Виртуальные машины

Лекция 3: Управление памятью

Олег Плисс
Oleg.Pliss@oracle.com

ноябрь 2011

Sun microsystems
We make the net work.

Зачем VM управление памятью

- Модель памяти VM ближе к уровню языка программирования, чем у аппаратуры и ОС
  - ОС: области адресного пространства, физической и виртуальной памяти, защита страниц, обработка прерываний доступа к памяти
  - VM: объекты языковых типов

Управление памятью

- Статическое
  - Смещение присваивается в процессе компиляции или линковки
  - Адрес присваивается не позднее, чем в момент загрузки
- Динамическое
  - Отведение и освобождение во время выполнения программы

Динамическое управление памятью

- Ручное
  - Пулы
  - «Куча»
- Автоматическое
  - Автоматизированное освобождение памяти при ручном отведении
  - Сборка мусора

Управление памятью в пулах

- Пул — область памяти, отводимая некоторой системой отведения памяти, в пределах которой организуется вторичное распределение в соответствии с некоторой специфической дисциплиной
  - Память может отводиться ОС, в другом пуле или куче
  - Отведенная память может быть непрерывной или кусочной
- Примеры дисциплин
  - Стек — отведение с дисциплиной LIFO
  - Отведение и освобождение объектов фиксированного размера
  - Отведение объектов со сходными временами жизни (scoped memory)

Зачем отводить память в пулах

- Преимущества специализированного распределения памяти
  - Скорость
    - Для отведения достаточно нескольких машинных инструкций
    - Единовременное освобождение всего пула
  - Предсказуемость
    - Худший случай не сильно хуже среднего
    - Из-за простоты реализации легко поддается анализу
  - Минимизация внутренней фрагментации (заголовков объектов, служебных структур, потерь на округление размера)
  - Минимизация внешней фрагментации

Ручное отведение в «куче»

- «Куча» - обобщение пула для произвольной дисциплины отведений-освобождений и произвольных типов объектов
- Для мелких объектов внутренняя фрагментация может быть значительной
  - Округление размера
  - Header & footer
- Внешняя фрагментация
  - Коэффициент внешней фрагментации памяти пропорционален логарифму отношений максимального и минимального размеров объектов

    J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM,* 18(3): 416-423, July 1971.

Традиционные классы механизмов управления «кучей»

- Последовательный поиск подходящего блока
  - First fit, Next fit, Best fit, Worst fit с вариациями
- Разделение объектов по размерам
  - Segregated storage, Segregated fit
  - Segregated storage позволяет определить размер объекта по его адресу без использования заголовка
- Метод близнецов
  - Двоичных, взвешенных (несколько двоичных серий с малыми весами), Фибоначчиевых и двойных (3=2+1)
- Индексированный поиск
  - Indexed fit (структуры поиска блока по размеру)
- Поиск в битовом массиве
  - Bitmapped fit

Практический пример: dlmalloc (Doug Lea's malloc)

- Одна из лучших реализаций *malloc*
  - По скорости, фрагментации, локальности доступа, настраиваемости, легкости портирования, поддержке многопоточности
  - По результатам многих независимых тестов

    David Detlefs, Al Dosser, Benjamin Zorn. Memory Allocation Costs in Large C and C++ Programs. Technical Report CU-CS-665-93. Department of Computer Science, University of Colorado. August 1993
  - Бесплатная, с открытым хорошо документированным исходным кодом http://gee.cs.oswego.edu/pub/misc/malloc.html
  - Выбор применяемых стратегий, алгоритмов и структур данных осознан и ясно артикулирован
  - Встроенные опционально включаемые возможности для отладки и мониторинга
  - Расширенный программный интерфейс для создания множественных «куч» изменяемого размера и массивов однородных объектов

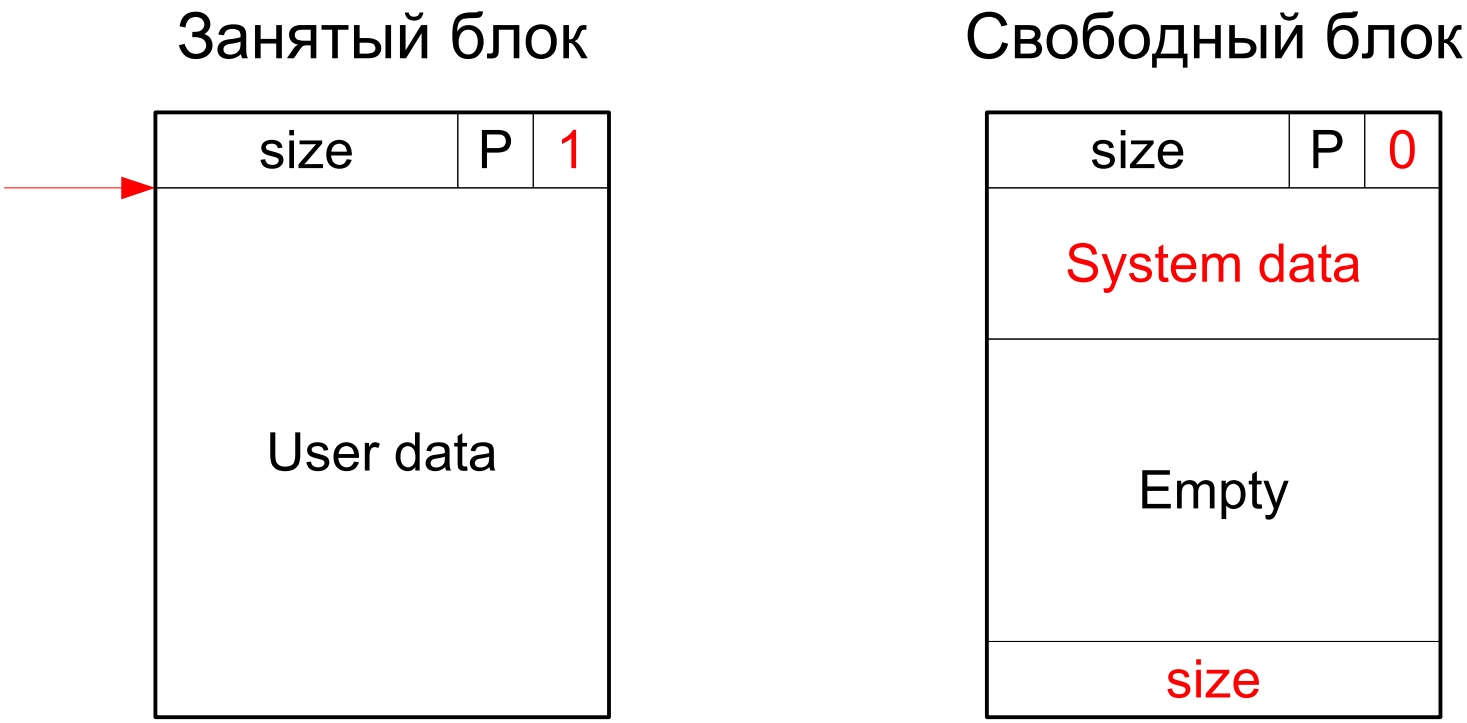
Doug Lea's malloc. Классификация

- Непрерывная куча переменного размера
  - Могут быть созданы несколько «куч»
  - Каждая «куча» может независимо не только увеличиваться, но и уменьшаться
  - Wilderness preservation - Для этого блок в конце кучи обрабатывается как бесконечно большого размера
- Поиск наилучшего подходящего блока (*BestFit*)
  - Для минимизации фрагментации
- Среди блоков одного размера выбирается ранее всех освобожденный (*FIFO*)
  - Для увеличения шансов слияния с освободившимися соседними блоками

Doug Lea's malloc. Классификация (2)

- Мгновенное слияние
  - Минимизация фрагментации в ущерб скорости обработки популярных размеров
  - Для повышения скорости можно кэшировать отдачу и отведение объектов некоторых типов и размеров в одном или нескольких пулах, реализованных поверх данной кучи
  - Эти пулы должны учитывать особенности конкретной программы
- Преимущественное деление
  - Если блок точного размера не найден, сначала пытаемся делить последний делившийся блок
  - Увеличивается локальность последовательных отведений

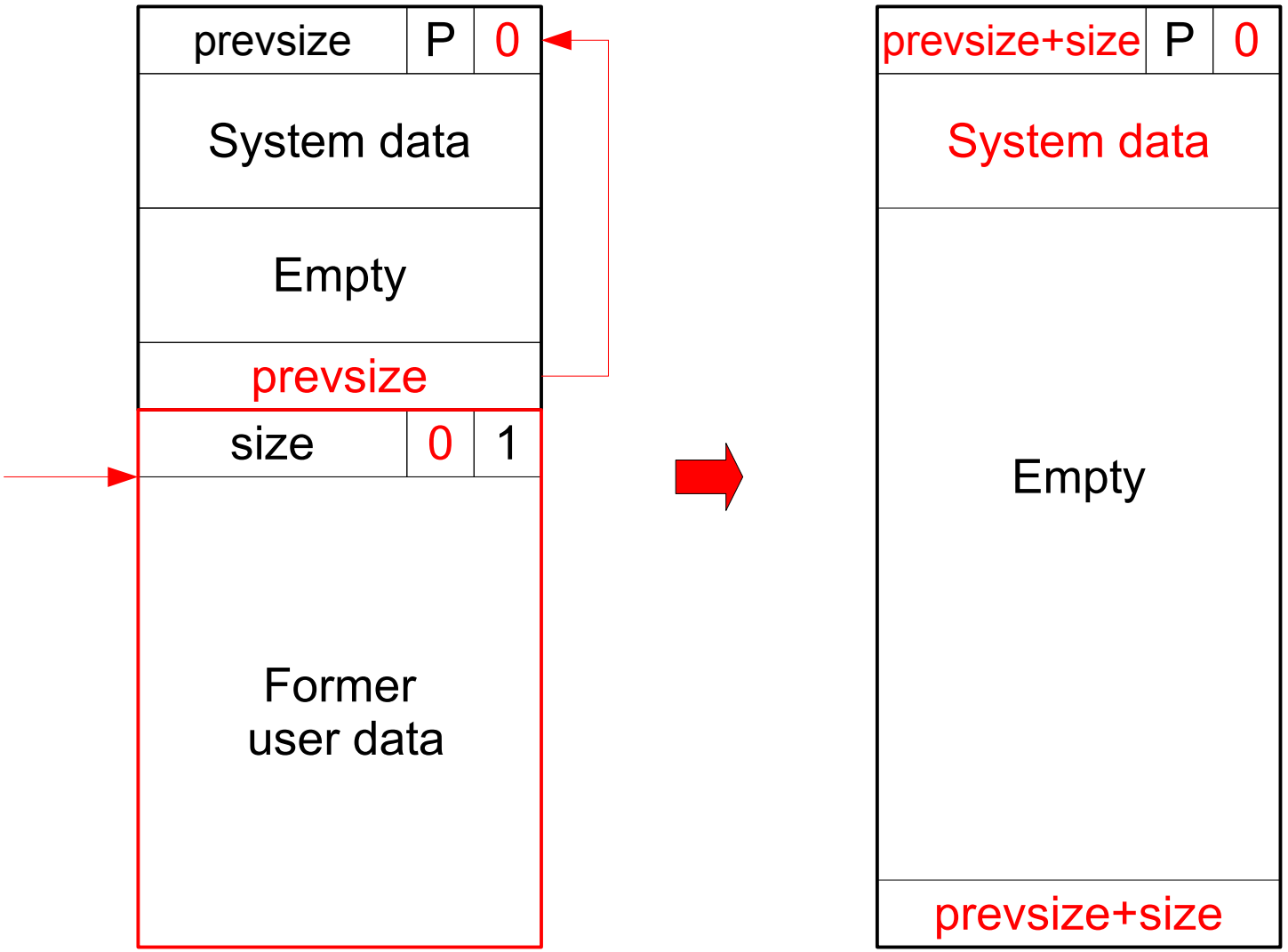
Doug Lea's malloc. Заголовки блоков



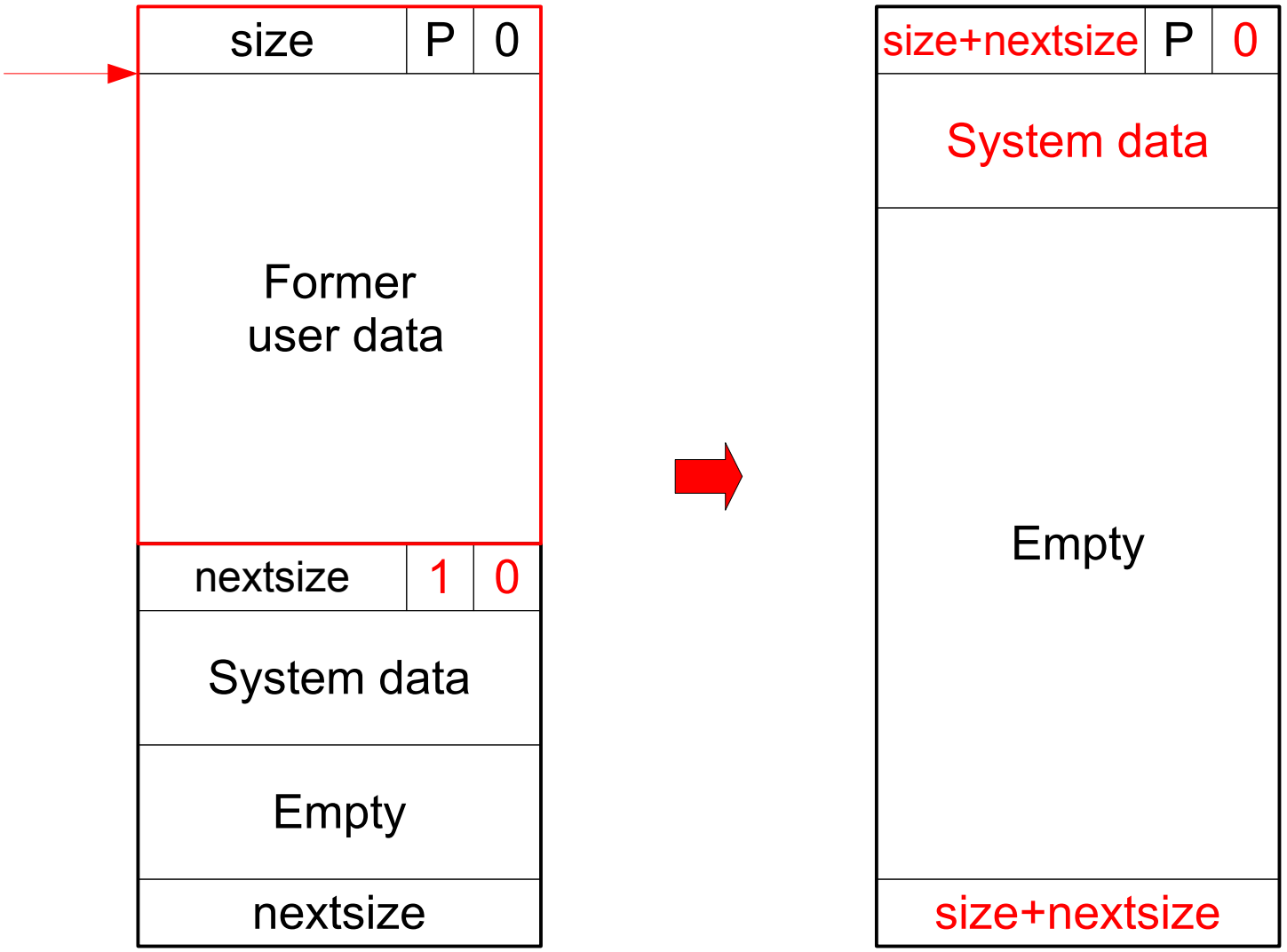
- Заголовок блока — одно слово
  - Блоки выровнены на границу двойного слова
  - Младшие 3 бита размера нулевые
  - Два из них используются как граничные признаки занятости данного и предыдущего блоков

    D.Knuth. The Art of Computer Programming. Volume 1: Fundamental Algorithms. Addison-Wesley, 1973.

Слияние с предыдущим свободным блоком в методе граничных признаков



Слияние со следующим свободным блоком в методе граничных признаков

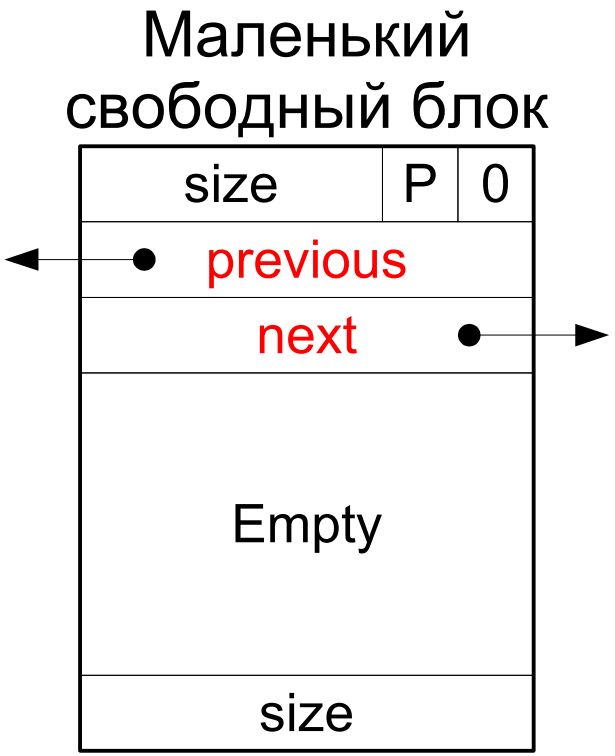


Doug Lea's malloc. Деление блоков по размеру

- Выравнивание блоков — двойное слово (8 байт)
- Блоки разделяются по размеру на маленькие, большие и огромные
  - Минимальный размер большого блока 256 байт
  - Минимальный размер огромного блока 1 МБ
  - Эти размеры управляются константами времени компиляции и могут быть изменены
  - К блокам разного размера применяются разные алгоритмы
  - В свободных блоках разного размера хранятся по-разному устроенные служебные данные
  - Огромные блоки отводятся отдельными вызовами *mmap*

Doug Lea's malloc. Маленькие блоки

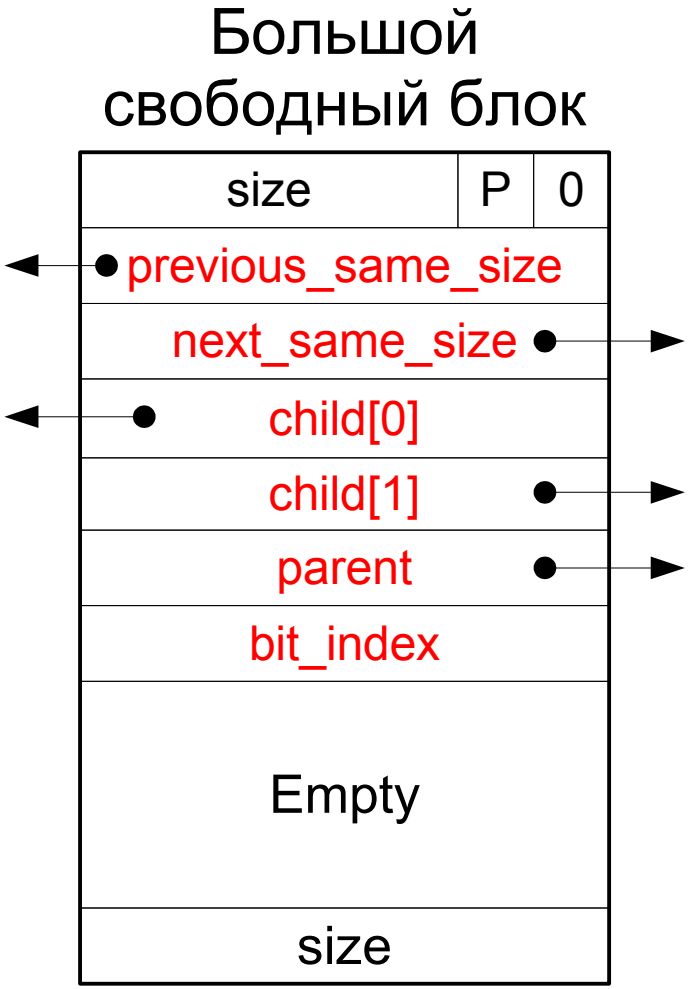
- Минимальный размер определяется структурой свободного блока - 4 слова
- Блоки разделяются на классы размеров от минимального маленького блока (16 байт) с шагом в округление (8 байт) до минимального большого, не включая его (256-8=248)



- В пределах каждого класса размеров свободные блоки объединяются в двусвязный циклический список
- Удаление из списка происходит при отведении блока или слиянии его с соседним блоком того же или другого размера

Doug Lea's malloc. Большие блоки

- Блоки разделяются на классы по round_down(log2(размер)) от 8 до 31 включительно
- В пределах класса блоки организованы в двоичное префиксное дерево (*бор*, *trie*).
- Блоки одного размера связаны в двусвязный циклический список.
- Поиск наиболее подходящего



- Если блоков найденного размера несколько, из них выбирается тот, который раньше всех добавлен в дерево

Автоматическое управление памятью

- Освобождает программиста от рутины ручного управления памятью
- Значительно снижает количество труднообнаружимых ошибок
  - Висящие ссылки исчезают как класс
  - Вероятность утечек памяти снижается
- Упрощает программные интерфейсы
- Часто *ускоряет* выполнение программы
  - По сравнению с ручным отведением в «куче»
- Часто *снижает* потребление памяти
  - По сравнению с фрагментированной «кучей» с ручным управлением

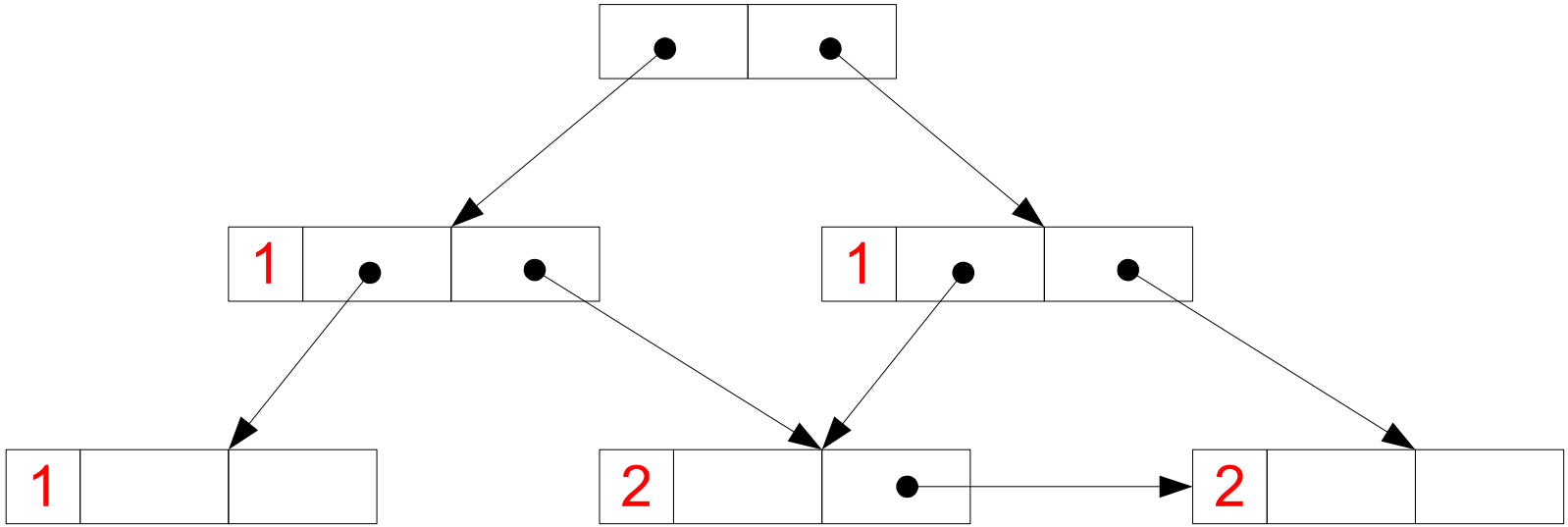
Ссылки на объекты в «куче»

