# Выделение памяти ядром

## Операционные системы 2011/12

Татьяна Романова

12-19 ноября 2011 г.

# План на сегодня

- Общие сведения о памяти
- Выделение памяти в ядре
- Требования к алгоритмам
- Алгоритмы:
  - Выделение произвольных блоков из кучи
  - Выделение блоков размером $2^n$
  - Слабовый аллокатор

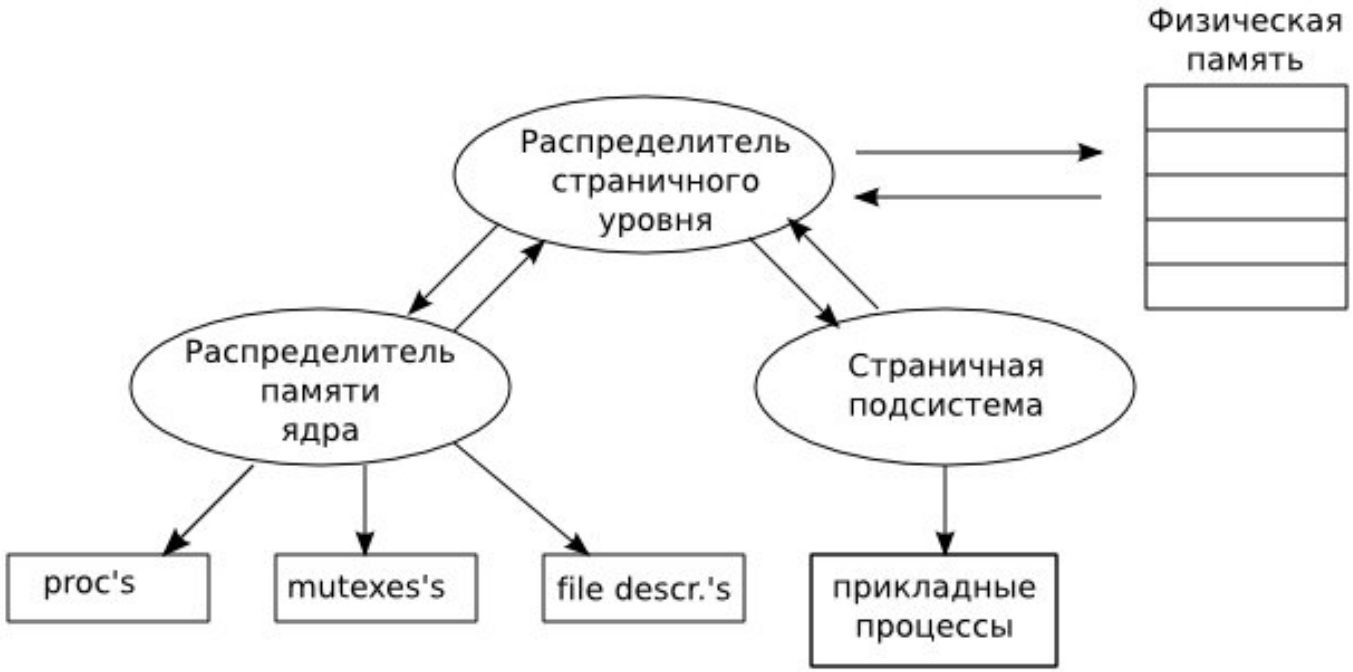
# Материалы

- Ю. Вахалия, UNIX изнутри, глава 12
- Д. Э. Кнут, Искусство программирования, т. 1, п. 2.5
- Р. Лав, Разработка ядра Linux, глава 11

# Общие сведения

- Существуют понятия физической и виртуальной памяти
- ОС управляет физической памятью
  - часть всегда занята кодом и статическими структурами ядра
  - часть выделяется/освобождается динамически по запросам процессов
- Память обычно разбита на страницы — блоки фиксированного размера (степень двойки, например, 4 Кбайта)
- Специальная микросхема процессора MMU (Memory Management Unit) аппаратно транслирует виртуальный адрес страницы в физический
- Для более быстрой трансляции используется кэш TLB (translation lookaside buffer).

