# Operating Systems

## Memory Allocation

Me

September 20, 2016

# Постановка задачи

- имеется регион памяти
  - или несколько регионов памяти;
  - границы регона/регионов известны;
- хотим научится отвечать на запросы:
  - *alloc(size)* - алокация свободного региона размером *size* байт;
  - *free(addr)* - освобождение занятого региона начинающегося по адресу addr;
  - *addr* - адрес возвращенный на запрос *alloc*;

# Выравнивание алоцированной памяти

- Адрес возвращенный в ответ на запрос *alloc* должен быть "выровнен":
  - некоторые архитектуры могут не уметь читать/писать данные по невыровненным адресам;
  - даже если конкретная архитектура умеет, это может приводить к падению производительности;
  - требуемое выравнивание зависит от операций, которые мы будем делать с этой памятью;
  - т. к. нам не известно, как будет использована алоцированная память, она должна быть выровнена под любые варианты использования.

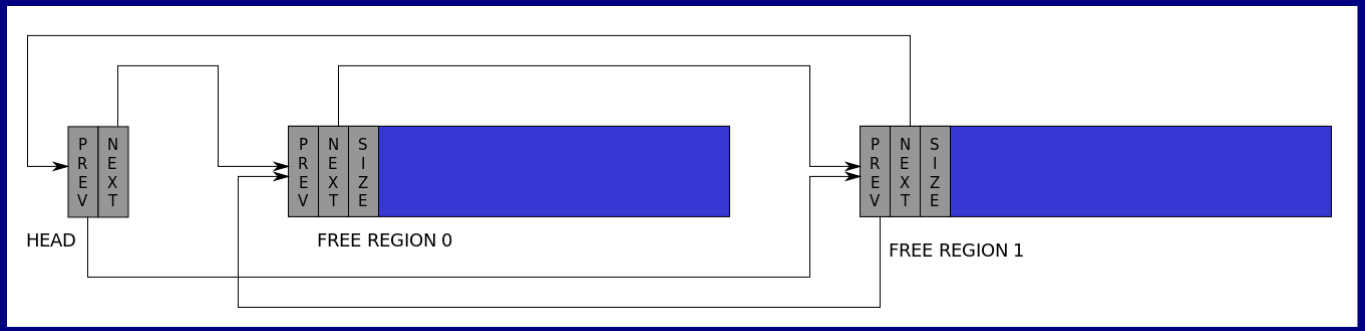
# x86 vs ARM

- x86 архитектура разрешает обращение по невыровненым адресам:
  - можно установить бит 18 в регистре RFLAGS, тогда в непривилигированном режиме обращение по невыровненым адресам будет генерировать исключение (для приложения выглядит как SIGBUS);
  - обычно это не заметно, если вы не используете невыровненный адрес слишком часто;
- ARM изначально запрещает доступ по невыровненным адресам:
  - ARM использует "натуральное" выравнивание, т. е. чтение/запись 1 байта не ограничивается, чтение/запись 2-ух байт должно быть выровнено на 2 байта и тд.

# C и выравнивание

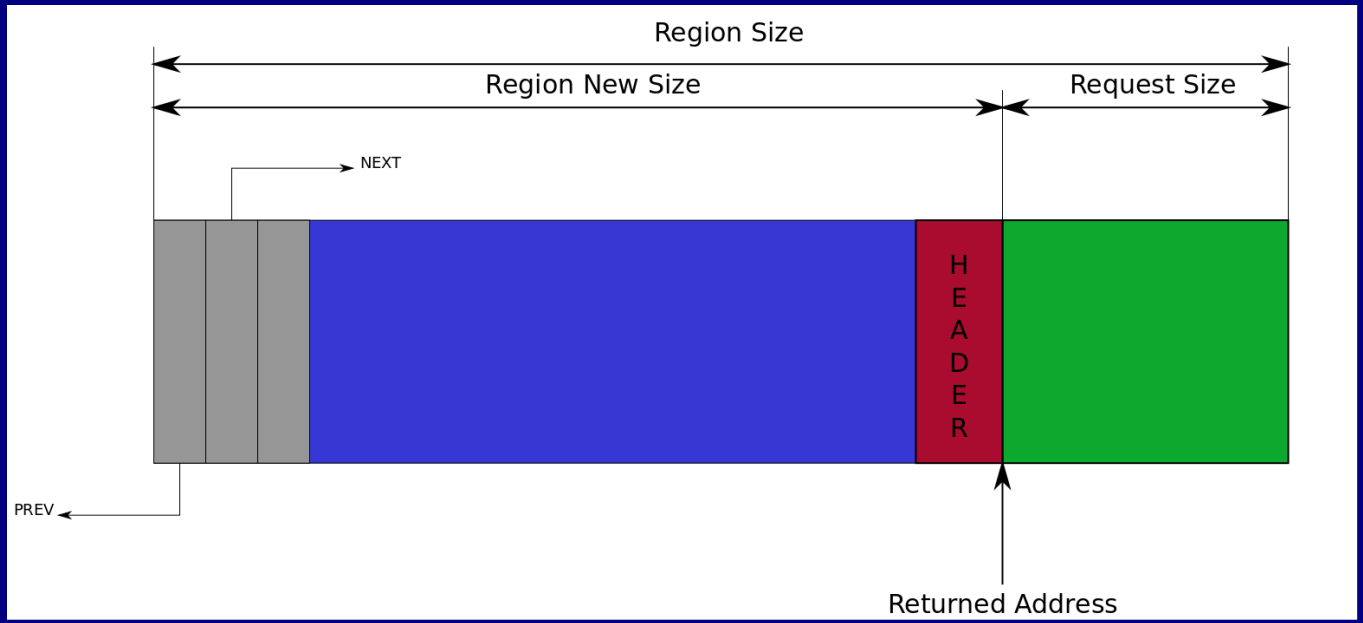
```
1  void foo(void)
2  {
3      /* char requires no alignment + stack
4         allocated = no alignment guaranteed */
5      char buf1[4];
6      /* malloc guarantees reasonable
7         alignment */
8      char *buf2 = malloc(4);
9
10     /* uint32_t might require 4 byte alignment,
11        works fine on x86, might fail on ARM */
12     uint32_t val1 = *((uint32_t *)buf1);
13     /* buf2 properly aligned, works fine */
14     uint32_t val2 = *((uint32_t *)buf2);
15 }
```