- Ссылки из полей одних объектов на другие
- Корни — ссылки извне «кучи»
  - Статические — глобалы
  - Динамические — локальные переменные, регистры
- Производные ссылки
  - Значения, произведенные от ссылок в «кучу»
  - Бегущий указатель на элементы массива
  - Адрес текущей инструкции при размещении кода в «куче»

Классификация сборщиков мусора

- Подсчет ссылок или сканирование указателей
- Точный или консервативный
- Direct pointers vs. Object handles
- Непрерывная или кусочная куча
- Полная или выборочная сборка
- Остановительная (Stop-the-World) или Конкурентная/Инкрементальная
- Одно- или многопоточная

Счетчики ссылок



Счетчики ссылок

- Самый старый из алгоритмов сборки мусора
  - George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM,* 3(12), December 1960
  - Используется во множестве систем (InterLisp, Smalltalk-80, Modula-2+), библиотек (строки в MFC), прикладных программ (Adobe Photoshop, awk, Perl).
- Каждый объект снабжается счетчиком ссылок
  - В простейшем случае битов счетчика должно быть достаточно, чтобы избежать переполнения
  - Счетчик увеличивает размер заголовка
  - При меньшем числе битов усложняются операции, необходимо предусмотреть возможность переполнения

Счетчики ссылок (2)

- При присваивании ссылки на объект его счетчик инкрементируется, при затирании — декрементируется
  - Утяжеление присваивания — две записи в память в добавление к простой пересылке
  - Запись в память мешает компиляторным оптимизациям (alias analysis, scheduling...)
  - Требует синхронизации при параллельной обработке

    David L. Detlefs, Paul A. Martin, Mark Moir, Guy L. Steele Jr. Lock-Free Reference Counting. Proceedings of the 12th annual ACM symposium on Principles of distributed computing. August 26-29, 2001
  - Чтобы не обрабатывать специальный случай указателя NULL, полезно заменить его ссылкой на статически размещенный NullObject
  - Присваивание локальной переменной и передача параметром — тоже присваивания
    - При этом передача указателя параметром по ссылке — не присваивание указателя на объект, но дополнительный уровень косвенности при доступе

Счетчики ссылок (3)

- При падении счетчика до нуля декрементировать ссылки из полей объекта и освободить его
  - Возможно лавинообразное освобождение объектов (пример - бинарное дерево)
  - Может случиться при любом присваивании указателя
  - Критическая непредсказуемость времени выполнения элементарных операций
- Для устранения этого дефекта освобождаем объекты инкрементально
  - Записываем освобождаемые объекты в очередь
  - При каждом отведении памяти освобождаем объем объектов, пропорциональный количеству отводимой памяти
  - При исчерпании памяти обрабатываем очередь до тех пор, пока не освободится достаточное количество памяти
  - Нужна дополнительная ссылка в заголовке
    - Можно совместить ее со счетчиком ссылок
  - Усложняется и замедляется отведение памяти

Счетчики ссылок (4)

- Для ускорения частых присваиваний локалам исключаем их из счетчика ссылок
  - Если счетчик ссылок упал до нуля, сканируем стеки
  - Если ссылок не нашлось, освобождаем объект
  - Иначе записываем его в специальный набор ZCT объектов с нулевыми счетчиками
  - Периодически (например, при выходе из секции активации или переполнении этого набора) согласуем стеки и ZCT
  - Если согласовании обнаруживается объект с ненулевым счетчиком, он исключается из ZCT
  - Если согласовании обнаруживается объект, на который больше нет локальных ссылок, он освобождается
- С учетом всех модификаций — сложный алгоритм с высокими накладными расходами

Счетчики ссылок (5)

- Циклические структуры не обрабатываются
  - Из-за взаимных ссылок счетчики элементов таких структур никогда не станут нулевыми
  - Объекты без ссылочных полей не могут образовывать циклических структур
- Для обработки циклических структур счетчики ссылок должны быть дополнены другим, лишенным этого недостатка алгоритмом
  - Счетчики ссылок как способ ускорить освобождение объектов
  - Однобитовые счетчики, размещаемые не в объекте, а в одном из свободных битов указателя на него
  - Нулевой бит означает уникальность ссылки на объект
  - Необходимо маскировать бит счетчика указателя при обращении к объекту

Сборка точная или консервативная

- Точная сборка — известны все указатели
  - Можно достоверно установить достижимость объекта от корней по ссылкам
  - Можно перемещать объекты
  - Куча может быть дефрагментирована
  - Простые и быстрые алгоритмы отведения
- Консервативная сборка — не все указатели известны
  - Вариант: никакие указатели не поддаются точной идентификации
  - Вариант: ссылки между объектами в «куче» и статические корни известны, а динамические корни — не вполне (*ambiguous roots collection*)
  - Не все объекты можно перемещать
  - Опасность фрагментации
  - Простое использование указателей
  - Более сложное и медленное отведение

Автоматизация освобождения памяти

- Automatic memory reclamation
  - Отдельная библиотека освобождения памяти
  - Без поддержки со стороны компилятора и «кучи»
- Память отводится в обычной ручной «куче»
- Библиотека следит за доступностью объектов и автоматически вызывает free
  - Можно освобождать и вручную, но это опасно
  - Специальное оформление ссылок вместо обычных указателей
  - Запрет или ограничение адресной арифметики, приведений типов и неразмеченных объединений
  - Счетчики ссылок, сканирование или их комбинация
- Простейший способ консервативной сборки
  - Никакие объекты не переместимы
  - Медленность отведения и опасность фрагментации

Direct pointers vs. Object handles

- Object handle — номер регистрационной записи объекта в таблице объектов
  - Ускоряют перемещение объектов
  - Легко осуществить подмену объекта
  - Легко перехватывать обращения к объекту при отладке
  - Замедляют доступ
  - Таблица объектов может переполниться

Куча непрерывная или кусочная

- Динамическое изменение размера кучи
  - При страничной адресации можно по максимуму зарезервировать диапазон адресного пространства и по мере необходимости отображать туда память
- При фиксированном размере кусков большой объект может не поместиться в один кусок
  - Нужны куски разных размеров или кусочное представление объектов
- Неиспользуемое место в конце кусков
- Проверка принадлежности адреса куче
  - Для непрерывной кучи это простая проверка принадлежности адреса диапазону
- Возможность освобождения куска целиком при утрате всех ссылок на его объекты
- Отдельный кусок как единица выборочной сборки и как работа при параллельной сборке

Сборка полная и выборочная

- Время выполнения полной сборки пропорционально суммарному размеру всех живых объектов
  - В худшем случае — размер кучи
- Выборочная сборка — в первую очередь собирать те объекты, которые с большой вероятностью мертвы
  - Повышается эффективность
  - Снижается средняя длительность пауз
  - В худшем случае вслед за выборочной сборкой следует полная — максимальная длительность паузы увеличивается
  - Усложняется реализация — необходимо следить за ссылками и сканировать их в дополнение к корням при выборочной сборке

Предсказание смертности объектов

- Слабая гипотеза поколений a.k.a. Гипотеза высокой детской смертности — большинство объектов умирает в младенчестве
  - Выполняется в большинстве практических программ
  - Легко написать программу, для которой она не верна
- Сильная гипотеза поколений — смертность объектов уменьшается с возрастом
  - Статистически не верна
- Гипотеза радиоактивного распада — смертность объектов растет с возрастом
  - Часто верна для очень старых объектов
  - За исключением некоторых вечноживущих объектов

Сборка мусора поколениями

- Выборочная сборка, использующая слабую гипотезу поколений
  - Как-либо отделить молодые объекты от прочих
  - Собирать их чаще
  - Достаточно повзрослевшие объекты продвигать в следующее поколение
  - В современных реализациях используется 2-3 поколения
- При сборке среди младших объектов нужно помнить о ссылках на них со стороны старших объектов
  - Сканировать все старые объекты долго
  - Перехватываем присваивания (*write barrier*), запоминаем ячейки, содержащие ссылки на молодые объекты, в специальном множестве (*remembered set*)
  - Если в языке программирования нет присваиваний, нет и ссылок из старшего поколения в младшее
- Модификация любого базового алгоритма

Точный барьер записи

```
inline void set( void** dst, void* src ) {
  if( is_old( dst ) ) {
    if( is_young( src ) ) {
      remember( dst );
    } else {
      forget( dst );
    }
  }
  *dst = src;
}
```

- Гарантирует, что все запомненные ссылки:
  - принадлежат объектам старого поколения
  - указывают на объекты младшего поколения
- Слишком длинный и неэффективный код
  - Две проверки условия, минимум по одному сравнению в каждой (особенно если «куча» кусочная)
  - Минимум два условных и один безусловный переход
  - Включает в себя добавление и исключение из множества
- Этот код порождается для каждого присваивания полю любого расположенного в «куче» объекта!

Упрощенный барьер записи №1

```
inline void set( void** dst, void* src ) {
  if( is_old( dst ) && is_young( src ) ) {
    remember( dst );
  }
  *dst = src;
}
```

- Избавляем барьер от дорогостоящего исключения из множества и безусловного перехода
- Запомненное множество теперь состоит из ссылочных полей старших объектов, которым с момента очистки множества присваивались указатели на младшие объекты
  - Больше размер множества, дольше его сканировать
  - Текущее значение запомненного поля не обязано указывать на младший объект
  - При сканировании выполняем проверку и исключаем из множества лишние поля

Упрощенный барьер записи №1. Дополнительные функции

```
inline void set_null( void** dst ) {
  if( is_old( dst ) ) forget( dst );
  *dst = null;
}

inline void set_young( void** dst, void* src ) {
  *dst = src;
}
```

- Даем программисту и оптимизирующему компилятору *возможность* в частных случаях использовать дополнительные функции
- При обнулении поля *полезно (но не обязательно)* исключить его из запомненного множества
- Не требуют барьера:
  - Инициализация полей создаваемого объекта
  - Присваивание нисходящей ссылки *(src <= dst)*

Упрощенный барьер записи №2

```
inline void set( void** dst, void* src ) {
  if( is_old( dst ) ) remember( dst );
  *dst = src;
}
```

- Продолжаем упрощать барьер — избавляемся от проверки принадлежности присваеваемой ссылки младшему поколению
- Запомненное множество теперь состоит из ссылочных полей старших объектов, которым что-либо присваивалось с момента очистки множества
- Чем проще барьер, тем больше размер запомненного множества и дольше его сканировать
  - Если реализация множества допускает переполнения, тем чаще они случаются

Предельно упрощенный барьер записи

```
inline void set( void** dst, void* src ) {
  remember( dst );
  *dst = src;
}

inline void set_null( void** dst ) {
  forget( dst );
  *dst = null;
}
```

- Избавляемся от всех проверок
  - Не всякая реализация запомненного множества позволяет включать в него произвольные ссылки
  - Если реализация допускает только расположенные в пределах кучи поля, а объекты могут быть отведены и вне ее (например, в стеке), то для присваивания полям таких объектов должны применяться другие функции
- Запомненное множество теперь состоит из всех ссылочных полей, которым что-либо присваивалось с момента очистки множества

Чуть более умный барьер записи

```
inline void set( void** dst, void* src ) {
  if( dst < src ) remember( dst );
  *dst = src;
}
```

- Фильтр нисходящих ссылок
  - Если выполнена слабая гипотеза поколений, в младшем поколении доминируют нисходящие ссылки
  - В языках или стиле программирования с преимущественно неизменными полями подавляющее большинство ссылок нисходящие
  - Сравнение двух расположенных в регистрах значений быстрее проверок принадлежности к поколениям
- Все ограничения предельно упрощенного барьера остаются в силе
- Особенно полезен при переполняющейся реализации запомненного множества

Способы запоминания ссылок

- Битовая карта указателей
- Пометка карт памяти
- Пометка страниц памяти
- Биты модификации страниц памяти
- Пометка заголовков объектов
- Связный список объектов
- Буфера последовательной записи
- Гибридная реализация
- Не все способы одинаково пригодны для разных алгоритмов и платформ

Битовая карта указателей

- «Куча» представляется как массив выровненных ячеек памяти размером в указатель
- Каждая ячейка снабжается своим битом пометки
  - Смысл этого бита может варьироваться в зависимости от области «кучи» и фазы сборки мусора
- Биты пометки хранятся в отдельном массиве битов (*битовой карте*)
  - Карта занимает 1/32 памяти «кучи»
- Индекс бита в массиве соответствует индексу ячейки памяти в «куче»
- Операции над битовой картой:
  - Проверка, установка и сброс бита про индексу
  - Очистка большого диапазона индексов
  - Итерация по индексам единичных битов в указанном большом диапазоне индексов с преимущественно нулевыми значениями битов

Операции над битами карты

```
inline unsigned heap_index ( const void* p ) {
  return ((unsigned*)p) – ((unsigned*)HeapBottom);
}
inline unsigned& bit_word ( const unsigned i ) {
  return ((unsigned*) HeapBitmap)[ i >> LogBitsPerWord ];
}
inline unsigned bit_index ( const unsigned i ) {
  return i & (BitsPerWord – 1);
}
inline unsigned bit_mask ( const unsigned i ) {
  return 1 << bit_index( i );
}
inline unsigned test_bit ( const void* p ) {
  const unsigned i = heap_index( p );
  return ( bit_word( i ) >> bit_index( i ) ) & 1;
}
inline void set_bit ( const void* p ) {
  const unsigned i = heap_index( p );
  bit_word( i ) |= bit_mask( i );
}
inline void clear_bit ( const void* p ) {
  const unsigned i = heap_index( p );
  bit_word( i ) &= ~bit_mask( i );
}
```

Ускорение операций над битами

- Выравниваем *HeapBottom* на *BitsPerWord* (32) слов
- Заранее вычисляем, запоминаем и используем *HeapBitmapStart* = &*word(-HeapBottom)*
- Вместо индекса используем исходный адрес

```
inline unsigned heap_index ( const void* p ) {
  return unsigned(p) >> LogBytesPerWord;
}
inline unsigned& bit_word ( const void* p ) {
  return ((unsigned*) HeapBitmapStart)
         [unsigned(p) >> (LogBytesPerWord+LogBitsPerWord)];
}
inline unsigned bit_index ( const void* p ) {
  return (unsigned(p) >> LogBytesPerWord) & (BitsPerWord-1);
}
```

- При наличии специальных процессорных инструкций для битовых операций переписываем функции на ассемблере (по 3-4 инструкции i386+ вместо 8)

Сканирование помеченных ссылок

- Если используется упрощенный барьер записи, среди запомненных ссылок могут содержаться *посторонние ссылки* - не указывающие на молодые объекты
- Во избежание накопления посторонних ссылок удаляем их из запомненного множества при его *первом* сканировании во время сборки мусора

```
void forEachRememberedRef ( void ref_do(Byte** ref) ) {
  forEachMarkedRef( ref ) {
    if( is_young( *ref ) ) {
      ref_do( ref );
    } else {
      clear_bit( ref );
    }
  }
}
```

Пометка карт памяти

- Не все процессоры снабжены инструкциями для установки бита в области памяти по его индексу
- Минимальной адресуемой единицей памяти обычно является байт
- Было бы расточительно отводить дополнительный байт на каждое выровненное слово «кучи»
  - Байтовая карта заняла бы 1/4 памяти «кучи»
- Делим память «кучи» не на слова, а на карты
  - Чем больше размер карты, тем меньше требуется байтов пометки, но тем ниже точность
  - Пометка карты означает, что среди входящих в нее слов *могут встретиться* ссылки на молодые объекты
  - Чаще всего используются 32, 64 или 128 слов

Барьер пометки карт памяти

- При очистке запомненного множества заполняем массив байтов пометки нечетным значением (например, 0xFF)
- Адрес записываемого поля выровнен и потому его младший байт четный
- Барьер записывает младший байт адреса поля в байт пометки соответствующей полю страницы
- Выравниваем *HeapBottom* на размер страницы
- Предварительно вычисляем, запоминаем и используем

```
Byte* CardTableStart = CardTable -
  ( unsigned(HeapBottom) >> LogBytesPerCard );
```

- Запоминание ссылки сводится к 2-4 машинным инструкциям

Итерация по помеченным картам

- Пройти по ссылочным полям всех объектов, частично или полностью содержащихся в помеченных картах
  - Некоторые алгоритмы сборки мусора требуют, чтобы просмотре запомненного множества каждая ссылка в младшее поколение посещалась ровно один раз
- Нужно уметь отыскать начало первого объекта, частично или полностью содержащегося в данной карте
  - Не содержащие ссылок объекты (например, массивы примитивных типов) полезно исключить
- Поскольку барьер неточный, во избежание накопления погрешностей полезно во время первого прохода удалять пометку карт, не содержаших ни одной ссылки на младшие объекты

Итерация по помеченным картам (1)

```
void forEachRememberedRef ( void ref_do(Byte** ref) ) {
  for( int i = 0; i < number_of_cards; i++ ) {
    if( is_marked_card( i ) ) {
      Byte* obj = find_first_object_for_card( i );
      bool young_ref_found = false;

      // if( obj ) - если бессылочные объекты исключены
      do {
        forEachRef( ref, obj ) {
          if( card_index( ref ) == i && is_young( *ref ) ) {
            young_ref_found = true;
            ref_do( ref );
          }
        }
        obj += allocation_size( obj );
      } while( card_index( obj ) == i );

      if( !young_ref_found ) {
        unmark_card( i );
      }
    }
  }
}
```

Ускорение итерации по помеченным картам

- Большинство карт не помечено, пропустим их в отдельном оптимизированном цикле
- Для исключения из цикла проверки выхода за границу массива полезно поместить за концом массива дополнительный помеченный байт

```
inline int next_marked_card ( int i ) {
  while( !is_marked_card( ++i ) );
  return i;
}
```

- Для дальнейшего ускорения пропуска непомеченных карт можно читать байты пометки выровненными словами или двойными словами
  - Предварительно дополнив сзади массив пометок соответствующим числом «помеченных» байтов

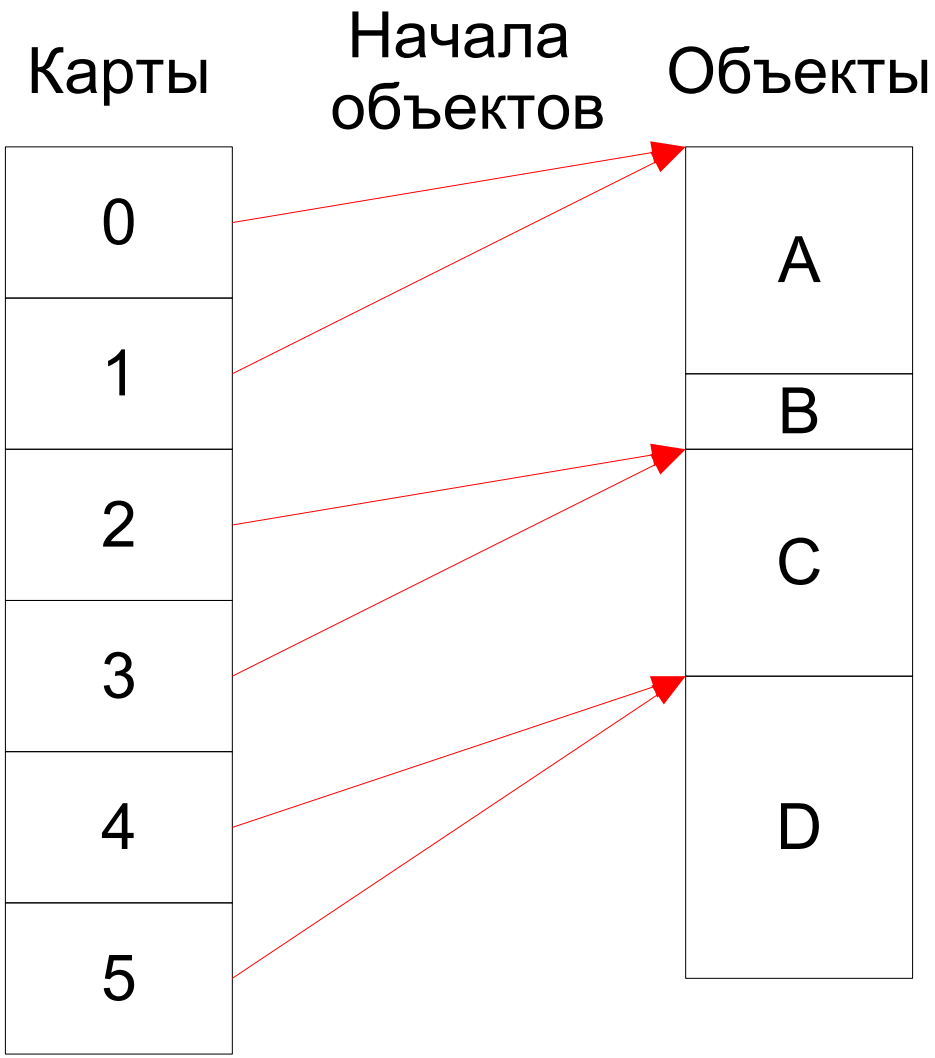
Итерация по помеченным картам (2)

```
void forEachRememberedRef ( void ref_do(Byte** ref) ) {
  for( int i = 0; (i = next_marked_card(i)) < number_of_cards; ){
    Byte* obj = find_first_object_for_card( i );
    // if( obj )
    for( bool young_ref_found = false;;) {
      forEachRef( ref, obj ) {
        const int next_card_index = card_index( ref );
        if( next_card_index > i ) {
          if( !young_ref_found ) unmark_card( i );
          i = next_card_index; young_ref_found = false;
          if( !is_marked_card( i ) ) break;
        }
        if( is_young( *ref ) ) {
          young_ref_found = true;
          ref_do( ref );
        }
      }
      obj += allocation_size( obj );
      const int next_card_index = card_index( obj );
      if( next_card_index > i ) {
        if( !young_ref_found ) unmark_card( i );
        i = next_card_index; young_ref_found = false;
        if( !is_marked_card( i ) ) break;
      }
    }
  }
}
```

Поиск начала первого объекта карты

- Просмотр запомненных ссылок производится при сборках мусора в младшем поколении
- Запомненные ссылки принадлежат объектам старшего поколения
- Расположение объектов старшего поколения может измениться только при сборке мусора в старшем поколении
- По окончании сборки в старшем поколении заполним таблицу указателей начал объектов для каждой карты этого поколения
  - Заполнение таблицы может быть совмещено с одной из фаз сборки мусора
  - Для ускорения просмотра запомненных ссылок полезно исключить не содержащие ссылок объекты

Простая таблица начал объектов



Сжатие таблицы начал объектов

- Каждый элемент простой таблицы занимает один указатель (4 байта). Нельзя ли его уместить в один байт?
- Чаще всего встречаются небольшие объекты
  - Первый объект карты обычно начинается в одной из предшествующих карт или с начала данной карты
  - Запишем в таблицу расстояние в словах от начала данной карты до начала объекта
- Большие объекты
  - Встречаются редко, занимают большое число карт
  - Наблюдение: у всех карт, занимаемых одним объектом, за исключением, быть может, первой, начальный объект совпадает
  - Попытаемся записать в таблицу число предыдущих страниц с тем же начальным объектом. Если оно в таблицу не поместится, для уменьшения числа скачков запишем максимальное помещающееся число

Сжатие таблицы начал объектов (1)

```
enum {
  MaxCardsPerSmallObject = 2,
  MaxWordsPerSmallObject = MaxCardsPerSmallObject * WordsPerCard
};

inline Byte* find_first_object_for_card ( int i ) {
  Byte x;
  while( (x = ObjectOriginTable[ i ]) > MaxWordsPerSmallObject )
    i -= x - (MaxWordsPerSmallObject-MaxCardsPerSmallObject);
  return card_start( i ) - ( x << LogBytesPerWord );
}
```

- Например, пусть размер карты 64 слова
- Будем использовать значения [0:128] для обозначения нулевого смещения в текущей и всех смещений в 2 предыдущих картах
- Большие беззнаковые значения байта *x* используем для скачка на **x-128+2** предыдущих карт
- Объекты длиной до **129** слов адресуются прямо, а большие требуют **(object_size/4-129)/(126*64)** скачков по таблице

