Лекция 0x10

2 апреля

Итоги
Ключевые правила выделения памяти

- Правила размещения:
  - Первый подходящий, следующий подходящий, наилучший, и др.
  - Компромисс между пропускной способностью и фрагментацией
  - ***Дальнейший материал***: раздельные списки свободных блоков приближение к поиску наилучшего блока без просмотра всего списка свободных блоков
- Правила расщепления:
  - При каких условиях следует расщеплять свободные блоки?
  - До какого уровня может быть доведена внутренняя фрагментация?
- Правила слияния:
  - ***Безотлагательное слияние:*** выполняем слияние каждый раз, когда вызываем функцию `free`
  - ***Отложенное слияние:*** можно попытаться улучшить производительность функции `free`, откладывая слияние на некоторое время. Примеры:
    - Объединяем при просмотре списка свободных блоков во время вызова функции `malloc`
    - Объединяем когда внешняя фрагментация достигает некоторого порогового значения

Итоги
Неявные списки

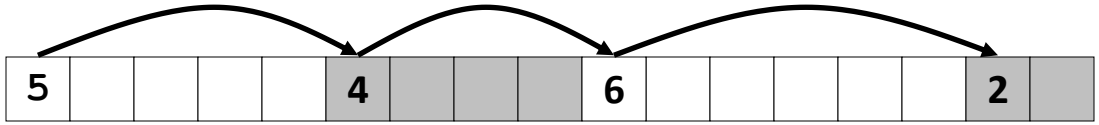
- Реализация: крайне простая
- Стоимость выделения памяти:
  - в худшем случае линейная сложность (время)
- Стоимость освобождения:
  - константное время
  - даже при выполнении слияния!
- Использование памяти:
  - зависит от правил (политики) размещения данных в свободных блоках
  - Первый подходящий, следующий подходящий, или наилучший

- На практике `malloc/free` не используют этот метод по причине линейной сложности, возникающей при выделении памяти
  - используется во многих других случаях

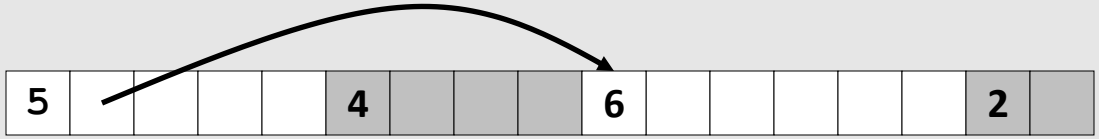
- Тем не менее, идеи расщепления, граничных тегов и слияния используются во *всех* менеджерах динамической памяти

Как отслеживать свободные блоки

- Метод 1: *неявный список* с использованием длины блока



- Метод 2: *Явный список* свободных блоков с использованием указателей



- Метод 3: *Раздельные списки*
  - Распределение блоков по раздельным спискам, исходя из размеров этих блоков

- Метод 4: *Сортировка блоков по размеру*
  - Можно использовать сбалансированное дерево (например, Красно-Черные деревья) с указателями в каждом свободном блоке, и с длиной блока в качестве ключа

Явный список свободных блоков

Занятый блок (как и ранее)

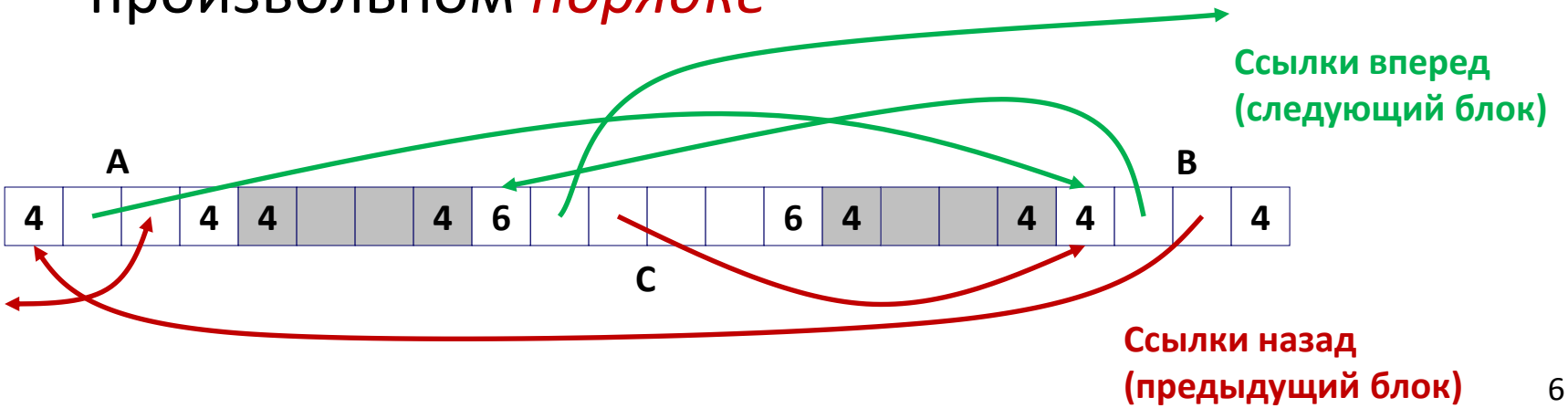
Свободный

- Поддерживаем список (списки) *свободных* блоков, а не *всех* существующих в памяти на данный момент
  - «Следующий» свободный блок может быть где угодно
    - Необходимо поддерживать не только размер текущего блока, но и указатели в оба направления: вперед и назад
  - Граничные теги все также необходимы для слияния
  - Поскольку отслеживаются только свободные блоки, можно хранить указатели в пространстве, отведенном под полезную нагрузку

Явный список свободных блоков

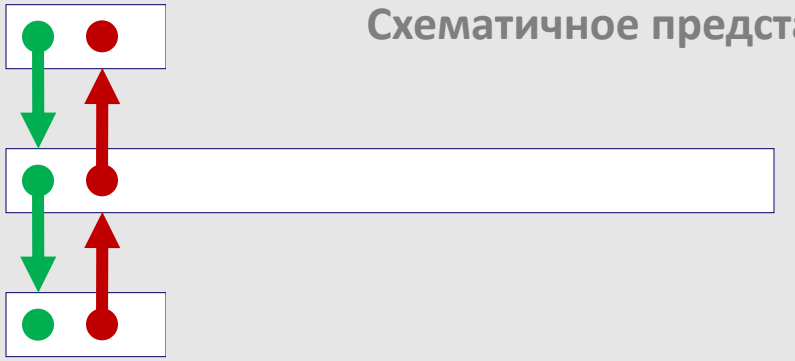
- Логическая организация

- Физическое размещение: блоки могут быть размещены в произвольных *местах* и в произвольном *порядке*



