Лекция 0xF

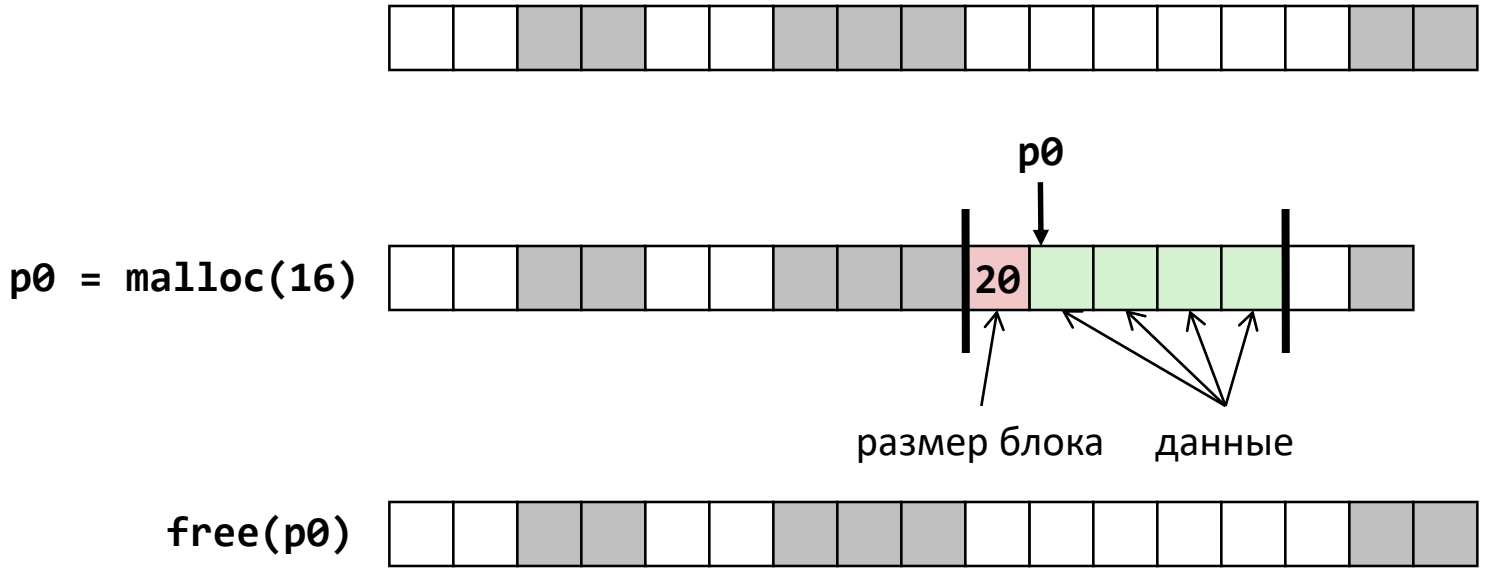
6 апреля

Проблемы реализации менеджера памяти

- Как следует запоминать, сколько памяти должно быть освобождено для данного адреса?
- Как лучше поддерживать информацию о свободных блоках?
- Если принято решение выделить блок большего размера, чем было запрошено, что делать с лишней памятью?
- Какой блок лучше выбрать для выделения?
- Как лучше распорядиться освобожденным блоком?

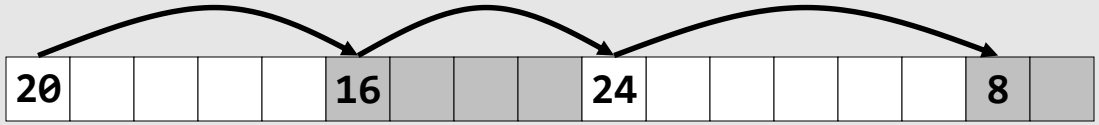
Сколько освобождать?

- Стандартный метод
  - Размещаем длину блока в слове, предшествующем блоку.
    - Такое слово называют ***заголовком***
  - Требуется дополнительное слово на каждый выделяемый блок

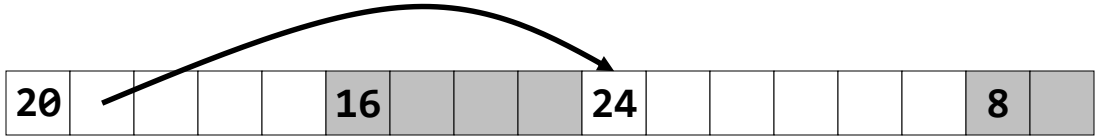


Как отслеживать свободные блоки

- 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

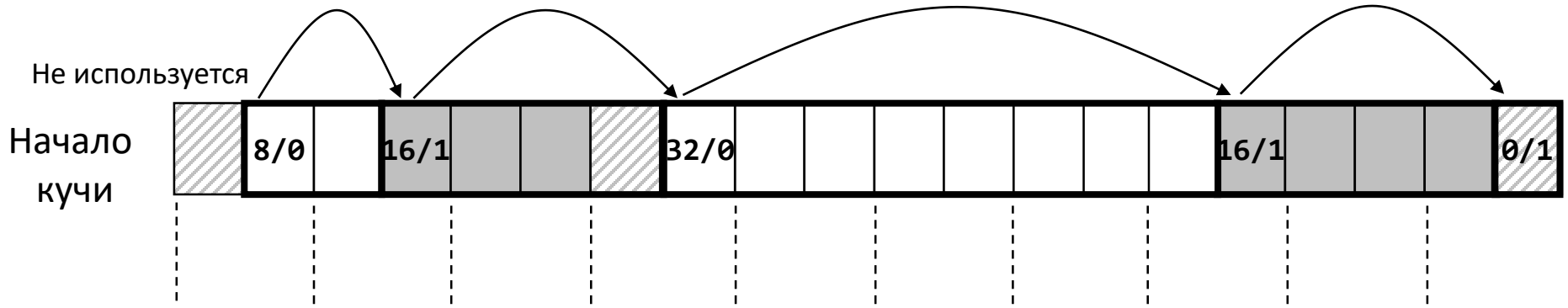
Метод 1: Неявный список

- Для каждого блока необходимо знать его длину и состояние - выделен/свободен
  - Расточительно использовать для этого два слова
- Стандартный прием
  - Если блоки выровнены в памяти, несколько младших битов адреса всегда 0, а размер блока кратен некоторой степени двойки
  - Вместо 0 храним в младшем бите заголовка флаг, выделен или свободен блок
  - Когда заголовок интерпретируется как размер блока, младший бит маскируется

Формат выделенных и свободных блоков

a = 1: блок занят
a = 0: блок свободен

Пример
Неявный список



Выровнено по границе двойного слова (8 байт)

Выделенные блоки: серая заливка
Свободные блоки: белое
Заголовки: обозначены размером в байтах /битом выделения

Неявный список
Поиск свободного блока

- *Первый подходящий:*
  - Проходим список с начала, выбираем ***первый*** подходящий блок:

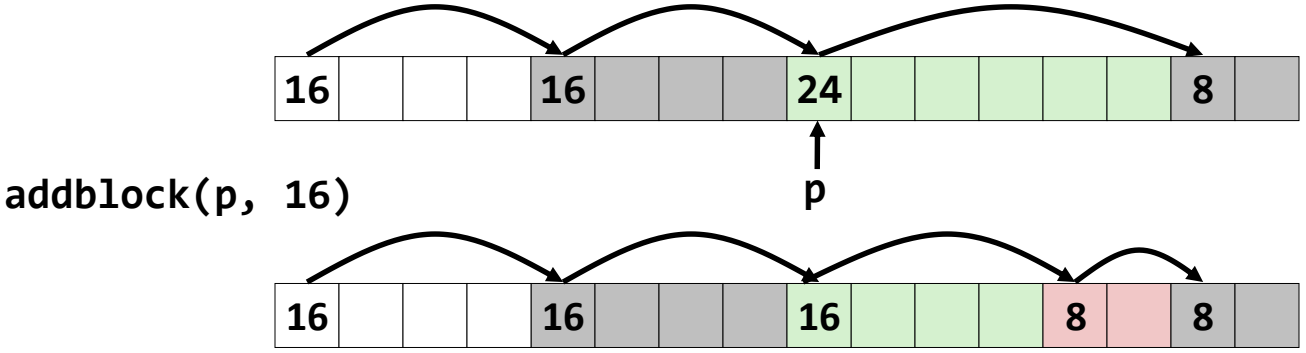
```
p = start;
while ((p < end) &&      \\ пока не дошли до конца
       ((*p & 1) ||      \\ уже выделен
       (*p < len)))      \\ маловато будет
  p = p + (*p & -2);     \\ переходим на следующий блок
```