# Связный список на свободной памяти



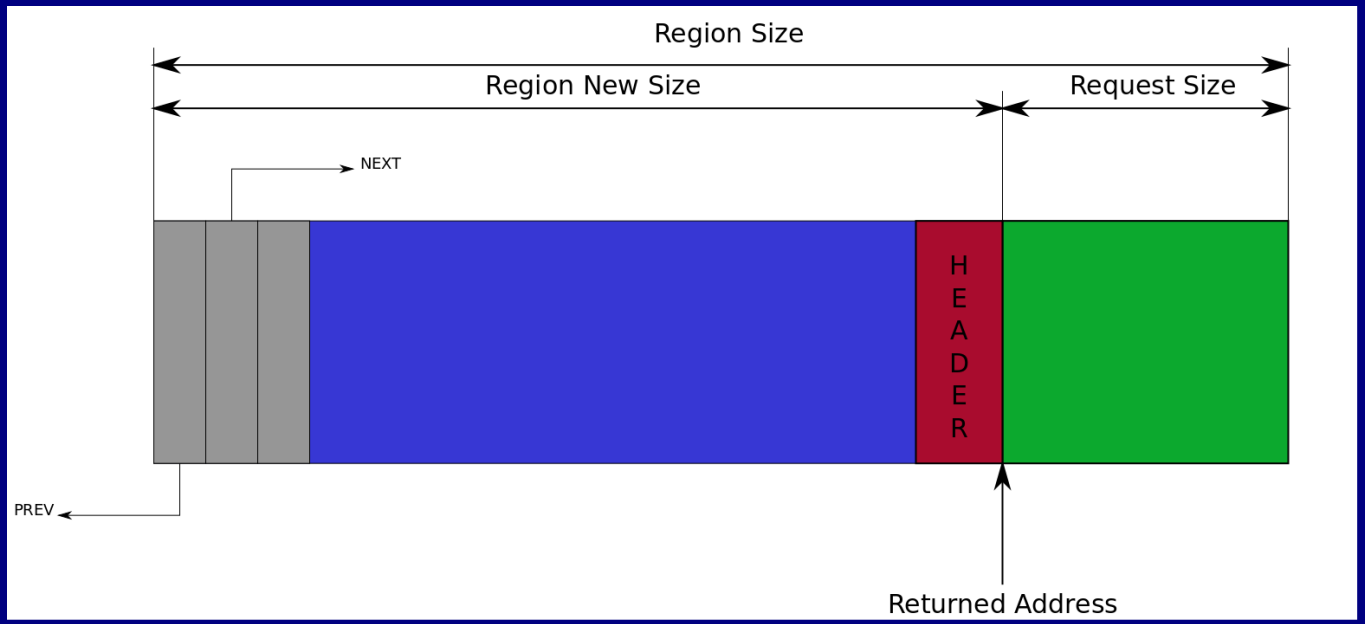

- узлы списка создаются прямо в свободных регионах памяти;
- дополнительно храним указание на размер регона.