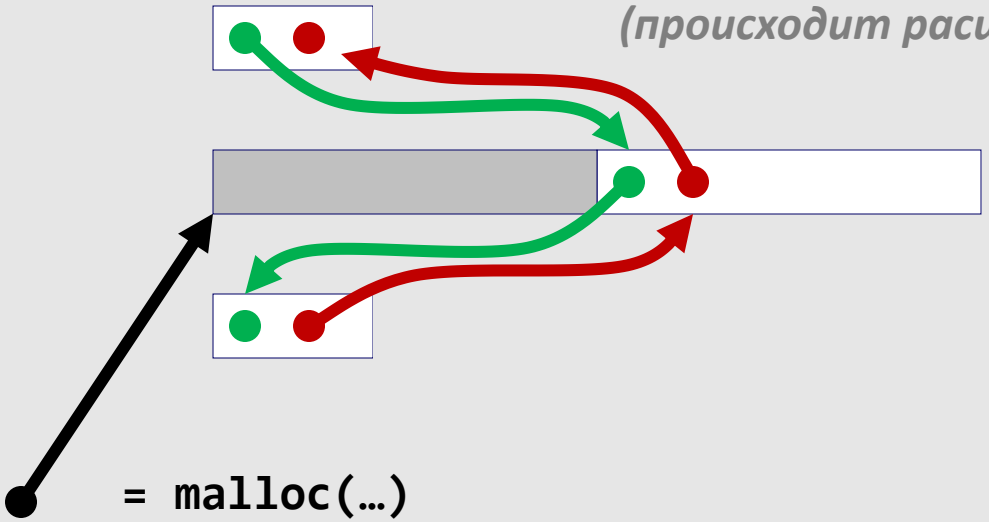
Явный список свободных блоков
Выделение памяти

До



Схематичное представление списка

После



(происходит расщепление блока)

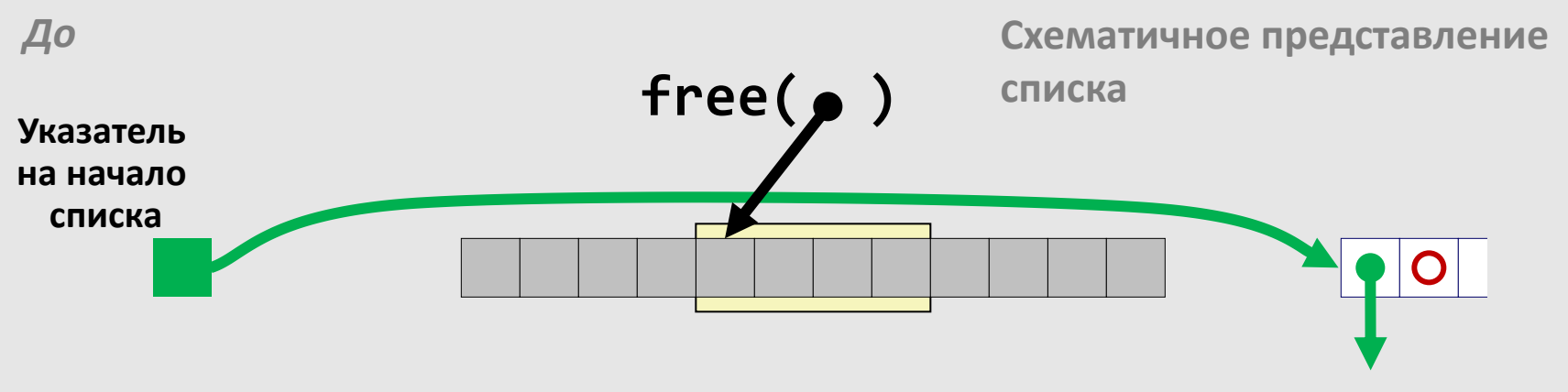
Явный список свободных блоков Освобождение памяти

- *Правила вставки блока*: В какое место списка следует поместить освобожденный блок?
  - **В порядке LIFO (last-in-first-out)**
    - Помещаем освобожденный блок в начало списка
    - ***За:*** простота реализации и константное время работы
    - ***Против:*** Исследования показывают, что возникает более сильная фрагментация по сравнению с тем, когда блоки упорядочены по адресам
  - **В порядке следования адресов**
    - Помещаем в список освобожденный блок так, что список всегда поддерживает упорядоченность по адресам:

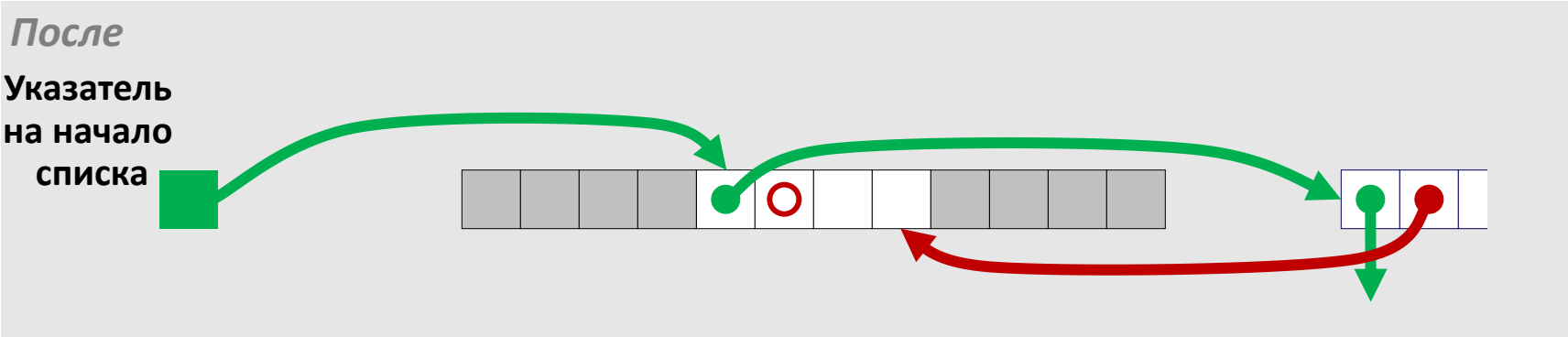
      $$addr(prev) < addr(curr) < addr(next)$$

    - ***Против:*** необходимо искать место вставки
    - ***За:*** см. вопрос фрагментации для дисциплины LIFO

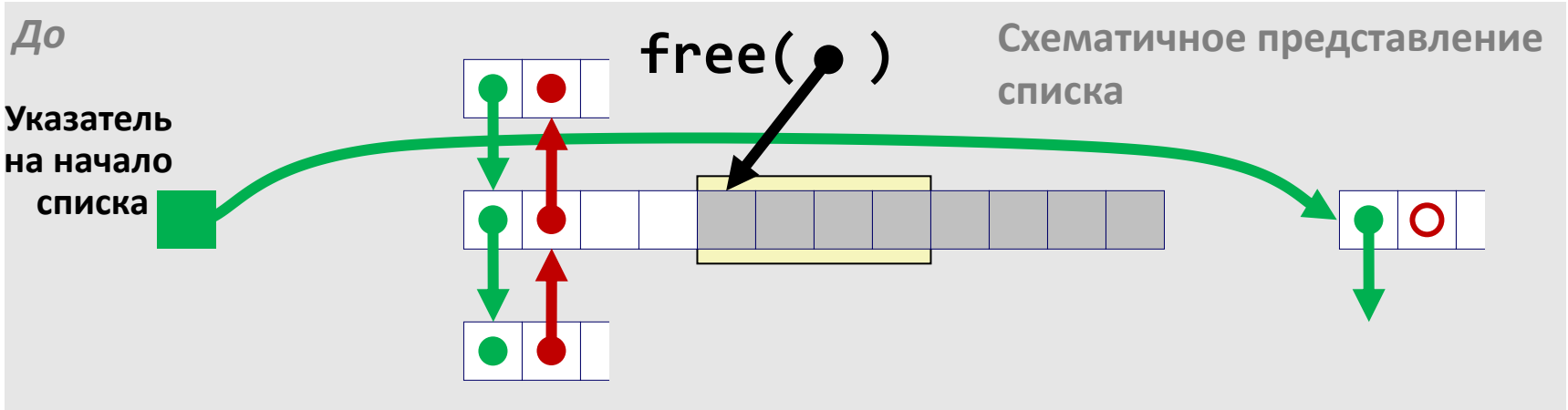
Освобождение блока в порядке LIFO (Случай 1)