# Распределение памяти




- Страничная подсистема выделяет память прикладным процессам, размещая страницы в их адресном пространстве
- Распределитель памяти ядра предоставляет буферы памяти разного размера для подсистем ядра (структуры proc, файловые дескрипторы, объекты синхронизации)

# Распределитель памяти ядра

- Особенность: частое выделение/освобождение структур из небольшого количества классов.
- Необходимо динамическое выделение объектов (раньше применялись статические таблицы фиксированного размера).
- Если нет свободной памяти:
  - заблокировать вызывающий процесс
  - вернуться с ошибкой без блокировки (при попытке выделить память в обработчиках прерываний)
- Должен остлеживать свободные части выделенного страничным аллокатором пула и уменьшать степень фрагментации.

# Критерии оценки аллокаторов

- Фактор использования:

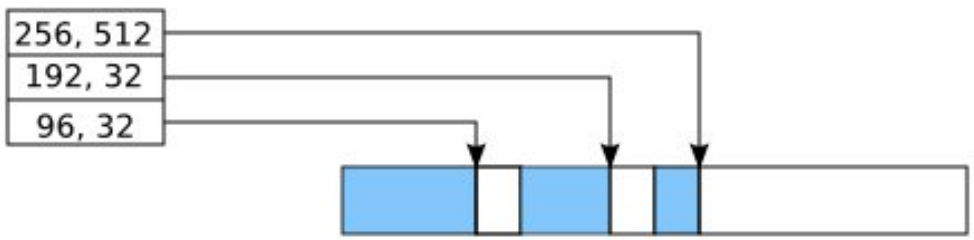
$$\frac{V_{avail}}{V_{tot}},$$

  $V_{avail}$ — доступный объем памяти (память может быть недоступна из-за фрагементации или особенностей аллокатора), $V_{tot}$ — общий объем памяти.
- Среднестатистическая и максимальная задержка при выделении/освобождении объекта.
- Простота интерфейса (например, стандартная функция free не требует указания размеров освобождаемого объекта).

# Выделение произвольных блоков из кучи

Постановка задачи: изначально есть один большой пул памяти, из которого нужно по требованию выделять объекты (и по требованию возвращать память назад в пул).

512 байт памяти после нескольких операций выделения/освобождения (alloc(128), alloc(128), free(96, 32), free(192, 32)):




Вопросы:

1. Как представить свободную память?
2. Какой алгоритм использовать для поиска блока из n свободных ячеек

# Список свободных элементов

1. Свободные блоки организуем в список, блок может хранить:
   - размер
   - адрес следующего свободного

2. Нужно пройти по списку свободных в поисках подходящего участка и
   - уменьшить размер свободного блока с m до $m - n$, если $m > n$
   - удалить блок из списка свободных, если $m = n$

   Методы выбора подходящего участка:
   - Первый подходящий
   - Следующий подходящий
   - Лучший подходящий

# Алгоритм выделения блока

Рассматривается метод первого подходящего (следующий и лучший делаются аналогично), см. Кнута:

- Элемент списка содержит size и next
- В реализации список представлен указателем на голову (нет структуры list, есть только структура elem)
- В функцию alloc передается адрес текущего указателя на голову (alloc(&head, N)), который может измениться.

# Alloc, первый подходящий

```
// *cur == head, N - size to alloc
void *alloc(elem **cur, int N)
{
  // search for first-fit elem
  while (*cur != 0 && (*cur)->size < N)
    cur = &(*cur)->next;
  // couldn't find memory
  if (*cur == 0)
    return 0;
  int diff = (*cur)->size - N;
  if (diff == 0) //delete from free-elems list
    *cur = (*cur)->next;
  else
    (*cur)->size -= N;
  return (*cur) + diff;
}
```

# Выделение блока, детали

Если есть свободный блок, размером $N + 1$, то при выделении блока размером N в оставшемся свободном элементе нельзя будет сохранить размер и указатель на следующий.

Возможные решения:

- Не разрешать выделять такие блоки, проходить вперед в поисках следующего подходящего по размеру.
- Выделять блок большего размера, и хранить размер в занятом блоке тоже. Это затратно, но полезно, теперь пользователь может писать free(T*), не указывая размер освобождаемого блока.