Пометка страниц памяти

- Используем вместо карт страницы виртуальной памяти, а вместо программного барьера - аппаратную защиту страниц от записи
  - Для этого аппаратура должна предоставлять, а ОС достаточно эффективно поддерживать защиту страниц виртуальной памяти от записи
- По завершении сборки в старшем поколении очищаем пометку всех его страниц и защищаем их от записи
  - Не защищаем не содержащие ссылок страницы
  - Достаточно одного бита пометки на страницу — этот код исполняется обработчиком прерывания, нет смысла экономить несколько инструкций
- При обработке прерывания защиты такой страницы помечаем ее, снимаем защиту и продолжаем выполнение

Пометка страниц памяти (2)

- Если при просмотре помеченных страниц встречается лишенная ссылок на младшие объекты страница, очищаем ее пометку и защищаем от записи
- Преимущество — полное отсутствие барьера записи в коде приложения
- Недостатки
  - Барьер срабатывает и страница помечается при записи любых данных, а не только ссылок
  - Низкая точность из-за большого размера страницы
  - Запомненное множество содержит очень большое количество посторонних ссылок в посторонних страницах и посторонних объектах
  - Высокие затраты на обработку прерываний, просмотр и фильтрацию запомненных ссылок
    - При небольшом размере младшего поколения во времени сборки мусора доминирует просмотр запомненных ссылок

Биты модификации страниц памяти

- По существу пометка страниц виртуальной памяти, но без защиты страниц и обработки прерываний
- Считаем страницу помеченной, если установлен ее бит модификации (*dirty bit*)
  - Этот бит выставляется аппаратурой для поддержки реализации виртуальной памяти
  - ОС должна предоставлять приложению возможности сбрасывать и просматривать биты модификации его страниц
  - Большинство современных ОС не предоставляет приложению таких возможностей
- По сравнению с пометкой страниц, исчезают накладные расходы на установку и снятие защиты страниц и обработку прерываний
- Все прочие преимущества и недостатки остаются

Пометка заголовков объектов

- Признак пометки ассоциируется не с полем, а с содержащим его объектом
  - Относится ко всем ссылочным полям этого объекта
  - Не может быть точным, за исключением объектов с единственным ссылочным полем
- Представляется битом в заголовке объекта старшего поколения
  - Тот же бит в заголовке объекта младшего поколения может иметь другой смысл во время сборки мусора

```
inline void remember( void* dst ) {
  mark( dst );
}

inline void set( void* dst, unsigned offset, void* src ) {
  if( is_old( dst ) && is_young( src ) ) {
    remember( dst );
  }
  set_field( dst, offset, src );
}
```

Упрощенная пометка заголовков объектов

```
inline void set_young( void* dst, const unsigned offset, void* src ) {
  set_field( dst, offset, src );
}

inline void set_null( void* dst, const unsigned offset ) {
  set_field( dst, offset, NULL );
}

inline void set( void* dst, const unsigned offset, void* src ) {
  if( is_old( dst ) ) {
    remember( dst );
  }
  set_field( dst, offset, src );
}

inline void set(void* dst, const unsigned offset, void* src) {
  if( dst < src ) {
    remember( dst );
  }
  set_field( dst, offset, src );
}

inline void set(void* dst, const unsigned offset, void* src) {
  remember( dst );
  set_field( dst, offset, src );
}
```

Итерация по ссылкам помеченных объектов

- Признак пометки как способ ускорить просмотр всех ссылочных полей в старшем поколении
  - Поскольку борьер неточный, во избежание накопления погрешностей полезно во время младшей сборки при первом просмотре запомненного множества снять пометку с объектов, не содержащих ссылок на младшие объекты
  - Вычисляется размер каждого старшего объекта

```
void forEachRememberedRef ( void ref_do(Byte** ref) ) {
  const Byte* top = OldGenerationEnd;
  for( Byte* p = HeapBottom; p < top; p += allocation_size( p ) ) {
    if( is_marked( p ) ) {
      bool young_ref_found = false;  // Только при первом просмотре
      forEachRef( ref, p ) {
        if( is_young( *ref ) ) {
          ref_do( ref );
          young_ref_found = true;    //
        }
      }
      if( !young_ref_found ) {       //
        unmark( p );                 //
      }                              //
    }
  }
}
```

Связный список объектов

- Зарезервируем в каждом объекте поле для связывания в список
  - Старшие объекты связываются в запомненное множество. В младших объектах это поле может использоваться иначе
  - Поколения не обязаны располагаться в разных областях адресного пространства «кучи»
- При присваивании ссылки на младший объект полю старшего объекта добавим старший объект в список
  - Перед добавлением в связный список необходимо убедиться, что объект в нем отсутствует
  - Для проверки присутствия будем проверять поле связи и использовать в качестве терминатора списка специальное ненулевое значение *Undef*
  - Барьер не может быть точным, за исключением объектов с единственным ссылочным полем

Связный список объектов (2)

```
#define UndefLink ((Object*)-1)

inline void initialize_remembered_list ( void ) {
  RememberedList = UndefLink;
}

inline void remember ( Object* p ) {
  if( p->next == NULL ) {
    p->next = RememberedList;
    RememberedList = p;
  }
}

inline void set( Object* dst, unsigned offset, Object* src ) {
  if( is_old( dst ) && is_young( src ) ) {
    remember( dst );
  }
  set_field( dst, offset, src );
}
```

- Упрощенные барьеры аналогичны применяемым при пометке заголовков объектов

Итерация по ссылкам запомненного списка объектов

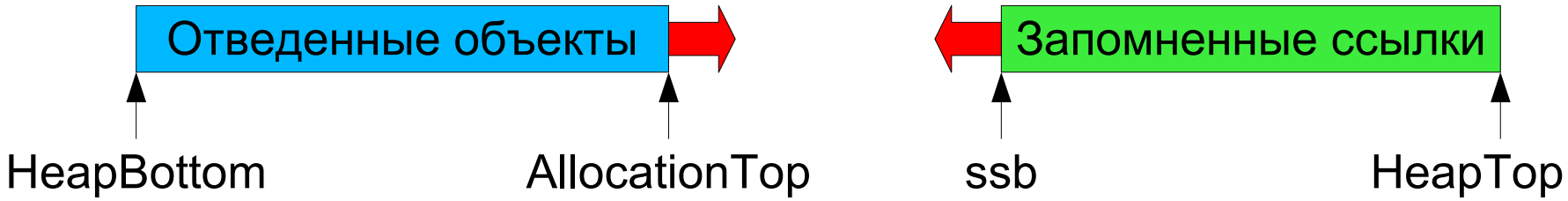
- Поскольку барьер неточный, во избежание накопления погрешностей при первом просмотре удаляем объекты, не содержащие ссылок в младшее поколение

```
void forEachRememberedRef ( void ref_do(Byte** ref) ) {
  Object* list = UndefRef;          // Только при первом просмотре
  for( Object* p = RememberedList; p != UndefRef; /* p = p->next */) {
    bool young_ref_found = false; //
    forEachRef( ref, p ) {
      if( is_young( *ref ) ) {
        ref_do( ref );
        young_ref_found = true;     //
      }
    }
    Object* next = p->next;         //
    if( young_ref_found ) {         //
      p->next = list; list = p;     //
    } else {                        //
      p->next = NULL;               //
    }                               //
    p = next;                       //
  }
  RememberedList = list;            //
}
```

Буфер последовательной записи (Sequential Store Buffer, SSB)

- Буфер последовательной записи
  - Одномерный массив с индексом текущей позиции
  - Первоначально индекс установлен в крайнюю позицию
    - Начало массива, если запись производится в направлении роста адресов
    - Конец массива, если запись производится в направлении убывания адресов
  - При записи элемента индекс перемещается на следующую позицию в соответствующем направлении
- Заводим глобальный буфер последовательной записи
  - Буфер может располагаться в пространстве кучи и записывать элементы в противоположном отведению памяти направлении
  - При многопоточной реализации во избежание взаимных блокировок у каждого потока свой буфер

Буфер последовательной записи (1)



- Обнаружение переполнения буфера
  - Программное (добавляет сравнение и условный вызов к коду барьера)
  - Или с использованием защиты страниц
- Обработка переполнения
  - Сборка мусора сначала в младшем поколении, потом при необходимости и в старшем (после полной сборки все запомненные ссылки забываются)
  - Уплотнение буфера с удалением посторонних ссылок
  - Сохранение данных во вторичном непереполняемом запомненном множестве с фильтрацией посторонних ссылок и дубликатов

Буфер последовательной записи (2)

```
inline void remember ( void** p ) {
  *--ssb = p;
  if( ssb == AllocationTop ) {    // Только если переполнение
    handle_ssb_overflow();        // обнаруживается программно
  }                               //
}
```

- В простейшем варианте 3-4 машинные инструкции
  - При использовании специализированных инструкций и глобальном отведении указателя *ssb* в специализированном регистре — 1 инструкция
- Высокая локальность при обработке
- Не требует синхронизации при многопоточности
- Но содержит дубликаты и быстро переполняется

Гибридная реализация запоминания ссылок

- Упрощение барьера сопровождается ростом числа посторонних ссылок
- Буфер последовательной записи очень быстр и не требует синхронизации, но переполняется
- Применим гибридную реализацию!
  - При записи в ссылочное поле без каких-либо проверок запомним его адрес в буфере последовательной записи
  - По возможности избегаем запоминания записей в поля объектов младшего поколения и значения *NULL*
  - Для обнаружения переполнения защищаем страницу
  - При переполнении переписываем, фильтруя посторонние ссылки, в битовую карту
  - При многопоточной реализации заполненный буфер ставится в очередь к переписывающему потоку, текущему потоку выделяется новый буфер, выполнение программы продолжается

Частичное сканирование программных стеков

- При большой глубине или количестве программных стеков затраты на сканирование всех содержащихся в них указателей могут быть значительными
- Наблюдение: Вызванная процедура не может изменить локальные данные вызывающей, если язык программирования не позволяет передавать локальную переменную параметром по ссылке или присваивать ее адрес
- Младшее поколение иногда бывает пустым
  - После полной сборки
  - После сборки в младшем поколении, если выживших объектов нет или все они продвинуты в старшее поколение

Частичное сканирование программных стеков (2)

- Если в момент, когда младшее поколение пусто, запомнить состояние стека, то при последующих младших сборках достаточно просматривать только изменившуюся его часть
- Как идентифицировать изменившуюся часть?
- *(Динамический) уровень секции активации* — ее номер в стеке, начиная от его дна
- *Глубина программного стека* — динамический уровень секции на его вершине
- Вычислим минимум глубины стека с момента опустошения младшего поколения
- Могли измениться только секции от верхней до вычисленного минимума глубины включительно

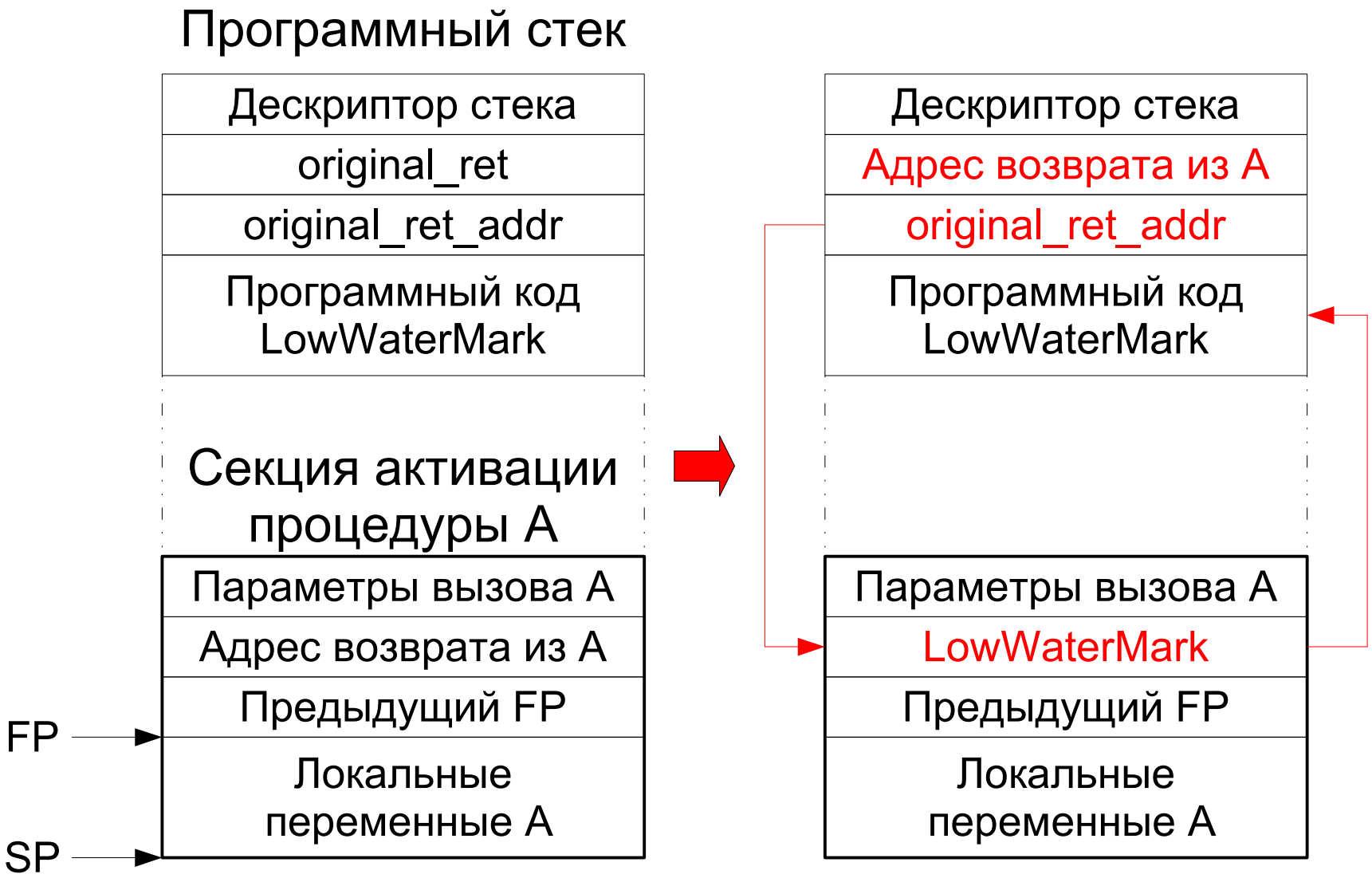
Вычисление минимума глубины программного стека

- Можно динамически отслеживать глубину стека
  - При входе в процедуру в ее коде увеличиваем счетчик глубины стека
  - Перед выходом в процедуре уменьшаем счетчик и вычисляем минимум
  - Как обрабатывать исключения?
  - В C++ можно применить локальный объект с конструктором и деструктором
  - В целом это неэффективно и требует изменений в исходном коде всех процедур
- Можно вычислять максимум адреса текущей секции активации
  - Или минимум, если стек растет вверх
  - Если при организации вызовов используются регистры *FP* и *SP*, этот адрес содержится в *FP*
  - Иначе *SP* + *размер_текущей_секции,* где размер секции определяется во время компиляции
  - Максимум вычисляется при выходе из процедуры

Вычисление минимума глубины программного стека (2)

- Подменим адрес возврата на программную метку минимума глубины стека
  - Сохраним в дескрипторе программного стека адрес подмененного адреса возврата и его исходное значение
  - Программная метка — локальная в стеке процедура, у каждого стека она своя
  - Может быть реализована как вызов одной глобальной процедуры, параметризованный адресом дескриптора программного стека
  - Программная метка переставляет себя на предыдущую секцию активации и передает управление по исходному адресу возврата
  - Адрес подмененного адреса возврата и есть максимум адресов секций активации

Подмена адреса возврата на программную метку



Когда продвигать объекты в следующее поколение?

- Если объекты продвигаются слишком быстро:
  - Они не успевают умереть в младшем поколении
  - Увеличивается частота дорогостоящих сборок в старших поколениях
- Если объекты продвигаются слишком медленно:
  - Увеличивается объем живых объектов, участвующих в частых сборках мусора в младшем поколении
  - Увеличивается длительность и снижается эффективность младших сборок
  - При слишком низкой частоте сборок в старшем поколении повышается доля запомненных ссылок, принадлежащих уже мертвым объектам старшего поколения
  - Молодые объекты по этим ссылкам не могут быть освобождены без сборки мусора в старшем поколении
- Критерии продвижения
  - Число пережитых объектом сборок в младшем поколении
  - Доля переживших сборку объектов в объеме младшего поколения
  - При малом объеме поколения доля высока, но нужно не ускоренно продвигать объекты, а увеличивать объем!

Какие объекты продвигать в следующее поколение?

- Продвижение может быть *массовым* или *выборочным*
- При массовом продвижении *все* пережившую сборку мусора в младшем поколении объекты либо остаются в том же поколении, либо продвигаются в старшее
  - Массовое продвижение не учитывает разницу в возрасте объектов в пределах поколения
- При выборочном продвижении объекты могут продвигаться в старшее поколение индивидуально
  - Для определения возраста объекты внутри поколения разбиваются на возрастные группы по числу пережитых сборок
  - Возрастная группа знает число пережитых ею сборок и способна перечислить входящие в нее объекты
  - Выборочно продвигаемые объекты должны быть просмотрены, чтобы запомнить содержащиеся в них ссылки на оставшиеся в предыдущем поколении объекты

Stop-the-World vs. Concurrent GC

- Collector — сборщик мусора
- Mutator — приложение
- Collector и Mutator — последовательные взаимодействующие процессы, модифицирующие совместно используемые данные
- Доступ к данным должен быть синхронизирован!
  - Collector перемещает ссылки и объекты. Mutator сломается, если обратится по некорректной ссылке
  - Mutator меняет конфигурацию графа ссылок между объектами. Если в процессе сканирования Collector упустит ссылку на объект, этот объект будет уничтожен
- Collector должен успевать освобождать память не медленнее, чем ее потребляет Mutator

Трехцветный инвариант

- В каждый момент сборки каждый объект окрашен в один из 3 цветов
- В начале сборки все объекты белые. Затем доступные из корней объекты красятся в серый. В конце сборки остаются только черные и белые. Белые не доступны и подлежат освобождению.
- Серые объекты образуют фронт сканирования
- Для корректности работы сборщика необходимо исключить прямые ссылки из полей черных объектов на белые
  - Достаточно сохранить хотя бы одну проходящую через фронт сканирования ссылку
- Альтернатива — красить объекты в белый и черный, а *поля* черных в черный и серый

Способы восстановления инварианта

- Не давать Mutator'у доступа к белым объектам
  - Перехватывать чтение ссылок и красить белое в серое или черное
  - Вновь отводимые объекты черные и поэтому не могут быть освобождены в том цикле сборки, во время которого они отведены.
  - Мешает использовать высокую смертность новорожденных объектов
- Не давать Mutator'у записывать белое в черное
  - Перехватывать запись ссылок, и если ссылка на белый объект присваивается полю черного, то:
  - либо перекрашивать присваиваемый объект из белого в серое или черное (Mutator выполняет часть работы Collector'а)
  - либо объект, полю которого присваивают, из черного в серое (дополнительная работа для Collector'а)
  - В последнем случае предпочтительна серая окраска полей, а не объектов
  - По окончании фазы пометки корни могут содержать ссылки на белые объекты

Способы восстановления инварианта (2)

- Snapshot-At-The-Beginning (SATB)
  - Собираем только те объекты, которые были недоступны в момент начала текущего цикла сборки
  - Отводим новые объекты черными
  - Перехватываем присваивания, перекрашиваем *старое* значение из белого в серое
  - Легче оценить сверху объем работы сборщика
    - В начале сборки известен объем собираемой памяти
    - Не нужно учитывать новые объекты
  - Меньше синхронизаций между Collector'ом и Mutator'ом
    - Mutator не обращается к старому значению или обращается к нему реже, чем к новому
  - Но больше объем *плавающего мусора* (недоступных объектов, не освобожденных в конце текущего цикла сборки)

Представление окраски объектов

- Разделение по адресам памяти
  - Копирование объектов и перемещение границ цветов
  - Годится только для объектов, но не для их полей
- Биты в заголовке объекта
  - Только для объектов
  - Можно сочетать биты в заголовке с битами в полях
- Списки
  - Только для объектов
- Битовые карты и последовательные буфера
  - Для объектов и полей
  - Не требуют битов в заголовке объекта
- Не каждому алгоритму сборки подойдут все из этих представлений!

Эффективность инкрементальной сборки

- Эффективность сборки мусора — отношение количества освобожденной памяти к произведенной для этого работе
- Инкрементальный сборщик производит больше работы из-за синхронизации
- … и освобождает меньше памяти из-за плавающего мусора
  - Поддержание инварианта слишком консервативно
- Для любого алгоритма сборки мусора его инкрементальная версия менее эффективна, чем остановительная
- … Зато паузы в среднем короче

Одно- или многопоточная сборка

- Однопоточная сборка
  - Гораздо проще реализовать
  - Не использует преимуществ многопроцессорности и многоядерности
- Многопоточная сборка
  - Требует синхронизации
  - При неумелой синхронизации многопоточная сборка медленнее однопоточной
- Work Stealing
  - Популярный способ динамического распределения асимметричной нагрузки
  - Каждый поток снабжается очередью работ. Если очередь опустошается, поток заимствует работу у соседей.
  - Нужно минимизировать синхронизации при операциях с очередью и при выполнении работ

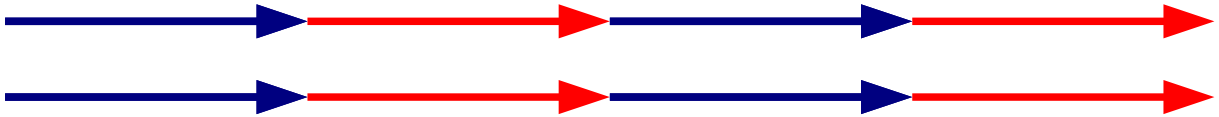
Idempotent Work Stealing

- Idempotent Work - работа, результат которой не меняется от её повторения
  - Такова природа операции — например, установка бита в единицу
  - … или работа содержит проверку, что она уже выполнена
- Для таких работ имеется более эффективная реализация очереди, требующая меньше синхронизаций
  - Maged Micheal, Martin Vechev, Vjay Saraswat. Idempotent work stealing. *PpoPP'09*, North Carolina.