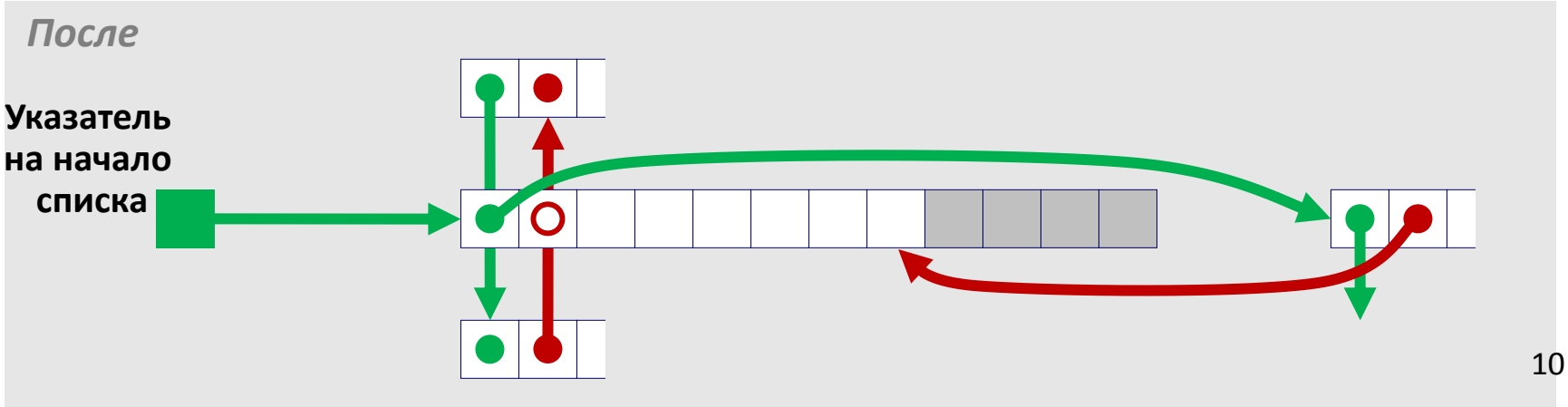
- Помещаем освобожденный блок в начало списка



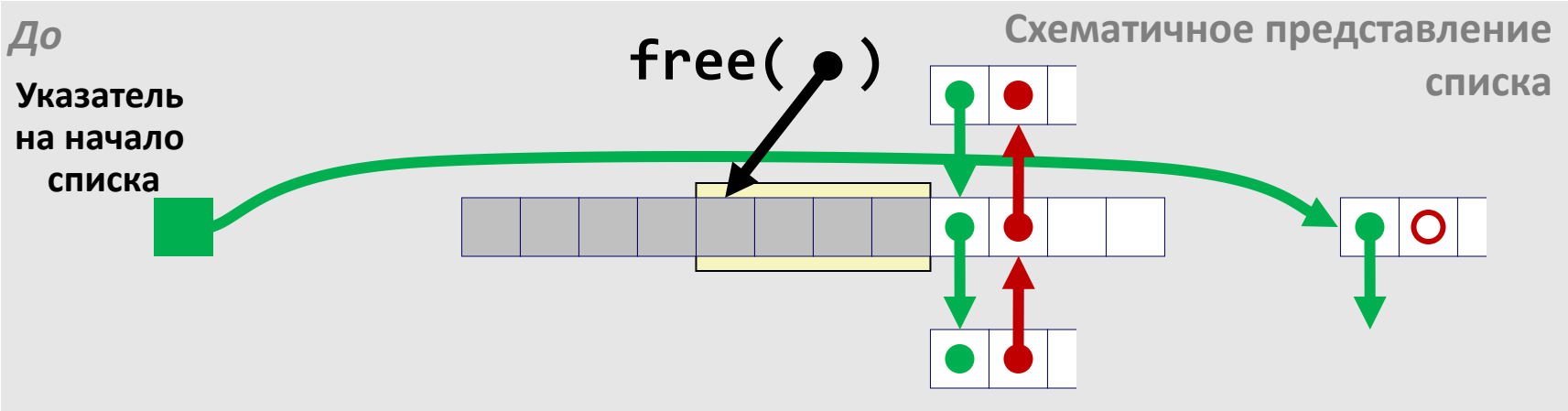
Освобождение блока в порядке LIFO (Случай 2)



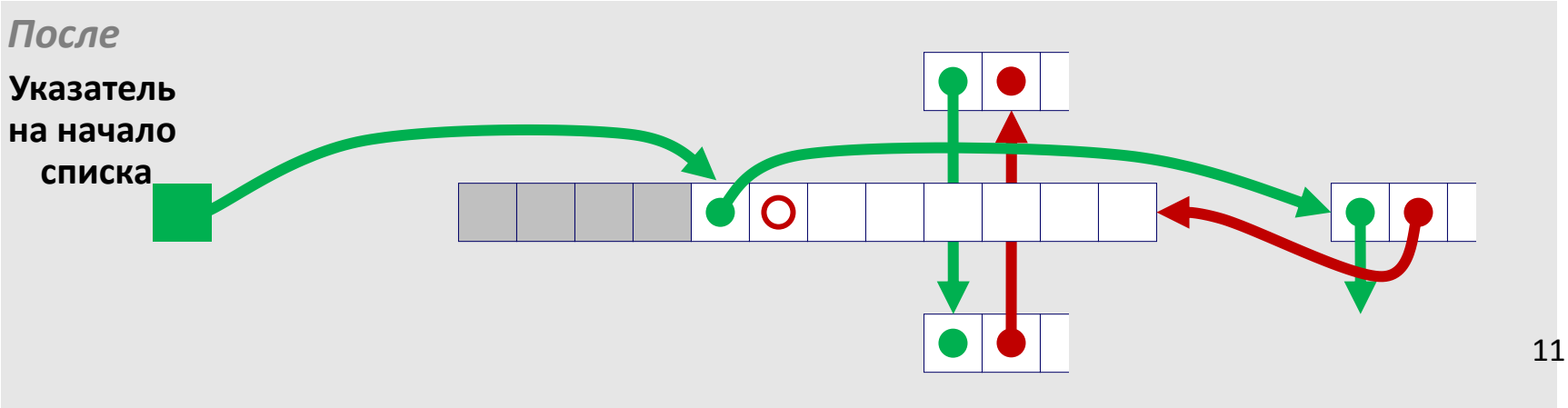
- Извлекаем из списка смежный (перед освобождаемым) в памяти блок, выполняем слияние, и вставляем образовавшийся блок в начало списка