Здесь и далее приводится Си-подобный псевдокод

  - Выделение за линейное время
  - На практике может вызывать «дробление» блоков в начале списка
- *Следующий подходящий:*
  - Аналогично предыдущему, поиск продолжается с позиции на которой он остановился ранее
  - Как правило работает быстрее: не происходит повторного просмотра неподходящих блоков
  - Некоторые исследования допускают худшую фрагментацию
- *Наилучший:*
  - Просмотр всего списка, выбор ***наилучшего*** свободного блока
    - меньше всего байт сверх запрошенного размера
  - Небольшой размер незанятых фрагментов
  - Как правило, работает медленнее, чем *первый подходящий*

Неявный список
Выделение свободного блока

- Выделение свободного блока: *расщепление*
  - Если размер требуемой памяти меньше, чем доступное в свободном блоке пространство, блок можно расщепить



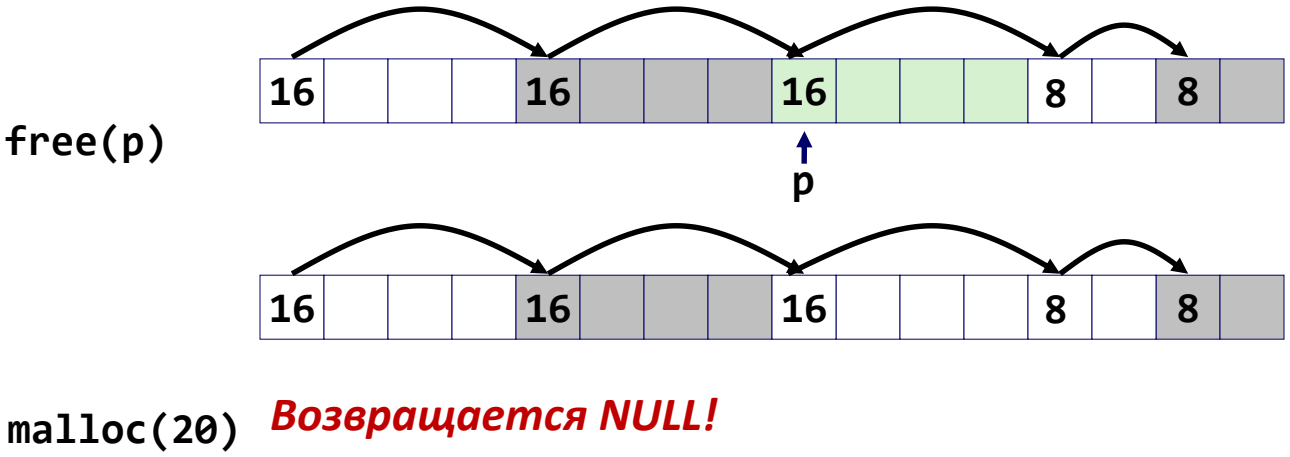

```
void addblock(ptr p, int len) {
  int newsize = ((7 + len) >> 3) << 3;   // «выравниваем вверх» по
                                         // 8 байтной границе
  int oldsize = *p & -2;                 // маскируем и считываем размер
  *p = newsize | 1;                      // выставляем новую длину блока
  if (newsize < oldsize)
    *(p + newsize) = oldsize - newsize;  // выставляем длину нового блока
}
```

Неявный список
Освобождение блока

- Невероятно простая реализация!
  - Всего лишь нужно сбросить флаг, показывающий выделен блок или свободен

```
void free_block(ptr p) { *p = *p & -2 }
```

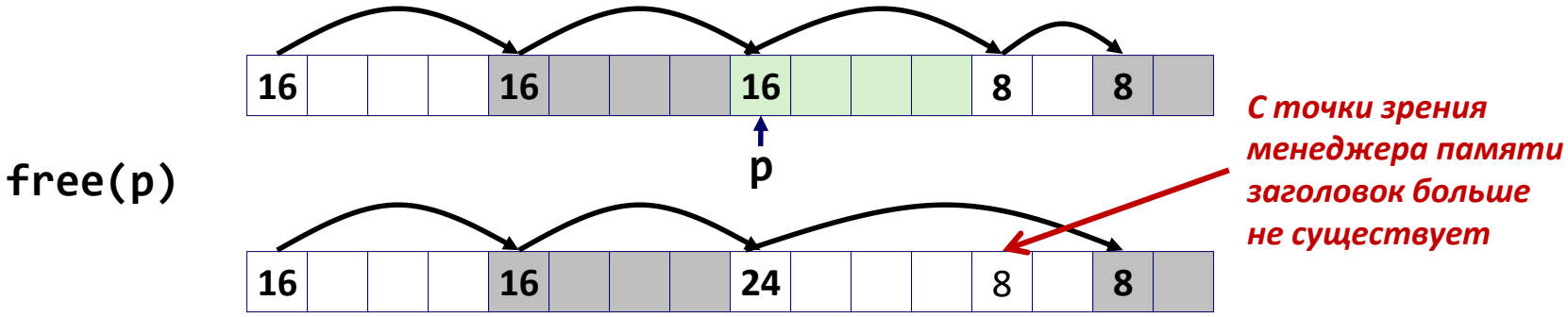
  - К сожалению, приходим к «ложной фрагментации»



Несмотря на то, что свободное пространство есть, менеджер памяти его не в состоянии найти

Неявный список
Слияние

- Объедение *(слияние)* со следующим/предыдущим блоком, если он свободен
  - Слияние со следующим блоком

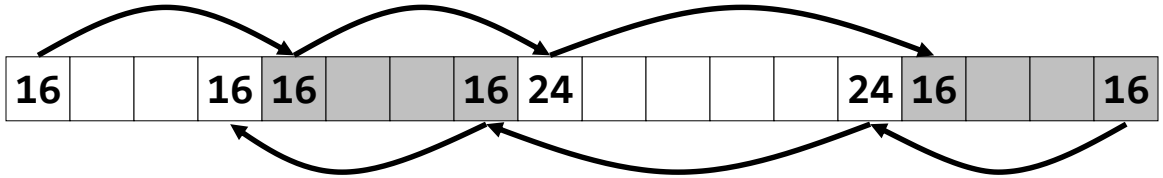



```
void free_block(ptr p) {
    *p = *p & -2;              // сбрасываем флаг
    next = p + *p;             // находим следующий блок
    if ((*next & 1) == 0)      // если он свободен
      *p = *p + *next;         // добавляем его к текущему блоку
}
```

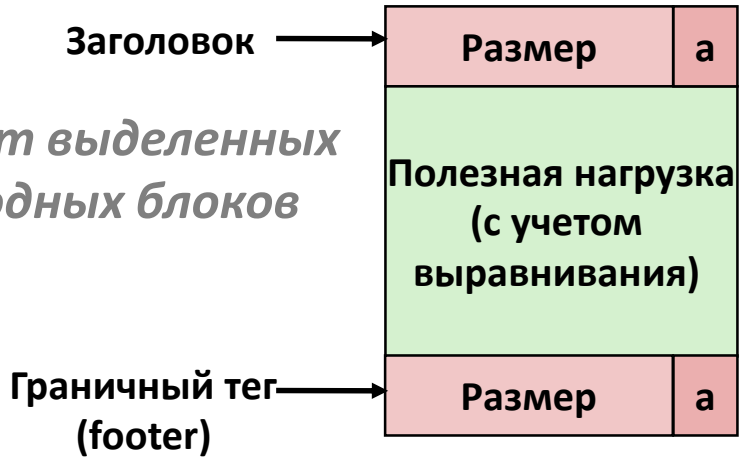
  - Как провести слияние с *предыдущим* блоком?

Неявный список
Двунаправленное слияние

- *Граничные теги* [Кнут 73]
  - Повторяем заголовок (размер/флаг) в конце блока
  - Появляется возможность проходить список в обратном направлении за счет дополнительного расходования памяти
  - Общеупотребительный технический прием