# Освобождение блока

- Список свободных упорядочен по адресам (cur $<$ cur $\rightarrow$ next)
- При вставке нужно иногда сливать свободные элементы
- Алгоритм free(ptr):
  - Идем по списку, поддерживая указатели на предыдущий prev и текущий cur свободные элементы, пока cur $<$ ptr
  - Слияние со следующим: если ptr + ptr->size == cur, то ptr->size += cur->size, ptr->next = cur->next, иначе ptr->next = cur
  - Слияние с предыдущим: если prev + prev->size == ptr, то prev->size += ptr->size, prev->next = ptr->next, иначе prev->next = ptr

# Освобождение блока, детали

Сложность освобождения с односвязным списком O(n):

- со след. можно слить за O(1)
- для поиска предыдущего необходим проход по списку

Модификация: используем двусвязный список

- в каждом занятом храним размер в начале и тег «занят» в начале и в конце
- в каждом свободном храним размер, тег «свободен» и два указателя на предыдущий и следующий свободные элементы

# Освобождение: модификация с двусвязным списком

Детали алгоритма необходимо посмотреть в Кнуте

- Зная адрес освобождаемого, смотрим на тег элемента перед ним
- Если элемент свободен, сливаем с ним
- Зная адрес освобождаемого и размер, смотрим на тег элемента за ним
- Если свободен, сливаем.

# Анализ рассмотренных алгоритмов

Плюсы:

- Простота
- Есть реализация, где можно выделять ровно запрошенное количество байтов
- Есть реализация, где можно освобождать меньше байтов, чем выделили
- Слияние при освобождении снижает степень фрагментации

Минусы:

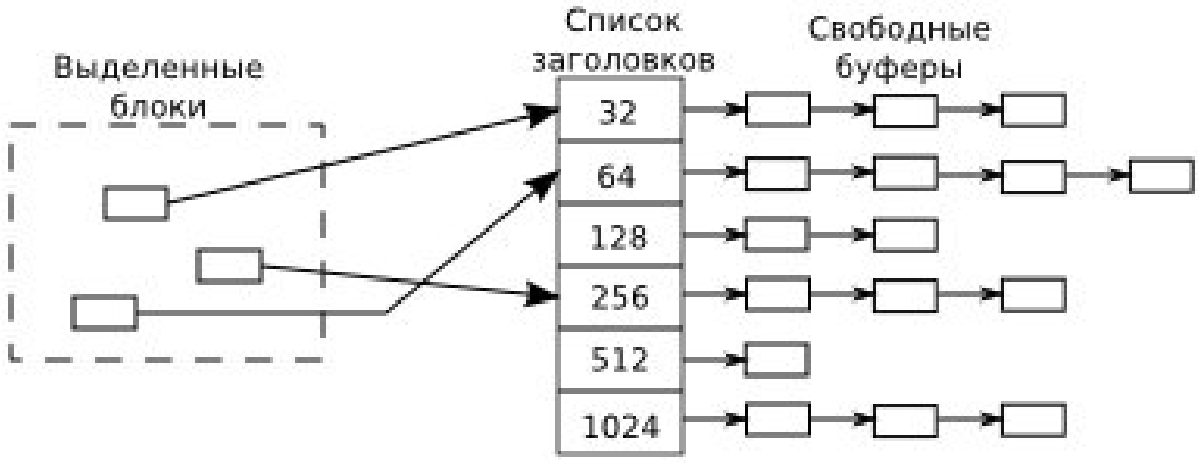
- Чем дальше работает со случайными освобождениями/выделениями, тем больше фрагментация и тем длиннее список свободных
- Требует дополнительной памяти для хранения размера блока
- Нетребовательные по памяти реализации нуждаются в линейном проходе по списку свободных при выделении и освобождении

# Выделение памяти блоками по $2^n$

Основная идея:

- Вся память делится на списки свободных элементов
- В каждом списке находятся элементы определенного размера
- У каждого элемента есть заголовок. Если элемент свободен, там хранится адрес следующего,если занят, адрес головы списка, в который нужно вернуть элемент
- При выделении нужно найти подходящий список $N_{list} = \lceil \log_2 n \rceil$, если в текущем списке нет свободных — искать в следующих
- При освобождении нужно довабить текущий блок к указанному в нем списку