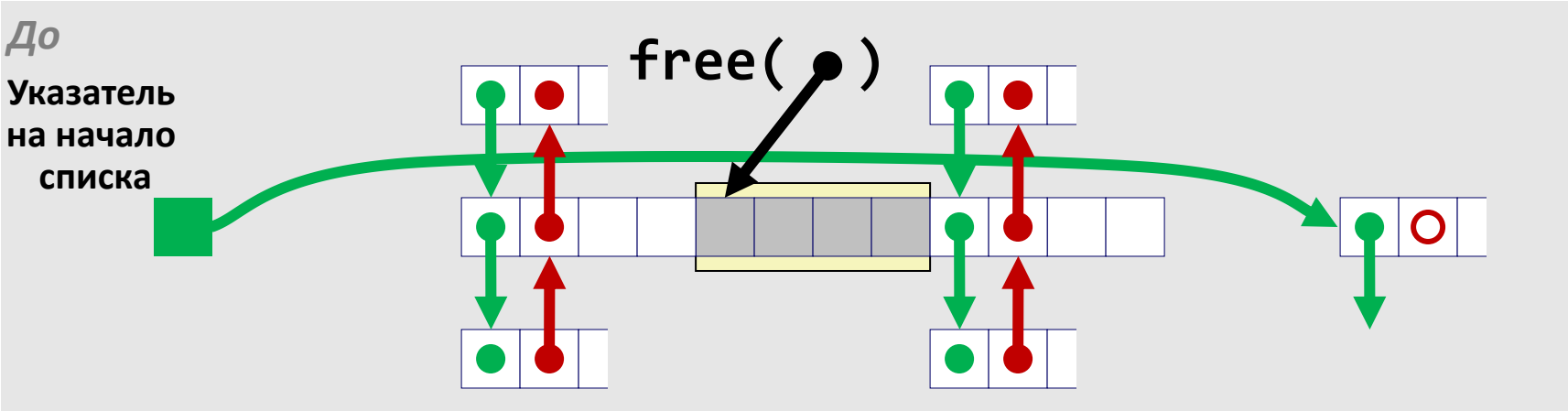
Освобождение блока в порядке LIFO (Случай 3)



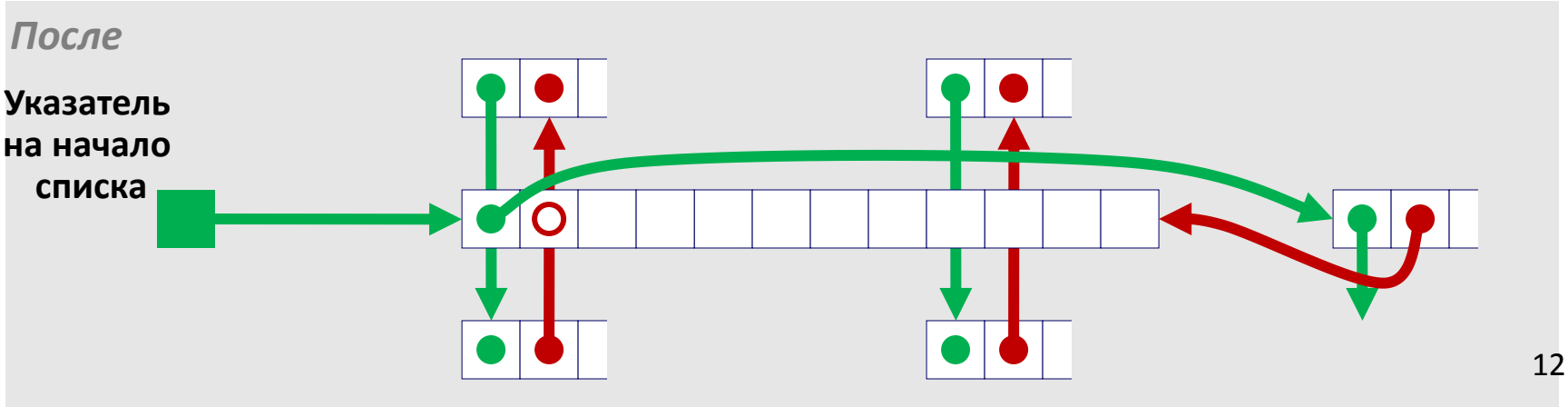
- Извлекаем из списка смежный (после освобождаемого) в памяти блок, выполняем слияние, и вставляем образовавшийся блок в начало списка



Освобождение блока в порядке LIFO (Случай 4)



- Извлекаем из списка смежные блоки, выполняем слияние трех блоков, и вставляем образовавшийся блок в начало списка



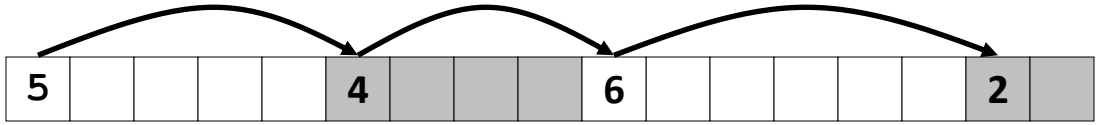
Итоги
Явный список

- В сравнении с неявным списком:
  - Выделение памяти занимает «линейное время» от числа ***свободных***, а не ***всех*** блоков
    - ***Гораздо быстрее*** работает, когда большая часть памяти занята
  - Незначительно усложнилось выделение и освобождение блоков, поскольку необходимо извлекать и добавлять элементы в список
  - Требуется дополнительное место для размещения указателей (2 машинных слова на каждый блок)
    - Увеличивается при этом внутренняя фрагментация?

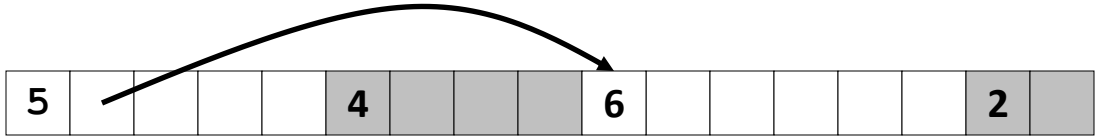
- Как правило подход с поддержкой явного списка комбинируют с разделением блоков по нескольким спискам
  - Блоки разделяют на несколько классов, в зависимости от их размера

