

+ - 2017 JVM | Алексей Рагозин - Сборка мусора в Java без пауз (Deutsche Bank)

https://www.youtube.com/watch?v=n89CZS0u6dY&ab_channel=HighLoadChannel

Важное:

1. Concurrent mark sweep выполняет остановку приложения для завершения маркировки

В докладе рассмотрено создание jvm работающей без пауз.

Сборка мусора

Mark-Sweep

- ✓ Фаза 1 – маркировка достижимых объектов
- ✓ Фаза 2 – “вычистка” мусора

Copy collector (сборка копированием)

- ✓ Использует две области памяти, но выполняется в один проход

Mark-Sweep-Compact

- ✓ Mark-Sweep + перемещение живых объектов

Барьеры

Барьеры – специальные процедуры, выполняемые VM **при каждом обращении к памяти**.

Барьер на запись – при записи указателя в память.

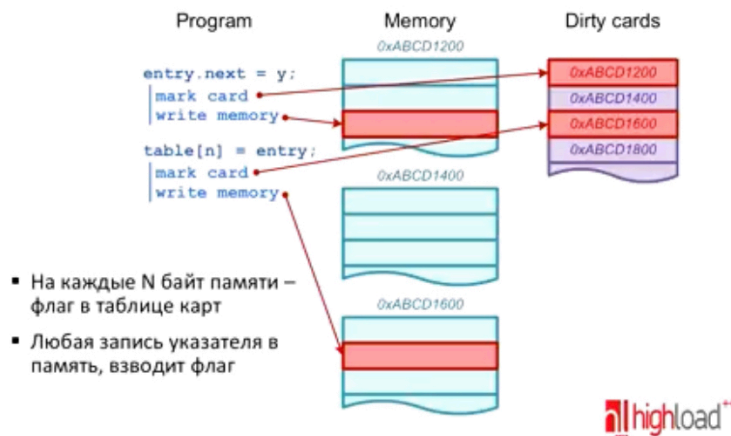
Барьер на чтение – при чтении указателя из памяти.

JIT компиляция позволяет частично сократить нагрузку барьеров за счёт глобальной оптимизации.

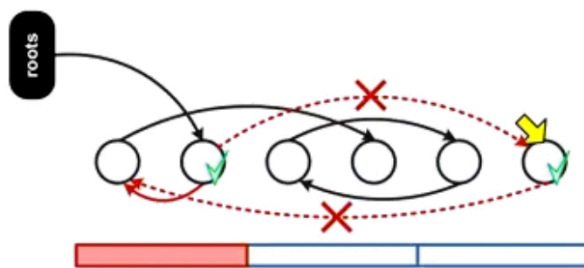
Крайне сложно найти живые объекты в графе по мере сборки мусора, поскольку граф постоянно меняется.

Реализация фоновой маркировки (половина алгоритма concurrent mark sweep) в HotSpotJvm основывается на **карточном барьере**. На каждые 512 байт памяти есть флажок, каждый раз когда код что-то пишет в память, он зануляет этот флажок.

Карточный барьер



Фоновая маркировка



После выполнения маркировки проверяется таблица карт и те участки памяти, которые были модифицированы помечаются заново.

Поскольку мы делаем барьер на запись, а он срабатывает уже пост фактум, приходится иногда останавливать приложение, чтобы закончить маркировку.

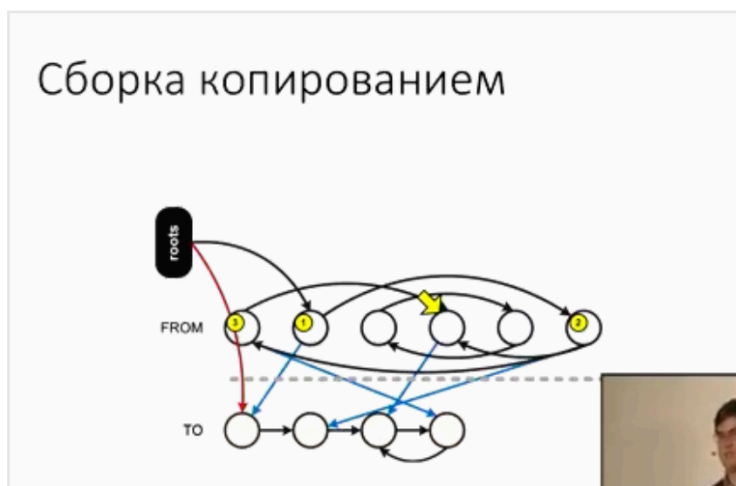
Карточный барьер на запись

Фоновая маркировка с карточным барьером на запись требует 2ух коротких пауз.