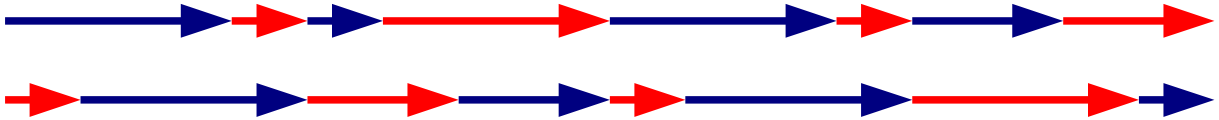
Ordinary GC

Concurrent GC

Parallel GC



Parallel Concurrent GC

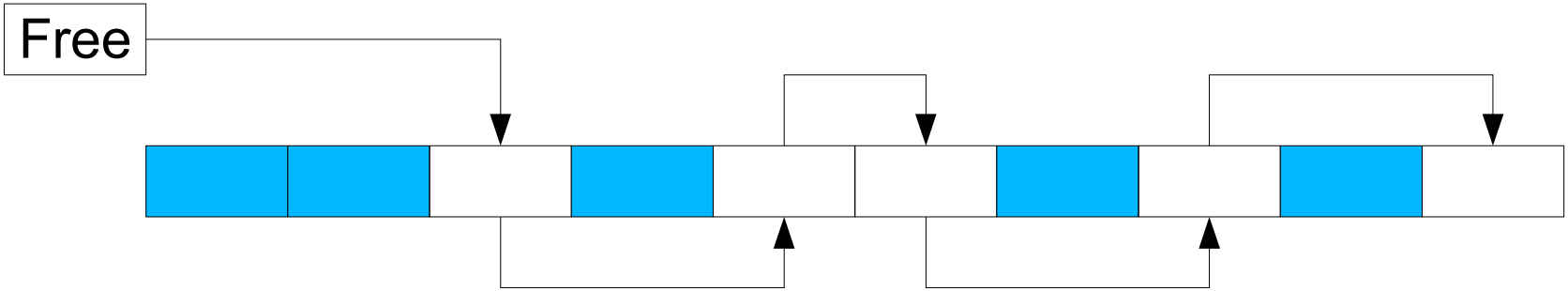


- Ускорение за счет лучшего использования аппаратных ресурсов, особенно когда параллельные возможности аппаратуры избыточны по отношению к потребностям программы

Основные классы алгоритмов сборки мусора

- Автоматическое освобождение памяти
  - Automatic Memory Reclamation
- Счётчики ссылок
  - Reference Counting Collection
- Пометка с последующим освобождением
  - Mark'n'Sweep
- Пометка с последующим уплотнением
  - Mark'n'Compact
- Копирование
  - Copying Collection

Пометка с последующим освобождением



Пометка с последующим освобождением

- Отводить объекты в каком-либо пуле
- Начиная с корней, обойти и пометить все доступные объекты
- Пройти по пространству кучи, освободить непомеченные объекты, снять пометку с помеченных
  - В отличие от Automatic Memory Reclamation, отведение и освобождение интегрированы в одну библиотеку
  - *Одновременное освобождение* недоступных объектов *в известном порядке*

Пометка с последующим освобождением 2

- Длительность пауз
  - Пропорциональна числу указателей в системе
  - Число указателей обычно хорошо аппроксимируется суммарным размером всех живых объектов
  - Верхняя граница — размер кучи
- Рекурсивный обход в фазе пометки
  - Возможно переполнение стека
- Неопределенность момента уничтожения объекта
  - После утраты последней ссылки уничтожение объекта откладывается на неопределенный срок
  - Если язык поддерживает выполнение действий при уничтожении объекта (деструкторы, финализаторы и т.п.), нужно учитывать эту особенность при их применении

Где хранить флажок пометки

- В заголовке объекта
  - У объекта должен быть заголовок, а в нем — свободный бит
  - Не годится для консервативной сборки — надо быть уверенным, что перед нами действительно объект
  - В последней фазе для очистки флага доступного объекта приходится писать в его память. Если эта память виртуальная, может потребоваться подкачка.
  - Но флажки доступных объектов можно не сбрасывать, если циклически менять их смысл!
- В отдельно лежащей битовой карте
  - В кусочной куче у каждого куска карта своя
  - Более сложная адресная арифметика
  - В последней фазе можно идти не по объектам, а по выставленным в карте единицам
  - Если отведение происходит в куче с Bitmapped Fit'ом, сброс флагов производится в начале GC

Борьба с переполнением стека

- Использовать отдельный стек
  - Исключаются накладные расходы на организацию рекурсивных вызовов (рамка стека вызовов помимо аргументов содержит локалы, адрес возврата и динамическую цепочку)
  - Облегчаются проверка и обработка переполнения
- Ранняя пометка
  - Исключить дублирование указателей в стеке
- Итеративная пометка односвязных списков
- Инкрементальная пометка массивов ссылок
  - По существу замена хвостовой рекурсии итерацией
- Восстановление после переполнения стека

Поздняя пометка

```
void markObject( Object* obj ) {
  push(obj);
  do {
    Object* p = pop();
    if( p && !is_marked(p) ) {
      mark(p);
      forEachRef(son,p) {
        push(son);
      }
    }
  } while( markingStackNotEmpty() );
}
```

Стек содержит указатели на доступные объекты вне зависимости от их маркировки. Стек может содержать множественные копии одного указателя. Пример: маркировка массива длиной N, заполненного одним указателем, приводит к помещению в стек N идентичных указателей.

Ранняя пометка

```
void markObject( Object* obj ) {
  if (obj && !is_marked(obj)) {
    mark(obj); push(obj);
    do {
      Object* p = pop();
      forEachRef(son,p) {
        if (son && !is_marked(son)) {
          mark(son); push(son);
        }
      }
    } while( markingStackNotEmpty() );
  }
}
```

Стек содержит указатели на помеченные объекты, сыновья которых еще не сканированы (*фронт сканирования*). Дублирование указателей исключено.

Итеративная пометка односвязных списков

- Пока из помечаемого объекта исходит не более одной ссылки, не помещать его в стек, а переходить по ссылке

```
void mark_and_push( Object* obj ) {
  for(;;) {
    mark(obj);
    const int n = numberOfSons(obj);
    if (n == 1 ) {
      Object* son = firstSon(obj);
      if (son && !is_marked(son)) {
        obj = son;
        continue;
      }
    } else if (n > 1) {
      push(obj);
    }
    break;
  }
}
```

Итеративная пометка односвязных списков (2)

```
void markObject( Object* obj ) {
  if (obj && !is_marked(obj)) {
    mark_and_push(obj);
    while( markingStackNotEmpty() ) {
      Object* p = pop();
      forEachRef(son,p) {
        if (son && !is_marked(son)) {
          mark_and_push(son);
        }
      }
    }
  }
}
```

- Вариант — в **`numberOfSons`** учитывать исходящие ссылки только на непомеченные объекты, причем нам интересно лишь, ссылок 0, 1 или много.

Инкрементальная пометка больших массивов ссылок

- При обработке массива ссылок необходимо поместить в стек все ранее не помеченные его элементы
- Для больших массивов размер стека может оказаться недостаточным
- Можно помечать массивы постепенно, за один шаг не более чем известное число элементов
  - Текущий индекс можно запомнить в отдельном стеке
  - Можно текущий индекс не запоминать ценой повторного просмотра начала массива

Восстановление после переполнения стека

- При возникновении переполнения:
  - Выставляем флаг, что оно произошло
  - Игнорируем не поместившиеся в стек элементы
  - Продолжаем маркировку
- По окончании фазы пометки:
  - Проверяем, не случилось ли переполнения
  - Если случилось, сбрасываем его флаг, последовательно просматриваем кучу и продолжаем обычный процесс маркировки, начиная с ранее помеченных объектов
  - Если при обработке переполнения возникло переполнение, обработка циклически повторяется
  - Обработка завершится за конечное число циклов, поскольку в каждом из них помечается минимум размер стека + 1 объектов, а их число конечно.

Связный стек пометки

- Если во время фазы пометки в каждом объекте имеется свободное поле-указатель, то можно его использовать для организации связного стека
- Вместо помещения в стек ранее не помеченный объект прописывается в начало списка
- Переполнение такого стека невозможно
  - Цена - дополнительный указатель в заголовке
  - Вне фазы пометки этот указатель может использоваться для других целей

Обращение указателей

- Бесстековый обход графа ссылок
  - Для ациклических и произвольных бинарных графов — путем вращения и обращения указателей без введения дополнительных полей
  - Для прочих необходимо поле с номером текущего сына (может использоваться вместо бита пометки)
  - По существу обход лабиринта методом левой или правой руки
- Производит слишком много записей в память
  - Имеет смысл использовать только как резервный алгоритм для обработки переполнения
- Красивый, но медленный алгоритм

Обращение указателей (2)

```
void mark_binary_tree( TreeNode* node ) {
  #define UndefNode ((TreeNode*)-1)
  TreeNode* prev = UndefNode;
  do {
    TreeNode* tmp;
    if( node ) {
      mark( node );
      // Rotate
      tmp = node->Sons[0];
      node->Sons[0] = node->Sons[1];
      node->Sons[1] = prev;
      prev = tmp;
    }
    // Swap
    tmp = node; node = prev; prev = tmp;
  } while( node != UndefNode );
  #undef UndefNode
}
```

Инкрементальная пометка: Алгоритм Юасы

Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3), 1990.

- Барьер чтения
  - Перехватывает все чтения ссылок программой
  - Если объект не помечен, он помечается и помещается в стек пометки
- Обработка стека пометки
  - При обращениях по ссылкам и отведениях памяти
- Не является алгоритмом реального времени
  - Если стек пометки ограниченной глубины, при любом чтении указателя может потребоваться обработка его переполнения
  - Скорость обработки стека пометки определяется переменным эвристическим коэффициентом
  - Если скорость сборки не достаточна, память может быть исчерпана, тогда выполнение программы приостанавливается до завершения сборки

Инкрементальная пометка: Стековый алгоритм Дейкстры

Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, № 46.* Springer-Verlag, New York, 1976.

Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM,* 21(11), November 1978.

- Упрощенный барьер записи
  - Перехватывает все записи ссылок программой
  - Если объект не помечен, он помечается и помещается в стек пометки вне зависимости от цвета объекта, в который записывается ссылка
- Обработка стека пометки
  - При записях ссылок и отведениях памяти

Инкрементальная пометка: Бесстековый алгоритм Дейкстры

- По существу модификация стекового алгоритма с постоянным применением восстановления при переполнении стека путем повторного сканирования кучи
- Для исключения повторного просмотра черных объектов полезно в дополнение к биту пометки ввести бит просмотра
  - Если оба бита установлены, объект черный
  - Если установлен только бит пометки, объект серый и требует просмотра, после чего помечается как просмотренный
  - Сочетание установленного бита просмотра при сброшенном бите пометки не имеет смысла
- Квадратичная сложность
  - После пометки сыновей повторить просмотр кучи

Инкрементальное освобождение непомеченных объектов

- Длительность сканирования всей «кучи» пропорциональна ее размеру
- Для сокращения средней длительности пауз производим сканирование инкрементально при отведении памяти
  - Отведение памяти усложняется, замедляется и становится менее предсказуемым
  - Признаки пометки должны храниться постоянно
  - Затрудняются сбор статистики о степени заполнения памяти и адаптивное изменение размера «кучи»
- В кусочной «куче» куски можно сканировать не только инкрементально, но и параллельно

Параллельная пометка

- Для ускорения сканирования используем несколько параллельных потоков, каждый со своим фронтом сканирования
  - Множество адресов помеченных объектов с непросмотренными сыновьями в стеке маркировки
  - *Очередь работ* — стек для данного потока, расширенный удалением со дна для прочих потоков
- При исчерпании своей очереди поток заимствует работу из очереди другого потока
  - Найти поток с достаточно длинной очередью, извлечь и удалить работу с ее конца
- Доступ к чужой очереди в данном случае не требует синхронизации между потоками
  - Пометка объектов идемпотентна
  - При правильной реализации в худшем случае эти два потока станут сканировать один и тот же объект

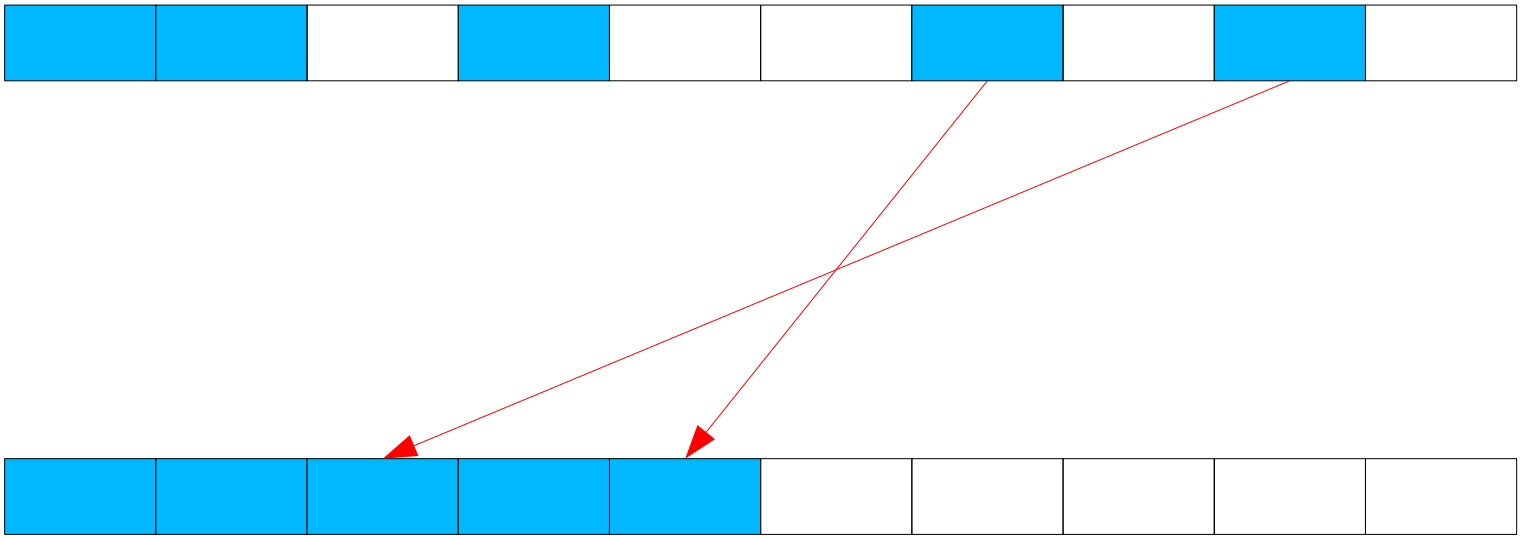
Параллельное освобождение

- На время освобождения разделить кучу на крупные сегменты
  - При блочной организации - блоки
  - При непрерывной организации — выровненные на границы объектов диапазоны адресов (нужно уметь быстро находить начало ближайшего к заданному адресу объекта)
- Сканировать сегменты в несколько потоков
  - Накапливать свободные объекты в локальных для сегмента/потока структурах
- Полученные локальные результаты объединять в глобальные структуры
  - Потери на синхронизацию невелики из-за величины сегментов
  - Для объединения можно использовать дополнительный поток

Практический пример: консервативный сборщик Боэма-Демерса-Уэйзера

- На время освобождения разделить кучу на крупные сегменты
  - При блочной организации - блоки
  - При непрерывной организации — выровненные на границы объектов диапазоны адресов (нужно уметь быстро находить начало ближайшего к заданному адресу объекта)
- Сканировать сегменты в несколько потоков
  - Накапливать свободные объекты в локальных для сегмента/потока структурах
- Полученные локальные результаты объединять в глобальные структуры
  - Потери на синхронизацию невелики из-за величины

Пометка с последующим уплотнением



Пометка с последующим уплотнением

- Отводим объекты в каком-либо пуле
  - Обычно путем перемещения указателя AllocationTop (Pointer Bumping)
- Начиная с корней, обходим и помечаем все доступные объекты
- Перемещаем живые объекты в один конец кучи
- Корректируем указатели, чтобы они указывали на новое место расположения объектов
  - В вариациях алгоритма корректировка указателей может производиться до, после или в той же фазе, что и перемещение объектов
- Не избегаем фрагментации, а устраняем ее!

Перемещение объектов

- Нужна высокая пропускная способность памяти
  - Неэффективно при узкой шине памяти
- Изменяется физический адрес объекта
  - Некоторые контроллеры используют физические адреса — нужна поддержка временной фиксации расположения объекта (object pinning)
- Устраняется внутренняя фрагментация страниц виртуальной памяти
- Возможно снижение энергопотребления за счет выключения свободных банков памяти
- Возможно увеличение локальности ссылок за счет переупорядочения объектов
  - Сохранение порядка отведения, обход графа ссылок, использование динамической статистики

Труднопереместимые объекты

- Долгоживущие объекты
  - В течение жизни многократно перемещаются
  - К счастью, стремятся осесть на дне кучи
- Большие объекты
  - Высокие затраты на пересылку в фазе уплотнения
  - Часто бывают долгоживущими
- Популярные объекты
  - Объекты, на которые часто ссылаются
  - Высокие затраты в фазе корректировки указателей
- Объекты, требующие временной фиксации расположения в физической памяти
  - Операции pin и unpin
  - Часто являются большими
- Эти классы объектов являются *устойчивыми*.
  - Распознавать, отдельно хранить, не перемещать

Способы корректировки указателей

- Запоминание нового адреса на месте старого расположения объекта
  - До или после перемещения объекта
- Табличный поиск
- Прошивка указателей (*Pointer Threading*)

Алгоритмы пометки с последующим уплотнением

- «Двух пальцев» (Эдвардса)
- Таблицы разрывов (Хэддона-Уэйта)
- Связной таблицы разрывов (Фитча-Нормана)
- Запланированного размещения (отсылочных указателей)
- Прошивки указателей (Йонкерса)

Алгоритм Эдвардса («двух пальцев»)

- Для объектов фиксированного размера
  - Segregated Storage
  - Перемещение и корректировка указателей внутри разных блоков могут производиться параллельно
- Фазы алгоритма
  - Пометка живых объектов
  - Уплотнение
  - Корректировка указателей
    - И стирание пометки
- Не требует дополнительных полей
  - Кроме временного хранения бита пометки

Robert A. Saunders. The LISP system for Q-32 compiler. In *The Programming Language LISP: Its Operation and Applications*. Information International, Inc., Cambridge, MA, fourth edition, 1974.

Алгоритм Эдвардса - уплотнение

- Два сходящихся указателя («пальца»)
  - Top движется вниз, начиная с вершины «кучи»
  - Bottom движется вверх, начиная с дна «кучи»
- С помощью Top найти очередной живой объект
- С помощью Bottom найти для него пустое место
- Если указатели встретились, уплотнение завершено
  - Живые объекты расположены в адресном интервале от дна кучи до указателя Bottom
- Скопировать живой объект на пустое место
- Запомнить новый адрес объекта на месте его старого расположения
  - Размер объекта должен быть достаточным для хранения одного указателя

Алгоритм Эдвардса — уплотнение (2)

```
Object* compact(void) {
  Object* bottom = HeapBottom-1;
  Object* top = HeapTop;

  for(;;) {
    do {
      if( --top <= bottom ) break;
    } while( !is_marked(top) );

    do {
      if( ++bottom > top ) return bottom;
    } while( is_marked(bottom) );

    *bottom = *top;
    *(Object**) top = bottom;
  }
}
```

Перемешивает объекты!

Алгоритм Эдвардса — корректировка указателей

- Пройти по корням и ссылочным полям всех выживших объектов
- Если ссылка указывает в область «кучи» над живыми объектами, то заменяем ссылку на там хранящийся адрес нового расположения объекта

```
Object* new_top = compact();
for( Object* obj = HeapBottom; obj < new_top; obj++ ) {
  forEachRef(ref, obj) {
    Object* p = *ref;
    if( p > new_top ) {                    // && p < HeapTop
      *ref = *(Object**)p;
    }
  }
}
```

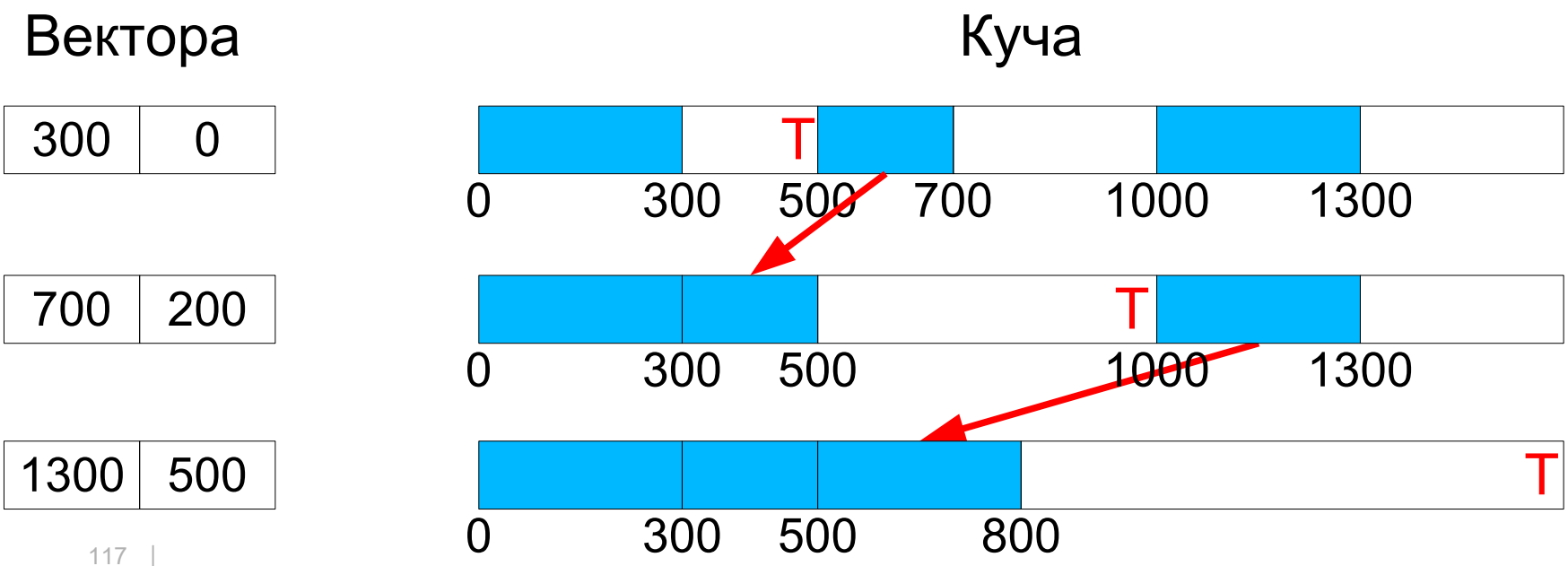
Алгоритм Хэддона-Уэйта (таблицы разрывов)

- Наблюдение: после пометки живые и мертвые объекты образуют чередующиеся интервалы
  - Большой объект сам по себе интервал адресов
  - Элементы составных ссылочных структур обычно отводятся подряд и одновременно умирают
- Уплотняем живые интервалы, запоминая вектор перемещения каждого из них в таблице
  - Таблицу разрывов (*Break Table*) храним на месте мертвых объектов
  - Сохраняется порядок следования объектов
- Фазы алгоритма
  - Пометка живых объектов
  - Уплотнение с построением таблицы разрывов
  - Корректировка ссылок при помощи таблицы

B.K.Haddon and W.M.Waite. A compaction procedure for variable length storage elements. *Computer Journal,* August 1967

Таблица разрывов

- Упорядоченный по возрастанию адресов набор векторов перемещений живых интервалов
  - (Адрес *конца* интервала, Смещение при уплотнении)
  - Таблица размещается в первом свободном интервале
  - Минимальный размер объекта должен быть достаточен для хранения вектора (при наивной реализации 2 слова)
    - Упаковка: заменяем адрес смещением относительно начала кучи, отбрасываем биты выравнивания
    - Двухуровневая таблица с корнем в стеке пометки



Фаза уплотнения алгоритма Хэддона-Уэйта

- Введем вспомогательную функцию для пропуска живого интервала с одновременным стиранием пометки объектов
  - Чтобы избежать проверки граничного условия в коротком цикле, заранее зарезервируем и не станем отводить одно слово на конце кучи

```
Byte* skip_live( Byte* p ) {
  for(; is_marked( p ); p += allocation_size( p ) ) {
    unmark( p );
  }
  return p;
}
```

Фаза уплотнения алгоритма Хэддона-Уэйта (2)

- Живой интервал на дне кучи образует неподвижный осадок (*Solid Prefix*), состоящий из долгоживущих объектов
- Запомним его верхнюю границу и исключим из таблицы разрывов
  - Чтобы зря не вычитать нулевое смещение при корректировке указателей
  - Страница виртуальной памяти при этом помечается как измененная и потребует записи при вытеснении
  - Лишняя синхронизация при параллельной корректировке указателей

```
inline Byte* skip_solid_prefix( void ) {
  return skip_live( HeapBottom );
}
```

Фаза уплотнения алгоритма Хэддона-Уэйта (3)

- Введем вспомогательную функцию для пропуска мёртвого интервала
  - Чтобы избежать проверки граничного условия в коротком цикле, временно пометим текущий указатель отведения памяти AllocationTop
    - Он всегда указывает внутрь кучи, поскольку мы зарезервировали одно слово в ее конце
  - Если биты пометки хранятся в отдельной битовой шкале, сканируем не объекты, а шкалу в поиске первого единичного бита