# Блоки по $2^n$: пример и анализ




Плюсы:

- Простота
- Скорость работы
- При освобождении не нужно указывать размер

Минусы:

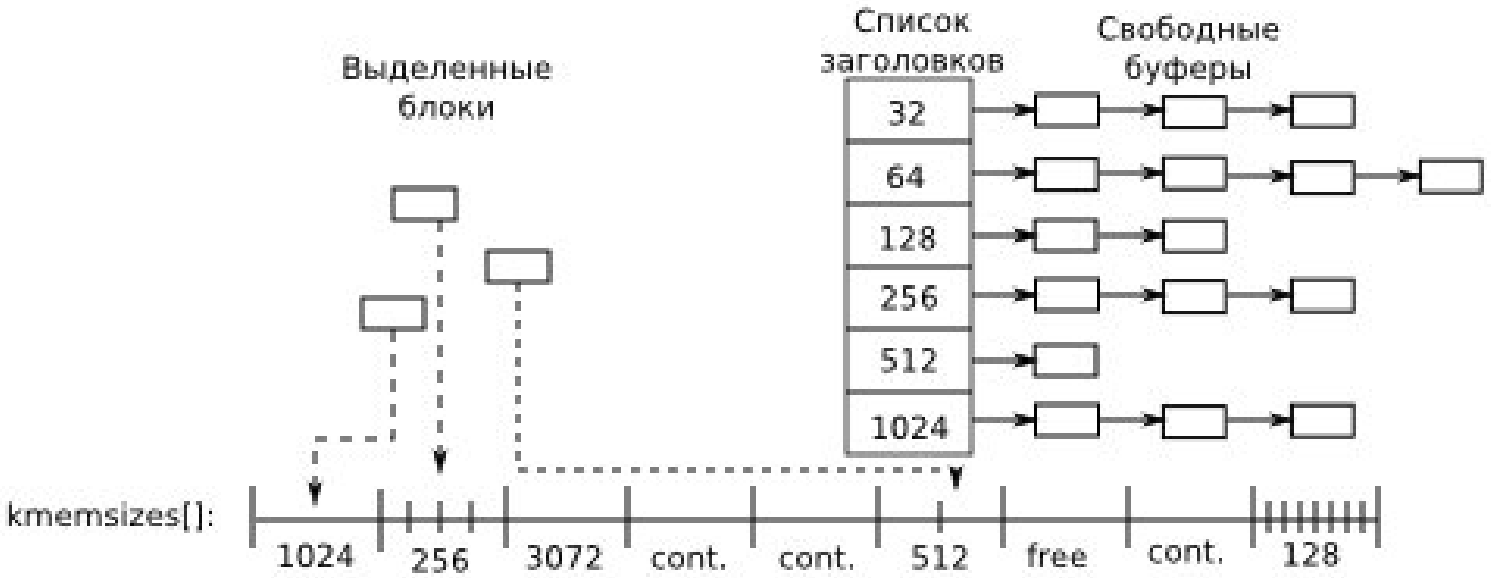
- Округление до $2^n$ приводит к использованию лишь части блока
- В занятых элементах хранится указатель на голову списка $\Rightarrow$ худший случай — выделение $2^n$ байт.
- Нет слияния буферов и возвращения памяти страничному аллокатору.

# Алгоритм Мак-Кьюзика-Кэрелса

Предыдущий алгоритм использоваться в 4.2BSD в качестве пользовательского аллокатора.
Аллокатор ядра 4.3BSD — гибрид предыдущей версии и первого подходящего.
Улучшения:

- пул памяти поделен на страницы (по 1 kB), страницами управлял массив kmemsizes[]
- для маленьких блоков — как из 4.2BSD (но поиск нужного списка более быстрый засчет использования макроса)
- при освобождении маленького блока не требуется знать его размер, он хранился в kmemsizes[]
- для блоков больших 2 kB, выделяются несколько последовательных страниц методом первого подходящего

# Алгоритм Мак-Кьюзика-Кэрелса: иллюстрация




При освобождении (free(addr)) страница ищется как (addr – kmembase)/PAGESIZE, где kmembase — адрес начала массива kmemsizes. Когда нашли страницу, стал известен размер блока, нужно добавить addr в список свободных блоков соответствующего размера.