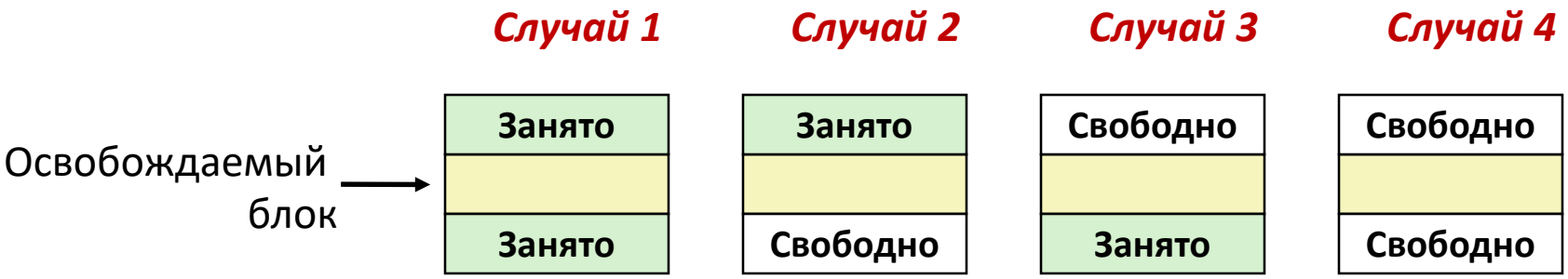
Формат выделенных и свободных блоков



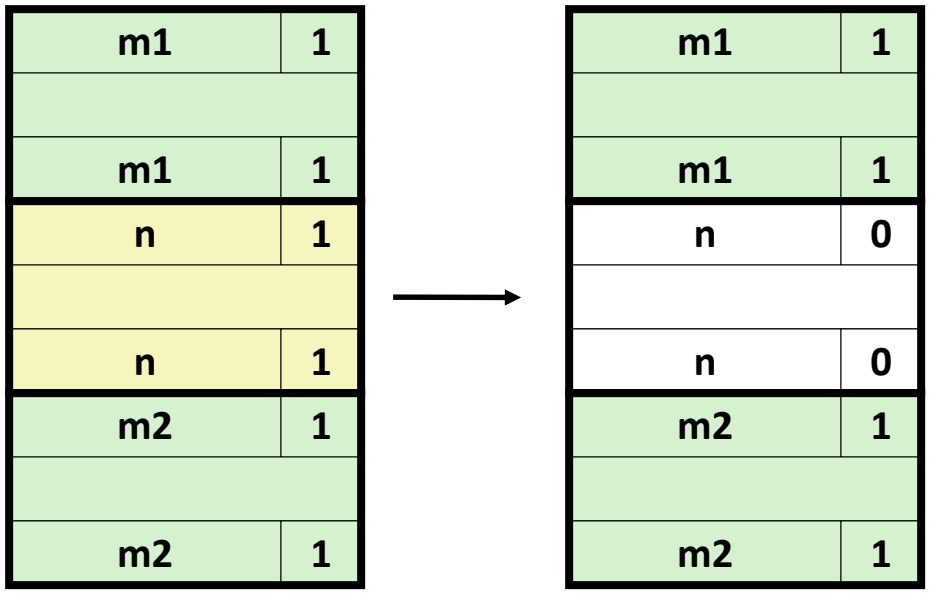
a = 1: Блок занят
a = 0: Блок свободен

Размер: Общий размер всего блока

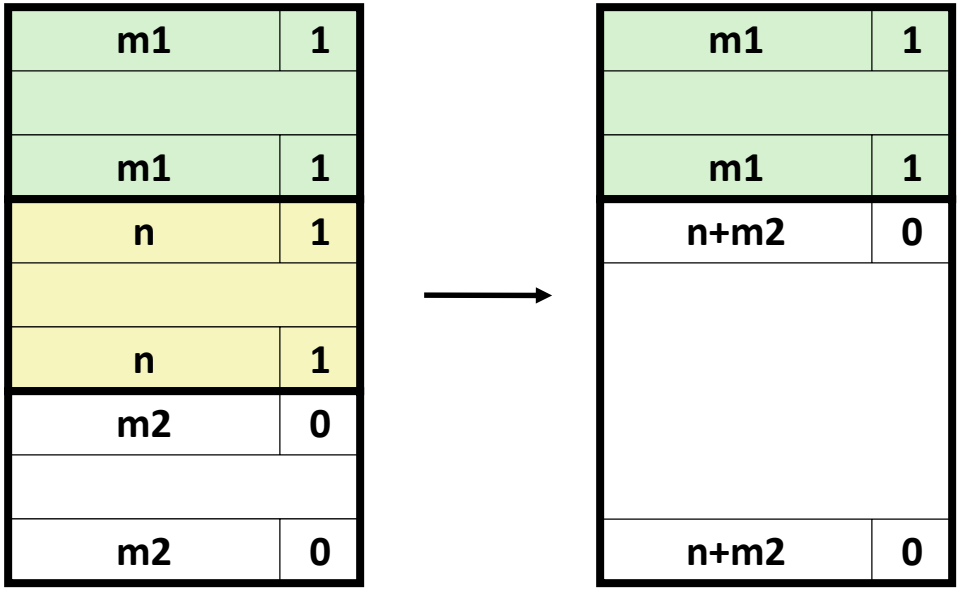
Слияние за фиксированное время



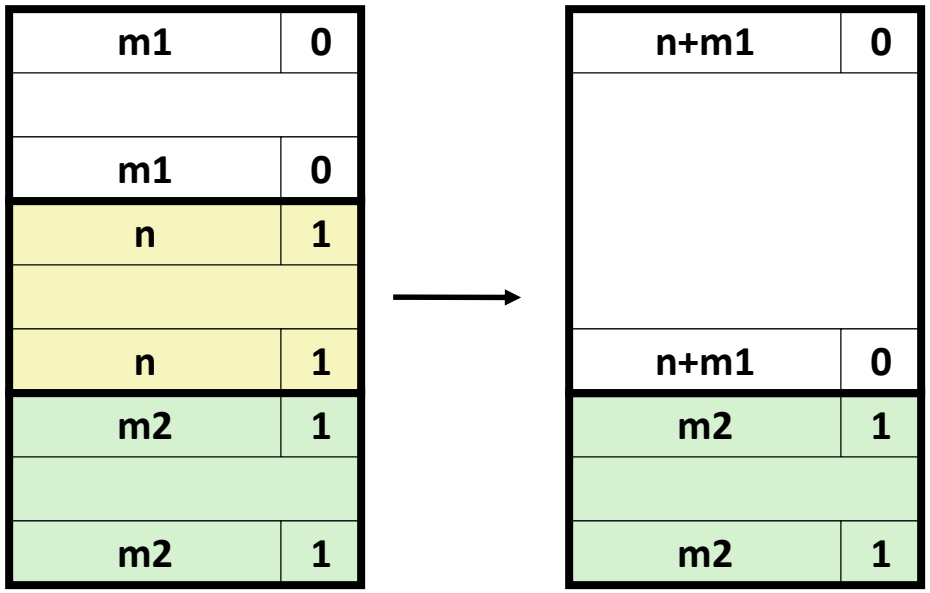
Слияние за фиксированное время (Случай 1)



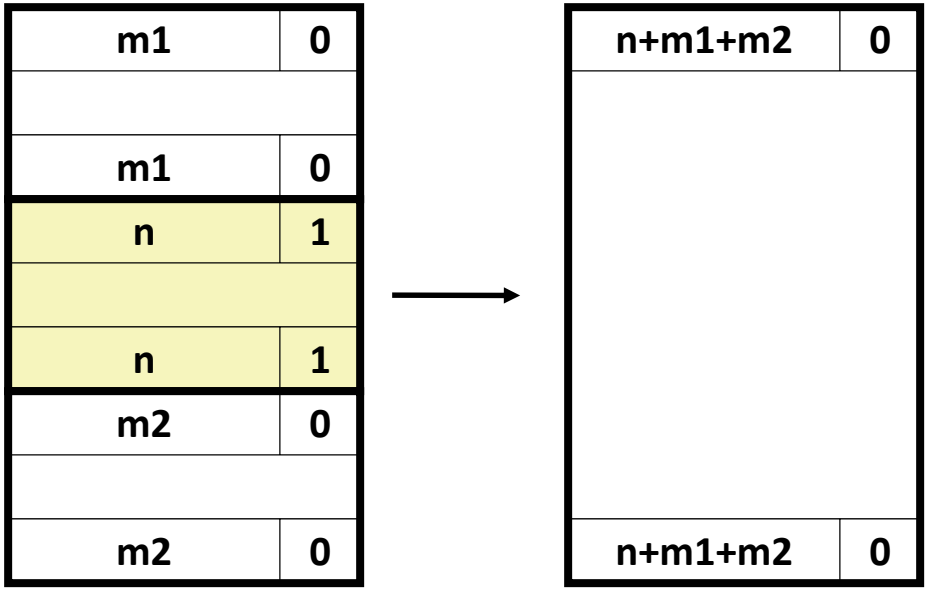
Слияние за фиксированное время (Случай 2)



Слияние за фиксированное время (Случай 3)



Слияние за фиксированное время (Случай 4)



Недостатки Граничных Тегов

- Внутренняя фрагментация

- Есть ли возможности для оптимизации?
  - Каким блокам нужен тег нижней границы?
  - … И что это значит?