# Алокация



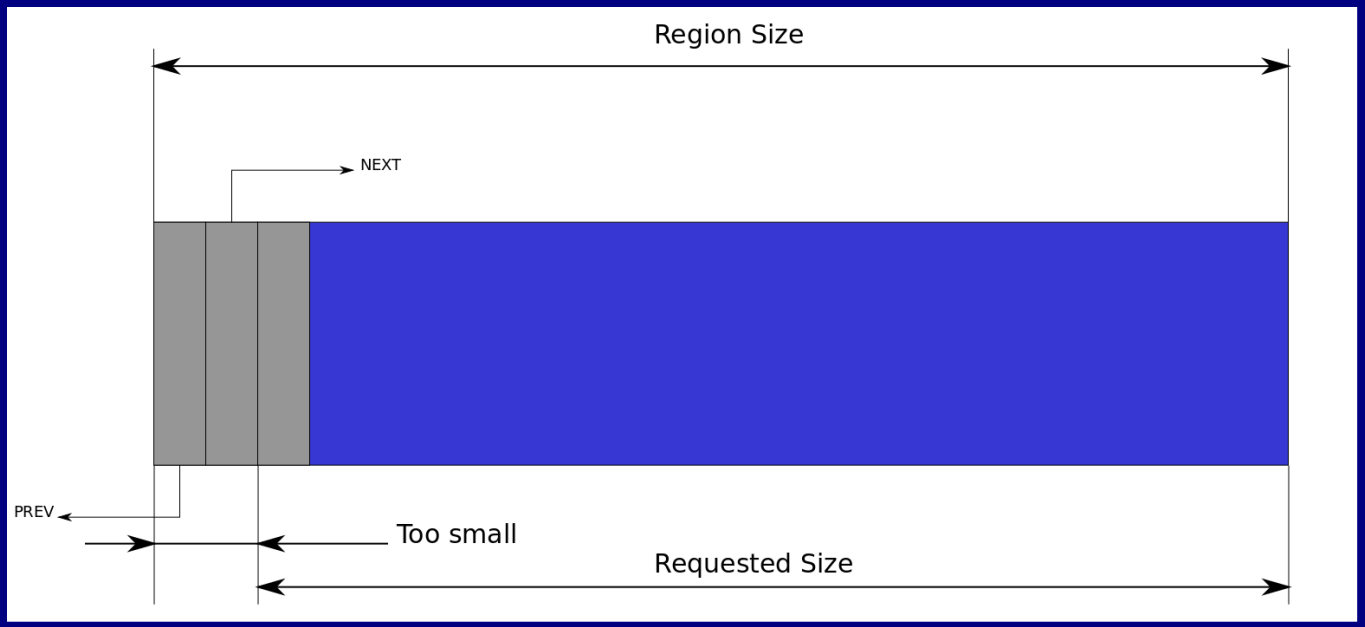

- Итерируемся по списку и ищем блок достаточного размера;
- от найденного свободного блока "отрезаем" участок нужного размера;
  - возможно придется удалить блок из списка.

# Заголовок алоцированной памяти



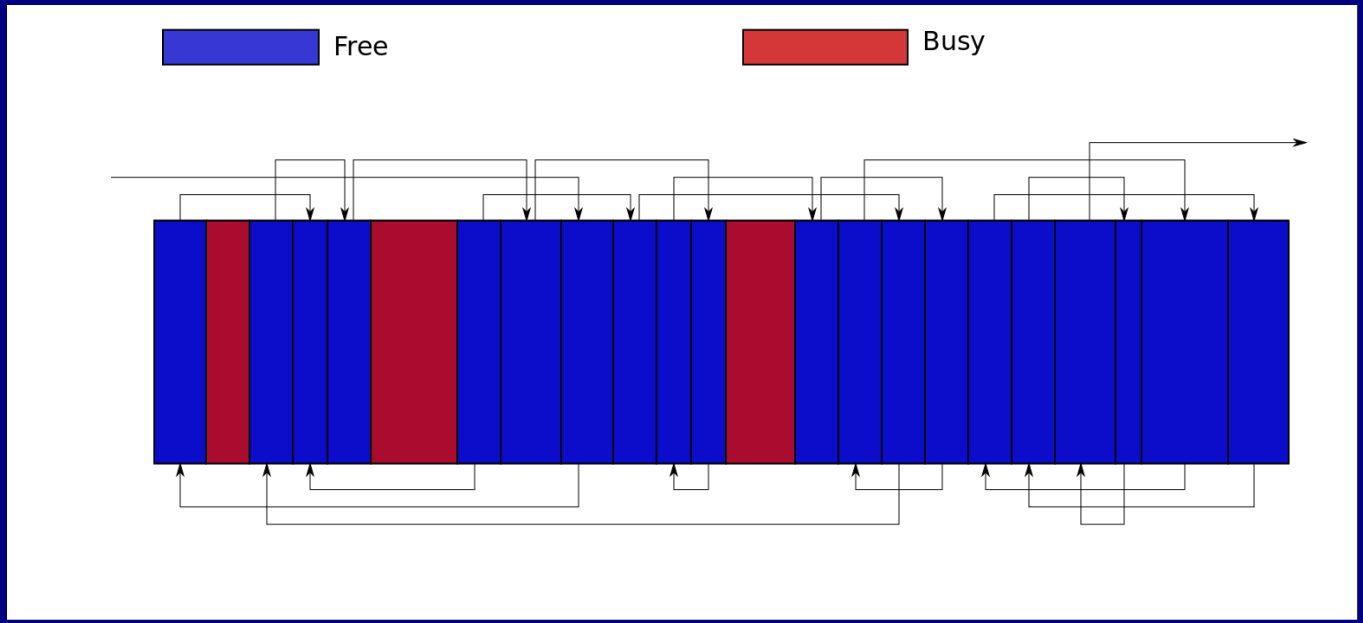

- Перед участком (и/или после) можно добавить заголовок, который может хранить:
  - размер алоцированного участка;
  - magic значение для перехвата ошибок;
  - border tag для ускорения освобождения.

# Размер алоцированого участка




- Оставшегося места может быть не достаточно:
  - меньше чем размер узла списка;
  - такой участок придется удалить из списка;
- вернем в ответ на запрос чуть больше памяти
  - придется хранить размер алоцированного участка.