Так же, барьер на запись не позволяет реализовать фоновое перемещение объектов.

Можно ли придумать лучший алгоритм, используя барьер на чтение?

Для алгоритма работающего без пауз нужен другой тип барьера - **барьер на чтение**.



Сборка копирование выполняется для молодого поколения в hotSpotJvm.

Выполняется проход по объектам, к которым есть доступ из корневых ссылок, выполняется их копирование в другое пространство и вместе с этим чинятся ссылки на скопированные объекты. На практике данный алгоритм хорошо распараллеливается. Всю старую область памяти можно очищать. Однако этот алгоритм требует stop the world.

Большая часть академических работ по алгоритмам сборки мусора проводилась в 70-80 годах прошлого века и

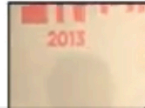
Алгоритм метроном - позволяет собирать мусор без пауз, основан на сборке копированием.

Алгоритм метроном

Алгоритм основан на сборке копированием.

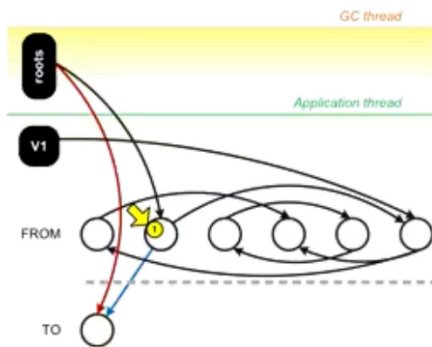
В течение сборки каждый прикладной поток

- При чтении указателя, каждый поток проверяет какой области он принадлежит
- Если адрес принадлежит "from" пространству, поток выполняет нерекурсивное копирование объекта в "to" и корректирует адрес.
- Продолжает работу



Отличие метронома от копирующего сборщика заключается в том, что после того как начался цикл сборки мусора все прикладные потоки работают только с данными в пространстве to.

Метроном



Проблемы алгоритма метроном:

1. Низкая скорость работы
2. Сложно обеспечить транзакционность в системе без блокировок
3. Сборка мусора с одним пространством не эффективная. Должно быть несколько поколений сборки.

На пути к решению

- Стоимость барьера на чтение
- Время копирования объекта
 - Например, большого массива
- Атомарное обнуление слабых ссылок
- Стоимость сборки мусора
 - Generational vs single spaced

В 2007 году компания Azul разработала суперкомпьютер, использующий java от HotSpot, но с нестандартным барьером на

Azul Vega – Java суперкомпьютер

2007 – Vega 2, 7200 series

- up to 768 cores
- 768 GiB RAM
- SPARC микорархитектура
- hardware assisted garbage collection

LVB - барьер на загрузку указателя

Load Value Barrier

- Срабатывает при загрузке указателя из памяти в регистр
- При срабатывании барьера происходит коррекция источника в памяти
- Прикладной код работает только с проверенными указателями
- При изменении правил барьера потоки ревалидируют указатели в регистрах

В HotSpot все потоки должны остановиться, у azul все потоки должны отметить, что прочитали актуальное состояние, перефильтровали свои регистры. Данная операция может выполняться асинхронно.

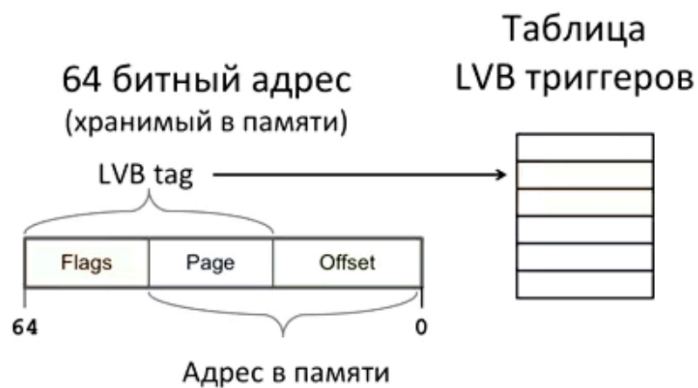
Адрес хранимый в памяти представляет собой своеобразную структуру.

Вся память разбита на 2МБ страницу, есть смещение внутри страницы, адрес страницы, биты не участвующие в адресации, например маркирующие.