Как отслеживать свободные блоки

- Метод 1: *неявный список* с использованием длины блока



- Метод 2: *Явный список* свободных блоков с использованием указателей

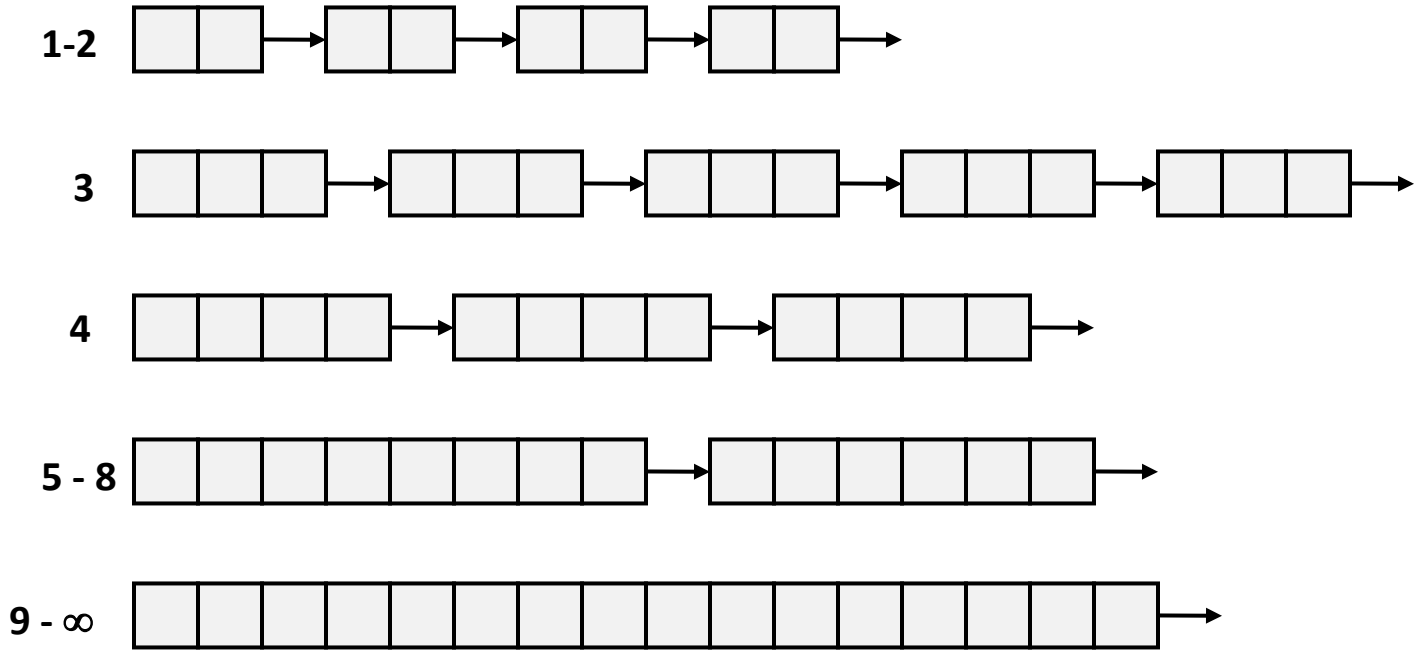


- Метод 3: *Раздельные списки*
  - Распределение блоков по раздельным спискам, исходя из размеров этих блоков

- Метод 4: *Сортировка блоков по размеру*
  - Можно использовать сбалансированное дерево (например, Красно-Черные деревья) с указателями в каждом свободном блоке, и с длиной блока в качестве ключа

Раздельные списки (Seglist)

- Блоки каждого *класса* образуют отдельный список



- Для блоков малого размера заводят по отдельному классу для каждого размера
- Для блоков достаточного большого размера границы классов идут по степеням двойки

Выделение памяти по методу Seglist

- Дан массив список, для каждого класса блоков
- Чтобы выделить блок размера *n* байт:
  - В соответствующем списке ищем блок размера $m > n$
  - Если подходящий блок найден:
    - Расщепляем блок и помещаем оставшийся фрагмент в список соответствующего класса
  - Если блок найти не удалось, ищем его в списке следующего класса
  - Повторяем до тех пор, пока не найдем
- Если после просмотра всех списков блок так и не найден:
  - Запрашиваем у ОС дополнительную память для кучи (используя функцию `sbrk()`)
  - В предоставленной памяти создаем блок размера *n* байт
  - Всю оставшуюся память занимаем одним свободным блоком и помещаем его в список класса наибольших по размеру блоков (из числа подходящих).

Выделение памяти по методу Seglist

- Чтобы освободить блок:
  - При необходимости выполняем слияние и помещаем блок в список подходящего класса размеров
- Преимущества метода Seglist
  - Более высокая пропускная способность
    - Логарифмическая сложность поиска для классов большого размера (граница по степеням двойки)
  - Лучшее использование памяти
    - Поиск первого подходящего в раздельных списках показывает результаты, схожие с поиском наилучшего в рамка всей кучи
    - Предельная ситуация: если для каждого размера блока завести отдельный класс эффективность расходования памяти будет совпадать с поиском наилучшего

Динамическое выделение памяти на стеке

```c
#include <alloca.h>

int f(int dataSize, int iter)
{
   for (int i = 0; i < iter; ++i) {
      char *p = alloca( dataSize );
// выделенная память не будет
// освобождена после закрывающей
// скобки на следующей строке
   }
   return 0;
}
// память, выделенная alloca(),
// освобождается здесь
```

```
f:
   push    ebp
   mov     ebp, esp
   sub     esp, 8
   xor     edx, edx
   mov     eax, dword [ebp+8]
   mov     ecx, dword [ebp+12]
   add     eax, 30
   and     eax, -16
   jmp     .L2
.L3:
   sub     esp, eax
   inc     edx
.L2:
   cmp     edx, ecx
   jl      .L3
   xor     eax, eax
   leave
   ret
```

`alloca` не входит в стандарт языка Си

Далее …

- Явные списки свободных блоков
- Раздельные списки свободных блоков
- ***Сборка мусора***
- Типичные ошибки при работе с памятью

Неявное управление памятью: сборка мусора

- *Сборка мусора:* автоматическое освобождение памяти, выделенной на куче, после того, как программа гарантированно не будет этой памятью пользоваться

```
void foo() {
   int *p = malloc(128);
   return; /* p указывает на блок, ставший «мусором» */
}
```

- Общая практика в функциональных, скриптовых, современных объектно-ориентированных языках:
  - Lisp, ML, Java, Python, Mathematica
- Вариации (“консервативные” сборщики мусора) были созданы для Си и Си++
  - Без гарантий сборки ***всего*** мусора

Сборка мусора

- Каким образом сборщик мусора узнает, что память может быть освобождена?
  - В общем случае невозможно спрогнозировать, что будет происходить в будущем в программе (например, это будет определяться внешними событиями)
  - Но можно однозначно заключить, что блок памяти не будет использоваться, если в программе больше нет ни одного указателя, на этот блок ссылающегося

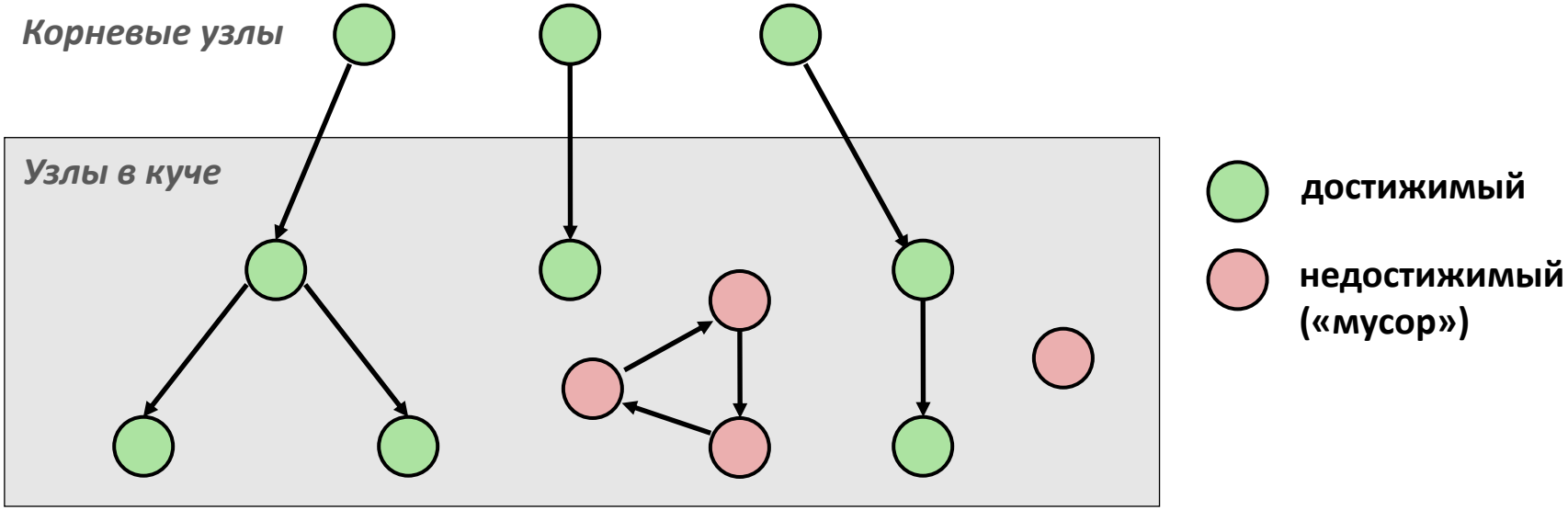
- Требуются некоторые допущения об использующихся в программе указателях
  - Менеджер памяти должен уметь различать указатели и все остальное (целые числа)
  - Все указатели должны указывать на начало блока
  - Недопустимо прятать указатели (например, приведением типа к `int`, а затем обратно к указателю)