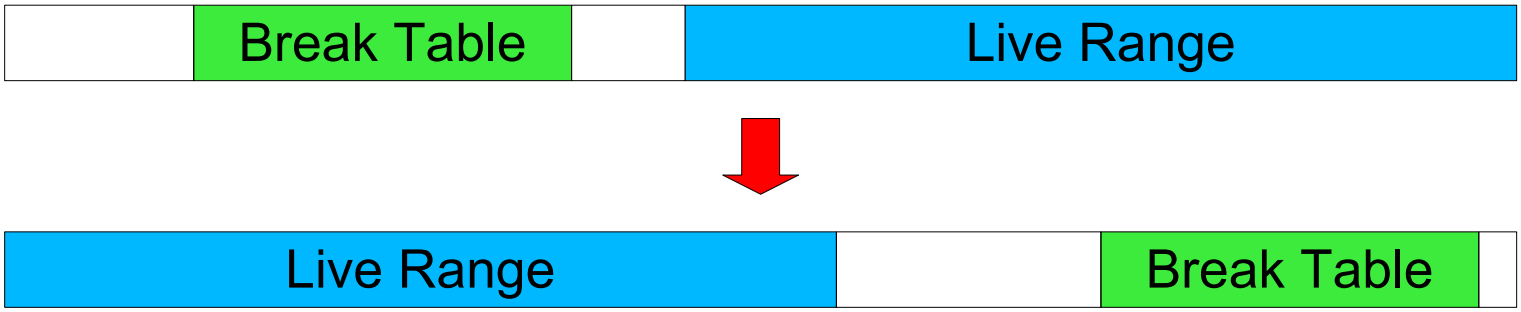
```
inline Byte* skip_dead( Byte* p ) {
  mark( AllocationTop );
  for(; !is_marked( p ); p += allocation_size( p ) );
  unmark( AllocationTop );
  return p;
}
```

Фаза уплотнения алгоритма Хэддона-Уэйта (4)

```
inline Byte* compact( Byte* dst, const Byte* top ) {
  for( Byte* src = dst; (src = skip_dead(src)) < top; ) {
    Byte* end = skip_live( src );
    if( relocation_table == NULL ) {
      // При первой итерации цикла таблица пуста
      // Этот случай следовало бы обработать вне цикла
      set_relocation_table_end( end );
    }
    const unsigned size = end - src;
    relocate( dst, src, size );
    append_relocation_record( end, src - dst );
    src = end; dst += size;
  }

  return dst;
}
```

Эвакуация таблицы разрывов

- Очередной живой интервал перемещается в начало текущего пустого интервала
- В этом же интервале расположена таблица
- Если свободного места под таблицей недостаточно для размещения живого интервала, таблица требует эвакуации
- Обычно для освобождения места достаточно переместить таблицу вверх под живой интервал
- Эвакуация не всегда сводится к простому копированию таблицы

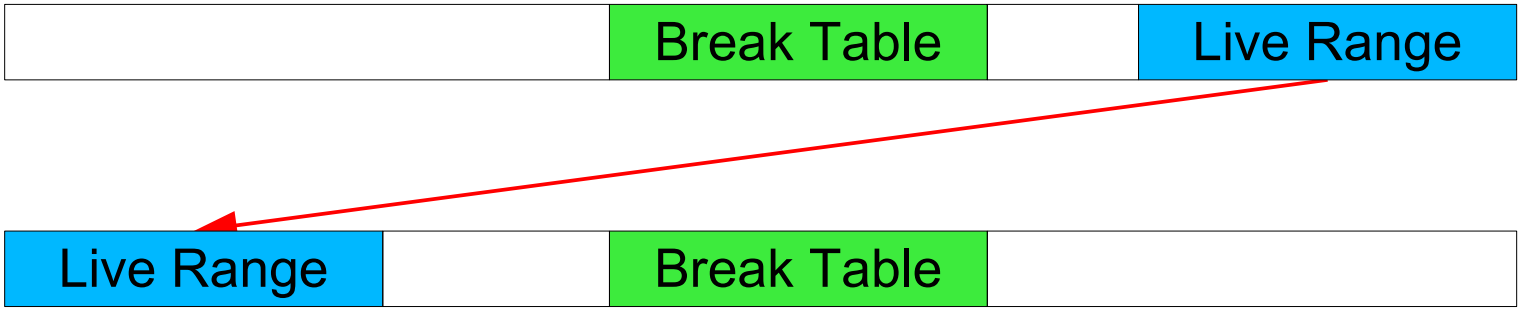


- Размер таблицы и число ее эвакуаций пропорциональны размеру кучи — квадратичная сложность фазы уплотнения

Эвакуация таблицы разрывов. Случаи (1) и (2)

1) Специальный случай — таблица нулевой длины
   - Встречается только в самом начале фазы
   - Пустая таблица не содержит данных и может быть перемещена куда угодно простой установкой указателя

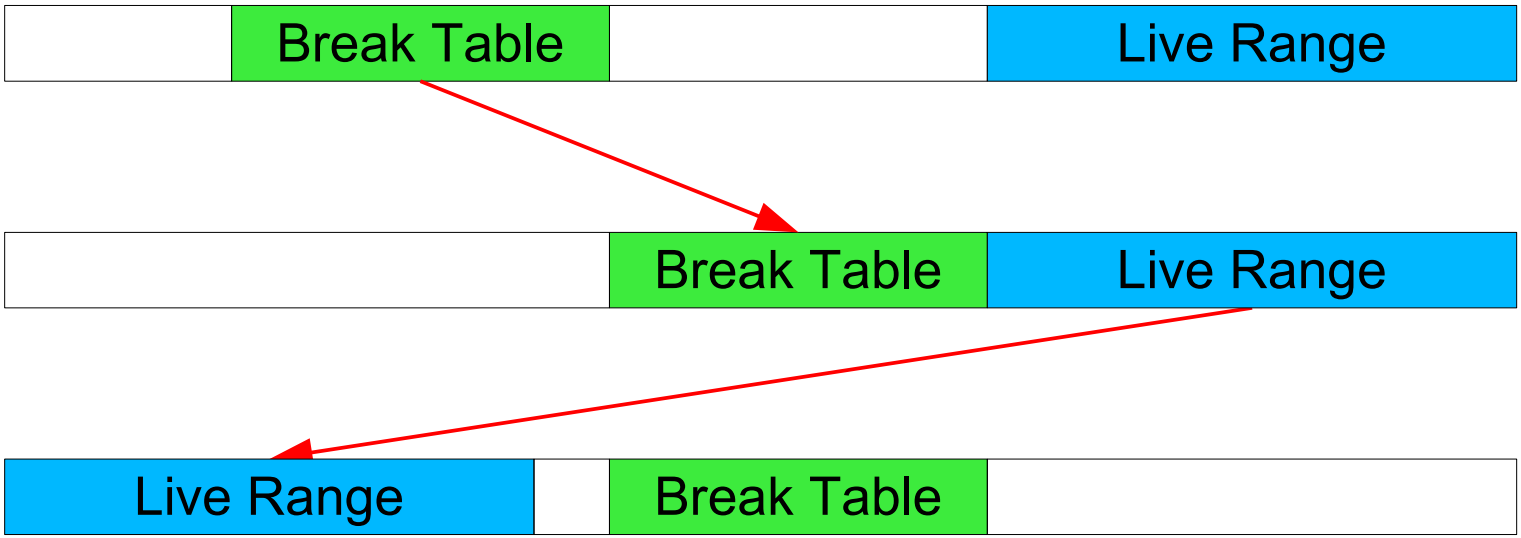
2) Места под таблицей достаточно для размещения живого интервала
   - Пересылаем туда живой интервал
   - Таблица остается на месте
   - Стоимость минимальна — 1 присваивание на слово перемещаемого живого интервала



Эвакуация таблицы разрывов. Случай (3)

3) Суммы места под и над таблицей достаточно для размещения живого интервала

- Перемещаем таблицу под свободный интервал
- Далее случай сводится к предыдущему — пересылаем вниз свободный интервал
- Стоимость — по 1 присваиванию на каждое слово живого интервала и таблицы



Эвакуация таблицы разрывов. Случаи (4) и (5)

4) Таблица длиной в несколько слов

- Может встретиться в начале фазы
- Сохраняем таблицу в локальных переменных
- Как в предыдущем случае, пересылаем вниз свободный интервал
- Устанавливаем указатель на таблицу, восстанавливаем ее содержимое из локалов

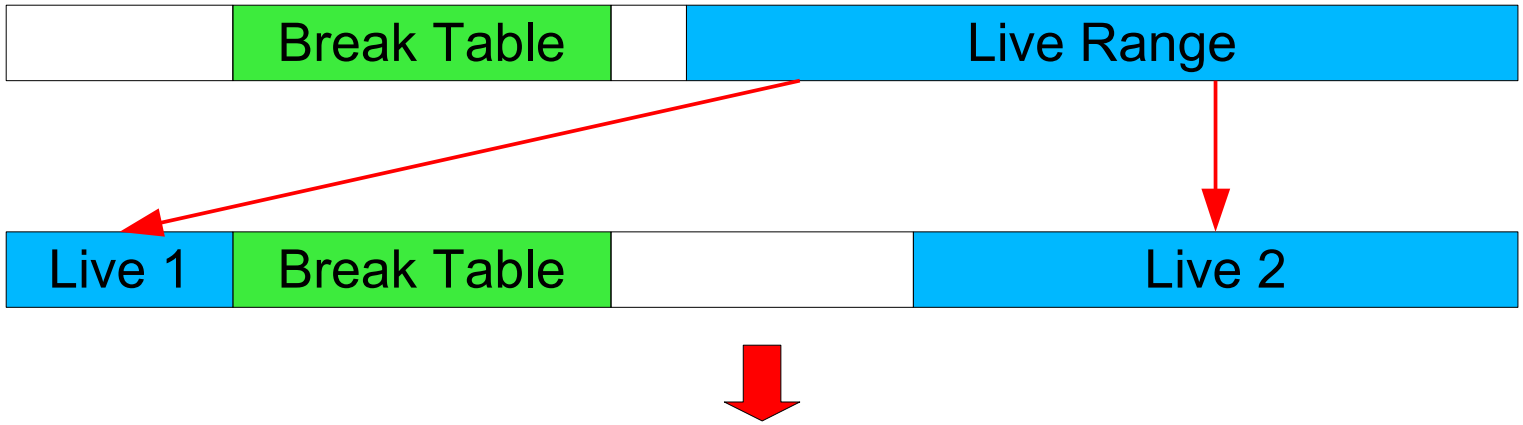
5) Живой интервал длиной в несколько слов

- Сохраняем живой интервал в локалах
- Пересылаем таблицу
- Восстанавливаем живой интервал

Эвакуация таблицы разрывов. Общий случай

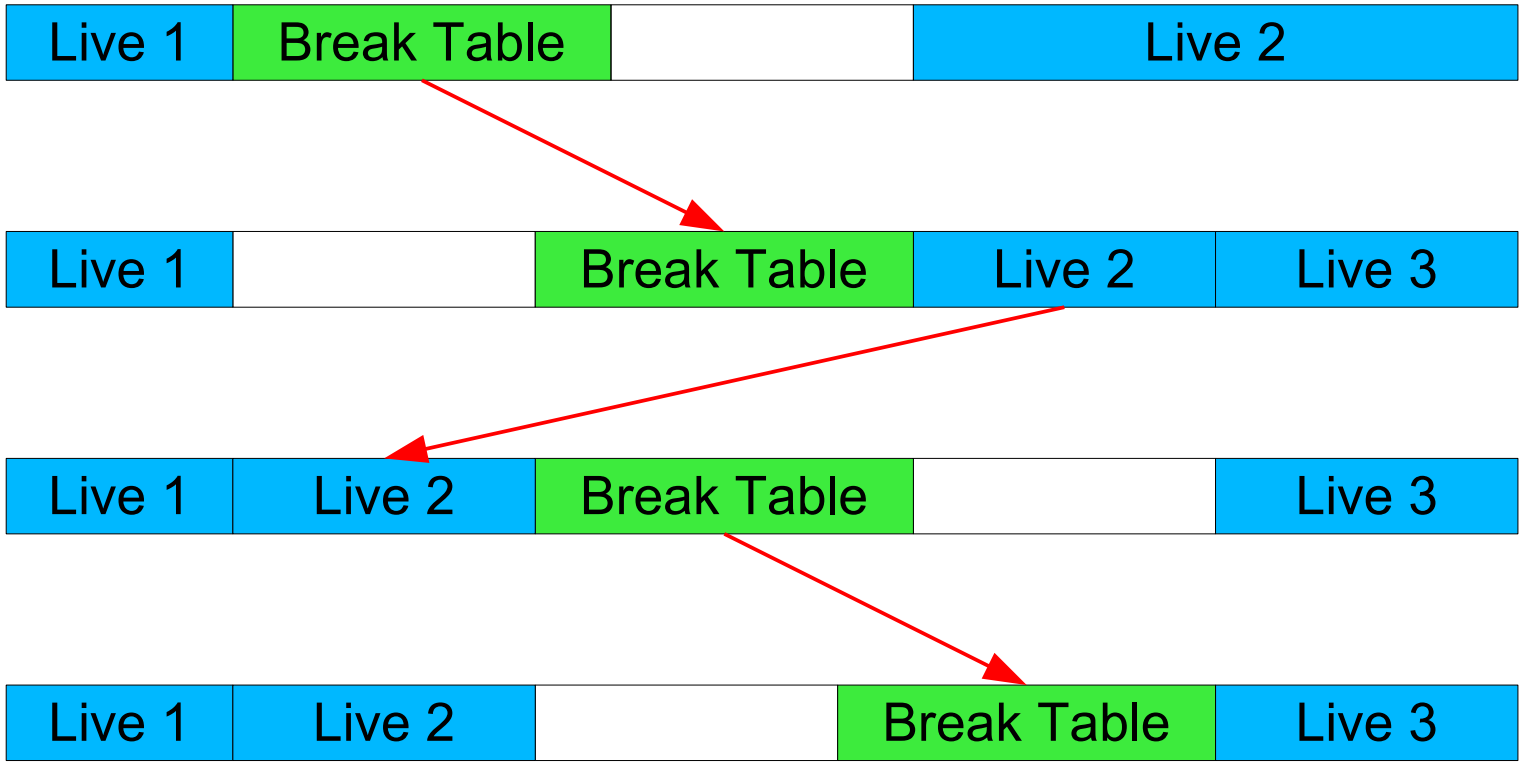
6) Подготовительная фаза

- Если под таблицей имеется пустое место, отделяем от живого интервала головную часть соотвествующего размера и пересылаем ее на это место
- Проверяем, не свелась ли задача к случаю малого интервала
- Иначе выбираем по минимуму стоимости один из вариантов обработки общего случая



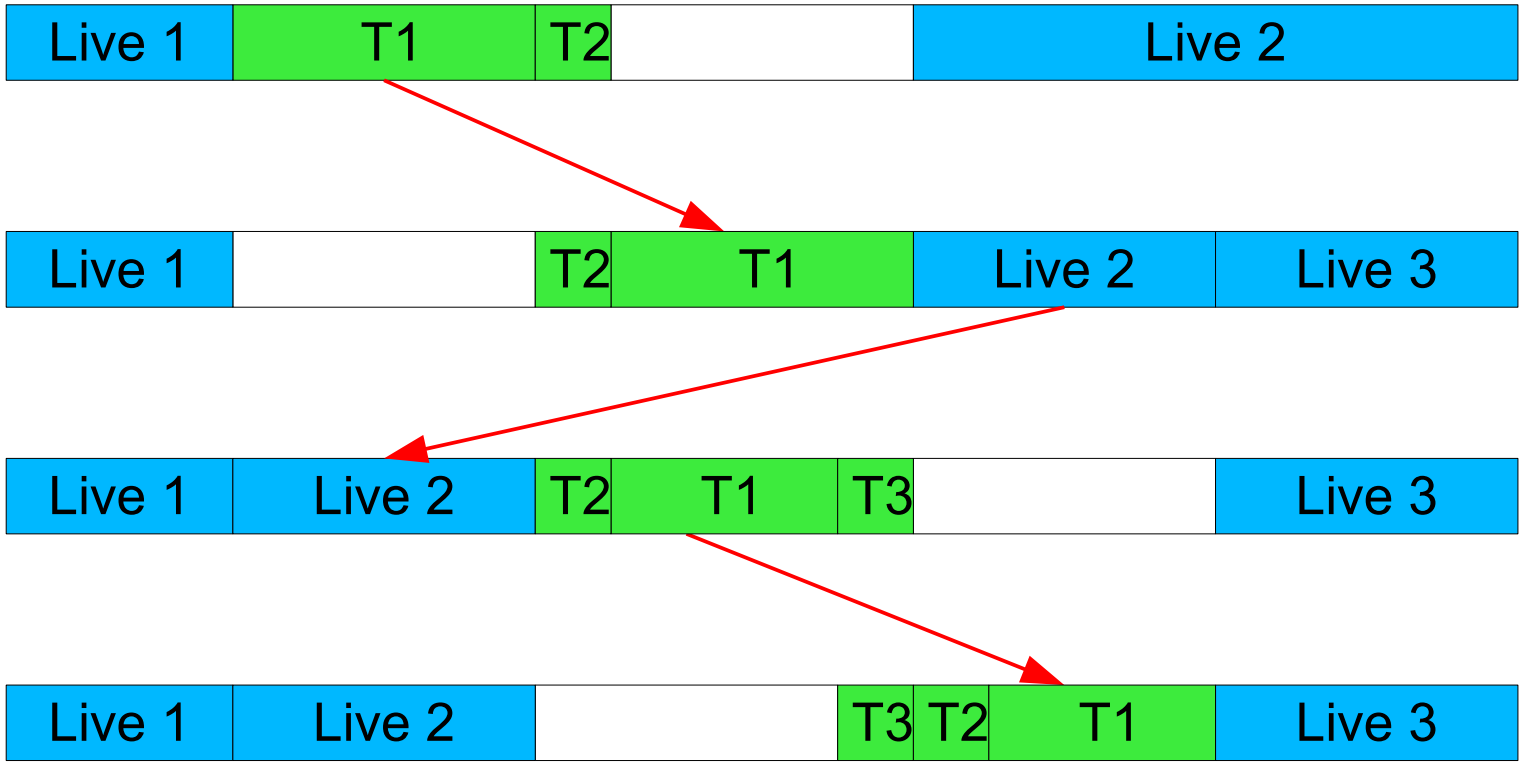
Эвакуация таблицы разрывов. Общий случай

- Многократная пересылка таблицы
  - В цикле пересылаем таблицу под интервал, отделяем от интервала головную часть, пересылаем ее под таблицу
  - Число итераций N = round_up(live_range_size / gap_size)
  - Стоимость: live_range_size + N • break_table_size



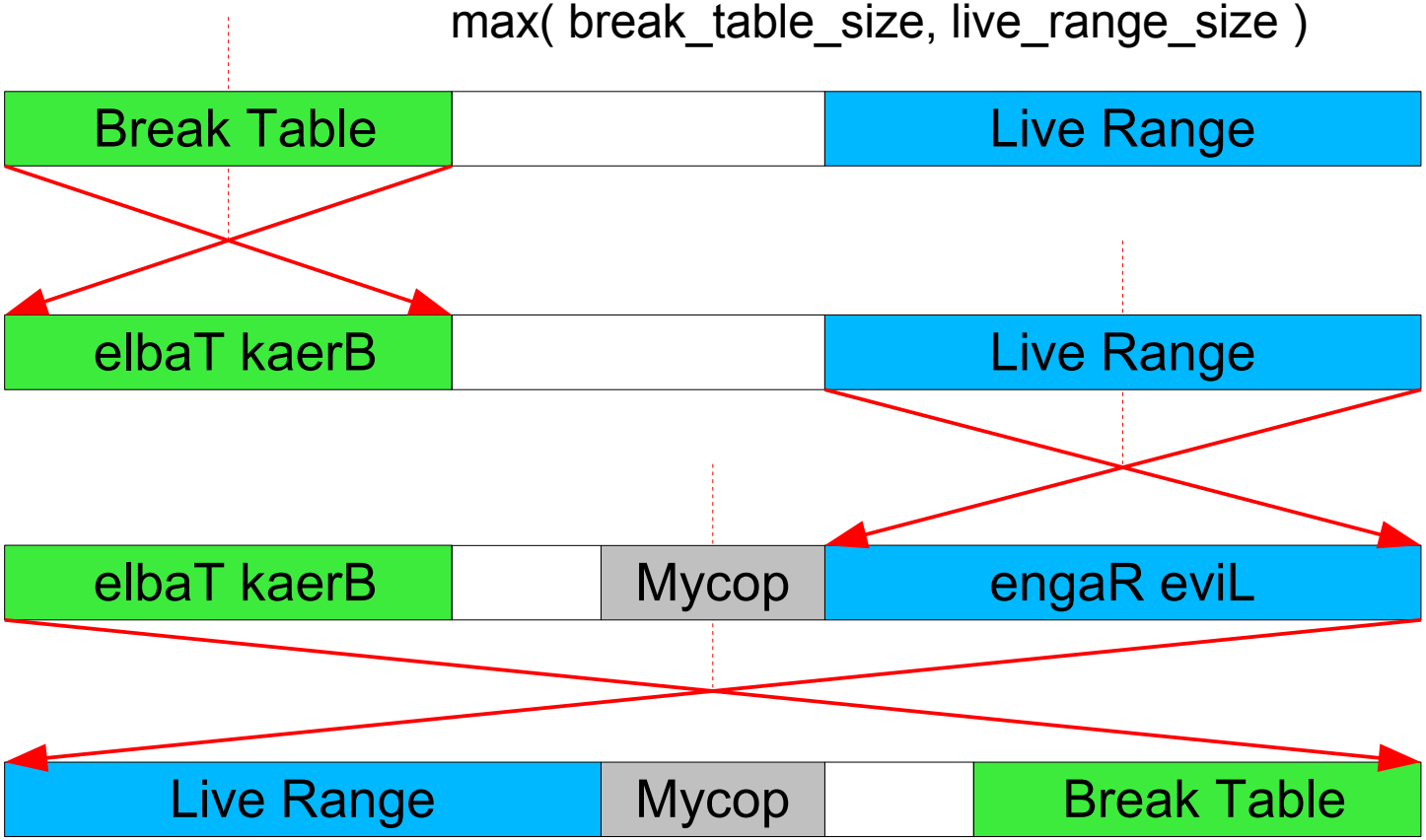
Эвакуация таблицы разрывов. Общий случай

- Альтернатива: перекатывание таблицы
  - В отличие от предыдущего варианта, в цикле пересылаем не всю таблицу целиком, а только е головную часть размером в пустое пространство
  - Элементы таблицы перемешиваются!
  - По окончании фазы уплотнения сортируем таблицу для восстановления порядка



Эвакуация таблицы разрывов. Общий случай

- Альтернатива: три переворота на месте
  - Переворачиваем таблицу, живой интервал и отрезок от начала таблицы до конца интервала
  - При последнем перевороте исключаем симметричный центр, целиком состоящий из мусора
  - Стоимость: break_table_size + live_range_size + max(break_table_size, live_range_size)



Корректировка ссылок при помощи непрерывной таблицы разрывов

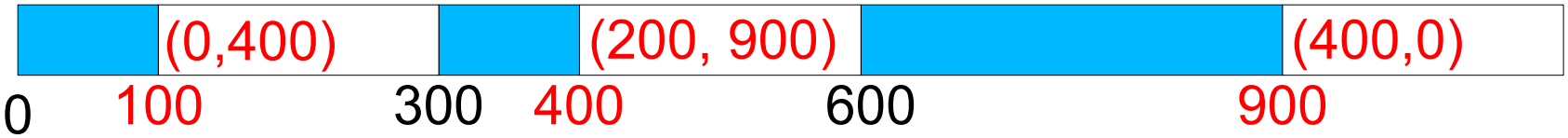
- Если адрес указывает в перемещаемую часть пространства кучи, найти в таблице вектор с минимальной базой, превосходящей адрес, и вычесть из адреса смещение
- Вектора упорядочены по возрастанию баз, можно использовать двоичный поиск
  - Логарифмическая сложность
  - Для увеличения локальности полезно пребразовать таблицу в массив пределов и массив смещений
- Одинаково хорошо корректируются ссылки на начало и внутрь объектов
- Легко распараллеливается
  - Разделяем пережившие сборку объекты на независимо обрабатываемые сегменты

Корректировка ссылок

```
inline void collect( void ) {
  mark_objects();
  initialize_relocation_table();

  Byte* moving_bottom = skip_solid_prefix();
  MovingBottom = moving_bottom;

  Byte* top = compact( moving_bottom, AllocationTop );
  AllocationTop = top;

  if( moving_bottom != top ) {
    for( Byte* p = HeapBottom; p < top; p += allocation_size(p)){
      forEachRef( ref, p ) {
        update_pointer( ref );
      }
    }
    forEachRoot( ref ) {
      update_pointer( ref );
    }
  }
}
```