# Алгоритм двойников

- Выделяемые блоки имеют размер $2^n$, изначально есть пул размером $2^m$
- Организуются отдельные списки блоков размера $2^k, 0 \leqslant k \leqslant m$
- При требовании $2^k$ слов и отсутствии такого свободного блока, бо̀льшие свободные блоки бьются на 2
- Части разделенного блока называют двойниками (buddies), если оба двойника свободны, их можно объединить.
- В каждом блоке нужно хранить тег «свободен/занят»
- Ключевой момент: если известен адрес блока и размер, то можно найти адрес двойника.

# Алгоритм двойников: пояснения

Пример: двойником блока с адресом 10100000 размером 8 будет блок с адресом 10101000.

Почему так?

- Адрес блока размером $2^k$ кратен $2^k$ (доказывается по индукции: для базового случая $2^m$ это верно. Если это верно для $2^{k+1}$, то это верно и половинок, размером $2^k$: на месте $k+1$ нуля остается 0 для одной половины и появляется 1 для другой.
- Если блок размером 16 имеет адрес xxxx0000, то при разделении получим два блока по 8: xxxx0000 и xxxx1000
- Имеем:

$$\mathrm{buddy_k}(x) = \begin{cases} x + 2^k, & \text{если } x \bmod 2^{k+1} = 0; \\ x - 2^k, & \text{если } x \bmod 2^{k+1} = 2^k \end{cases}$$

Вычисляется битовыми операциями (как?)

# Алгоритм двойников: выделение памяти

Для свободных блоков используются двусвязные списки, каждый блок помечен тегом «свободен/занят».

- Находим наименьшее j, такое что $k \leqslant j \leqslant m$, для которого список свободных блоков не пуст (если такого нет, выделение невозможно)
- Удаляем найденный блок из списка свободных
- Если $j = k$, то разделения не требуется, возвращаем найденный блок
- Если $j > k$, то уменьшаем j на 1, получаем двойников, один из них становится первым в j-м списке свободных, со вторым переходим к предыдущему шагу (возможно, нужно более мелкое разделение).

# Алгоритм двойников: освобождение

- Найти двойника и по тегу проверить, свободен ли он.
- Если свободен, объединить блоки, удалив двойника из списка свободных, получить новый блок, большего размера, вернуться к предыдущему шагу.
- Если занят, добавить блок в список свободных соответствующего размера.

# Алгоритм двойников: анализ и улучшения

Плюсы:

- Объединяет смежные буферы, получаем гибкое автоматическое распределение памяти по спискам свободных блоков заданного размера.
- Легко обмениваться памятью со страничным распределителем.

Минусы:

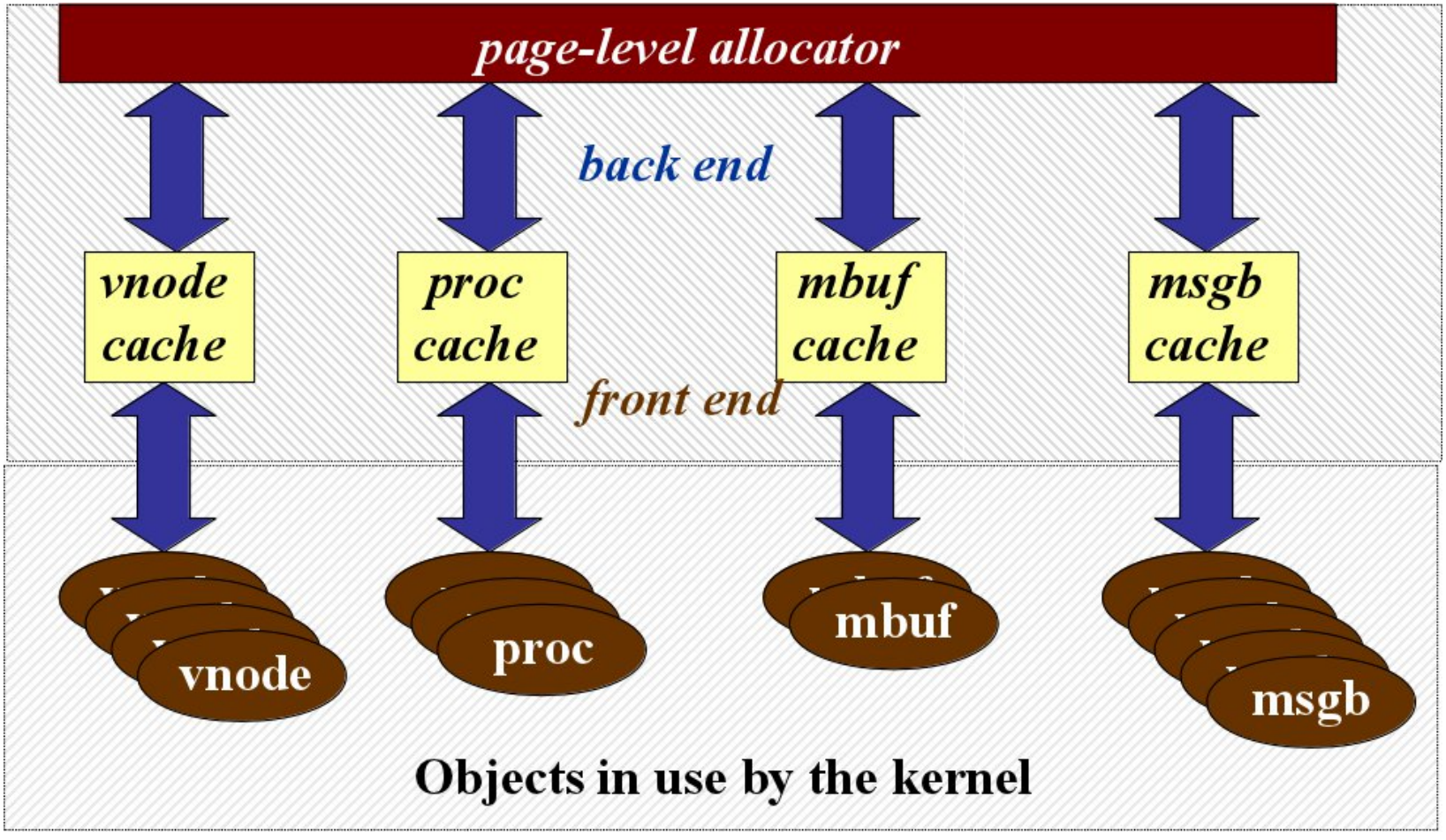
- При чередовании выделения/освобождения низкая производительность, так как идет постоянное разбиение и сливание блоков
- Можно выделить только блок кратный степени двойки
- При удалении обязательно указывать размер

В качетсве улучшения можно испльзовать отложенное слияние: можно сливать блоки не сразу, а только в тот момент, когда будет нехватка блоков большего размера.

# Слябовый аллокатор (slab allocator)

- Используется во многих ОС.
- Объекты используемые ядром проходят один и тот цикл: выделение памяти - создание - использование - уничтожение - освобождение.
- Возможно повторное использование объекта без повторной инициализации.
- Объекты ядра делятся на группы-кэши, кэш делится на слябы (один сляб — одна или несколько последовательных страниц), которые содержат объекты. При создании пытаемся достать объект из сляба. При удалении объект в слябе помечается как свободный.
- Новый кэш можно создать с помощью функции ядра: kmem_cache_create(name, size, offset, flags, ctor, dtor); Получить объект: kmem_cache_alloc(cachep, flags);

# Слябовый аллокатор: иллюстрация



# Заключение

- Выделение памяти в ядре обладает некоторыми особенностями:
  - Фиксированный небольшой стек
  - Максимальная скорость операций
  - Нужно стремиться к уменьшению расходов на структуры ядра
  - Нужен универсальный механизм для различных объектов
  - Набор объектов ядра фиксирован
- Первый подходящий (и аналоги) подходят для редкого выделения больших объектов, требуется линейный поиск, доп. затраты могут быть малы.
- Блоки по степеням двойки — быстро работает, для больших блоков может тратить много лишней памяти.
- Алгоритм двойников более гибкий за счет динамического слияния, разделения блоков.
- Слябовый аллокатор быстрый и экономный за счет повторного использования инициализировнных объектов.