Классические алгоритмы сборки мусора

- Алгоритм пометок (Mark-and-sweep, McCarthy, 1960)
  - Не перемещает блоки
- Подсчет ссылок (Collins, 1960)
  - Не перемещает блоки (не рассматривается)
- Сборка копированием (Minsky, 1963)
  - Перемещает блоки (не рассматривается)
- Поколения объектов (Lieberman and Hewitt, 1983)
  - Алгоритм сборки учитывает время жизни объектов (выделенных блоков)
    - Большинство выделенных блоков данных крайне скоро освобождается
    - Поиск недостижимых объектов в первую очередь просматривает пространство памяти с недавно выделенными блоками

Рассматриваем память как граф …

- Рассматриваем память как направленный граф
  - Каждый выделенный блок – вершина графа
  - Каждый указатель – ребро графа
  - Память вне кучи, в которой содержатся указатели, рассматривается в качестве ***корневых*** узлов (регистры, автоматические локальные переменные на стеке, глобальные переменные)

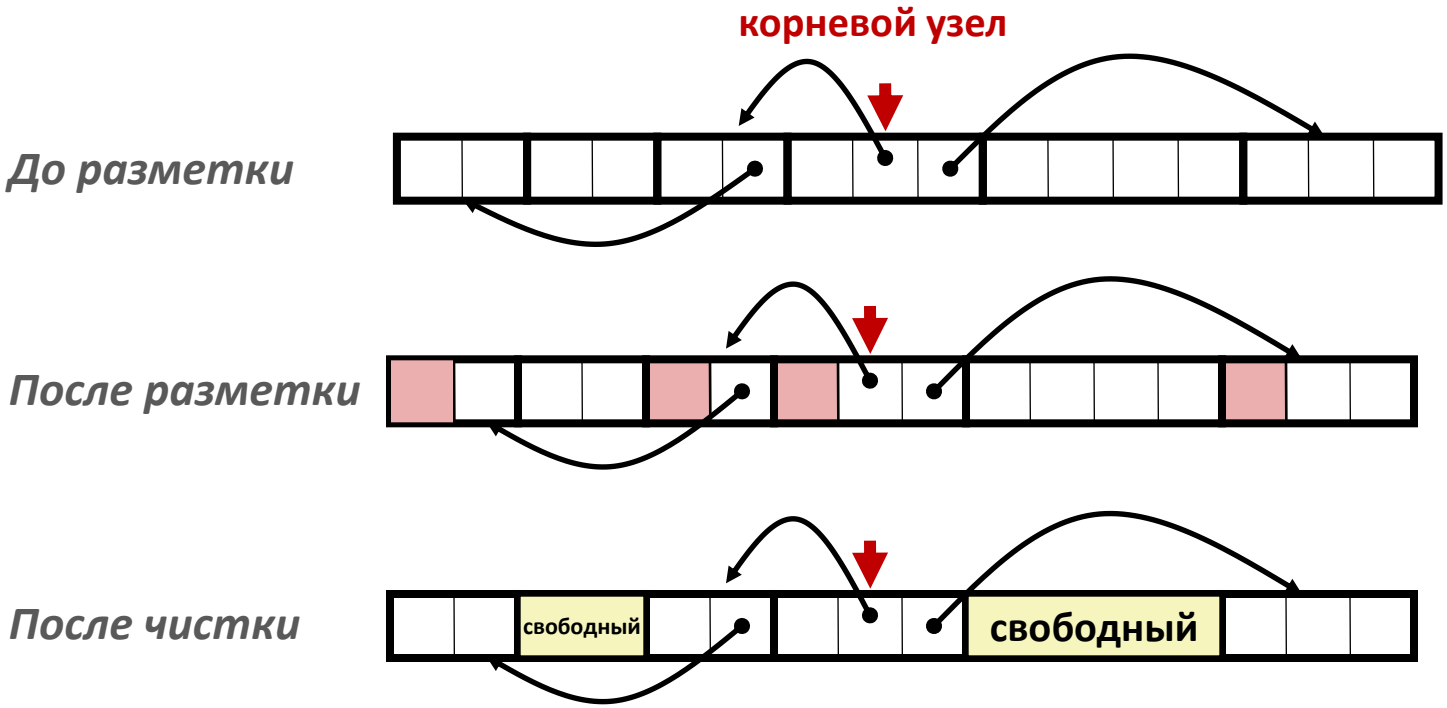


Узел (блок выделенной памяти) *достижим*, если существует путь от некоторого корневого узла до него.

Недостижимые узлы рассматриваются как *мусор* (поскольку не могут быть использованы программой)

Алгоритм пометок

- Может быть реализован поверх функций `malloc/free`
  - Выделяем память функцией `malloc` до тех пор, пока вся память не «кончится»
- Когда свободного места не осталось:
  - Используем дополнительный ***бит пометки*** в заголовке каждого блока
  - ***Разметка:*** Начиная с корневых узлов, проходим по всем достижимым блокам и ставим пометки
  - ***Чистка:*** Просматриваем все блоки и освобождаем все непомеченные



Замечание: в данном примере стрелками обозначены указатели на память, а не только на свободные блоки

Установлен бит пометки

Предположения, необходимые для модельной реализации

- Команды, доступные пользовательской программе
  - **`new(n)`** возвращает указатель на новый блок с пустым содержимым
  - **`read(b,i)`** считывает значение, размещенное по смещению **`i`** в блоке **`b`**
  - **`write(b,i,v)`** записывает величину **`v`** по смещению **`i`** в блок **`b`**

- Каждый блок снабжен заголовком
  - Заголовок (одно слово) адресуется как **`b[-1]`**, для любого блока **`b`**
  - В различных алгоритмах сборки мусора заголовок может использоваться по-разному

- Команды, используемые Сборщиком Мусора
  - **`is_ptr(p)`** определяет, является ли **`p`** указателем
  - **`length(b)`** возвращает длину блока **`b`**, не включая заголовок
  - **`get_roots()`** возвращает список всех корневых объектов