Алгоритм Фитча-Нормана (связной таблицы разрывов)

- Вариация алгоритма таблицы разрывов, в которой векторы перемещений хранятся раздельно в начале свободных интервалов и связаны в список
  - Адрес вектора равен его базе
  - (Смещение, Адрес следующего вектора)



John P. Fitch and Arthur C. Norman. Note on compacting collection. *Computer Journal,* 21(1), February 1978

Алгоритм Фитча-Нормана (связной таблицы разрывов) (2)

- Фазы алгоритма
  - Пометка живых объектов
  - Связывание интервалов в список
  - Корректировка ссылок при помощи списка интервалов
  - Уплотнение
- Преимущества:
  - Таблицу не нужно эвакуировать (избегаем квадратичной сложности фазы построения таблицы)
- Недостатки:
  - Требуется дополнительный проход
  - Поиск в связном списке затруднен (линейная сложность обновления одного указателя)
- Ускорение поиска
  - В освободившемся стеке пометки или найденном во время связывания максимальном свободном интервале строим хеш-таблицу входов в список и начальных смещений

Алгоритм Фитча-Нормана

```
inline void collect( void ) {
  mark_objects();

  Byte* dead = skip_live( HeapBottom );
  const Byte* top = AllocationTop;
  if( dead < top ) {          // Есть освобожденные объекты?
    Byte* live = skip_dead( dead );
    if( live < top ) {        // Есть перемещаемые объекты?
      link_ranges( dead, live, top );
      update_pointers( top );
      AllocationTop = compact( top );
    } else {
      AllocationTop = dead;
    }
  }
}
```

Представление списка интервалов

- Элемент списка должен помещаться в любом свободном интервале
  - Минимальный интервал состоит из единственного объекта минимального размера
  - Как в него уместить его длину и адрес следующего интервала?
- Используем представление переменной длины
  - Первое слово - адрес следующего элемента
  - Этот адрес выровнен, используем его младший бит для обозначения свободного интервала длиной в одно слово
    - Если бит установлен, это интервал длиной в слово
    - Иначе это интервал большей длины, хранящейся в следующем слове

Доступ к списку интервалов

```
inline void set_range( unsigned* p, unsigned live, unsigned dead ) {
  if( ( ((unsigned*)live) - p ) == 1 ) {
    p[0] = dead | 1;
  } else {
    p[0] = dead;
    p[1] = live;
  }
}
inline void get_range(const unsigned* p, unsigned& live, unsigned& dead){
  dead = p[0];
  if( dead & 1 ) {
    dead &= ~1;
    live = unsigned(p+1);
  } else {
    live = p[1];
  }
}

inline void set_range( void* p, Byte* live, Byte* dead ) {
  set_range( (unsigned*)p, (unsigned)live, (unsigned)dead );
}
inline void get_range( void* p, Byte*& live, Byte*& dead ) {
  get_range( (unsigned*)p, (unsigned&)live, (unsigned&)dead );
}
```

Связывание интервалов в список

```
inline void link_ranges(Byte* dead, Byte* live, const Byte* top){
  first_dead = dead;

  for(;;) {
    Byte* next_dead = skip_live( live );
    set_range( dead, live, next_dead );
    if( next_dead == top ) {
      break;
    }
    dead = next_dead;
    live = skip_dead( next_dead );
  }
}
```

Корректировка ссылок при помощи списка интервалов

```
inline void update_pointers( const Byte* top ) {
  Byte* live = HeapBottom;
  for( Byte* dead = first_dead;; ) {
    for( ; live < dead; live += allocation_size( live ) ) {
      forEachRef( ref, live ) {
        update_pointer( ref );
      }
    }
    if( dead == top ) {
      break;
    }
    const void* r = dead;
    get_range(r, live, dead);
  }

  forEachRoot( ref ) {
    update_pointer( ref );
  }
}
```

Корректировка ссылки при помощи списка интервалов

```
void update_pointer( Byte** ref ) {
  Byte* p = *ref;
  const Byte* dead = first_dead;

  // Указывает на перемещаемый объект?
  if( p > dead ) { // && p < HeapTop
    unsigned offset = 0;

    do {
      Byte* live; Byte* next;
      get_range( dead, live, next );
      offset += live - dead;
      dead = next;
    } while( p > dead );

    *ref = p - offset;
  }
}
```

Ускорение и распараллеливание корректировки ссылок

- Для ускорения линейного поиска в связном списке строим индекс
  - В стеке пометки, который свободен вне фазы пометки
  - Или в максимальном свободном интервале, найденном при построении списка интервалов
  - По числу входов в индекс делим на равные сегменты диапазон адресов от самого нижнего подвижного объекта до *allocation_top* или *HeapTop*
  - Для каждого сегмента храним в индексе адрес соответствующего элемента списка и его смещение при уплотнении
- Индекс списка может быть использован для параллельной корректировки ссылок
  - Корректировка ссылок в каждом сегменте производится независимо от других сегментов
  - Перераспределение работ между потоками требует осторожности — корректировка ссылки не идемпотентна, ее нельзя повторять

Уплотнение списка интервалов

```
inline Byte* compact ( const Byte* top ) {
  Byte* dead = first_dead;
  Byte* dst = dead;

  do {
    Byte* live; Byte* next;
    get_range( dead, live, next );
    const unsigned size = next - live;
    move( dst, live, size );
    dst += size;
    dead = next;
  } while( dead < top );

  return dst;
}
```

Алгоритм отсылочных указателей

- Применим к объектам любого размера
- Сохраняет порядок следования объектов
  - Сохраняет локальность ссылок
  - Объекты разного возраста располагаются в разных диапазонах адресов
  - Облегчает инлайновую подстановку объектов и другие оптимизации
- Фазы алгоритма
  - Пометка живых объектов
  - Вычисление нового места расположения объектов
  - Корректировка ссылок
  - Уплотнение (и стирание пометки)
- Требует дополнительного поля в заголовке
  - Указатель на новое место расположения объекта (*Forwarding Pointer*)

Вычисление расположения объектов

```
inline Byte* compute_locations( void ) {
  Byte* dst = HeapBottom;
  Byte* src = dst;
  while( src < AllocationTop ) {
    const unsigned size = allocation_size( src );
    if( is_marked( src ) ) {
      set_forwarding_pointer( src, dst );
      dst += size;
    }
    src += size;
  }
  return dst;
}
```

- Объекты могут либо оставаться неподвижными, либо опускаться ближе к дну кучи
  - Для сокращения расходов при изменении страниц виртуальной памяти, уплотнении и корректировке ссылок полезно запомнить адрес первого перемещающегося объекта

Вычисление расположения объектов (2)

```
inline Byte* skip_solid_prefix( void ) {
  Byte* src = HeapBottom;
  while( src < AllocationTop && is_marked( src ) ) {
    src += allocation_size( src ) )
  }
  return src;
}
inline Byte* compute_locations( void ) {
  Byte* src = skip_solid_prefix();
  Byte* dst = src;
  CompactionBottom = src;
  while( src < AllocationTop ) {
    const unsigned size = allocation_size( src );
    if( is_marked( src ) ) {
      set_forwarding_pointer( src, dst );
      dst += size;
    }
    src += size;
  }
  return dst;
}
```

Корректировка ссылок

- Пройти по выжившим объектам и всем корням и обновить ссылки из них

```
inline void update_pointers( void ) {
  const Byte* top = AllocationTop;
  for( Byte* p = HeapBottom; p < top; p += allocation_size(p) ) {
    if( is_marked( p ) ) {
      forEachRef( ref, p ) {
        update_pointer( ref );
      }
    }
  }

  forEachRoot( ref ) {
    update_pointer( ref );
  }
}
```

Корректировка одного указателя

```
inline void update_pointer( Byte** ref ) {
  const Byte* p = *ref;
  if( CompactionBottom <= p ) { // && p < AllocationTop
    *ref = get_forwarding_pointer( p );
  }
}
```

- Если объект не перемещается, нет необходимости обновлять указатель на него
  - Все перемещающиеся объекты расположены в диапазоне между *CompactionBottom* и *AllocationTop*
    - Если нет ссылок вне данной непрерывной кучи, верхнюю границу можно не проверять
  - Значения *NULL* находятся вне этого диапазона
  - При практической реализации эту функцию полезно подставить в места вызова при помощи директивы компилятору или вручную с последующим кэшированием читаемых глобалов в регистрах

Уплотнение

- Снова симулируем отведение живых объектов и копируем их по новому адресу

```
inline void compact( void ) {
  const Byte* src = CompactionBottom;
  Byte* dst = src;
  while( src < AllocationTop ) {
    // Обязательно нужно запомнить размер -
    // при перемещении вниз объект может затереть
    // собственный заголовок
    const unsigned size = allocation_size( src );
    if( is_marked( src ) ) {     // test_and_unmark( src )
      unmark( src );             // unpack( src )
      move( dst, src, size );
      dst += size;
    }
    src += size;
  }
}
```

Уплотнение (альтернативный вариант)

```
inline void compact( void ) {
  const Byte* src = CompactionBottom;
  while( src < AllocationTop ) {
    // Обязательно нужно запомнить размер -
    // при перемещении вниз объект может затереть
    // собственный заголовок
    const unsigned size = allocation_size( src );
    if( is_marked( src ) ) {
      unmark( src );
      Byte* dst = get_forwarding_pointer( src );
      move( dst, src, size );
    }
    src += size;
  }
}
```

- При повторении симуляции отведения ее результат не меняется
  - Отсылочный указатель незачем копировать на новое место размещения объекта

Ускорение сканирования кучи

- При вычислении нового размещения, корректировке ссылок и уплотнении производится полное сканирование кучи
- При этом вычисляется размер *каждого* объекта (живого или мертвого)
  - Если размер не записан в заголовке, его вычисление может быть не очень простой операцией (массивы и т.п.)
- Если признаки пометки хранятся в битовой шкале, лучше сканировать ее, а не кучу
  - Выше локальность ссылок, специальные инструкции
  - Не нужно вычислять размеры мертвых объектов
- Иначе можно переписать первый объект большого мертвого интервала
  - Байтовый массив соответствующего размера
- Или связать мертвые интервалы в список
  - Пересылка интервалов быстрее пересылки объектов
  - Но больше работы при вычислении размещений
  - Трудно совместимо с упаковкой отсылочных указателей — распаковку удобно производить при проходе по живым объектам в фазе уплотнения

Упаковка отсылочных указателей

- Объекты часто бывают снабжены заголовками
  - Типовой тег, размер, атрибуты, бит пометки
- Но в заголовке редко можно найти место для хранения *полного* указателя
- Выравнивание
  - Несколько младших битов указателей свободны
  - По окончанию сборки все поля заголовка должны быть восстановлены!
- Беззнаковое смещение
  - Направление перемещения известно
  - С учетом выравнивания 16 битами можно закодировать смещение в 256KB области
- Логическое сегментирование
  - Порядок следования объектов сохраняется
  - Смещения не увеличиваются
  - Диапазон переходит в диапазон того же или меньшего размера

Логическое сегментирование

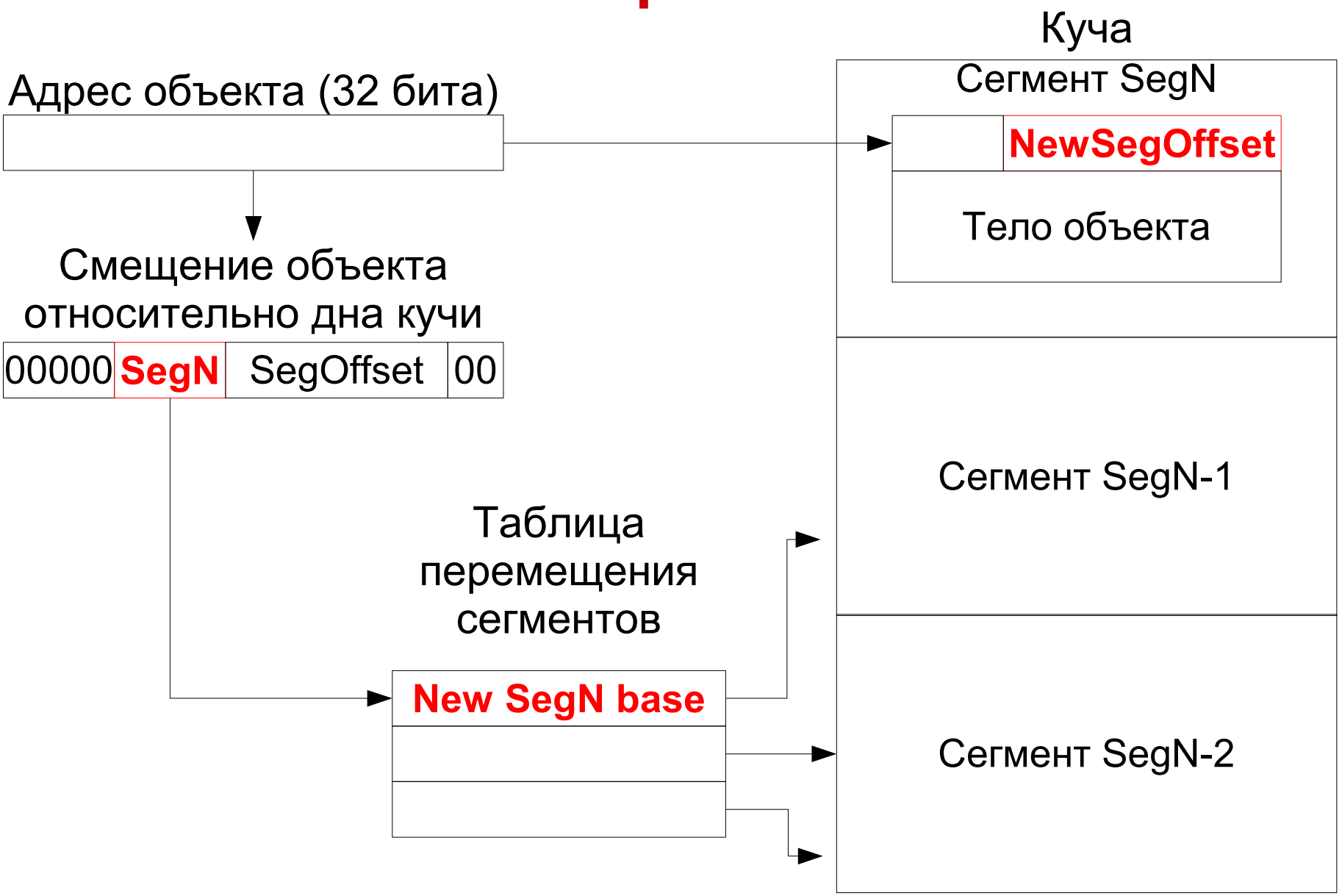


Таблица перемещения сегментов

- Число сегментов определяется размером сегмента и размером кучи
- Размер сегмента определяется выравниванием объектов и числом битов смещения в заголовке объекта
  - Для определенности предположим выравнивание на границу 4-байтового слова
  - Если заголовок объекта ссылается на класс или дескриптор объекта, необходимое для его кодирования число битов растет с размером кучи
  - … а число битов для смещения соответственно уменьшается

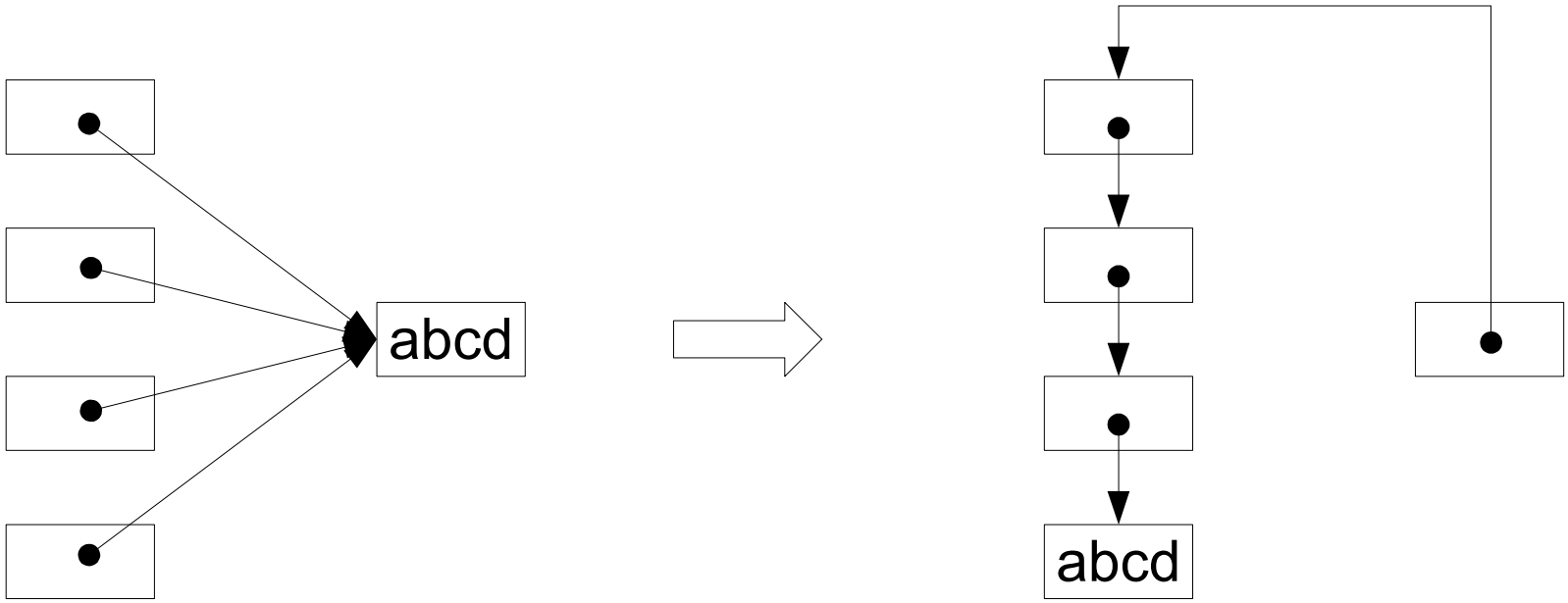
Размер кучи	Битов класса	Битов смещения	Число сегментов
256 K	16	16	1
1 MB	18	14	16
16 MB	22	10	4096

Распараллеливание

- Все фазы, кроме уплотнения, легко поддаются распараллеливанию с синхронизацией в конце фазы
- Вычисление расположения объектов
  - Разделить кучу на сегменты
  - Каждый поток при симуляции отведения использует собственный указатель текущего адреса
  - У самого нижнего сегмента этот адрес настоящий, у остальных начинаются с нуля
  - По окончанию обработки сегмента в соответствующий ему элемент таблицы перемещения записать конечное значение указателя отведения, т. е. *новый размер* сегмента
  - По завершению обработки всех сегментов вычислить *новую базу* каждого сегмента как сумму *новых размеров* предыдущих сегментов
- При корректировке ссылок независимо обрабатывать сегменты

Обращение и прошивка указателей

- Способ обращения отношения «ссылается на» без использования дополнительной памяти
  - Поставить в соответствие объекту множество ссылающихся на него указателей
  - Представить это множество односвязным списком
  - Корень списка разместить в объекте
  - В качестве полей ссылок в списке использовать сами указатели



Обращение и прошивка указателей (2)

```
void link( void** pp ) {
  void** p = (void**)*pp;
  if( p ) {
    *pp = *p;
    *p = (void*)pp;
  }
}

#define is_pointer(p) ( (unsigned(p) & 1) == 0 )

void resolve(void* dst, void* obj) {
  void* p = *(void**) obj;
  while( is_pointer(p) ) {
    void** q = (void**)p;
    p = *q;
    *q = dst;
  }
  *(void**)obj = p;
}
```

Обращение и прошивка указателей (3)

- Широко используется в загрузчиках, линковщиках, однопроходных ассемблерах и компиляторах для разрешения ссылок вперед на метки
- Необходимые условия
  - Указатели могут ссылаться только на начало объектов
  - Можно отличить указатель от хранящегося в объекте значения (например, поля-указатели выровнены, а объект снабжен заголовком с единицей в младшем бите)
  - Размер объекта достаточен для размещения в нем указателя

F. Lockwood Morris. A time- and space-efficient garbage collection algorithms. *Communications of the ACM,* 21(8), August 1978.

Алгоритм Йонкерса

H. B. M. Jonkers. A fast garbage collection algorithm. *Information Processing Letters,* 9(1), July 1979.

- Фазы алгоритма
  - Пометка доступных объектов
  - Корректировка ссылок вперед
  - Уплотнение, корректировка ссылок назад
- Первоначально сформулирован для объектов фиксированного размера
  - Легко обобщается, но менее красив для объектов переменного размера

Корректировка ссылок вперед

```
inline void update_forward_pointers( void ) {
  forEachRoot( ref ) link( ref );

  Byte* dst = HeapBottom;
  for( Byte* src = dst; src < AllocationTop; ) {
    if( is_marked( src ) ) {
      resolve( dst, src );
      // Заголовок восстановлен - можно вычислять размер
      const unsigned size = allocation_size( src );
      forEachRef( ref, src ) link( ref );

      // В процессе итерации заголовок может быть переписан
      // Нужно учитывать это при реализации forEachRef
      // Размер вычислять здесь нельзя
      src += size;
      dst += size;
    } else {
      src += allocation_size( src );
    }
  }
}
```

Корректировка ссылок назад

```
inline Byte* compact_and_update_backward_pointers(void) {
  Byte* dst = HeapBottom;
  for( Byte* src = dst; src < AllocationTop; ) {
    if( is_marked( src ) ) {
      resolve( dst, src );
      // Заголовок восстановлен - можно вычислять размер
      const unsigned size = allocation_size( src );
      move( dst, src, size );
      src += size;
      dst += size;
    } else {
      src += allocation_size(src);
    }
  }
  return dst;
}
```

Модифицированный алгоритм Йонкерса

- Сравнение алгоритма Йонкерса с алгоритмом отсылочных указателей
  - \+ Одним сканированием кучи меньше
  - – В остальных проходах совершается значительно больше работы
  - – Низкая локальность обращений к памяти
  - – За исключением фазы пометки, не поддается распараллеливанию
- Модифицированный алгоритм
  - Обход доступных объектов с прошивкой указателей
  - Корректировка ссылок и связывание в список интервалов
  - Уплотнение интервалов
- Анализ модифицированного алгоритма
  - Не требует отдельного бита пометки
  - Ни в какой фазе не поддается распараллеливанию
  - Труднее обобщить на объекты разных типов и размеров
  - За исключением ускорения фазы уплотнения, сохраняет все недостатки исходного алгоритма

Обход с прошивкой указателей

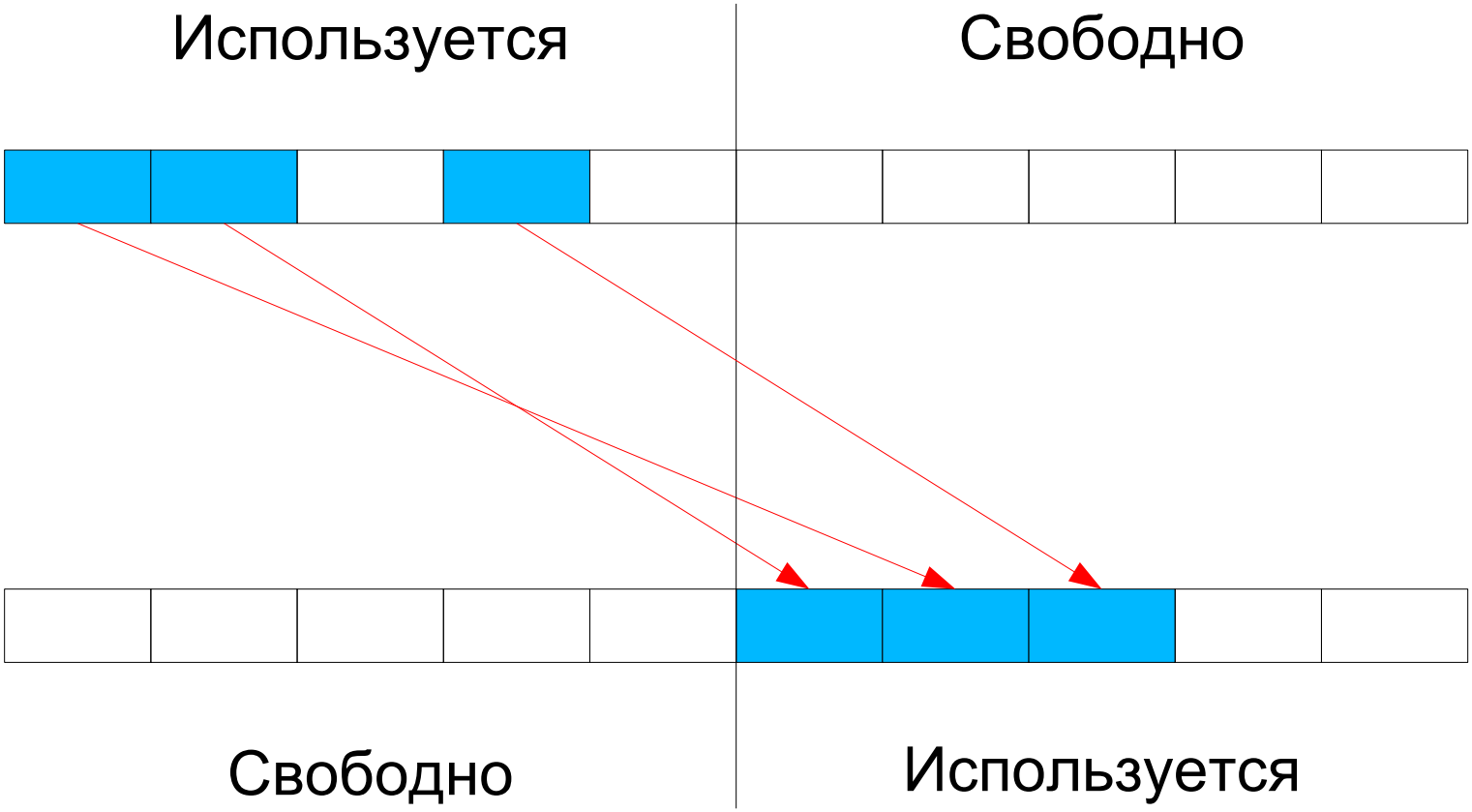
```
#define is_marked(p) is_pointer(*(void*)p)
Object* link_and_push( Object** ref ) {
  Object* obj = *ref;
  // Если возможны ссылки на объекты вне кучи, вместо NULL
  // проверить принадлежность диапазону адресов кучи
  if( obj ) {
    *ref = *(Object**)obj;
    *(Object**)obj = (Object*)ref;
    push( obj );
  }
  return obj;
}

void mark_root( Object** root ) {
  if( link_and_push(root) ) {
    do {
      // Заголовок объекта переписан. Исходный заголовок
      // хранится в конце списка прошитых указателей
      forEachRef( ref, pop() ) link_and_push( ref );
    } while( markingStackNotEmpty() );
  }
}
```

Инкрементальная параллельная пометка с последующим уплотнением: алгоритм Стила

Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM,* 18(9), September 1975.