# Освобождение алоцированного участка

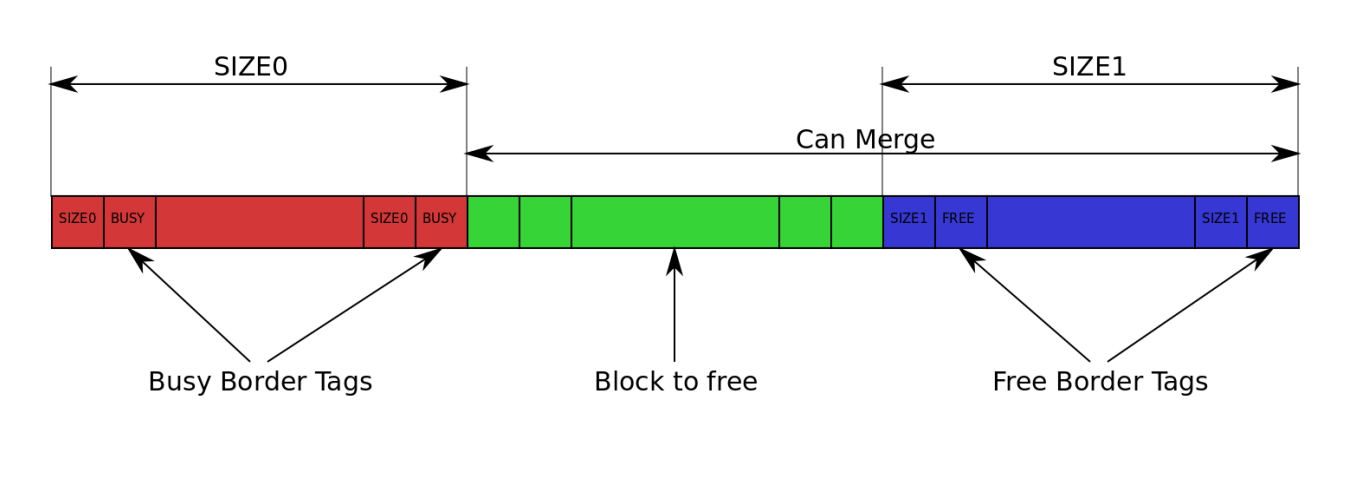



- Из заголовка легко найти размер участка
  - а заголовок легко находится по адресу участка;
- просто добавим новый элемент в список блоков
  - приведет к фрагментации свободного места;
  - нужно объединять смежные свободные блоки.

# Объединение смежных блоков

- Проход по списку блоков
  - медленно, т. е. мы можем гораздо лучше;
- использовать дерево вместо списка:
  - освобождение за $O(logn)$, где $n$ - количество свободных блоков;
  - можно амортизировать, выполняя объединение переодически;
  - обычно $O(logn)$ не недостаток, но есть решение за $O(1)$.
- использовать border tag-и.

# Border Tag




- Добавим заголовок в начало и конец свободных и занятых блоков
  - в заголовок добавим специальное поле Border Tag - признак свободности/занятости блока;
  - имея на руках блок (свободный или занятый) мы всегда можем найти Border Tag-и смежных блоков;
  - зная Border Tag мы можем проверить свободен или занят смежный блок.

# Ошибки при работе с памятью

- Типичные ошибки при работе с памятью:
  - запись за пределы алоцированного участка;
  - чтение за пределами алоцированного участка;
  - освобождение неправильного указателя (free на адресе, который не был возвращен из alloc).
- Алокатор хранит служебные данные рядом с пользовательскими данными:
  - если пользователь ошибется пострадает работа алокатора;
  - ошибка может проявится в неожиданном месте - трудно понять причину;
  - по крайней мере часть проблем мы можем отследить простым методом.

# Magic Numbers

- Добавим в заголовок магическое число:
  - это может быть любое фиксированное число, но лучше не использовать часто встречающиеся паттерны, например, 0 или 0xffffffff;
  - например, 0x13134242;
- при освобождении проверяем магическое число освобождаемого блока, а так же смежных блоков:
  - таким образом легко отловить освобождение некорректного укзатателя;
  - некоторые часто встречающиеся случаи записи за пределы выделенной памяти.

# Разделение на большие и маленькие алокации

- До сих пор мы не учитывали никак сколько и зачем мы алоцируем память:
  - часто нужно алоцировать память под объекты одного типа и, соответственно, размера;
  - большие участки памяти алоцируются реже и живут, зачастую, долго;
- Разобьем задачу на несколько более простых задач:
  1. научимся эффективно алоцировать большие регионы памяти;
  2. научимся эффективно алоцировать регоны фиксированного размера.

# Алокация больших регионов

- Будем алоцировать только регионы кратные некоторому размеру *PAGE_SIZE*:
  - *PAGE_SIZE* выбирается исходя из задачи;
  - например, *PAGE_SIZE* может быть 4Kb или около того;
- не будем добавлять к аллоцированной памяти никаких заголовков и прочего:
  - в отличие от предыдущего алгоритма в этом нет необходимости;
  - мы всегда будем алоцировать столько, сколько запросили и пользователь сам может хранить размер;
- алоцированные регионы всегда выровнены на границу *PAGE_SIZE*.

# Buddy Allocator

- Будем алоцировать блоки не просто кратные $PAGE\_SIZE$, а блоки размера $2^i \times PAGE\_SIZE$:
  - не смотря на ограничение, алгоритм очень практичный и широко применяется;
- Разделим всю память на регоны размером $PAGE\_SIZE$:
  - пронумеруем регионы начиная с 0;
  - номера регонов однозначно сопоствляются адресам, далее мы будем работать только с номерами.

# Порядок блока

- $2^i$ смежных регонов назовем блоком порядка $i$:
  - блок порядка 0 имеет размер $PAGE\_SIZE$, блок порядка 1 имеет размер $2 \times PAGE\_SIZE$, для порядка 2 размер $4 \times PAGE\_SIZE$ и т. д.;
  - блок однозначно определяется порядком и номером первого региона (далее просто номер);
  - нас не интересуют блоки не выровненные на свой размер (например, блок порядка 1 с начинающийся в регионе 1 нас не интересует);
- Свободные блоки одного порядка "связаны" в список:
  - для каждого возможного порядка по отдельному списку (их не может быть очень много).

