

# Dynamic Memory Allocation

Программная инженерия  
20 ноября 2023

# Сегодня в программе

- Какие аллокаторы бывают
- Какие есть нюансы при разработке аллокатора
- Рассмотрим подходы, алгоритмы
- Немного посмотрим на код
- ~~• Страшная сложная большая задача на дом~~



# Итак

- Вы уже имели дело с аллокаторами памяти
- Они нужны, чтобы работать со структурами данных динамического размера
- Аллокатор работает с кучей (heap)
- Для аллокатора куча — множество блоков разного размера
- Блок — непрерывный кусок виртуальной памяти, который может быть либо выделенным (allocated), либо свободным (free).
- НУО предполагаем, что куча растет по увеличению количества адресов.



# Какие бывают аллокаторы

- Явные (Explicit)
  - Программа явно освобождает выделенный блок
  - malloc и free в C, new и delete в C++ ...
- Неявные (Implicit)
  - Аллокатор сам берет на себя обнаружение и освобождение неиспользованных блоков
  - Сборщики мусора в Java, .NET, Lisp ...
- Мы будем говорить о явных



# Напомним...

- `void* malloc(size_t size)` — выделяет блок памяти размера не менее `size`
  - Return: указатель на выделенный блок или `NULL` в случае ошибки
  - Блок памяти обычно выравнивается (В Unix системах - по 8 байт)
  - Не занимается инициализацией! Для этого есть `calloc`
  - В рамках лекции считаем размер слова равным 4 байта
- `void free(*ptr)` — освобождает блок памяти, на который указывает `ptr`
  - `ptr` должен указывать на начало выделенного блока - иначе UB
  - `free` не сигнализирует об ошибке — с ним надо быть аккуратнее



# Как аллокатор может выделять память?

- `mmap` и `mmapr`
  - Отображение объекта в физической памяти в адресное пространство процесса
  - `/proc/[pid]/maps` — показать отображенные участки памяти процесса
- Управление размером кучи: `void* sbrk(intptr_t incr)`
  - `brk` — указатель на конец кучи
  - `sbrk` просто прибавляет `incr` к этому указателю
  - Надо, чтобы запросить у ОС больше памяти в куче



# Требования к аллокатору

- Последовательность запросов `malloc` и `free` — произвольная
  - Нельзя полагаться на порядок запросов
  - Но мы предполагаем, что `free` вызывается на участке, который был выделен
- Немедленный ответ на запрос
  - Нельзя буферизировать запросы или переупорядочивать
- Используется только куча
  - Все нескаллярные структуры данных, которыми пользуется аллокатор, должны лежать в куче
- Выравнивание блоков
  - Нужно, чтобы в блоке могли размещаться данные любого типа
  - В большинстве систем выравнивается по 8 байт
- Нельзя модифицировать выделенные блоки
  - Можно манипулировать только свободными блоками



# Цели аллокатора

- Максимизация пропускной способности (throughput)
  - Количество запросов, выполняющихся в единицу времени
  - Нужно уменьшать среднее время на запрос к аллокатору
- Максимальная утилизация памяти (memory utilization)
  - Полезная нагрузка (payload) — сколько памяти действительно было запрошено
  - Нам нужно максимизировать суммированную полезную нагрузку для всех запросов относительно размера кучи

Эти цели противоречат друг другу. Нужно искать баланс.





# Фрагментация

- Главная причина плохой утилизации кучи
- Неиспользованная память не соответствует требованиям запросов аллокатора
- Внутренняя фрагментация
  - Выделили больше, чем было запрошено (больше, чем `payload`)
  - Минимальный размер блока
  - Выравнивание
- Внешняя фрагментация
  - В куче есть место, чтобы выделить память, но нет доступных свободных блоков
  - Зависит в том числе от будущих запросов

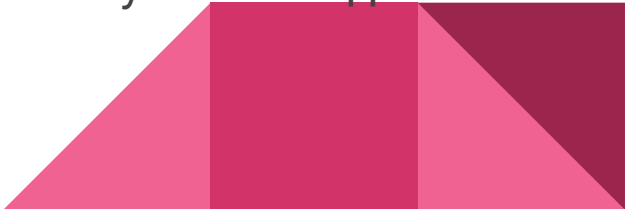


# Нюансы реализации

Мы могли бы сделать простой аллокатор:


- Куча — массив с указателем `p` на начало
- `malloc(size)`: увеличить указатель `p` на `size`, вернуть новый указатель
- `free(ptr)`: просто `return`, ничего не делать

Что мы получили:

- Хорошая пропускная способность, все запросы за константу
  - Отвратительная утилизация памяти — не переиспользуем свободные блоки
- 

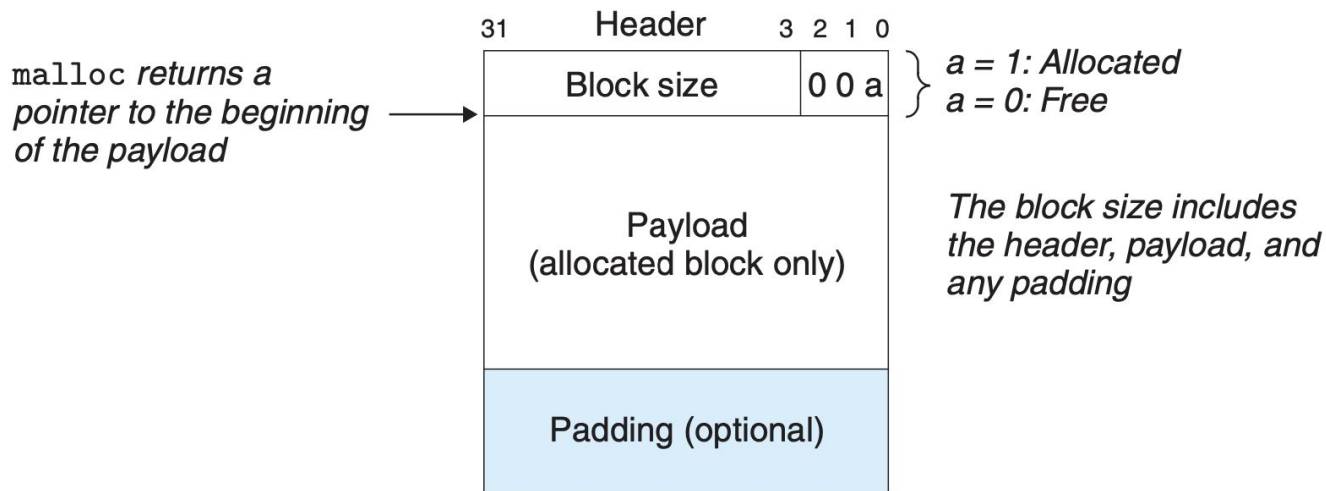
# Нюансы реализации

Если мы хотим добиться баланса между целями, нужно ответить на следующие вопросы:

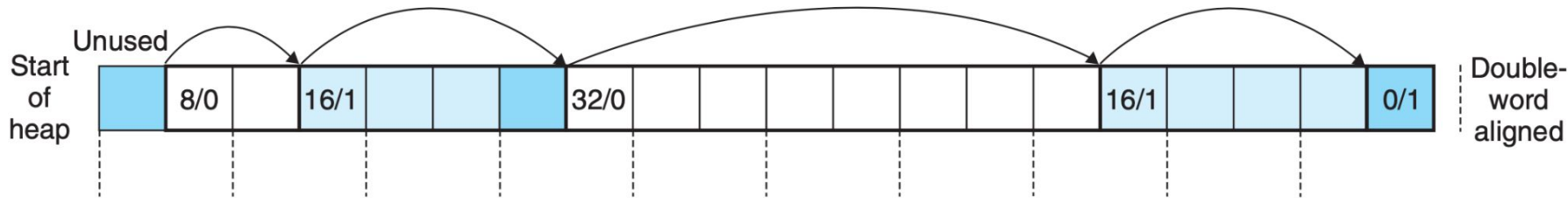
- Free block organization.
    - Как отслеживать свободные блоки?
  - Placement.
    - Как выбрать свободный блок, куда мы будем аллоцировать?
  - Splitting.
    - Как только заняли свободный блок, что делать с оставшейся частью?
  - Coalescing.
    - Что делать с только что освобожденным блоком?
- 

# Implicit Free List

- Как различать границы блоков и понимать, свободен блок или нет?
- Будем хранить все нужное в самом блоке:



# Как выглядит куча? (Free block organization)



- Последовательность занятых и свободных блоков
  - Односвязный список свободных блоков
  - В конце — маркер окончания списка
- Время на операцию - линейное от количества всех блоков
  - Это недостаток, с которым в будущем будем бороться
- Выравнивание
  - Появляется минимальный размер блока — 2 слова

# Размещение выделенного блока (placement)

При запросе аллокации - поиск подходящего свободного блока.