Флаг и номер страницы представляет собой тэг, в памяти существует таблица тегов, в которой на каждый тег true или false.

Как работает барьер? Читает указатель, смотрит в таблицу, если true, выполняет барьер, иначе ничего не делает. Таким образом процессор выполняет проверку барьера и при этом исполняет код. Барьер срабатывает редко, поэтому не накладывает ограничений на производительность.

Azul JVM



Azul JVM

C4 (Continuously Concurrent Compacting Collector)

- Инкрементальная сборка мусора регионами
- Алгоритм сборки копированием
 - ✓ Бартер на чтение обеспечивает фоновую сборку
- Generational
 - ✓ Регионы классифицируются как молодые и старые
- Фоновая маркировка для оценки "замусоренности" региона

C4 не является алгоритмом реального времени



Однако для проверки необходимости запуска цикла сборки по старому поколению необходимо выполнить маркировку, которая требует остановки приложения, пусть и гораздо более быстрого, чем в HotSpot

Azul JVM

2010 Zing 4.0 – *virtual appliance*

2011 Zing 5.0

- x86 (64 бит) архитектура
- Linux

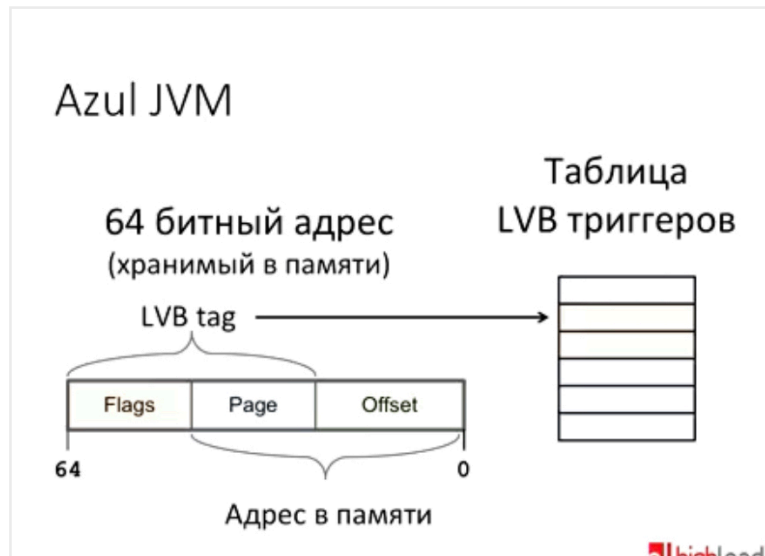
Механизм виртуальной памяти используется для эмуляции "незначащих" битов адреса.

На AZUL нельзя было запускать ничего кроме java и использовались специфичные процессоры.

В последующих версиях они это решили используя процессоры intel.

Была проблема при переходе с процессоров vega на intel, что процессоры vega позволяли хранить в адресе несколько незначащих битов. В архитектуре x86 такого нет.

Чтобы решить эту проблему Azul берет все возможные комбинации этого поля флагов и реплицирует адресное пространство виртуальной памяти. Независимо от того какие биты будут в поле флагов, после преобразования через таблицу адресов микропроцессор получит один и тот же физический адрес в памяти.



Также при такой архитектуре есть проблема копирования больших объектов.

Azul JVM

3 стратегии копирования

- до 256 Kb – копирование байт между страницами
- 256 Kb – 16 Mb – перемещаются 4k страницы виртуальной памяти с последующей дефрагментацией
- Более 16 Mb – не перемещаются 2M страницы виртуальной памяти

Можно подытожить, что этот алгоритм не является алгоритмом сборки в режиме реально времени

Без пауз ≠ Без проблем

“Безпаузный” алгоритм не спасет от

- утечек памяти
- неправильного сайзинга памяти JVM
- свопинга
- псевдо-свопинга (*ожидания записи дискового кэша*)
- кривых рук (*хотя существенно снижает риск*)

Почему все используют hotSpot а не zing? Zing сложнее настраивать, уходит больше времени на настройку. Вероятны сложности при переходе с одной jvm на другую. Zing - путь вертикально масштабирования. Если зинг не будет успевать очищать память без пауз, то они начнутся. Сборщик мусора предугадывать есть ли мусор в куче. Это достигается за счет анализа динамики. Если приложение аллоцирует 1гб, из которых долгоживущих 10%, то при сбросе кэша с этим возникает проблема.

Один из способов избавиться от проблем со сборкой мусора - добавить больше памяти.