Промежуточные 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

Промежуточные 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

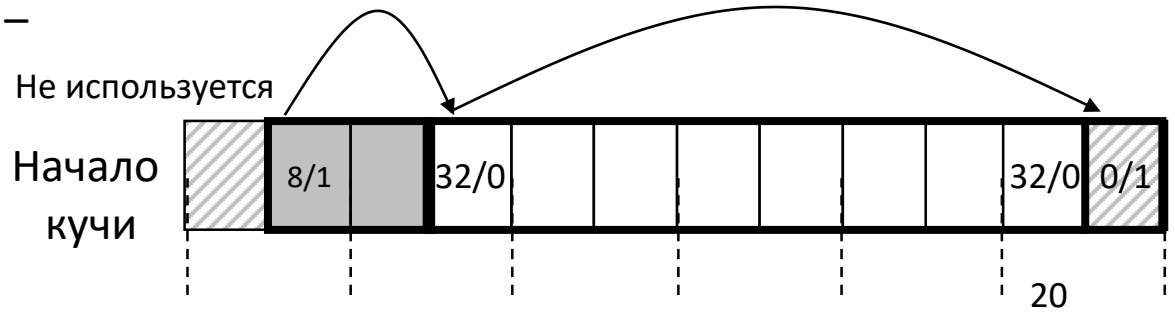
- Тем не менее, идеи расщепления, граничных тегов и слияния используются во *всех* менеджерах динамической памяти

Задача на моделирование работы менеджера динамической памяти

- Размер кучи – 12 четырехбайтных слов
- Неявный список
- Размер в заголовке и граничном теге
- В выделенных блоках граничный тег не используется
- Память выравнивается по 8-ми байтной границе
- Поиск свободного блока:
  с начала / с текущей позиции
- Выбирается первый подходящий свободный блок
- При расщеплении используется 1я часть
- Слияние проводится незамедлительно
- Первый выделенный блок – служебный, неудаляемая «голова» списка

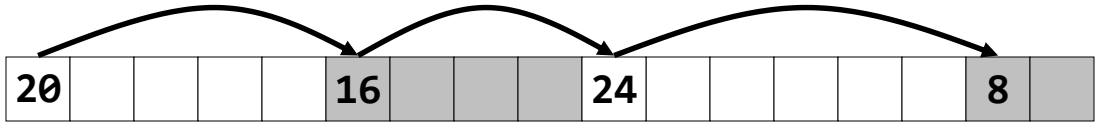
- Требуется определить
  - Состояние кучи после выполнения запросов
    1. p1=malloc(5)
    2. p2=malloc(11)
    3. free(p1)
    4. p3=malloc(4)
    5. p4=malloc(5)
    6. free(p2)
  - Пиковое использование памяти U_6

Начальное состояние кучи

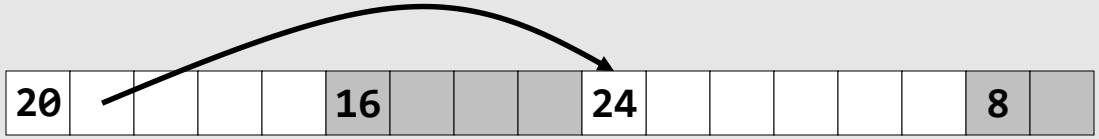


Как отслеживать свободные блоки

- 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)

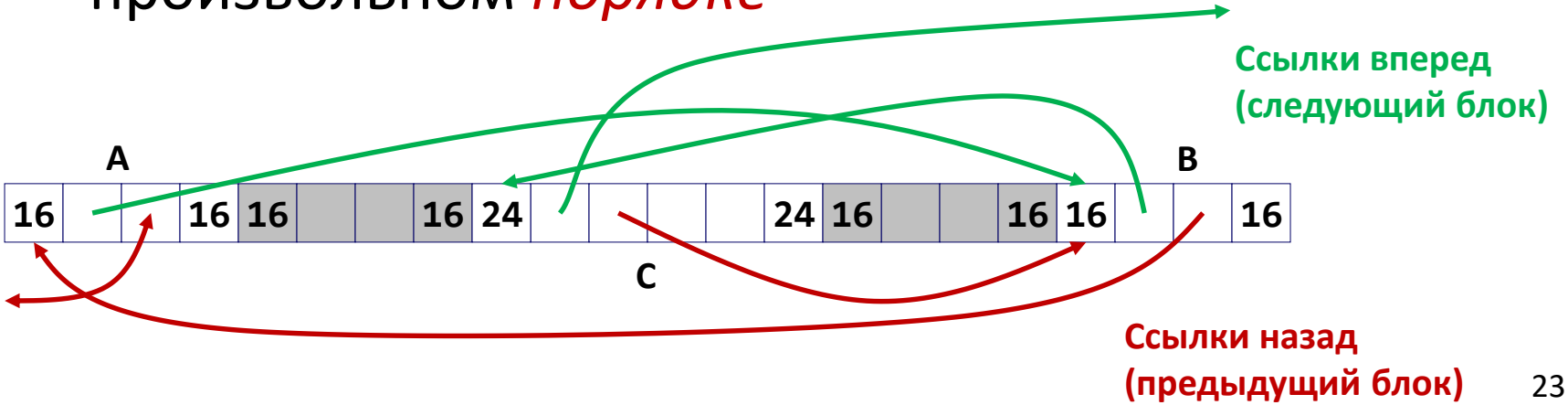
Свободный

- 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

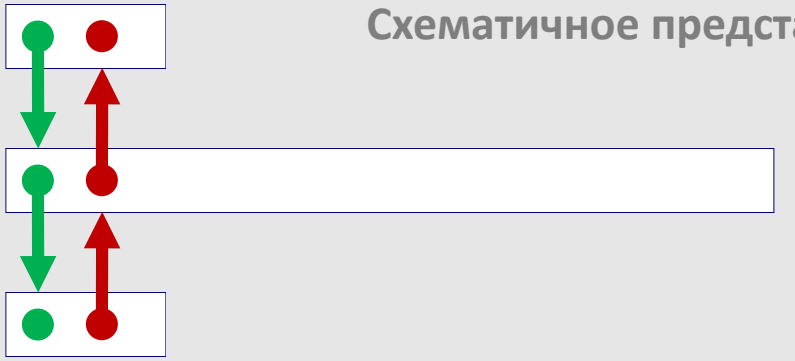
- Логическая организация

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



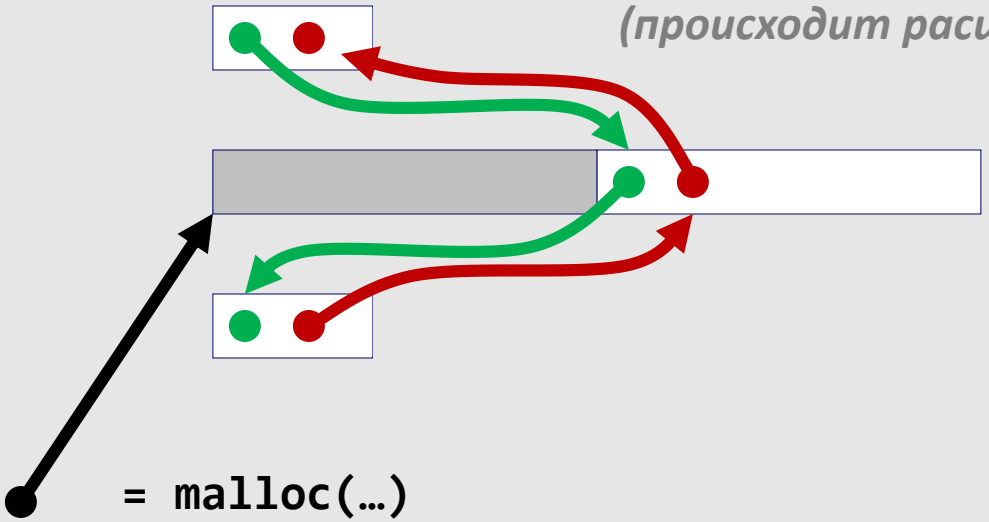
Явный список свободных блоков
Выделение памяти

До



Схематичное представление списка

После

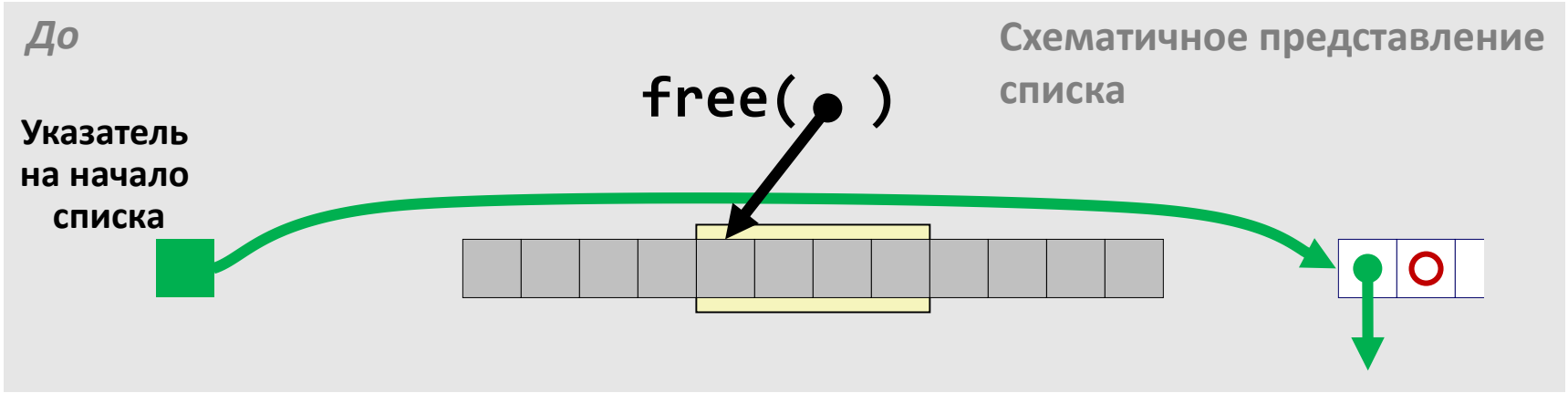


(происходит расщепление блока)