А как искать? Есть разные **политики размещения!**

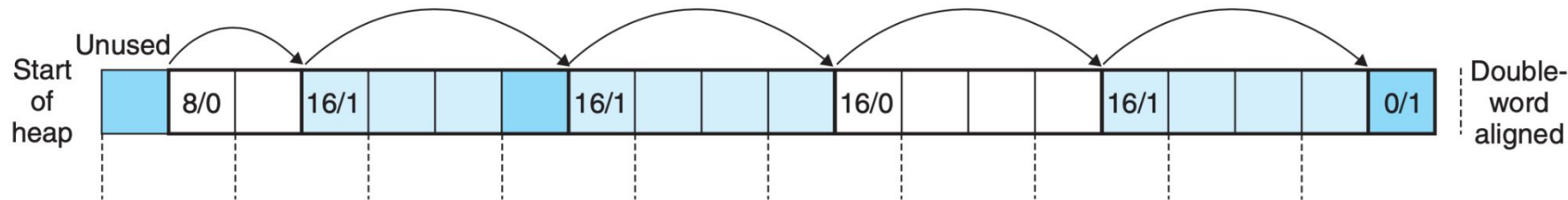
- first fit — берем первый попавшийся подходящего размера
  - Обычно свободные блоки побольше оказываются в конце списка
  - В начале свободные блоки меньше => поиск блока побольше займет больше времени
- next fit — начинаем поиск там, где закончился предыдущий
  - Есть вероятность, что следующий подходящий блок - остаток предыдущего
  - Может работать быстрее, чем first fit
  - Хуже утилизирует память
- best fit — перебираем все блоки и ищем подходящий с наименьшим размером
  - Лучше утилизирует память
  - Хуже по времени - бежим по всей куче



# Разделение свободных блоков (splitting)

Нашли подходящий свободный блок — сколько надо занять?

- Можем весь блок — плохо. Внутренняя фрагментация
- Можем делить на 2 части — выделенный блок и остаток.
- На примере — запрос на 3 слова:



# Получение дополнительной памяти в куче

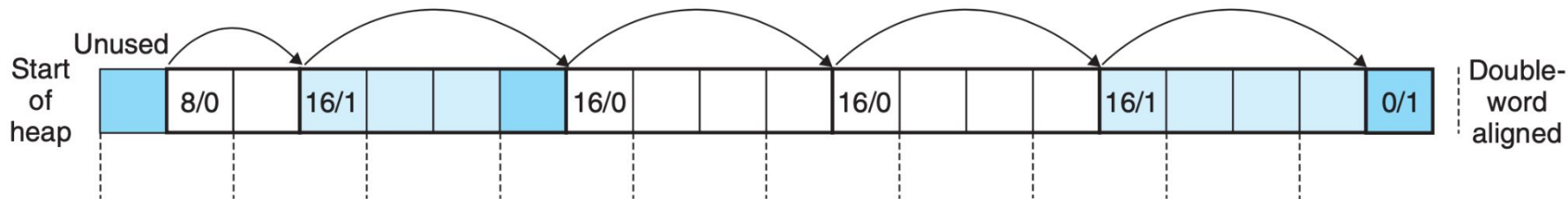
- Что, если так и не нашли подходящий свободный блок?
- Объединение свободных блоков
  - Об этом дальше
- Если не сработало — просим у ОС больше памяти через `sbrk`
- Превращаем новый кусок памяти в большой свободный блок
- Вставляем блок в список и используем его для выделения.





# Объединение свободных блоков (coalescing)

- При освобождении свободные блоки могут оказаться рядом:



- Получили ложную фрагментацию:
  - Запрос на 4 слова не выполнится, хотя место есть
  - У нас два блока с payload = 3 слова
- Блоки надо объединять

# Нюансы объединения

Когда это можно делать?

- Сразу при запросе
  - Быстро
  - Может привести к лишним действиям
- Когда-то позже
  - Если не нашли свободный блок, например
  - Требуется отдельного прохода по куче



