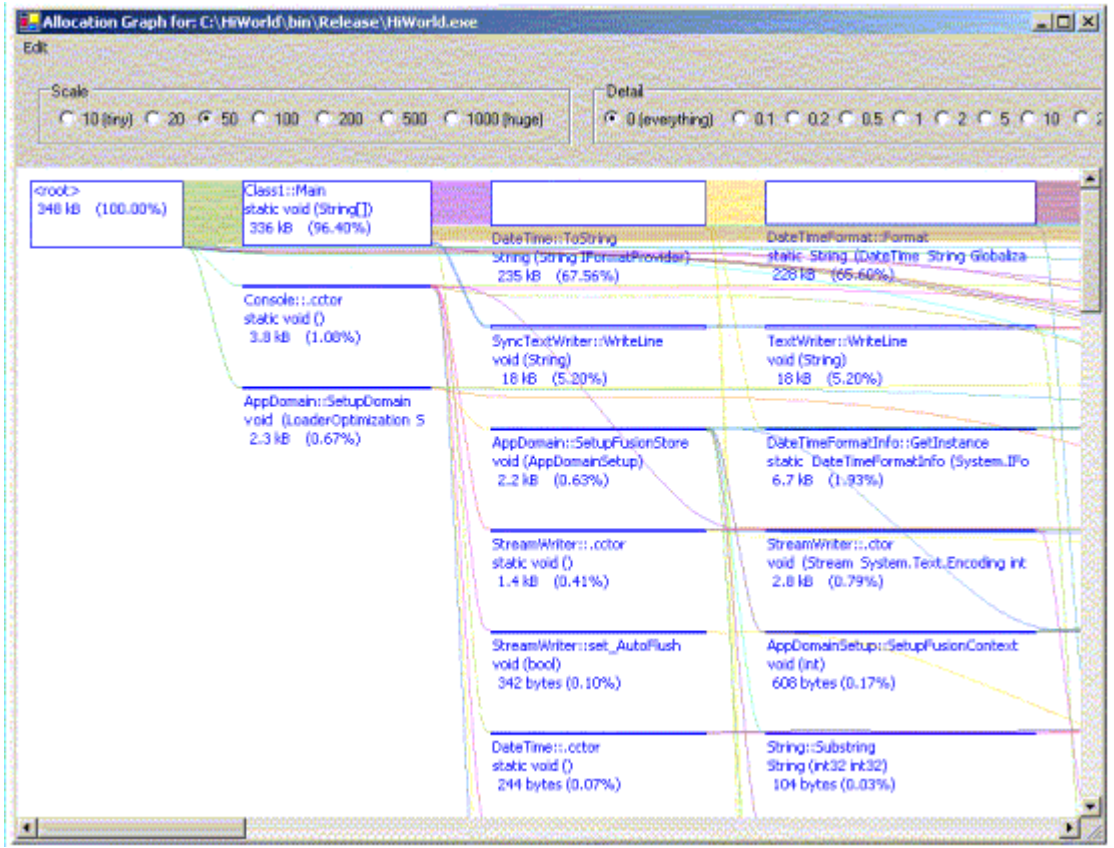
После

[GC 910364K->909188K(1097920K), 0.0294890 secs]
[GC 940292K->940420K(1097920K), 0.0514670 secs]
[Full GC 940420K->940401K(1177792K), 0.0779090 secs]
[GC 971505K->971638K(1198144K), 0.0561410 secs]
[GC 1023094K->1022790K(1208704K), 0.0542470 secs]
[Full GC 1022790K->1022770K(1317248K), 0.1424450 secs]
[GC 1075815K->1075298K(1379904K), 0.0560660 secs]
[GC 1153506K->1153154K(1386432K), 0.0722250 secs]
[Full GC 1153154K->1148852K(1546560K), 0.3228770 secs]
[GC 1349940K->1345908K(1637504K), 0.0532410 secs]
[Full GC 1345908K->1264360K(1765568K), 0.3047320 secs]
[GC 1702184K->1703632K(2015296K), 0.1122350 secs]
[Full GC 1703632K->1514315K(2031808K), 0.4659840 secs]

CLR Profiler



CLR Profiler



Результаты тестов (после)

911681 per second (time lag: 0)
873617 per second (time lag: 0)
794956 per second (time lag: 0)
813831 per second (time lag: 0)
860524 per second (time lag: 0)
842524 per second (time lag: 0)
826807 per second (time lag: 0)
695091 per second (time lag: 0)
844197 per second (time lag: 0)
384675 per second (time lag: 1)
820852 per second (time lag: 0)
809632 per second (time lag: 0)
779316 per second (time lag: 0)

762793 per second (time lag: 0)
701988 per second (time lag: 0)
739456 per second (time lag: 0)
669018 per second (time lag: 0)
210585 per second (time lag: 0)
748317 per second (time lag: 0)
742608 per second (time lag: 0)
739628 per second (time lag: 0)
695432 per second (time lag: 0)
726261 per second (time lag: 0)
719516 per second (time lag: 0)
714206 per second (time lag: 0)
643562 per second (time lag: 0)

Инструменты

- Visualvm (http://visualvm.java.net/)
- HeapTracker (http://highperformancejava.blogspot.com/)
- CLR Profiler (http://en.wikipedia.org/wiki/CLR_Profiler)

Чтиво

- What Every Programmer Should Know About Memory www.akkadia.org/drepper/cpumemory.pdf
- Virtual Machine Garbage Collection Tuning
  http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

Контакты

- Примеры доклада: https://github.com/alsor/simple-market-data
- Доработки в fastutils: https://github.com/alsor/data-collections
- alsor.net@gmail.com