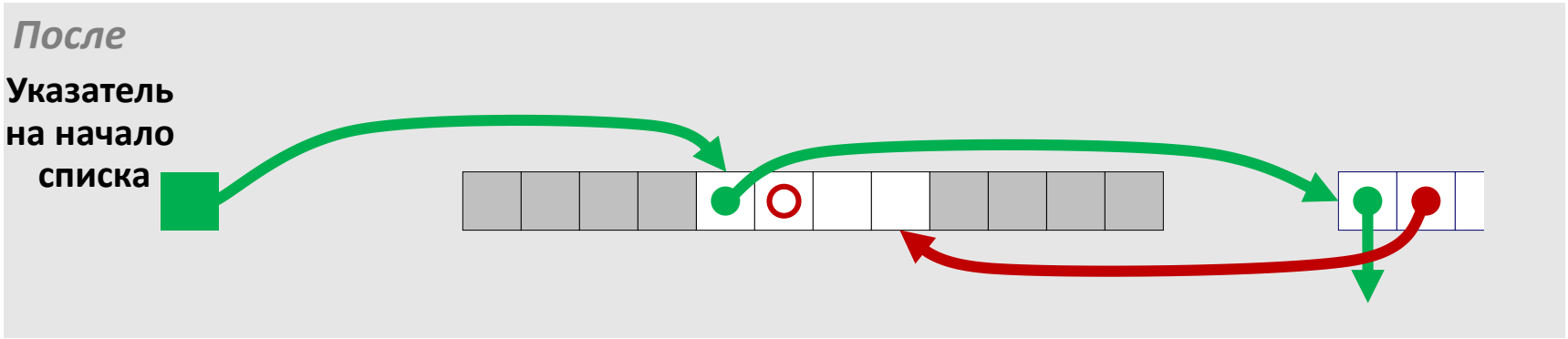
Явный список свободных блоков Освобождение памяти

- *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

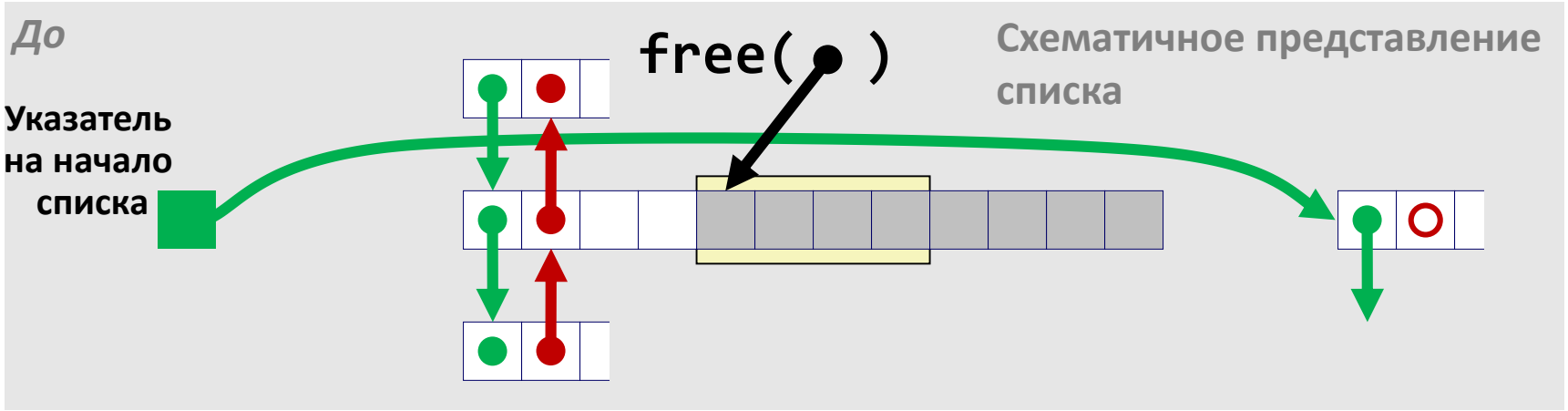
Освобождение блока в порядке LIFO (Случай 1)



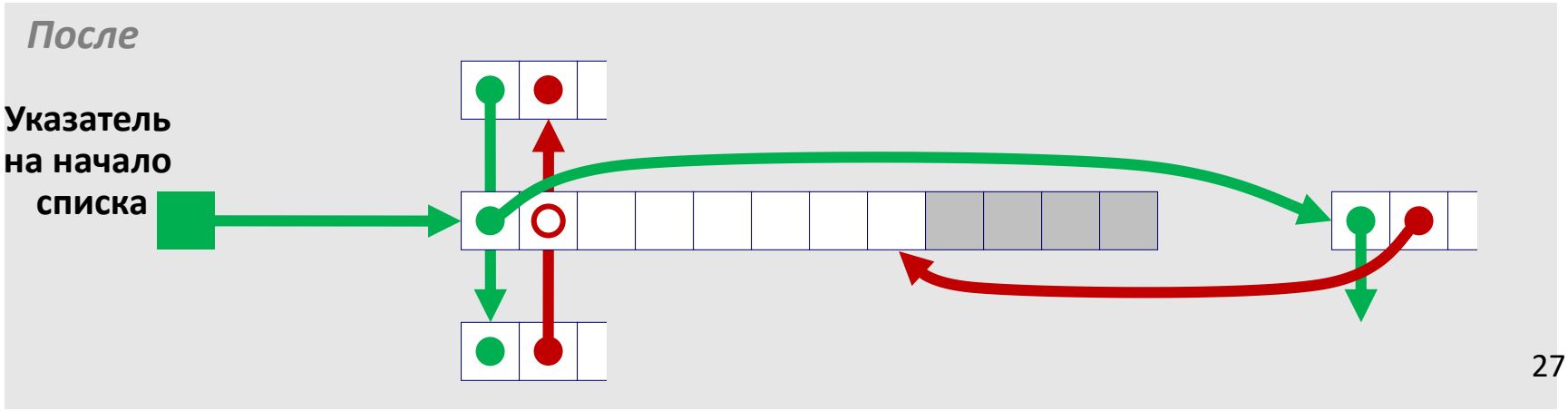
- Помещаем освобожденный блок в начало списка



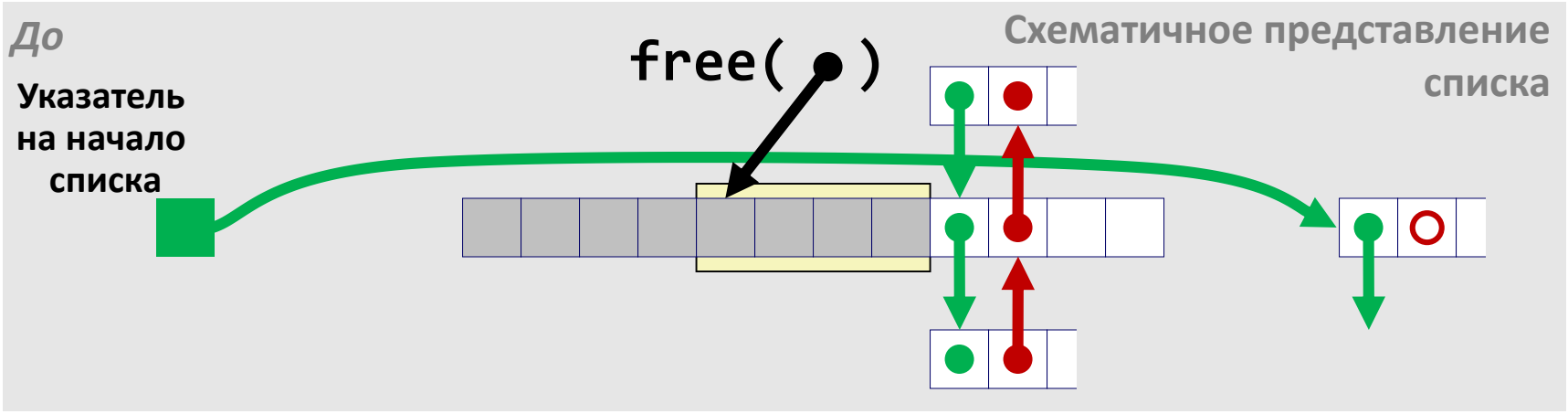
Освобождение блока в порядке LIFO (Случай 2)