# Нюансы объединения

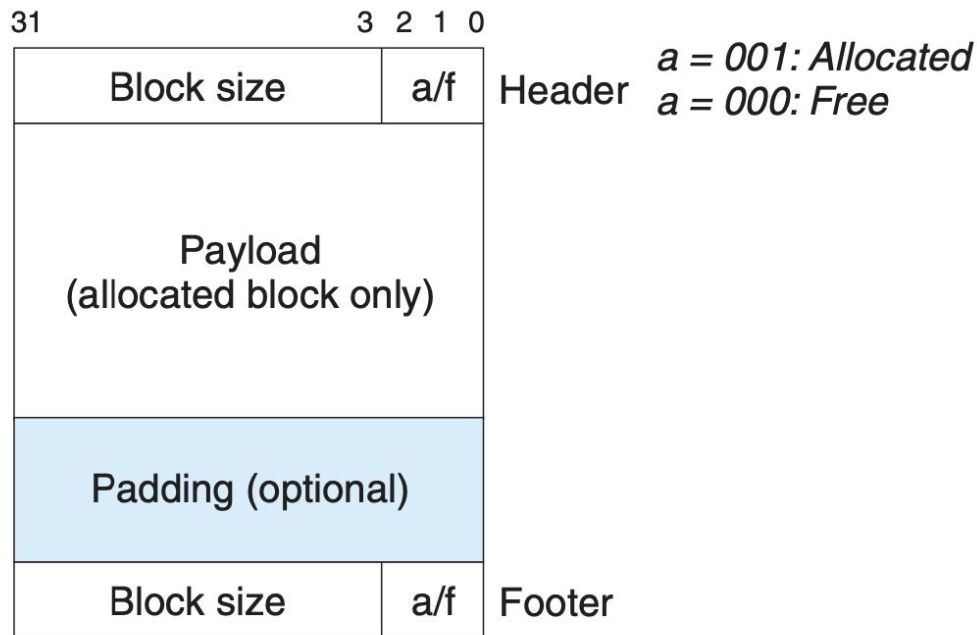
Допустим, освободили текущий блок

- Объединиться со следующим - легко за константное время
- А как объединяться с предыдущим?
- С текущей реализацией — только за линейное время от количества
  - Получаем довольно медленный free
- Как добиться константы?
- Нам поможет Дональд Кнут!



# Метод граничных маркеров

- Улучшим наш блок:



Теперь можем получить  
размер предыдущего блока!



# Метод граничных маркеров - случаи

m1	a
m1	a
n	a
n	a
m2	a
m2	a



m1	a
m1	a
n	f
n	f
m2	a
m2	a

Case 1

m1	a
m1	a
n	a
n	a
m2	f
m2	f



m1	a
m1	a
n+m2	f
n+m2	f

Case 2

m1	f
m1	f
n	a
n	a
m2	a
m2	a



n+m1	f
n+m1	f
m2	a
m2	a

Case 3

m1	f
m1	f
n	a
n	a
m2	f
m2	f



n+m1+m2	f
n+m1+m2	f

Case 4

# Метод граничных маркеров

- Получили константное время в каждом случае
- Подход легко обобщить на разные типы аллокаторов
- Тратим много памяти на header и footer
- Можем оптимизировать:
  - Можем избавиться от футера у выделенных блоков



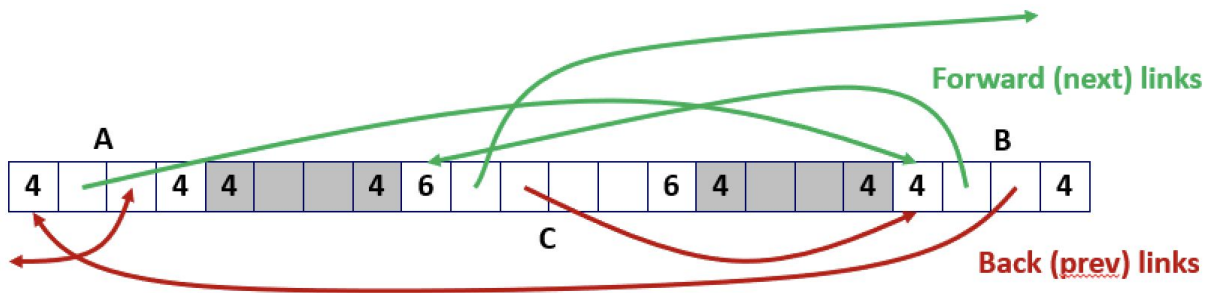
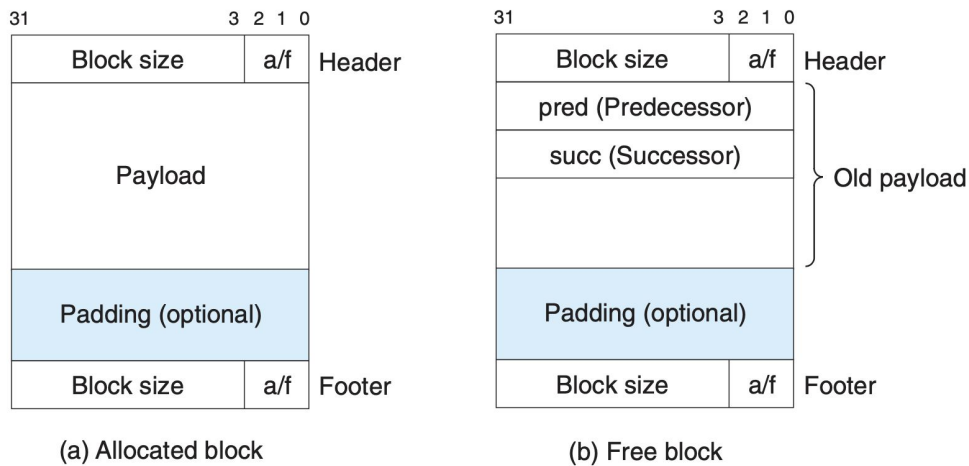
# Implicit Free Lists - что получили?