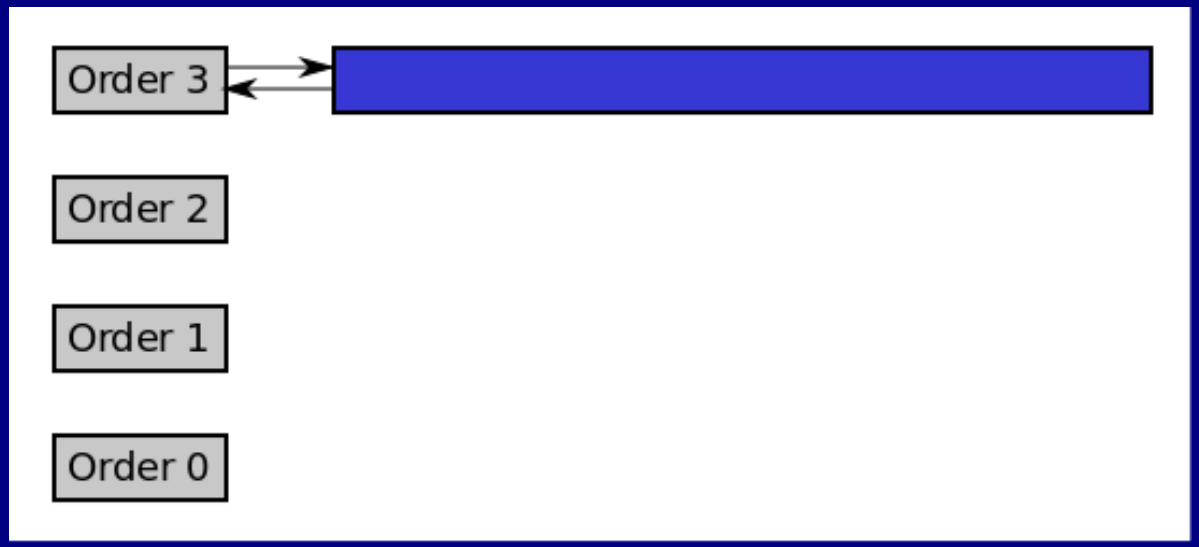
# Buddies 1/2

- Некоторые пары блоков будем называть buddies (товарищи, дружбаны и т. д.):
  - два блока поряка $i$ товарищи, если их номера отличаются только $i$-ым битом считая с 0, где 0-ой бит самый младший;
  - блоки порядка 0 с номерами 0 и 1 - товарищи, а блоки с номерами 1 и 2 нет;
  - блоки порядка 1 с номерами 0 и 2 - товарищи, а блоки с номерами 1 и 3 нет, и т. д.
- Для блока с номером $Block_{no}$ и порядком $i$ найти номер товарища легко:
  - $Buddy_{no} = Block_{no} \oplus 2^i$;
  - $Buddy_{no}$ - номер товарища;
  - $\oplus$ - побитовый *xor*.

# Buddies 2/2

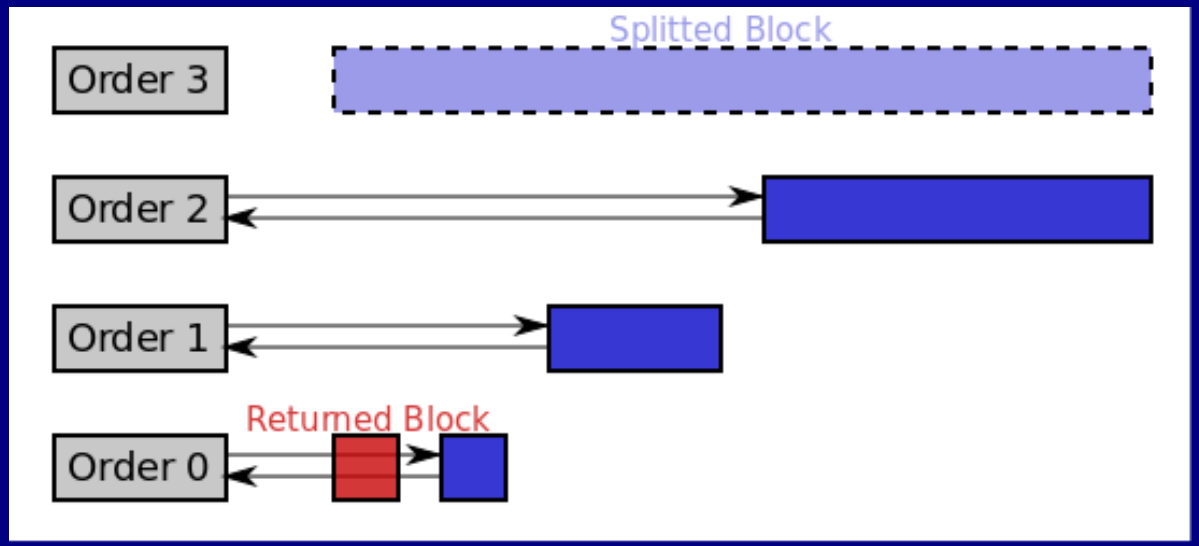
- Два buddy блока порядка $i$ в объединение дают блок порядка $i + 1$:
  - при освобождении мы будем объединять свободные buddy блоки в блоки большего порядка;
  - будем продолжать объединение рекурсивно, пока можно объединять или пока не достигнем максимального порядка;
- и наоборот, блок порядка $i + 1$ можно разбить на два buddy блока порядка $i$:
  - при алокации мы будем делить большие блоки на меньшие, пока не дойдем до нужного порядка.