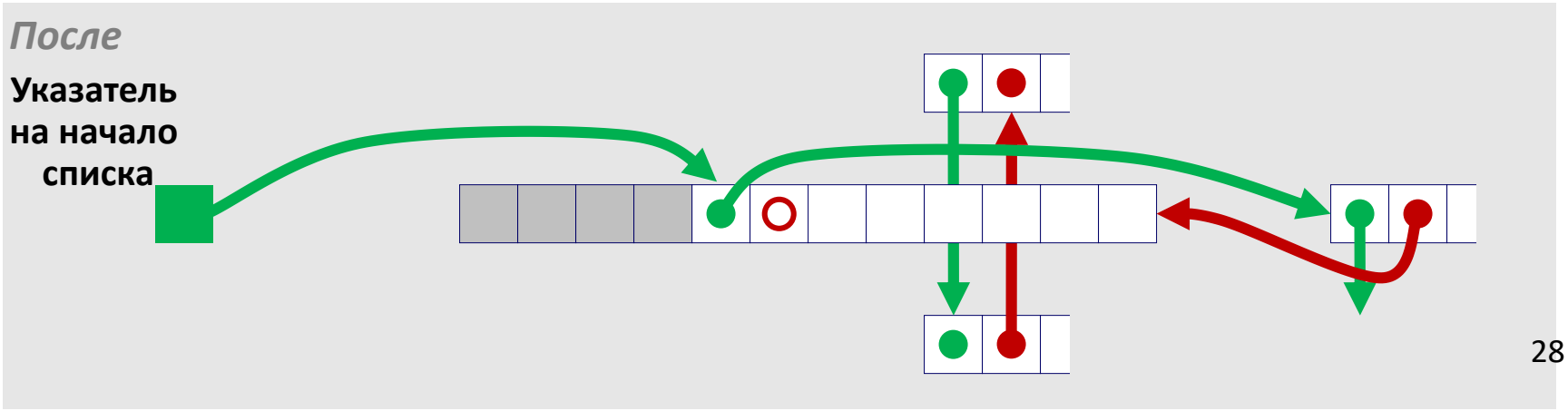
- Извлекаем из списка смежный (перед освобождаемым) в памяти блок, выполняем слияние, и вставляем образовавшийся блок в начало списка



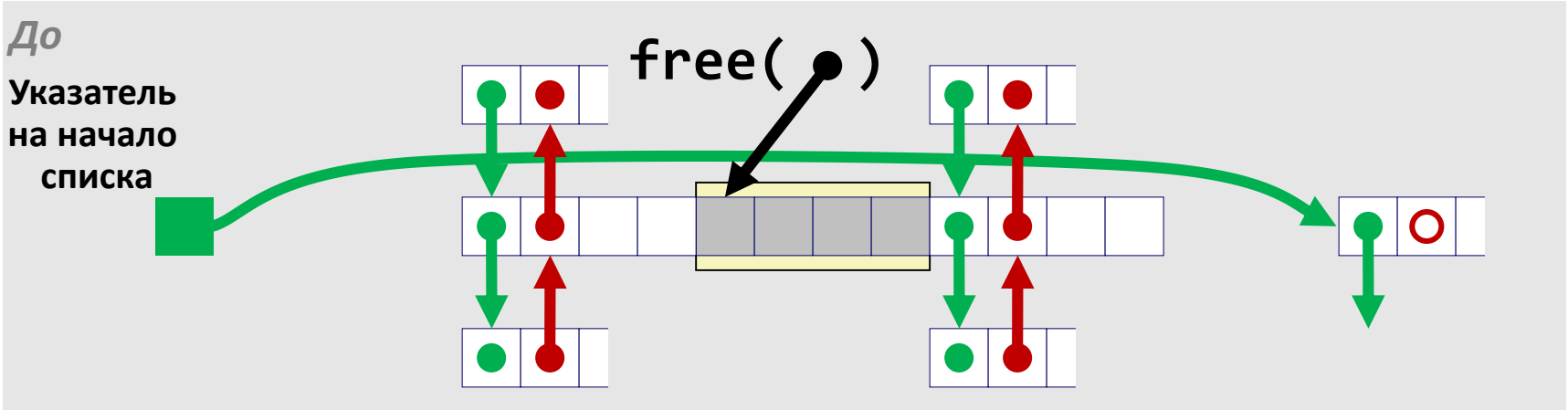
Освобождение блока в порядке LIFO (Случай 3)



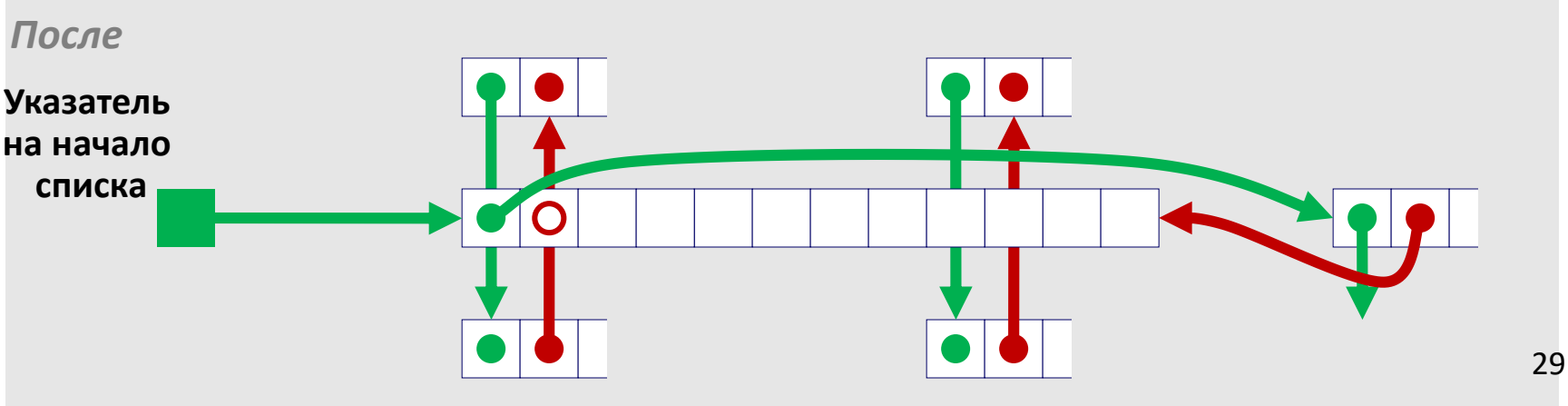
- Извлекаем из списка смежный (после освобождаемого) в памяти блок, выполняем слияние, и вставляем образовавшийся блок в начало списка



Освобождение блока в порядке LIFO (Случай 4)



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



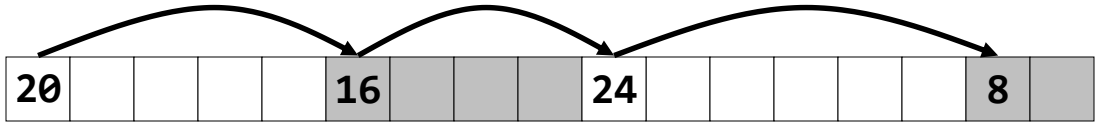
Промежуточные итоги
Явный список

- 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?