- Линейное время от количества всех блоков на аллокацию
- Константное время на освобождение
- Очень просто реализовать
- Редко используется из-за скорости malloc, но при этом довольно в определенных случаях может подойти
- Разделение и объединение может распространяется почти на все аллокаторы!



# Explicit Free List

- Строим двусвязный список из свободных блоков!





# Explicit Free List — аллокация блока

- Также, как в Implicit Free List
- Опять же, политики размещения бывают разные!



# Explicit Free List — освобождение блока

- Зависит от **политики вставки в список**
  - Last In First Out (LIFO) — вставляем новый блок в начало
    - free работает за константу
  - Address order — блоки в списке упорядочены по адресам
    - free работает за линию — проходим по списку
    - trade-off: мы лучше утилизируем память: проход first fit приближается к best fit!
- Граничные маркеры все еще нужны для объединения блоков



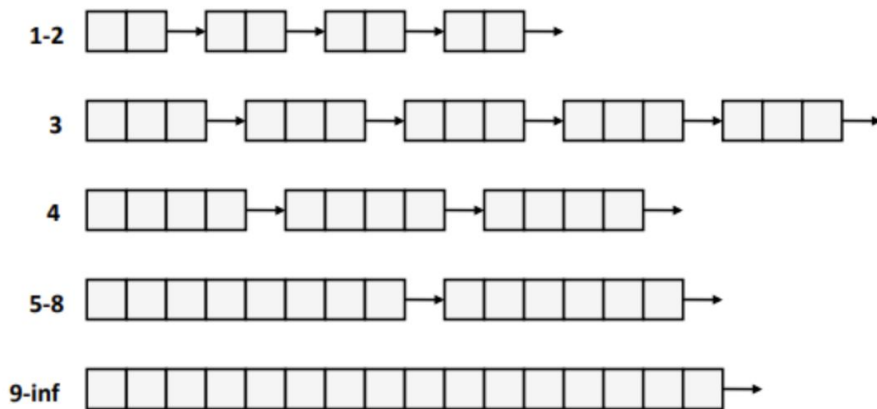
# Explicit Free List - что получили?

- По сравнению с Implicit Free List:
  - Улучшили время аллокации: линейное время от количества свободных блоков, а не от количества всех блоков
  - Нужно больше места на свободный блок, ведь мы храним указатели
  - Это может увеличивать внутреннюю фрагментацию!
- Время на аллокацию можно еще улучшить!



# Segregated Free Lists

- Массив списков со свободными блоками определенных размеров
- Множество блоков разбиваем на классы по размерам
- Политик того, как разбить блоки на классы — много
  - Можно по степеням двойки
  - Можно маленькие размеры выделять в отдельные классы



# Segregated Free Lists — аллокация и освобождение

- Чтобы выделить блок размера  $n$ 
  - Ищем в списке свободных блоков подходящего класса
  - Если нашли свободный блок — выделяем
    - Разделить, а остаток поместить в нужный список - опционально
  - Если не нашли, пробуем искать в списке блоков большего размера
- Перебрали все списки и не нашли — что делать?
  - Просим больше памяти у ОС при помощи `sbrk()`
  - Выделяем из новой памяти блок нужного размера
  - Остаток выделяем в отдельный блок и вставляем в нужный список
- Освобождение блока
  - Вставка блока в нужный список
  - Можем объединять блоки со вставкой в нужный список — опционально



# Segregated Free Lists - что мы уже получили?

- Время на запрос стало ниже
  - Теперь исследуем не всю кучу, а какую-то часть
- Лучше используем память
  - Проход с политикой first fit приближает best fit

То, что мы сейчас описали — концепция. Рассмотрим более конкретные реализации.