# Алокация 1/3



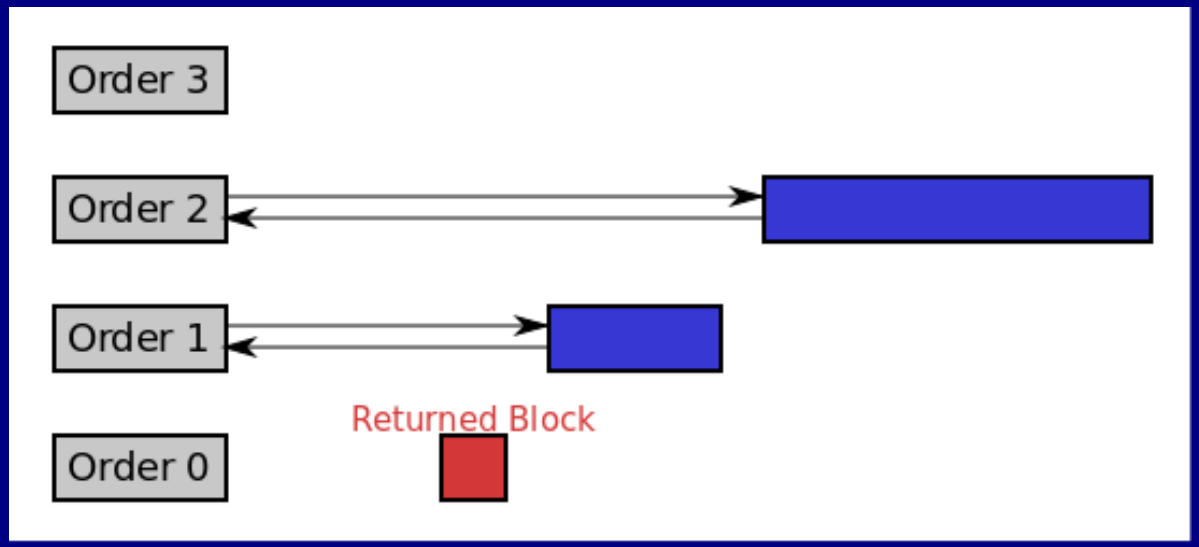

- Для примера ограничемся только порядками от 0 до 3;
- допустим вся память описывается блоком порядка 3:
  - начальная конфигурация может состоять из нескольких блоков;
- будем алоцировать блоки порядка 0:
  - можно алоцировать блоки произвольного порядка.