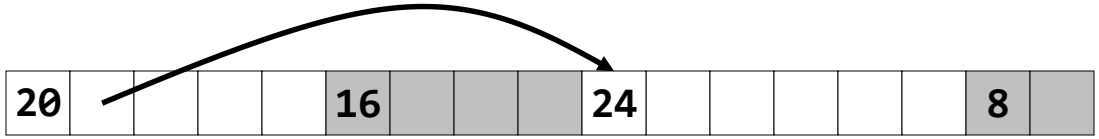
- Как правило подход с поддержкой явного списка комбинируют с разделением блоков по нескольким спискам
  - Блоки разделяют на несколько классов, в зависимости от их размера

Как отслеживать свободные блоки

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



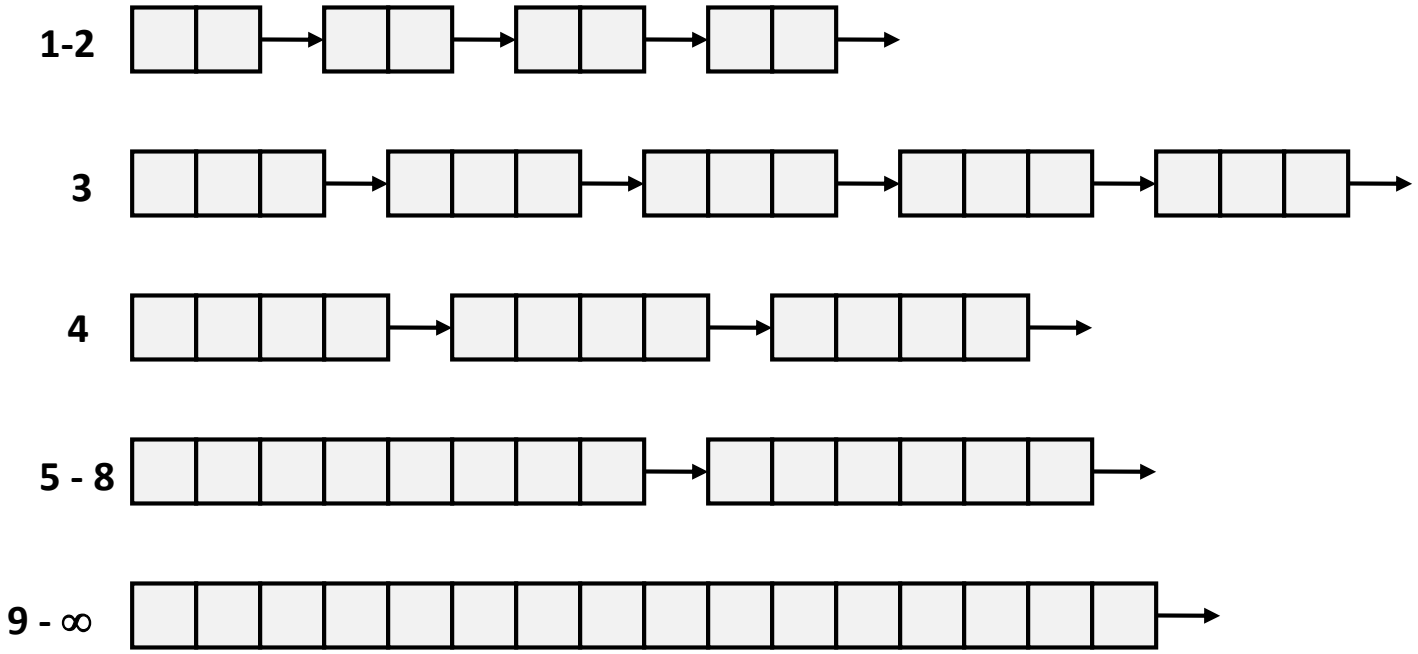
- Метод 2: *Явный список* свободных блоков с использованием указателей



- 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).

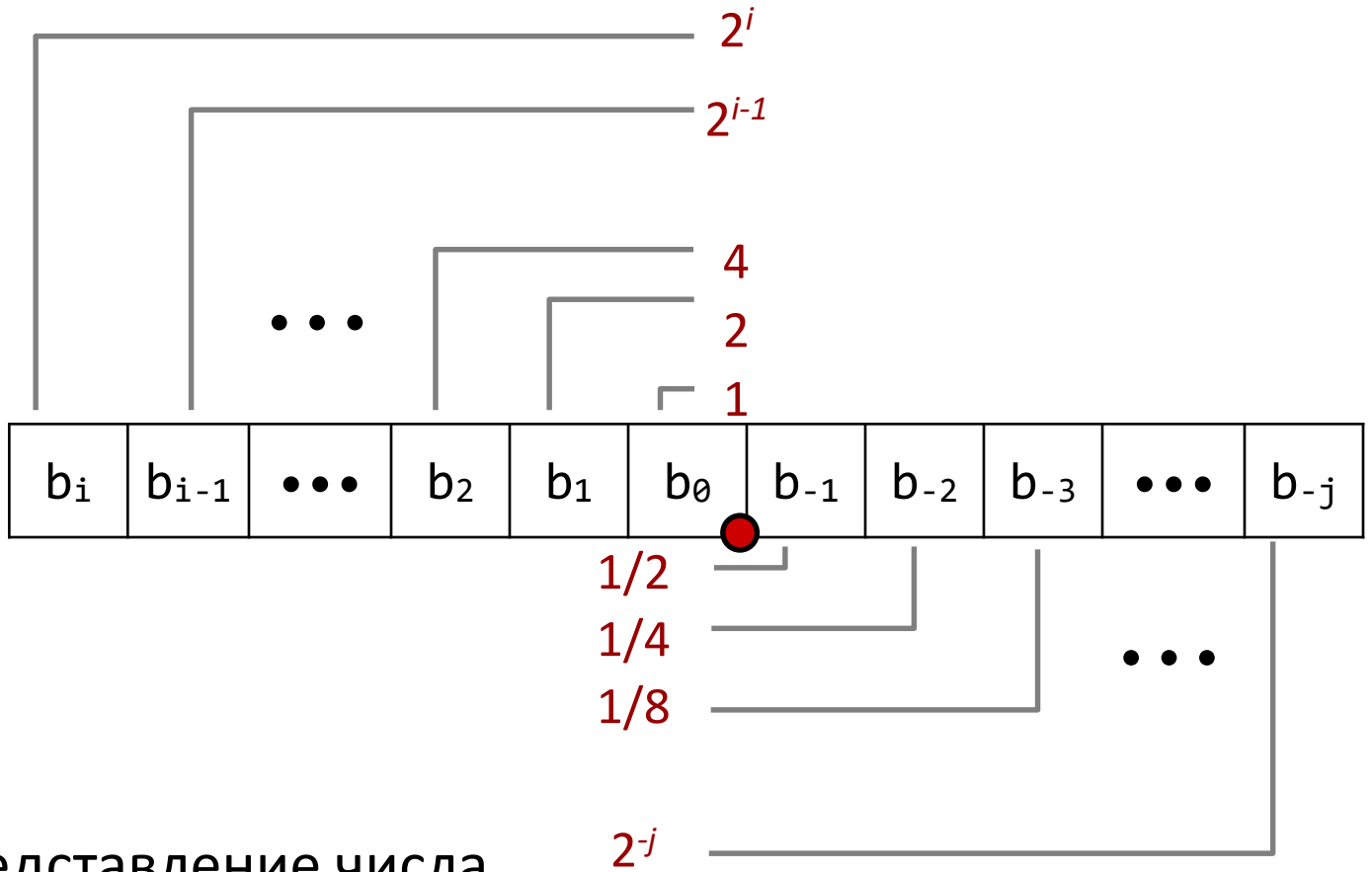
Выделение памяти по методу Seglist

- 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

Далее ...

- *Динамическая память*
  - *Организация и управление*
  - *Численные характеристики*
  - *Управление свободными блоками*
- ***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 $1.0 - \varepsilon$

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` `0.00110011001[0011]`$\ldots_2$

  `1/10` `0.000110011001[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:

s **exp** **frac**

Денормализованное число

- Условие: exp = 000…0
- Значение порядка: `E = –Смещение + 1` (вместо `E = 0 – Смещение`)
- Мантисса кодируется с ведущим `0`: `M = 0.`xxx…x_2
  - xxx…x: биты поля `frac`
- Примеры
  - exp = 000…0, frac = 000…0
    - Представляет число ноль
    - Различные кодировки для +0 и –0
  - exp = 000…0, frac ≠ 000…0
    - Кодируются числа близкие к 0.0
    - Распределены по числовой прямой с равным шагом

Особые числа

- Условие: exp = 111…1
- Пример: exp = 111…1, frac = 000…0
  - Представляет бесконечно большое число ∞
    (как положительное, так и отрицательное)
  - Требуются для операций в которых может произойти переполнение

    `1.0/0.0 = –1.0/–0.0 = +∞`

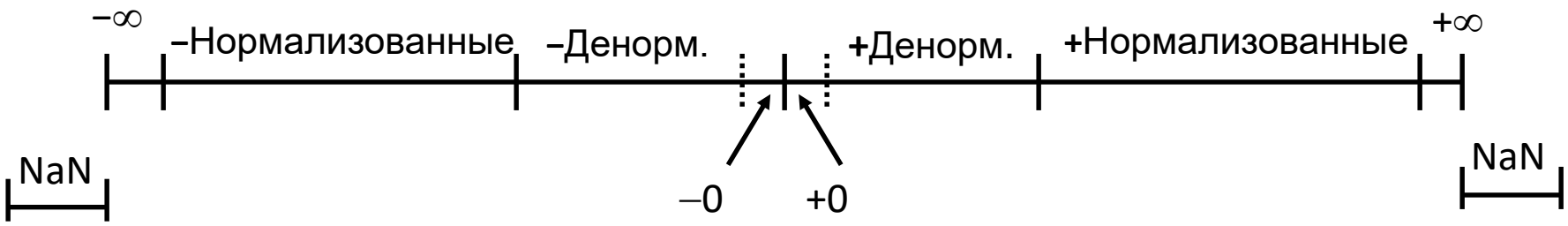
    `1.0/–0.0 = –∞`
- Пример: exp = 111…1, frac ≠ 000…0
  - Not-a-Number (NaN)
  - Используется в ситуациях, когда значение операции не определено