Модельная реализация алгоритма пометок

Помечаем, обходя в глубину граф памяти

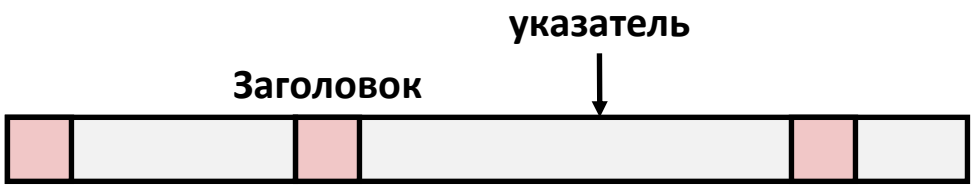
```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // если не указатель, то ничего не делаем
   if (markBitSet(p)) return;       // нет ли уже выставленной пометки?
   setMarkBit(p);                   // выставляем пометку
   for (i=0; i < length(p); i++)    // рекурсивно вызываем на всех словах
     mark(p[i]);                    // данного блока
   return;
}
```

Чистим, используя функцию длины для перехода на следующий блок

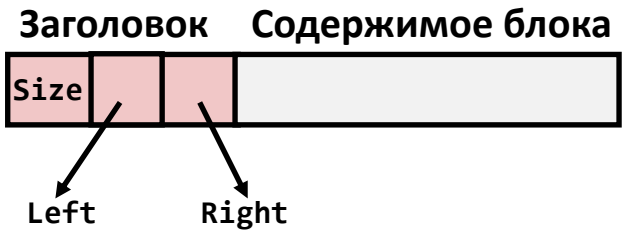
```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

Консервативный алгоритм пометок для Си программ

- «Консервативный сборщик мусора» для Си программ
  - **`is_ptr()`** определяет, что машинное слово – указатель, через проверку, является ли это число начальным адресом выделенного блока памяти
  - Но в языке Си указатели могут указывать в середину блока



- Как можно найти начало блока?
  - Можно поддерживать в памяти сбалансированное двоичное дерево для отслеживания всех выделенных блоков памяти (ключом является адрес начала блока)
  - Указатели сбалансированного дерева могут храниться в заголовке (потребуются два дополнительных слова)



`Left`: меньшие адреса
`Right`: большие адреса

Далее …

- Явные списки свободных блоков
- Раздельные списки свободных блоков
- Сборка мусора
- ***Типичные ошибки при работе с памятью***

Типичные ошибки при работе с памятью

- Разыменование дефектного указателя
- Чтение неинициализированной памяти
- Перезапись памяти
- Ссылки на не существующие переменные
- Многократное освобождение блоков памяти
- Ссылки на уже освобожденные блоки
- Ошибки, связанные с освобождением блоков

Операции в языке Си

Операторы	*Ассоциативность*
`() [] -> .`	слева направо
`! ~ ++ -- + - * & (type) sizeof`	справа налево
`* / %`	слева направо
`+ -`	слева направо
`<< >>`	слева направо
`< <= > >=`	слева направо
`== !=`	слева направо
`&`	слева направо
`^`	слева направо
`|`	слева направо
`&&`	слева направо
`||`	слева направо
`?:`	справа налево
`= += -= *= /= %= &= ^= != <<= >>=`	справа налево
`,`	слева направо

`*p->`

- `->`, `( )`, и `[ ]` имеют более высокий приоритет, чем `*` и `&`
- Унарные операции `+`, `-`, и `*` имеют больший приоритет, нежели их бинарные аналоги

Источник: «Керниган & Ричи»

Объявление Си указателей: самопроверка

`int *p`	`p` – указатель на `int`
`int *p[13]`	`p` – массив из `[13]` указателей на `int`
`int *(p[13])`	`p` – массив из `[13]` указателей на `int`
`int **p`	`p` – указатель на указатель на `int`
`int (*p)[13]`	`p` – указатель на массив из `[13]` элементов `int`
`int *f()`	`f` – функция, возвращающая указатель на `int`
`int (*f)()`	`f` – указатель на функцию, возвращающую `int`
`int (*(*f())[13])()`	`f` – функция возвращающая указатель на массив `[13]` указателей на функции, возвращающие `int`
`int (*(*x[3])())[5]`	`x` – массив `[3]` указателей на функции, возвращающие указатели на массив `[5]` элементов `int`

Источник: «Керниган & Ричи» раздел 5.12

Разыменование дефектного указателя

- Классическая ошибка при использовании функции `scanf`

```
int val;

...

scanf(“%d”, val);
```

Чтение неинициализированной памяти

- Ошибочное предположение, что память, полученная из кучи, предварительно была заполнена нулями

```
/* вычисляем y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

Перезапись памяти

- Выделение памяти с неправильным определением размеров (в некоторых случаях)

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Что будет, если такой код перенести с IA-32 на x86-64?

Перезапись памяти

- Ошибка диапазона

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Перезапись памяти

- Не проверяется превышение максимального размера вводимой строки

```
char s[8];
int i;

gets(s);  /* читаем “123456789” со стандартного входа */
```

- Основа для реализации классической атаки «переполнение буфера»

Перезапись памяти

- Неправильное понимание адресной арифметики

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

Перезапись памяти

- Использование указателя вместо объекта, на который он указывает

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

Ссылки на не существующие переменные

- Автоматические локальные переменные перестают существовать после выхода из функции

```
int *foo () {
   int val;

   return &val;
}
```

Многократное освобождение блоков памяти

- Фу!

```
x = malloc(N*sizeof(int));
      // что-то делаем с x
free(x);

y = malloc(M*sizeof(int));
      // что-то делаем с y
free(x);
```

Ссылки на уже освобожденные блоки

- Фу-фу-фу!

```
x = malloc(N*sizeof(int));
   // что-то делаем с x
free(x);
   ...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Ошибки, связанные с освобождением блоков (утечка памяти)

- Долгое, но верное «убийство» программы

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```

Ошибки, связанные с освобождением блоков (утечка памяти)

- Освобождаем только часть сложно устроенной структуры данных

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head = malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   // создаем и что-то делаем с остальной частью списка
    ...
   free(head);
   return;
}
```

И что же со всем этим делать?

- Традиционный отладчик (gdb)
  - Удобно находить разыменование дефектных указателей
  - Совсем неудобно выявлять все остальные виды ошибок

При срабатывании ошибки крайне важна выдаваемая диагностика!

- Отладочная версия `malloc`
  - Обертка вокруг обычной функции `malloc`
  - Выявление ошибок в рамках функций `malloc` и `free`
    - Повреждение внутренних структур кучи в результате перезаписи
    - В некоторых случаях – многократное освобождение памяти
    - Утечки памяти
  - Не может выявлять все остальные ошибки
    - Перезаписи внутри выделенных блоков
    - Использование уже освобожденных блоков
    - …

И что же со всем этим делать?

- Некоторые реализации malloc уже содержат код дополнительных проверок
  - glibc malloc в ОС Linux: `setenv MALLOC_CHECK_ 2`
  - FreeBSD: `setenv MALLOC_OPTIONS AJR`
- Программные инструменты динамического анализа
  - valgrind (ОС Linux), Purify
    - Двоичная трансляция
  - Работают с исполняемы кодом программы
  - Может выявлять все ошибки, что и отладочный `malloc`
  - А также проверяет все обращения к памяти во время выполнения программы
    - Дефектные указатели
    - Перезапись памяти
    - Ссылка вне выделенных блоков памяти

- ***Дают детальную диагностику, если ошибка сработала***
- ***Нужны начальные данные, на которых ошибки будут проявляться***

И что же со всем этим делать?