- Параллельная реализация инкрементальной пометки в стиле Дейкстры
- Последующее уплотнение не инкрементально и не параллельно

Копирующая сборка мусора



Копирующая сборка мусора

- Делим память для хранения объектов на два *полупространства* равного размера
- Одно из них играет роль *активного*
- Память в активном полупространстве отводится путем простого перемещения указателя AllocationTop (*Pointer Bumping*)
- По исчерпании памяти в активном полупространстве копируем все достижимые по ссылкам из корней объекты в другое полупространство
- Во избежание повторного копирования новый адрес объекта запоминаем в месте его старого расположения
- Меняем полупространства ролями

Рекурсивный алгоритм Фенихеля-Йохельсона

```
void copy( Object** ref ) {
  Object* src = *ref;
  if( src ) {
    Object* dst = get_forwarding_pointer(src);
    if( !dst ) {    // Проверка наличия отсылочного указателя
      const unsigned size = allocation_size(src);
      dst = AllocationTop; AllocationTop += size;
      copy(dst, src, size); // Без перекрытия
      set_forwarding_pointer(src, dst);
      forEachRef(son, dst) copy(son);
    }
    *ref = dst;
  }
}
void collect( void ) {
  swap(ActiveTop, PassiveTop);
  swap(ActiveBottom, PassiveBottom);
  AllocationTop = ActiveBottom;
  forEachRoot(ref) copy(ref);
}
```

Рекурсивный алгоритм Фенихеля-Йохельсона (2)

Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM,* 17(11), November 1969.

- Сборка мусора происходит в один проход по достижимым объектам
- Отсылочный указатель пишется на место старого расположения объекта
  - Размер объекта должен быть достаточен для размещения указателя
- Отсылочный указатель должен быть отличим от обычного заголовка объекта
  - Проверка выравнивания - если объекты выровнены и снабжены заголовками с единицей в младшем бите
  - Или проверка диапазона - если первым полем каждого объекта является указатель

Рекурсивный алгоритм Фенихеля-Йохельсона (3)

- Вне сборки мусора половина адресного пространства кучи не используется
  - ОС об этом не может догадаться сама
  - Для уменьшения нагрузки на виртуальную память полезно отменить (uncommit) ее отображение в пассивное полупространство
  - Для снижения энергопотребления можно выключить соответствующие банки памяти
- Все объекты копируются
  - Не эффективно для больших и долгоживущих объектов
  - Невозможно временно зафиксировать физический адрес объекта
  - При низкой ассоциативности кэша данных возможны конфликты адресов разных полупространств
  - Если оба полупространства одновременно не помещаются в виртуальную память, возможно аномальное вытеснение страниц

Рекурсивный алгоритм Фенихеля-Йохельсона (4)

- Объекты переупорядочиваются в соответствии с порядком обхода
  - Порядок обхода можно варьировать
  - От него зависит пространственная локальность ссылок
  - Оптимальный порядок обхода зависит от приложения
  - По результатам экспериментальных исследований обход в глубину заметно лучше обхода в ширину, но хуже сохранения исходного порядка

David A. Moon. Garbage collection in a large LISP system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, Austin, TX, August 1984.

James W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(3), May 1984.

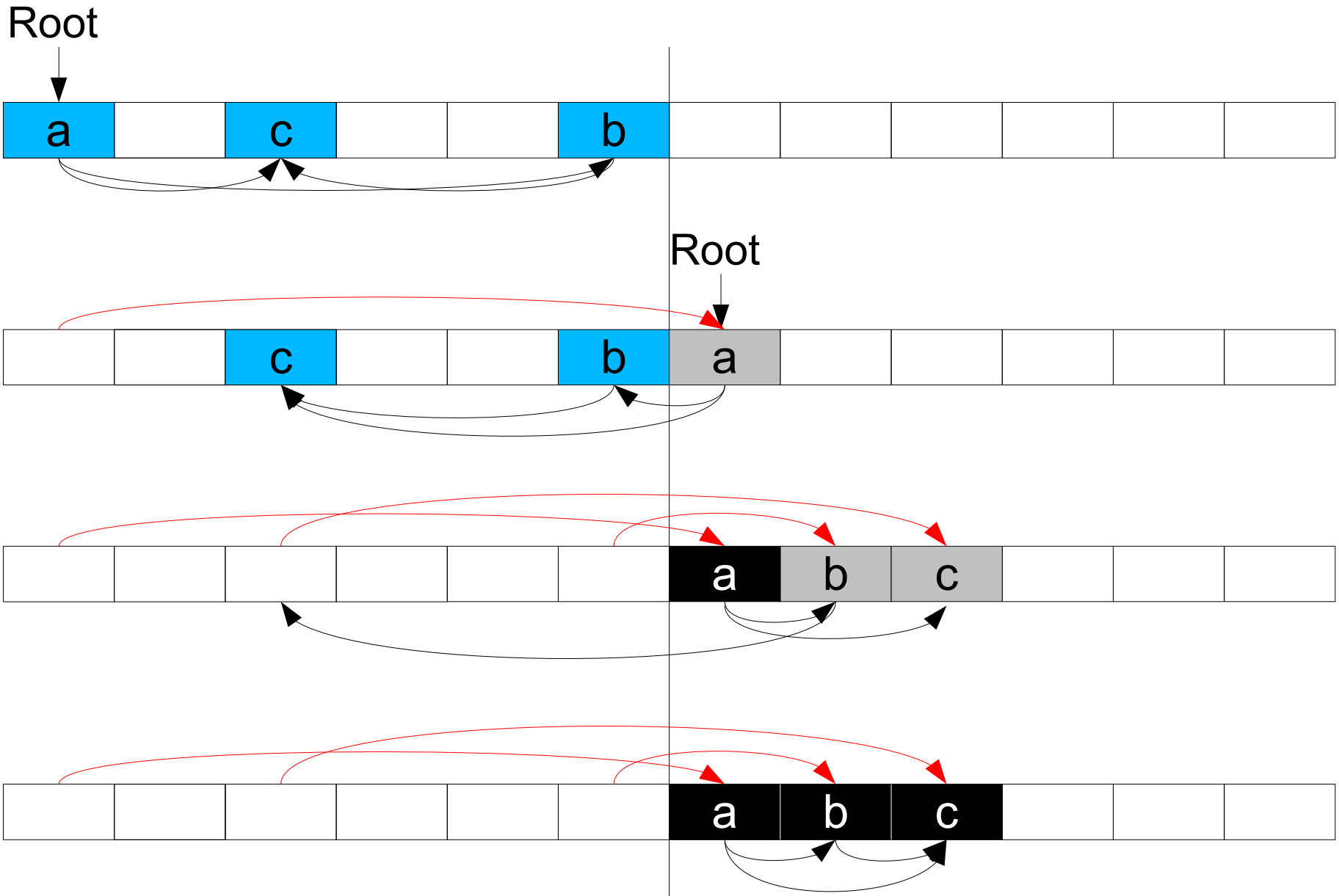
Рекурсивный алгоритм Фенихеля-Йохельсона (5)

- Трудно распараллелить
  - Требуется синхронизация при отведении памяти, записи и чтении отсылочных указателей
- При рекурсии возможно переполнение программного стека
  - Можно использовать ранее рассмотренные отдельный стек, способы минимизации его глубины и обработки его переполнения
  - Или разработать итеративный алгоритм

C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM,* 13(11), November 1970

Итеративный алгоритм Чейни («двух пальцев»)

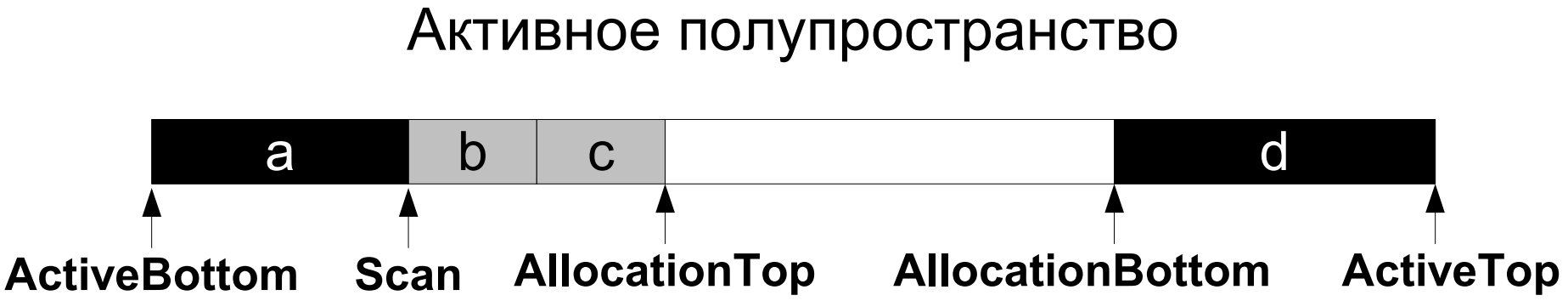
- Вместо рекурсивного обхода сыновей копируем их и линейно просматриваем копии в цикле
- Для этого на время сборки мусора заводим переменную *scan*, которая указывает на первый скопированный, но еще не просмотренный объект
- Переменные *scan* и *AllocationTop* («пальцы») разбивают полупространство на три интервала
  - Черный (от *Bottom* до *scan*) — скопированные просмотренные объекты
  - Серый (от *scan* до *AllocationTop*) — скопированные непросмотренные объекты
  - Белый (от *AllocationTop* до *Top*) — пустая память
- Сборка завершена, когда серых объектов не осталось (т.е. «пальцы» сошлись)


Root
a
c
b
Root
c
b
a
a
b
c
a
b
c

Итеративный алгоритм Чейни (3)

```
void copy( Byte** ref ) {
  Byte* src = *ref;
  if( src ) {
    Byte* dst = get_forwarding_pointer(src);
    if( !dst ) {
      const unsigned size = allocation_size(src);
      dst = AllocationTop; AllocationTop += size;
      copy(dst, src, size);
      set_forwarding_pointer(src, dst);
    }
    *ref = dst;
  }
}
void collect( void ) {
  swap(ActiveTop, PassiveTop);
  swap(ActiveBottom, PassiveBottom);

  AllocationTop = ActiveBottom;
  forEachRoot(ref) copy(ref);

  for( Byte* scan = ActiveBottom;
         scan < AllocationTop; scan += allocation_size( scan ) ) {
    forEachRefOf(son, scan) copy(son);
  }
}
```

Инкрементальный копирующий алгоритм Бэйкера



- Инкрементальная модификация алгоритма Чейни
- Collector и Mutator отводят память в активном полупространстве навстречу друг другу

Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4), 1978.

Инкрементальный копирующий алгоритм Бэйкера (2)

- Трехцветная окраска определяется интервалами адресов
  - Черные объекты занимают интервалы от *ActiveBottom* до *Scan* и от *AllocationBottom* до *ActiveTop*
  - Серые объекты в диапазоне от *Scan* до *AllocationTop*
  - Белые объекты в пассивном полупространстве (диапазон от *PassiveBottom* до *PassiveTop*)
- Сканирование серых объектов производится при отведении памяти Mutator'ом
  - Если все серые объекты обработаны, сборка завершена. Меняем полупространства ролями, начинаем новую сборку
  - Скорость сканирования определяется эвристически
  - Ситуация исчерпания памяти в активном полупространстве до завершения сборки мусора данным алгоритмом не обрабатывается

Инкрементальный копирующий алгоритм Бэйкера (3)

- Барьер чтения: если объект белый, то копировать его в серые, запоминая на его месте отсылочный указатель и корректируя ссылку

```
Object* read_barrier( Object* obj ) {
  if( is_white( obj ) ) {
    Object* dst = get_forwarding_pointer(obj);
    if( !is_valid_forwarding_pointer( obj ) ) {
      const unsigned size = allocation_size(src);
      if( AllocationTop + size > AllocationBottom )
        HeapOverflow();
      dst = AllocationTop; AllocationTop += size;
      copy(dst, obj, size);
      set_forwarding_pointer(obj, dst);
    }
    obj = dst;
  }
  return obj;
}
```

Инкрементальный копирующий алгоритм Бэйкера (4)

- Непрерывно копируются объекты и переключаются полупространства
  - Пример — несколько долгоживущих объектов в большой куче
- Можно по завершении сканирования не переключать полупространства до тех пор, пока программа не исчерпает память между *AllocationTop* и *AllocationBottom*
  - Барьер можно выключить
  - Объекты при этом отводятся черными

Инкрементальный копирующий алгоритм Брукса

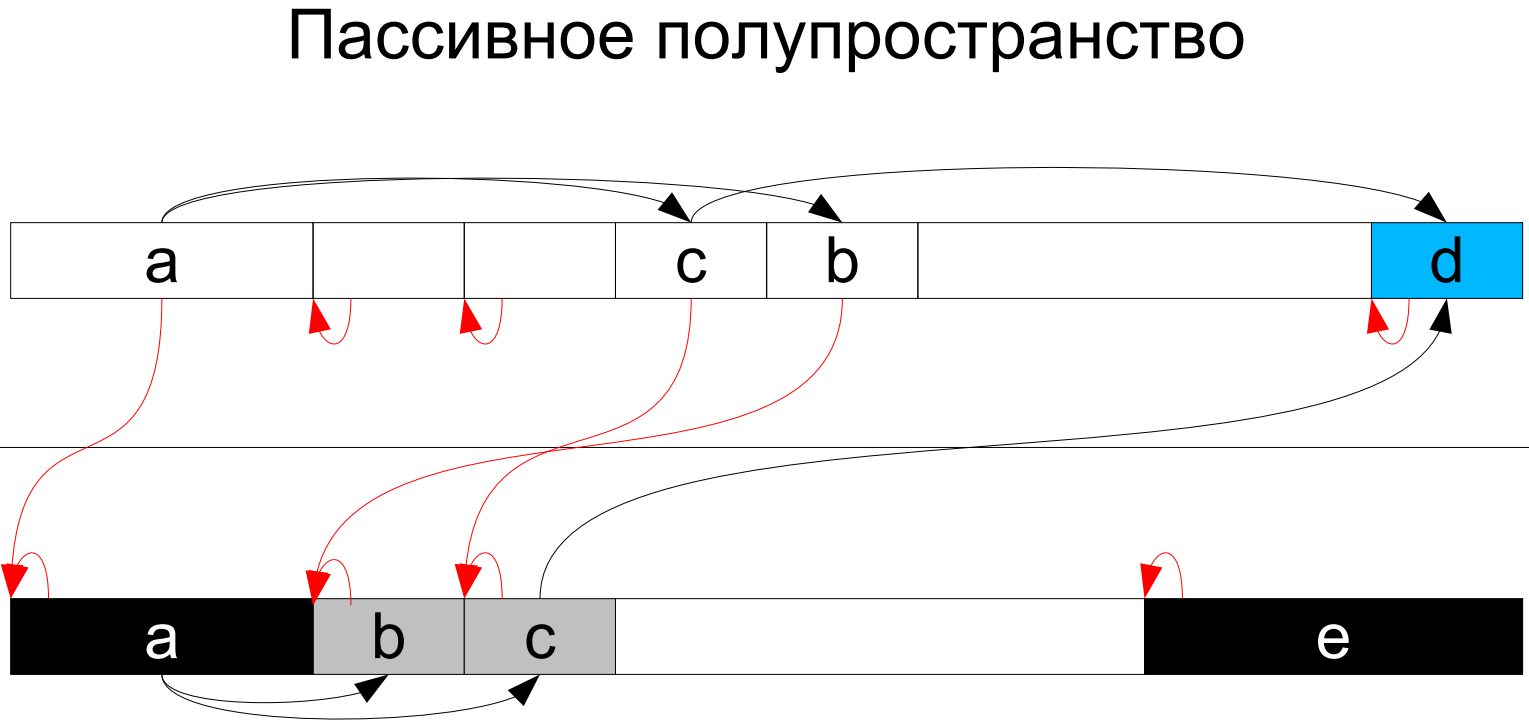
Rodney A. Brooks. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, Austin, TX, August 1984.

- Вариация алгоритма Бэйкера с заменой барьера чтения на барьер записи
  - Барьер записи вызывается гораздо реже
  - Перехватывает присваивание белого объекта полю черного объекта
  - Как и в алгоритме Бэйкера, объект копируется в область серых объектов
- Проблема: могут остаться ссылки на старое место расположения объекта
  - Барьер чтения сработал бы при первой ссылке, а барьер записи может сработать позже

Инкрементальный копирующий алгоритм Брукса (2)

- Увеличиваем косвенность доступа к объекту
  - Каждый объект снабжается отсылочным указателем
  - Изначально он ссылается на заголовок содержащего его объекта
  - Если объект скопирован, то ссылается на копию
  - При любом обращении к объекту переходим по отсылочному указателю
  - По существу косвенный доступ — предельно упрощенный барьер чтения
- Замедление доступа
- Усложнение оптимизаций доступа к полям
- Дополнительный указатель в каждом объекте
  - В алгоритмах Чейни и Бэйкера отсылочный указатель записывается после копирования на место старого расположения объекта

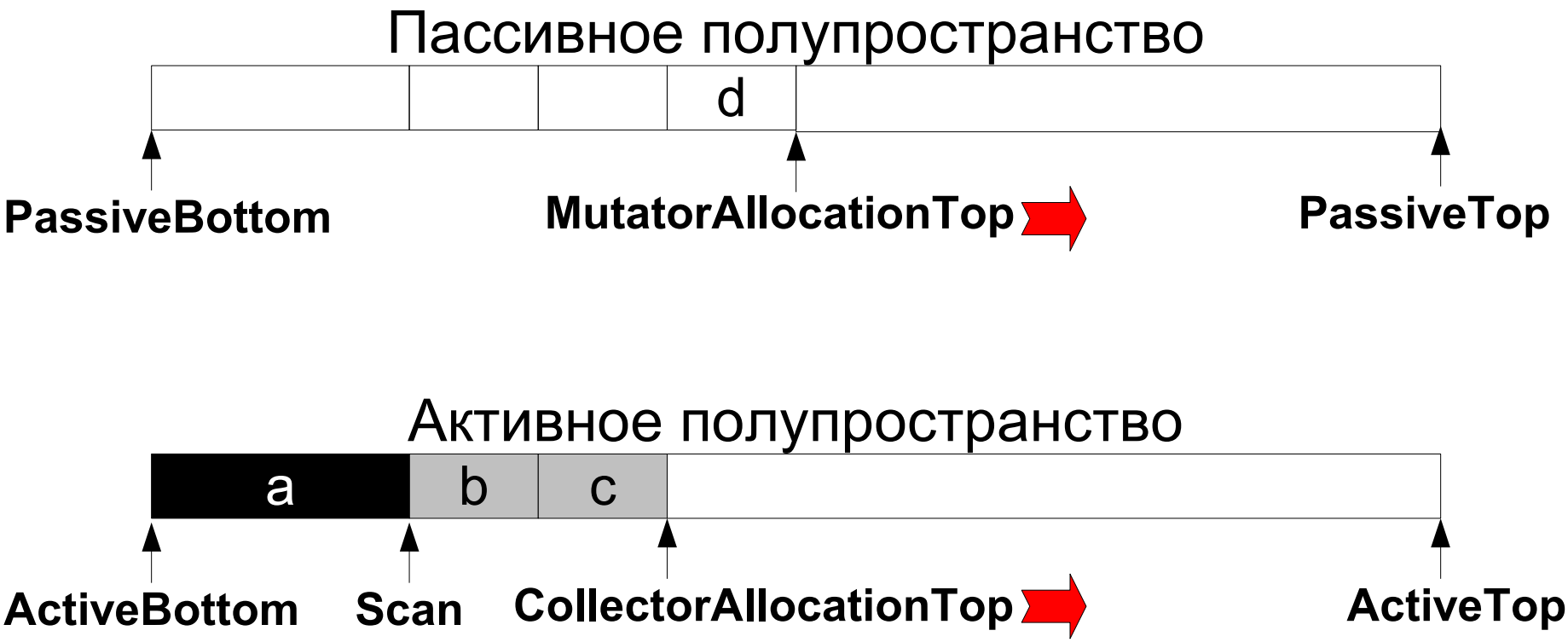
Инкрементальный копирующий алгоритм Брукса (3)



Отведение новых объектов белыми в алгоритме Брукса

- В алгоритме Бэйкера новые объекты отводятся в активном полупространстве черными
  - Не могут быть белыми из-за мгновенного перекрашивания барьером чтения
- В алгоритме Брукса возможно отводить объекты белыми или черными
  - Полупространства переключаются в момент исчерпания серых объектов
  - В этот момент в активном полупространстве имеется достаточное количество памяти
  - После переключения полупространств можно в этой памяти отводить белые объекты
  - Новые объекты могут успеть умереть белыми и тогда будут освобождены в текущем цикле сборки мусора
  - Для присваиваний полям белых объектов не нужен барьер записи (оптимизация конструкторов)

Отведение новых объектов белыми в алгоритме Брукса (2)



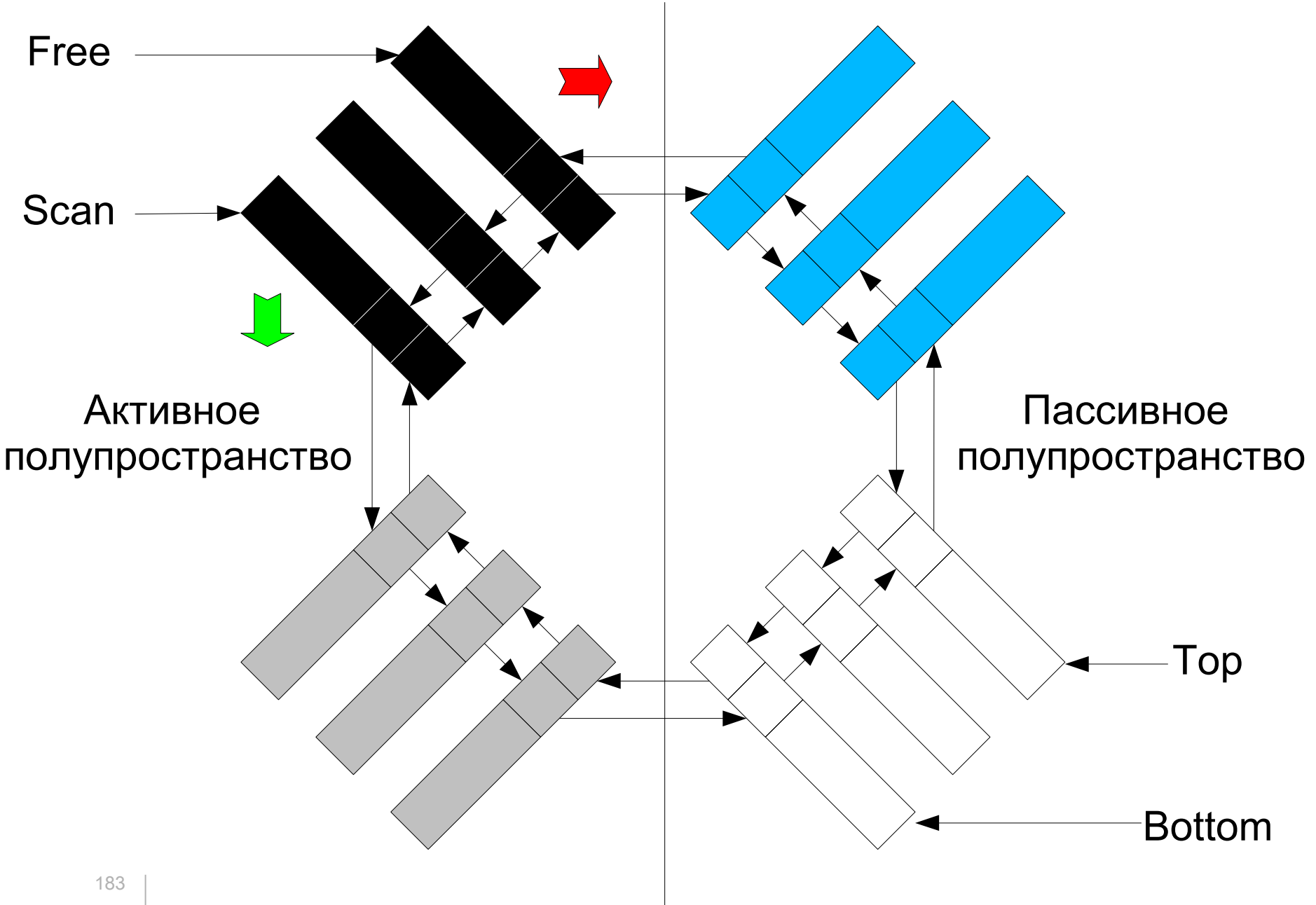
Jeffrey L. Dawson. Improved effectiveness from a real-time LISP garbage collector. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, San Francisco, CA, June 1992. ACM Press.