    `sqrt(–1)`

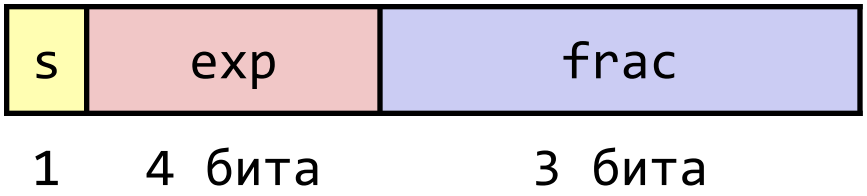
    `∞ – ∞`

    `∞ × 0`

Диапазоны значений



Пример



- 8-разрядные числа с плавающей точкой
  - Знаковый бит – старший бит
  - Следующие четыре бита – порядок, смещение – 7
  - Последние три бита – дробная часть (мантисса)

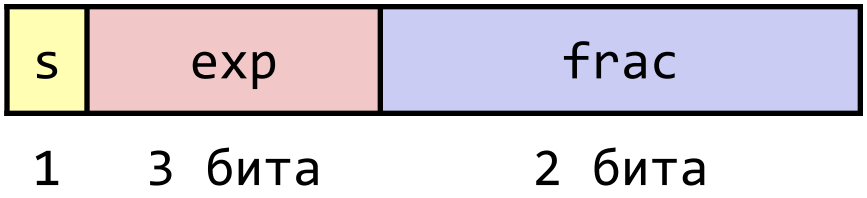
- Выполнены все требования стандарта IEEE 754 к формату числа
  - Реализованы нормализованные и денормализованные числа
  - Представлены значения 0, NaN, бесконечность

Диапазоны значений (только для положительных чисел)

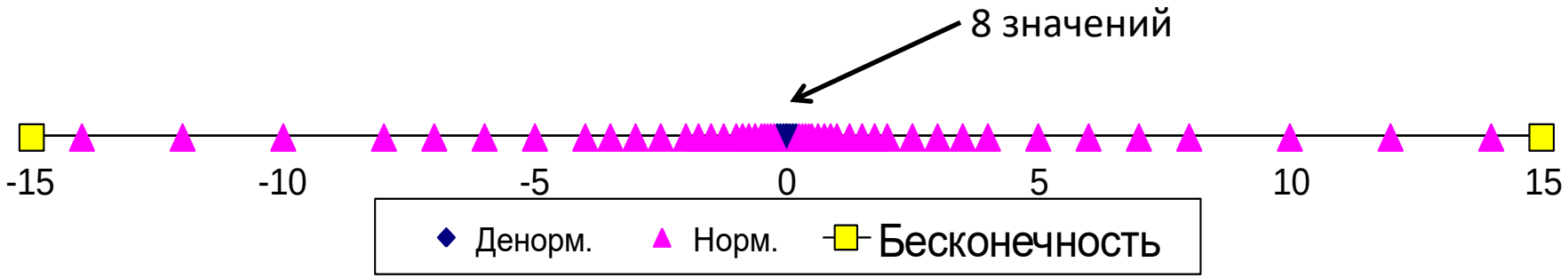
	s	exp	frac	E	Значения	
Денормализованные числа	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	Ближайшее к 0
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	Наибольшее денорм.
Нормализованные числа	0	0001	000	-6	8/8*1/64 = 8/512	Наименьшее норм.
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	Ближайшее к 1 «снизу»
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	Ближайшее к 1 «сверху»
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	Наибольшее норм.
	0	1111	000	n/a	inf	

Распределение значений по числовой прямой

- 6-разрядный формат
  - e = 3 бита порядка
  - f = 2 бита мантиссы
  - Смещение $2^{3-1}-1 = 3$

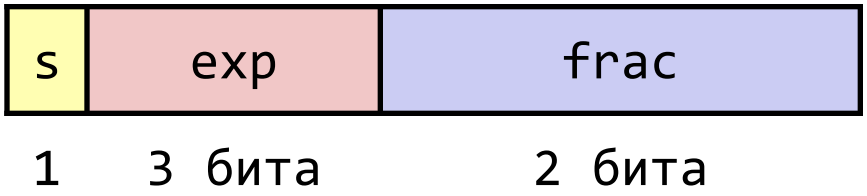


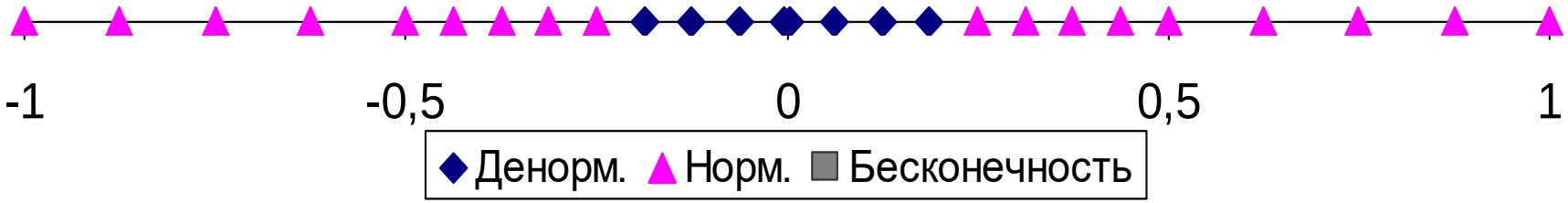
- Распределение сильно «сгущается» в окрестности 0



Распределение значений по числовой прямой (вид вблизи)

- 6-разрядный формат
  - e = 3 бита порядка
  - f = 2 бита мантиссы
  - Смещение 3





Некоторые числа

Описание	*exp*	*frac*	*Численное значение*
• Ноль	00…00	00…00	0.0
• Наименьшее «+» денорм.	00…00	00…01	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
– Одинарная точность ≈ 1.4×10^{-45}			
– Двойная точность ≈ 4.9×10^{-324}			
• Наибольшее денорм.	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
– Одинарная точность ≈ 1.18×10^{-38}			
– Двойная точность ≈ 2.2×10^{-308}			
• Наименьшее «+» норм.	00…01	00…00	$1.0 \times 2^{-\{126,1022\}}$
– Немногим больше чем наибольшее денормализованное			
• Единица	01…11	00…00	1.0
• Наибольшее норм.	11…10	11…11	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
– Одинарная точность ≈ 3.4×10^{38}			
– Двойная точность ≈ 1.8×10^{308}			

Точность {одинарная,двойная}

Особенности кодировки

- FP ноль совпадает с целочисленным нулем
  - Все биты = 0
- Допустимо (в большинстве случаев) использовать беззнаковое целочисленное сравнение
  - Сперва сравниваем знаковые биты
  - Необходимо рассматривать −0 = 0
  - NaNs
    - В целочисленной интерпретации больше, чем любые другие числа
    - Что необходимо выдавать в качестве результата сравнения?
  - В противном случае …
    - Денормализованные vs. Нормализованные
    - Нормализованные vs. Бесконечность

Операции над числами с плавающей точкой

- $x +_f y = \mathrm{Round}(x + y)$
- $x \times_f y = \mathrm{Round}(x \times y)$
- Основная идея
  - Сперва вычислить точный результат
  - Поместить результат в требуемый размер точности
    - Переполнение, если порядок слишком большой
    - Возможно придется округлять поле frac

Округление

- Способы округления

	1.40	1.60	1.50	2.50	–1.50
– К нулю	1	1	1	2	–1
– К наименьшему (−∞)	1	1	1	2	–2
– К наибольшему (+∞)	2	2	2	3	–1
– К ближайшему (✓)	1	2	2	2	–2

Округление к ближайшему целому числу

- Основной способ округления
  - Все остальные способы дают статистическое смещение
    - Пример: суммирование положительных чисел будет давать устойчивую недо- или пере- оценку результата
- Применимо при округлении в произвольной позиции дроби
  - Когда число расположено точно посредине двух значений к которым можно округлить
    - Округляют к тому числу, у которого наименьшая значащая цифра четная
  - Например, округление до ближайших сотых

1.2349999	1.23	
1.2350001	1.24	
1.2350000	1.24	(середина — округляем к большему)
1.2450000	1.24	(середина — округляем к меньшему)

Округление двоичных чисел

- Двоичные дробные числа
  - “Четные” числа у которых младший значащий бит 0
  - “Середина” – когда биты справа от позиции к которой происходит округление = $100\ldots_2$

- Примеры
  - Округление до ближайшей 1/4 (2 бита справа от бинарной точки)

Число	Двоичное	Окр.	Действие	Окр. число
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2