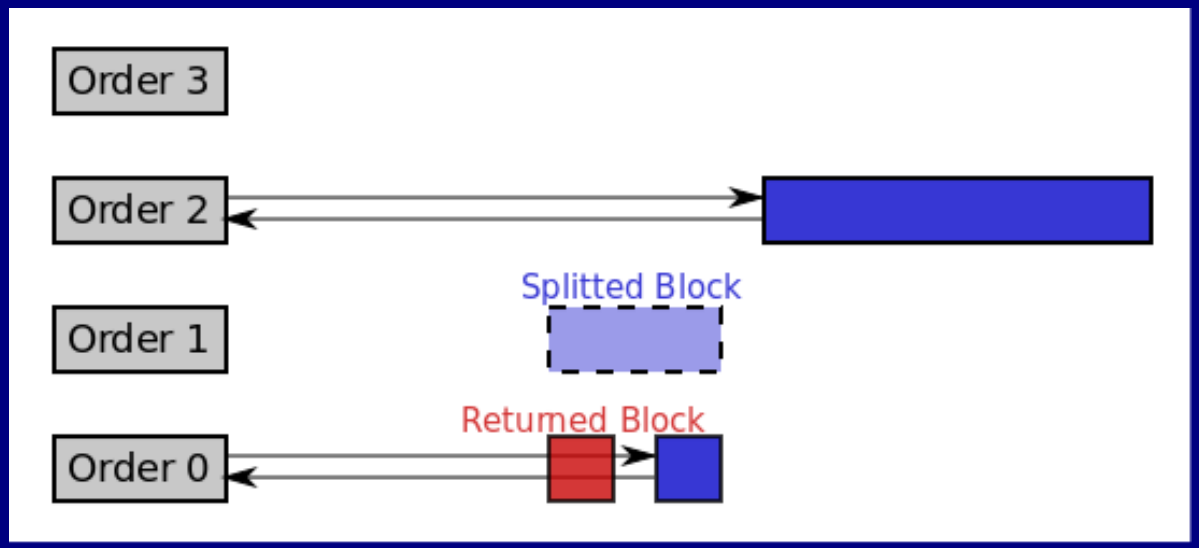
# Алокация 2/3



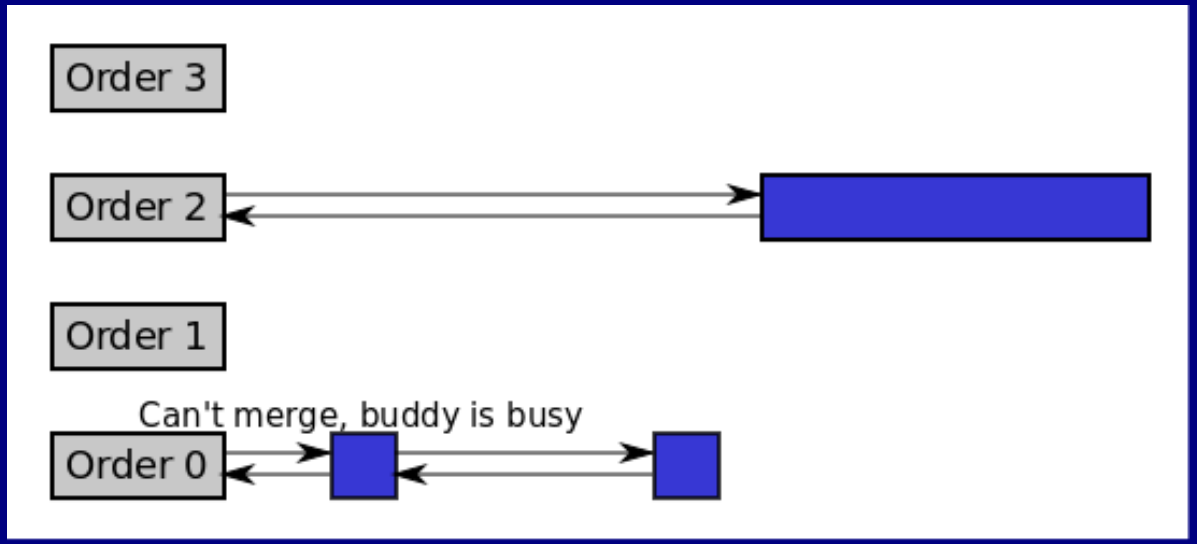

- Список с блоками нужного размера может быть пуст
  - можем взять блок большего размера и разбить его;
- делим блок пока не останется блок нужного размера:
  - ненужную половину добавляем в соответствующий список.

# Алокация 3/3


Order 3
Order 2
Order 1
Returned Block
Order 0

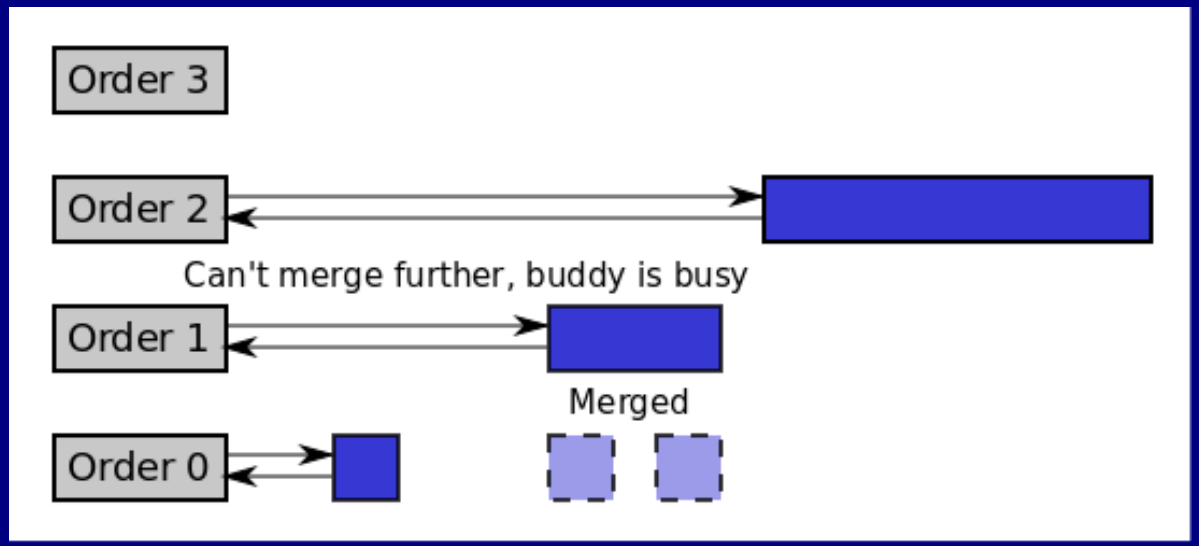

Order 3
Order 2
Splitted Block
Order 1
Returned Block
Order 0

# Освобождение 1/2




- При освобождении блок добавляется в соответствующий список
  - при освобождении нужно проверить свободен ли buddy блок.

# Освобождение 2/2




- Если buddy блок свободен, то можно блоки можно вновь объединить
  - для объединенного блока также нужно проверить buddy блок.

# Дескрипторы регионов

- Для блоков нам необходимо уметь:
  - определять свободен блок или нет;
  - уметь связать блоки в список;
- заведем дескриптор для каждого региона размера *PAGE_SIZE*:
  - дескрипторы можно связывать в список (хранят *prev* и *next*);
- дескриптор первого региона блока - представитель блока:
  - в нем хранится признак свободности/занятости;
  - порядок блока, представителем которого он является;
  - блок порядка $i$ свободен, если *представитель свободен и порядок представителя равен* $i$.

# Проблема курицы и яйца

- Где брать память под дескрипторы?
  - алоцировать фиксированное заранее количество дескрипторов
    - возможно придется алоцировать с большим запасом;
  - использовать другой алокатор памяти, чтобы алоцировать дескрипторы
    - например, описанный ранее алгоритм или более простой;
    - нам не нужно освобождение, чтобы инициализировать Buddy алокатор.