- Автоматическая сборка мусора
  - Консервативный сборщик Boehm-GC для Си/Си++
- Статический анализ исходного кода
  - Легковесный анализ
    - Встраивается в среду разработки
  - Глубокий анализ: построение потоков управления, данных, межпроцедурный анализ всего кода программы, и т.д.
    - Встраивается в систему сборки

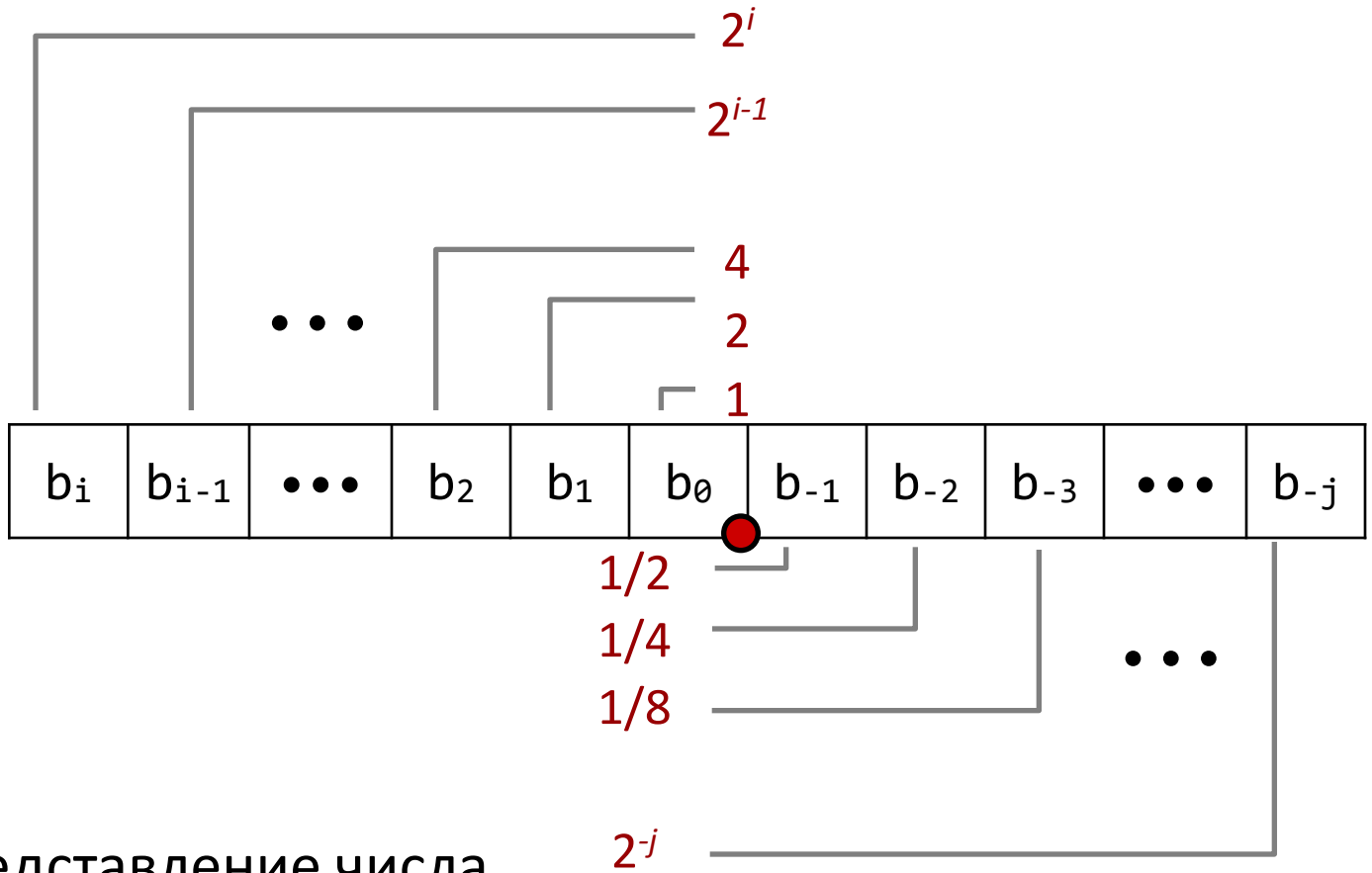
- ***Анализ всех возможных путей выполнения при произвольных входных данных***
- ***Для понимания диагностики требуется хорошая квалификация***

- На практике профессиональные разработчики программ комбинируют всё перечисленное
  - А также кое-что еще…

Далее ...

- ***Динамическая память***
  - ***Организация и управление***
  - ***Численные характеристики***
  - ***Управление свободными блоками***
  - ***Сборка мусора***
  - ***Типовые ошибки в Си программах при работе с памятью***
- ***Числа с плавающей точкой***
  - ***Представления для вещественных чисел***
    - ***Дробные двоичные числа***
    - ***Числа с плавающей точкой***
  - ***Сопроцессор x87***
    - ***Устройство***
    - ***Примеры программ***

Дробные двоичные числа



- Представление числа
  - Биты справа от “двоичной точки” представляют отрицательные степени 2
  - Точное представление для рациональных чисел вида : $\sum_{k=-j}^{i} b_k \times 2^k$

Примеры дробных двоичных чисел

Число	Представление
5 3/4	101.11_2
2 7/8	10.111_2
63/64	0.111111_2

- Деление на 2 может выполняться сдвигом вправо, …
- … а умножение на 2 – сдвигом влево
- Числа вида $0.11111\ldots_2$
  - На один «шаг» меньше чем `1.0`
  - Используется специальное обозначение `1.0 – ε`

Представимые рациональные числа

- Ограничение
  - Можно представить рациональные числа только вида $x/2^k$
  - Другие рациональные числа представляются повторяющимися группами бит

- Число Представление

1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

Представление чисел с плавающей точкой

- Численное представление

$$(-1)^s \times M \times 2^E$$

  - Знаковый бит s определяет, является число положительным или отрицательным
  - Мантисса M – дробное число в полуинтервале [1.0,2.0).
  - Порядок E определяет степень 2 в третьем множителе

- Кодировка
  - Наибольший значащий бит s – знаковый бит s
  - Поле exp кодирует порядок E
  - Поле frac кодирует мантиссу M

Размеры чисел

- Одинарная точность: 32 бита. Тип – `float`.
  - Знак s 1 бит
  - Мантисса M 23 бита
  - Порядок E 8 битов
- Двойная точность: 64 бита. Тип – `double`.
  - Знак s 1 бит
  - Мантисса M 52 бита
  - Порядок E 11 битов
- Нормализация чисел
  - Нормализованное значение – порядок не принимает «крайние» значения (одни нули или одни единицы)
  - Денормализованное значение – порядок либо ноль, либо 11…11

Нормализованное число

- Значение: `float f = 15213.0;`

  $15213_{10} = 11101101101101_2$

  $= 1.1101101101101_2 \times 2^{13}$

- Мантисса

  M = 1.1101101101101_2

  frac = $11011011011010000000000_2$

- Порядок

  E = 13

  Смещение = 127

  Exp = E + Смещение = 140 = 10001100_2

- Итого:

0	10001100	11011011011010000000000
s	**exp**	**frac**