# Simple Segregated Storage

- Каждый список хранит блоки одного размера
  - Пример: размеры {17-32} — округляем до 32
- Выделение блока:
  - Смотрим нужный список. Если не пуст — берем первый блок. Не разделяем.
  - Список пуст — запрашиваем у ОС кусок памяти, и делим его на блоки нужного размера, теперь список не пуст.
- Освобождение:
  - Вставляем новый свободный блок в начало нужного списка
  - Никакого объединения



# Simple Segregated Storage - что получили?

- malloc и free за константное время - круто!
- Уменьшили минимальный размер блока
  - Нам нужен только указатель на следующий блок
- Страдаем от внутренней фрагментации
  - Не разделяем же блоки
- Страдаем от внешней фрагментации
  - Есть конкретные сценарии
  - Много запросов на размер 1, много запросов на размер 2 ...





# Segregated Fits

- Каждый список — явный или неявный (как описывалось ранее)
  - В списке — блоки разных размеров!
- Выделение блока:
  - Бежим по нужному списку по политике first fit
  - Нашли — делим, остаток отправляем в нужный список
  - Не нашли — ищем в списке класса больших размеров
  - Перебрали все списки? Просим памяти у ОС, выделяем нужный блок, остаток — помещаем в нужный список
- Освобождение блока:
  - Объединяем блоки и результат отправляем в нужный список



# Segregated Fits - что получили?

- Поиск не по всей кучи, а по ее части
- first fit здесь приближается к best fit по всей куче
- Популярный подход
  - Используется в пакете malloc стандартной библиотеки Си



# Двоичные близнецы

- Segregated Fits, только каждый класс - степень двойки
  - Округляем размеры
- Пусть в куче  $2^m$  слов
- Держим список для блоков размеров  $2^k$ ,  $0 \leq k \leq m$
- Изначально у нас один блок размером  $2^m$





# Двоичные близнецы — освобождение блока

- Объединяем блоки, пока не дойдем до близнеца
- Пример: освобождаем блок D

6	A: $2^0$	C: $2^0$	$2^1$	D: $2^1$	$2^1$	$2^3$
7.1	A: $2^0$	C: $2^0$	$2^1$	$2^1$	$2^1$	$2^3$
7.2	A: $2^0$	C: $2^0$	$2^1$	$2^2$		$2^3$

# Двоичные близнецы — что получили?

- Быстрый поиск и объединение
  - Знаем адрес и размер блока — легко посчитать адрес близнеца
  - xxx...x00000 — адрес блока размером 32 байта
  - xxx...x10000 — адрес близнеца
  - Отличие в одном бите!
- Страдаем от внутренней фрагментации
- Может подойти, когда размеры выделенных блоков известны и они близки к степеням двойки.



# Итог

- Разных аллокаторов много — и они нужны
  - Разным вариантам ОС
  - СУБД
  - Реализациям языков программирования
  - Разным программам на Си
- Здесь рассмотрена только малая часть, их гораздо больше






А как закодировать?



# Implicit Free List: Реализация на Си

- Реализуем подход, описанный ранее
  - First fit placement
  - Объединяем блоки сразу
  - Максимальный размер блока -  $2^{32} = 4 \text{ GB}$
- Сделаем так, чтобы не пересекался с malloc в Си
- Наше API:

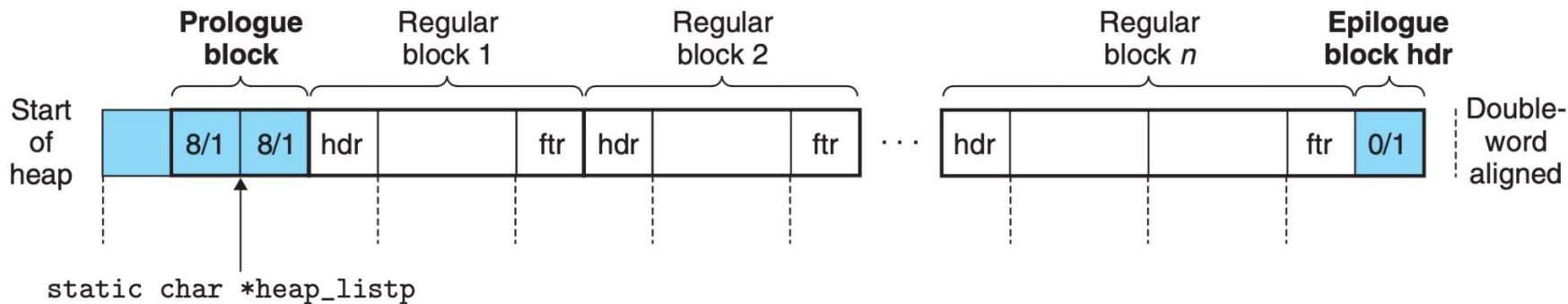
```
1  extern int mm_init(void);  
2  extern void *mm_malloc (size_t size);  
3  extern void mm_free (void *ptr);
```