Мельница Бэйкера

Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices,* 27(3), March 1992

- Только для объектов фиксированного размера
- Инкрементальная пометка с последующим освобождением
  - Алгоритм основан на копирующем алгоритме Бэйкера
  - Копирование заменено на перемещение объекта в двусвязном циклическом списке
  - Длительность выполнения большинства операций невелика и хорошо предсказуема
- Барьер чтения
  - Может быть заменен на барьер записи
- Четырехцветная раскраска
  - Бесцветное пустое пространство в дополнение к обычным трем цветам

Мельница Бэйкера (2)



Мельница Бэйкера (3)

- Память кучи разбита на элементы фиксированного размера, связанные в двусвязный циклический список
  - Операции удаления и вставки в произвольную позицию выполняются в несколько присваиваний

```
static inline void link( Object* prev, Object* next ) {
  prev->set_next( next );
  next->set_prev( prev );
}
inline void Object::remove( void ) {
  Object* prev = get_prev();
  Object* next = get_next();
  if( this == Bottom ) Bottom = prev;
  if( this == Top    ) Top    = next;
  link( prev, next );
}
inline void Object::insert( void ) {
  link( this, Bottom->get_next() );
  link( Bottom, this );
}
```

Мельница Бэйкера (4)

- Элементы не перемещаются и могут где угодно располагаться в адресном пространстве
  - Куча может менять размер и быть блочной
  - Принадлежность к полупространству определяется не адресом элемента, а его позицией в списке
  - Для ускорения проверки принадлежности к полупространству снабдим элементы битовым признаком *active*
  - Смысл этого бита будем менять на противоположный при смене ролей полупространств
  - При условии выравнивания элементов для экономии памяти можно разместить этот бит в одном из младших битов одной из ссылок в списке

Операции с принадлежностью к активному полупространству

```
static int CurrentActiveState;

inline int Object::is_active( void ) {
  return get_active_bit() == CurrentActiveState;
}

inline void Object::set_active( void ) {
  set_active_bit( CurrentActiveState );
}

inline void flip_active_state( void ) {
  CurrentActiveState ^= 1;
}
```

Мельница Бэйкера (5)

- Список разделен на сегменты указателями *Scan*, *Free*, *Bottom* и *Top*
  - Свободный сегмент от *Free* до *Top*
  - Белый сегмент от *Top* до *Bottom*
  - Серый сегмент от *Bottom* до *Scan*
  - Черный сегмент от *Scan* до *Free*
  - С точки зрения реализации удобнее ссылаться на граничные черные и белые элементы, поскольку они всегда есть
- Цвет элемента определяется принадлежностью к сегменту

Отведение памяти в мельнице Бэйкера

- Отведение памяти производится перемещением указателя *Free* на следующий элемент по часовой стрелке
- При исчерпании памяти расширить кучу

```
static Object* allocate( void ) {
  collect();
  Free = Free->get_next();
  if( Free == Top ) HandleOverflow();

  Object* obj = Free;
  obj->set_active();
  return obj;
}
```

Сборка мусора в мельнице Бэйкера

- Сканирование производится путем перемещения указателя *Scan* на очередной элемент против часовой стрелки
- Если серых элементов больше нет, полупространства меняются ролями и начинается следующий цикл сборки
- Иначе все сыновья сканируемого элемента, находящиеся в пассивном полупространстве, перемещаются в конец серого сегмента
- Количество элементов, сканируемых при отведении одного элемента, определяется адаптивно вычисляемым коэффициентом *SpeedFactor*

Сборка мусора в мельнице Бэйкера

```
void Object::move( void ) {
  if( this != NULL && !is_active() ) {
    set_active();
    remove(); insert();
  }
}

static void collect( void ) {
  int count = SpeedFactor;
  do {
    Scan = Scan->get_prev();
    if( Scan == Bottom ) {
      flip_spaces();
      forEachRoot( ref ) ref->move(); // Оценка числа корней?
      break;
    }
    forEachRef( ref, Scan ) ref->move();
  } while( --count );
}
```

Переключение полупространств

```
static inline void flip_spaces( void ) {
  Top = Bottom->get_next();
  Bottom = Free;
  Scan = Free->get_next();
  flip_active_state();
}
```

- Объекты, оставшиеся белыми в конце цикла сборки, добавляются к свободным
  - Белые и свободные расположены в соседних сегментах, поэтому объединение сводится к перестановке указателей
- Черные объекты становятся белыми
- Серых объектов в момент переключения полупространств быть не может

Барьер чтения в мельнице Бэйкера

```
static inline void read_barrier( Object* obj ) {
  obj->move();
}
```

- При обращении к белому объекту переместить его в конец списка серых
- Барьер чтения можно заменить на барьер записи
  - Барьер записи вызывается в несколько раз реже
  - Реализация барьера записи точно такая же, как и барьера чтения
  - Не требуется никаких изменений в алгоритме
  - При переключении полупространств объекты, доступные только из корней, попадают в свободный сегмент. Но сразу после этого при обходе корней они перемещаются в серый сегмент.

Анализ мельницы Бэйкера

- Длительность выполнения большинства функций невелика и хорошо предсказуема.
- Исключения:
  - Сканирование переменного числа корней
  - Обработка переполнения кучи
  - Оценка коэффициента *SpeedFactor* — с какой скоростью нужно сканировать объекты, чтобы цикл сборки был завершен до исчерпания свободного места программой
- Ускорение удаления из списка (*Object::remove)*
  - Перемещения указателей *Bottom* и *Top* можно избежать, если использовать для них фиктивные элементы
- Ускорение перемещения (*Object::move*)
  - Для устранения проверки на *NULL* использовать *NullObject*.

Гибридные алгоритмы

- Копирование наиболее эффективно для молодых объектов
  - Единственный проход по живым объектам
  - Вследствие высокой смертности доля памяти, занимаемой живыми объектами, низка
  - Нет долгоживущего плавающего мусора
- Пометка с освобождением и при необходимости последующим уплотнением
  - Если вследствие фрагментации после освобождения не удается отвести объект нужного размера, произвести уплотнение
- Пометка с освобождением для больших объектов

Практический пример: Управление памятью в Monty JVM (SUN Microsystems)

Управление памятью в Monty JVM

- JVM для мобильных устройств
- Организация «кучи»
  - Пометка с уплотнением, 2 поколения
  - Размер 100KB-64MB
  - Пространство непрерывно
  - При поддержке со стороны аппаратуры и ОС может динамически менять размер в соответствии с заданными диапазонами размера и заполненности кучи
- Несколько специалированных областей
  - Для объектов со специальными свойствами (сортов, размеров или времени жизни
  - Границы между областями устанавливаются динамически в соответствии с заданными параметрами
- Единственный нативный поток управления
  - Многоядерность и многопроцессорность редки и потому не используются

Строение кучи Monty JVM

Неподвижные нативные объекты

Динамически загружаемые модули

Кэш скомпилированного кода, временные данные компилятора

Место отведения новых объектов продвижением указателя вверх

Статически загружаемые модули

Специальные области кучи

- Область предзагрузки
  - Содержит вечноживущие большие объекты
  - Не содержит ссылок на другие области кучи
- Область больших объектов
  - Содержит долгоживущие большие объекты
  - Время жизни определяется стартом и завершением задач
  - Не содержит ссылок на другие области кучи
- Компиляторная область
  - Содержит кэш скомпилированного кода и пул временных данных компилятора
  - Порождение и освобождение скомпилированного кода управляются динамическим профилятором
  - Код может быть позиционно-зависим - требовать коррекции при перемещении
  - Все временные данные освобождаются по завершении компиляции.

Загрузка совместно используемых динамически загружаемых Java-модулей

- Если ОС не поддерживает отображение файлов в память, модули загружаются средствами виртуальной машины в область «кучи»
- Модули перемещаемы
- Модуль может содержать ссылки на другие модули, но не на данные в куче
- В однозадачном режиме
  - При старте приложения модули загружаются в неподвижную не собираемую область на дне кучи
- В многозадачном режиме
  - При старте задачи в перемещаемую область на вершине кучи (чтобы избежать большой сборки)
  - Указатели помечены битами в битвекторе

Заголовок объекта

- Варьируется в зависимости от области «кучи»
  - У модуля есть размер, нет типа, перемещаемые указатели помечены в битвекторе
  - Скомпилированные методы все однотипные
  - В пуле временных данных компилятора заголовки не требуются
- В основной «куче»
  - Обычно 1 слово, у массивов 2
  - Представляет класс объекта, а кроме того:
  - При необходимости хеш-код
  - При синхронизации доступа ссылку на ассоциированный с объектом «замок»
  - Во время сборки мусора отсылочный указатель
- Как все это умещается в одном слове?
  - Дополнительная косвенность доступа к классу при использовании хеш-кода или синхронизации

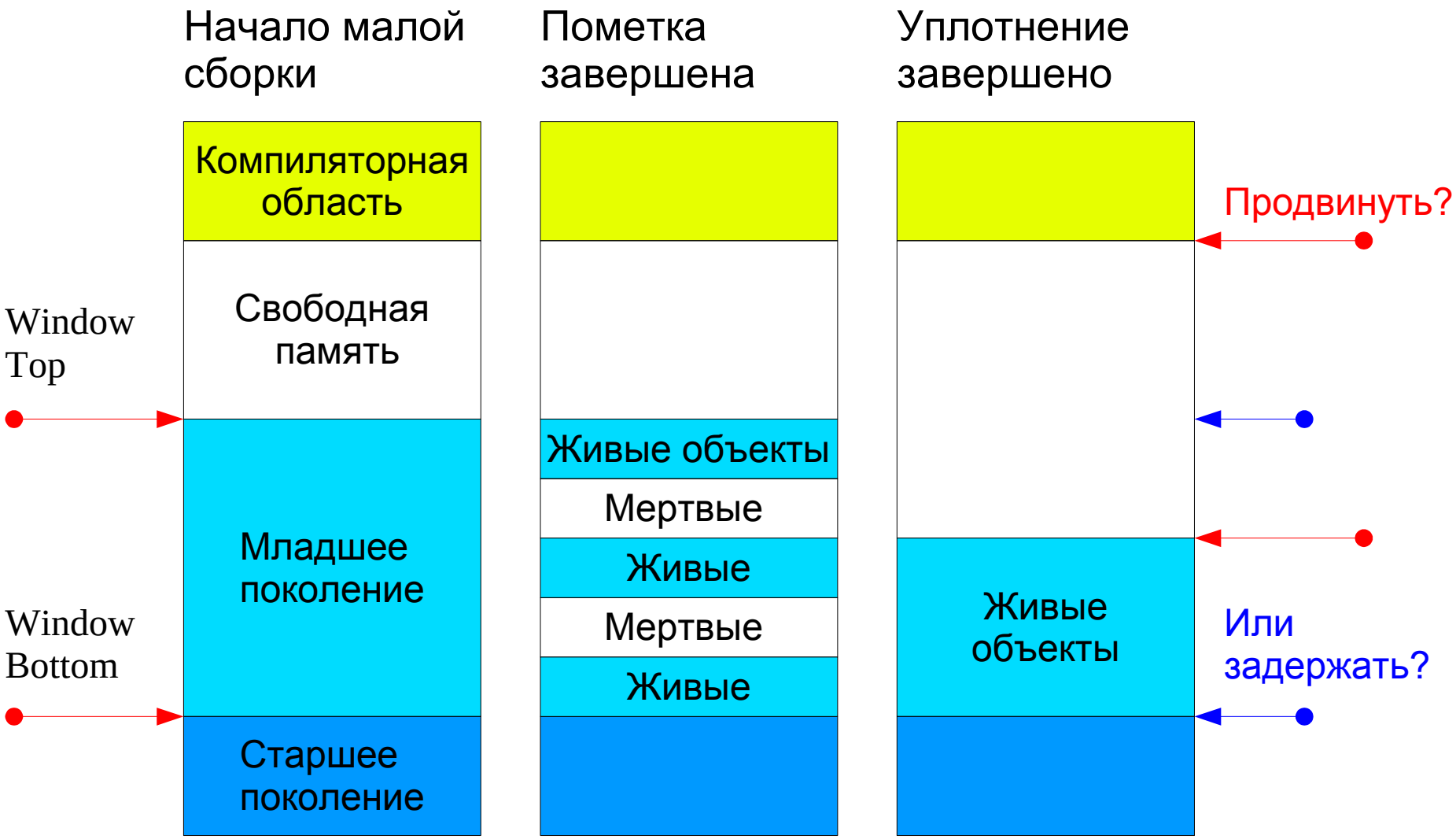
Generational mark'n'compact GC

- Последовательное отведение путем линейного перемещения указателя
  - Варианты зачистки нулями — после отведения или после сборки мусора
  - В данной виртуальной машине используется единственная нативная нить, поэтому нитям не требуется локальных буферов отведения для снижения числа синхронизаций
- Пометка с последующим уплотнением
  - Сохраняет порядок отведения объектов
  - Сжатый отсылочный указатель записывается в свободные биты заголовка объекта
  - Биты пометки хранятся в отдельном битвекторе

Generational mark'n'compact GC (2)

- Два поколения переменного размера
  - Младшее поколение — скользящее окно на вершине старшего поколения
  - Границей между поколениями и размером молодого поколения можно управлять
  - По завершении сборки мусора в младшем поколении его границы могут быть перемещены вверх
  - При этом выжившие объекты оказываются в старшем поколении
  - Указатели запомненного множества помечаются битами в том же битвекторе
- Использование битвектора
  - Вне сборки мусора хранит запомненное множество
  - В начале сборки часть битвектора, соответствующая *окну сборки*, очищается, потом хранит биты пометки
  - При малой сборке окном сборки является младшее поколение, при большой — вся куча

Продвижение объектов из младшего поколения в старшее после малой сборки



Управление младшим поколением

- Promotion policy
  - Определяет после малой сборки мусора, продвигать ли выжившие объекты в старшее поколение или оставить в младшем для последующих сборок
  - *En masse promotion* — продвигаются либо все объекты, либо ни одного
  - Используется эвристика доли памяти, занимаемой выжившими объектами в младшем поколении
  - Если суммарный объем выживших объектов превышает установленный процент, они продвигаются, иначе остаются
  - По умолчанию 15%
  - Если остаются, запомненное множество сохраняется.
  - Если продвигаются, молодое поколение пусто, запомненное множество очищается.
  - Если бы продвигалась часть объектов, прошлось бы ее сканировать для дополнения запомненного множества указателями на оставшуюся часть

Управление младшим поколением (2)

- Sizing policy
  - Верхнюю границу младшего поколения в любой момент можно перемещать вверх и вниз в пределах свободной памяти
  - При фиксированной скорости отведения объектов чем больше младшее поколение, тем больше промежуток между сборками, вероятность смерти объектов до продвижения в старшее поколение и эффективность сборки мусора
  - Но длительность сборки линейно растет с суммарным объемом живых объектов и (с меньшим коэффициентом) размером поколения
  - Размер можно менять, достигая максимума эффективности при заданной максимальной длительности
  - Для этого нужно уметь достаточно точно предсказывать по прошлой статистике объем выживших объектов в ближайшем будущем
  - Увы, в практических приложениях точность этого предсказания слишком низка

Практический пример: Управление памятью IBM J9 JVM

Сборка мусора в IBM J9 (1)

- Семейство виртуальных машин с заменяемыми модулями для разных применений
  - В т.ч. имеется несколько взаимозаменяемых сборщиков мусора, по умолчанию *gencon*
- Сборщик мусора *optthruput*
  - Оптимизирован для максимальной пропускной способности
  - Параллельная пометка с последующим освобождением и возможным уплотнением при высокой фрагментации
- Сборщик мусора *optavgpause*
  - Оптимизирован для минимизации средней длительности пауз
  - Параллельная инкрементальная (*mostly concurrent*) пометка с последующим освобождением
  - При срабатывании барьера работа выполняется непосредственно нитью приложения, а не ставится в очередь к пулу нитей сборщика

Сборка мусора в IBM J9 (2)

- Сборщик мусора *gencon*
  - Оптимизирован для минимизации пауз
  - Вариант *optavgpause* с двумя поколениями
- Сборщик мусора *subpool*
  - Вариант *optthruput* для уменьшения числа блокировок при большом числе процессоров
  - Для каждого процессора выделяется свой пул для отведения памяти внутри общей кучи
- Сборщик мусора Methronome
  - Входит в состав JVM реального времени
  - Оптимизирован для сокращения длительности и увеличения предсказуемости пауз
  - Сегментированная куча
  - Инкрементальная пометка с последующим освобождением и возможным копированием для снижения фрагментации

http://www.ibm.com/developerworks/java/library/j-ibmjava2/index.html?S_TACT=105AGX02&S_CMP=EDU

Практический пример: Управление памятью в Microsoft .NET

Сборка мусора в Microsoft .NET

- Для персональных компьютеров и серверов
  - Варианты общего алгоритма
- Организация «кучи»
  - Переменного размера от 1 GB
  - Сегментированная, размер сегмента 16 MB
  - Исключение — большие объекты, размещаемые в индивидуальных сегментах, не уплотняемые, освобождаемые при утрате ссылок на сегмент
- Пометка с последующими освобождением и частичным уплотнением
  - Поддерживаются неподвижные (pinned) объекты, не перемещаемые при уплотнении
  - Три поколения
  - Сборка в двух младших поколениях производится посегментно
  - Сборка в старшем поколении затрагивает все сегменты и может производиться инкрементально

Конфигурации сборки мусора в Microsoft .NET

- Workstation GC, concurrent GC off
  - Общий алгоритм для всех поколений
- Workstation GC, concurrent GC on
  - Старшее поколение собирается инкрементально
- Server GC
  - Параллельная сборка сегментов (число нитей соответствует числу процессоров)
  - Отдельная куча для каждого процессора

http://msdn.microsoft.com/en-us/magazine/bb985010.aspx

Практический пример: Управление памятью в JRockit JVM (Oracle)

Сборка мусора в JRockit JVM

- Для персональных компьютеров и серверов

  http://egeneration.beasys.com/jrockit/geninfo/diagnos/garbage_collect.html

- Множество вариантов сборки мусора
  - Выбор между инкрементальным или параллельным вариантами пометки с последующим освобождением
  - С поколениями или без
  - С возможным последующим уплотнением наиболее фрагментированных областей кучи
- Поддерживаемые комбинации
  - *singlepar* — параллельная пометка с освобождением
  - *genpar* — вариант *singlepar* с поколениями
  - *singlecon* — инкрементальная пометка с освобождением
  - *gencon* — параллельная сборка для младшего поколения, инкрементальная для старшего

Практический пример: Управление памятью в HotSpot JVM (SUN Microsystems)

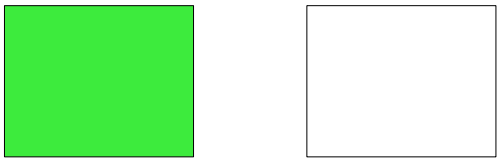
Сборка мусора в HotSpot JVM

- Для персональных компьютеров и серверов

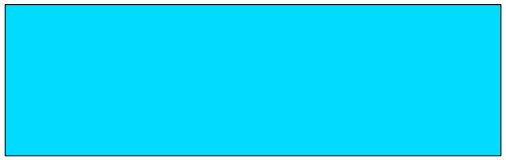
  http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf

- Три основных сборщика мусора
  - Общее строение кучи
  - Все с поколениями
  - В младшем поколении (Survival Spaces) производится копирующая сборка для отсеивания короткоживущих объектов
  - *Serial GC* — пометка с освобождением для персональных компьютеров, оптимизирован для максимальной пропускной способности
  - *Parallel GC* — параллельная пометка с освобождением для серверов, оптимизирован для максимальной пропускной способности
  - *Concurrent GC (CMS)* — инкрементальная пометка с освобождением для серверов

Строение кучи HotSpot JVM

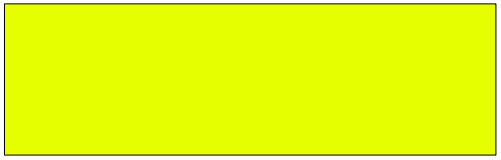


Младшее поколение (Survival Spaces)



Среднее поколение (Eden)

Старшее поколение (OldGen)



Вечное не собираемое поколение (PermGen)

Сборка мусора в HotSpot JVM (2)

- Экспериментальный алгоритм *GarbageFirst* (G1)
  - Цель — пропускная способность при высокой предсказуемости пауз
  - Куча из сегментов равного размера
  - Исключение — большие объекты
  - Параллельная инкрементальная пометка с освобождением и возможным уплотнением
  - Количество участвующих в сборке сегментов определяется в ее начале
  - Сборка может быть прервана и затем восстановлена с утратой работы текущей фазы
  - Для разных сегментов могут использоваться разные алгоритмы

Вопросы?

Sun microsystems
We make the net work.