# Работаем с памятью

```
1  /* Private global variables */
2  static char *mem_heap;      /* Points to first byte of heap */
3  static char *mem_brk;      /* Points to last byte of heap plus 1 */
4  static char *mem_max_addr; /* Max legal heap addr plus 1*/
16 /*
17  * mem_sbrk - Simple model of the sbrk function. Extends the heap
18  *    by incr bytes and returns the start address of the new area. In
19  *    this model, the heap cannot be shrunk.
20  */
21 void *mem_sbrk(int incr)
22 {
23     char *old_brk = mem_brk;
24
25     if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
26         errno = ENOMEM;
27         fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of memory...\n");
28         return (void *)-1;
29     }
30     mem_brk += incr;
31     return (void *)old_brk;
32 }
```

# Как выглядит наш Implicit Free List



- Блок пролога - состоит только из заголовка и футера
- Блок эпилога - выделенный блок нулевого размера
- Они нужны, чтобы удобнее работать с объединением (конфликты на границах кучи).
- `heap_listp` - указывает на пролог

# Полезные макросы

```
1  /* Basic constants and macros */
2  #define WSIZE      4      /* Word and header/footer size (bytes) */
3  #define DSIZE      8      /* Double word size (bytes) */
4  #define CHUNKSIZE (1<<12) /* Extend heap by this amount (bytes) */
5
6  #define MAX(x, y) ((x) > (y)? (x) : (y))
7
8  /* Pack a size and allocated bit into a word */
9  #define PACK(size, alloc) ((size) | (alloc))
10
11 /* Read and write a word at address p */
12 #define GET(p)      (*(unsigned int *)(p))
13 #define PUT(p, val) (*(unsigned int *)(p) = (val))
14
15 /* Read the size and allocated fields from address p */
16 #define GET_SIZE(p) (GET(p) & ~0x7)
17 #define GET_ALLOC(p) (GET(p) & 0x1)
18
19 /* Given block ptr bp, compute address of its header and footer */
20 #define HDRP(bp)     ((char *)(bp) - WSIZE)
21 #define FTRP(bp)     ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
22
23 /* Given block ptr bp, compute address of next and previous blocks */
24 #define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
25 #define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

# Инициализация списка

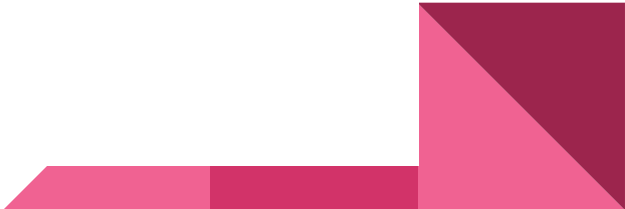
```
1  int mm_init(void)
2  {
3      /* Create the initial empty heap */
4      if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
5          return -1;
6      PUT(heap_listp, 0);                          /* Alignment padding */
7      PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1)); /* Prologue header */
8      PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1)); /* Prologue footer */
9      PUT(heap_listp + (3*WSIZE), PACK(0, 1));      /* Epilogue header */
10     heap_listp += (2*WSIZE);
11
12     /* Extend the empty heap with a free block of CHUNKSIZE bytes */
13     if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
14         return -1;
15     return 0;
16 }
```

# Расширение кучи

```
1  static void *extend_heap(size_t words)
2  {
3      char *bp;
4      size_t size;
5
6      /* Allocate an even number of words to maintain alignment */
7      size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
8      if ((long)(bp = mem_sbrk(size)) == -1)
9          return NULL;
10
11     /* Initialize free block header/footer and the epilogue header */
12     PUT(HDRP(bp), PACK(size, 0));          /* Free block header */
13     PUT(FTRP(bp), PACK(size, 0));          /* Free block footer */
14     PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */
15
16     /* Coalesce if the previous block was free */
17     return coalesce(bp);
18 }
```

# Освобождение

```
1 void mm_free(void *bp)
2 {
3     size_t size = GET_SIZE(HDRP(bp));
4
5     PUT(HDRP(bp), PACK(size, 0));
6     PUT(FTRP(bp), PACK(size, 0));
7     coalesce(bp);
8 }
9
```



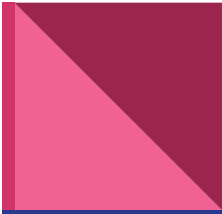
# Объединение (1)

```
10 static void *coalesce(void *bp)
11 {
12     size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKBP(bp)));
13     size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKBP(bp)));
14     size_t size = GET_SIZE(HDRP(bp));
15
16     if (prev_alloc && next_alloc) {           /* Case 1 */
17         return bp;
18     }
19
20     else if (prev_alloc && !next_alloc) {      /* Case 2 */
21         size += GET_SIZE(HDRP(NEXT_BLKBP(bp)));
22         PUT(HDRP(bp), PACK(size, 0));
23         PUT(FTRP(bp), PACK(size, 0));
24     }
```



## Объединение (2)

```
26     else if (!prev_alloc && next_alloc) {          /* Case 3 */
27         size += GET_SIZE(HDRP(PREV_BLKPTR(bp)));
28         PUT(FTRP(bp), PACK(size, 0));
29         PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
30         bp = PREV_BLKPTR(bp);
31     }
32
33     else {                                          /* Case 4 */
34         size += GET_SIZE(HDRP(PREV_BLKPTR(bp))) +
35                 GET_SIZE(FTRP(NEXT_BLKPTR(bp)));
36         PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0));
37         PUT(FTRP(NEXT_BLKPTR(bp)), PACK(size, 0));
38         bp = PREV_BLKPTR(bp);
39     }
40     return bp;
41 }
```



# Выделение: нахождение подходящего блока

```
1  static void *find_fit(size_t asize)
2  {
3      /* First fit search */
4      void *bp;
5
6      for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
7          if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
8              return bp;
9          }
10     }
11     return NULL; /* No fit */
```

# Выделение: размещение блока

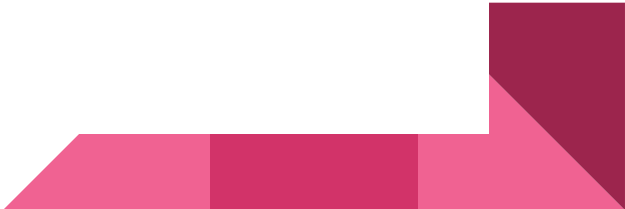
```
1  static void place(void *bp, size_t asize)
2  {
3      size_t csize = GET_SIZE(HDRP(bp));
4
5      if ((csize - asize) >= (2*DSIZE)) {
6          PUT(HDRP(bp), PACK(asize, 1));
7          PUT(FTRP(bp), PACK(asize, 1));
8          bp = NEXT_BLKP(bp);
9          PUT(HDRP(bp), PACK(csize-asize, 0));
10         PUT(FTRP(bp), PACK(csize-asize, 0));
11     }
12     else {
13         PUT(HDRP(bp), PACK(csize, 1));
14         PUT(FTRP(bp), PACK(csize, 1));
15     }
16 }
```

# Выделение: собираем все вместе (1)

```
1 void *mm_malloc(size_t size)
2 {
3     size_t asize;      /* Adjusted block size */
4     size_t extendsize; /* Amount to extend heap if no fit */
5     char *bp;
6
7     /* Ignore spurious requests */
8     if (size == 0)
9         return NULL;
10
11     /* Adjust block size to include overhead and alignment reqs. */
12     if (size <= DSIZE)
13         asize = 2*DSIZE;
14     else
15         asize = DSIZE * ((size + (DSIZE) + (DSIZE-1)) / DSIZE);
16
```

## Выделение: собираем все вместе (2)

```
17      /* Search the free list for a fit */
18      if ((bp = find_fit(asize)) != NULL) {
19          place(bp, asize);
20          return bp;
21      }
22
23      /* No fit found. Get more memory and place the block */
24      extendsize = MAX(asize, CHUNKSIZE);
25      if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
26          return NULL;
27      place(bp, asize);
28      return bp;
29  }
```



# Материалы

- Bryant, O'Hallaron — Computer Systems A Programmer's Perspective — глава 9.9
- Donald Knuth — Art of Computer Programming, The: Volume 1: Fundamental Algorithms — глава про Dynamic Memory Allocation
- <https://www.kernel.org/doc/gorman/html/understand/understand009.html>
- <https://github.com/str8outtaheap/heapwn/blob/master/TUCTF/mm.c>
- [https://github.com/giamo/segregated-fits-memory-allocator/blob/master/mm-segregated\\_fits.c](https://github.com/giamo/segregated-fits-memory-allocator/blob/master/mm-segregated_fits.c)
- <https://github.com/abhi195/Dynamic-Storage-Allocator>