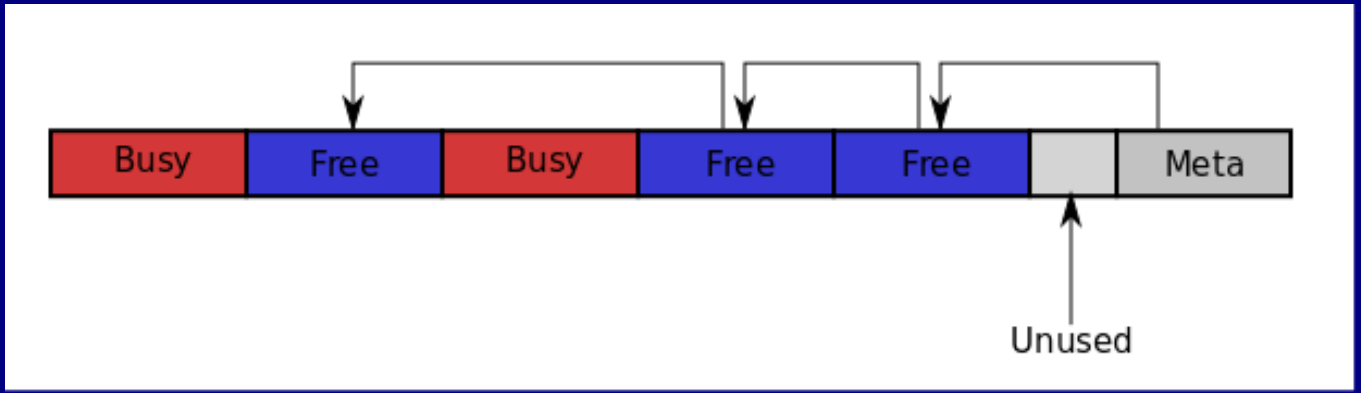
# Кеширующий алокатор

- Мы умеем алоцировать большие блоки памяти
  - мы можем алоцировать большие блоки и разделять их на меньшие блоки фиксированного размера;
  - работать с блоками фиксированного размера проще.
- Для объектов одного типа можно сэкономить на инициализации:
  - некоторые поля объектов при освобождении находятся в "исходном" состоянии;
  - например, mutex/lock должен быть отпущен перед освобождением памяти, а счетчик ссылок скорее всего равен 0/1;
  - инициализировать такие поля при повторной алокации не нужно;
  - Jeff Bonwik, The Slab Allocator: An Object-Caching Kernel Memory Allocator.

# Slab Allocator

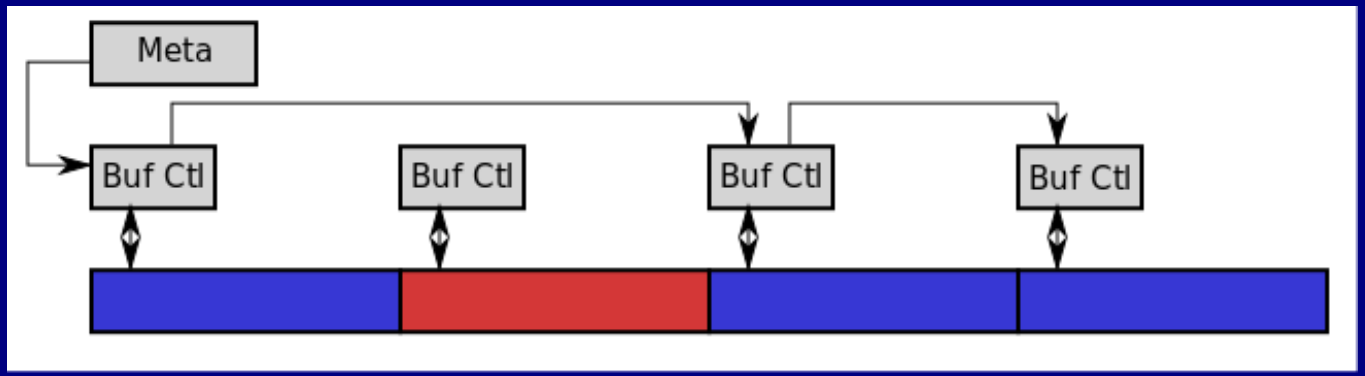
- Базовым понятие Slab Allocator-а является Slab:
  - Slab - пул/кеш объектов/регионов (далее просто объекты) памяти фиксированного размера;
  - Slab - большое регион памяти, который мы разделяем на меньшие кусочик, а также управляющие структуры необходимые для алокации;
- Для каждого объекта Slab-а есть дескриптор:
  - дескрипторы свободных объектов связаны в список;
  - все объекты одного размера, при алокации мы можем взять любой из них.

# Slab для маленьких объектов




- Для маленьких объектов сами объекты и управляющие структуры можно хранить вместе:
  - пока объект свободен мы можем его использовать как дескриптор;
  - занятые объекты мы никак не отслеживаем.
- Какие объекты считать маленькими?
  - в оригианльной статье объекты меньше $1/8 \times PAGE\_SIZE$.

# Slab для больших объектов




- Для больших объектов управляющие структуры алоцируются отдельно:
  - мы можем алоцировать их как маленькие объекты;
  - нам нужно знать был ли объект уже инициализирован или нет;
  - если вы не оптимизируете инициализацию - вам это не нужно.

# Информация о доступной памяти

- До сих пор мы знали границы памяти, из которой мы удовлетовряем запросы на алокацию
  - откуда берется это знание?
- Для ядра ОС:
  - из спецификации оборудования;
  - от загрузчика/BIOS/UEFI или любого другого ПО, которое запускает ОС;
  - например, multiboot загрузчик может предоставить карту памяти;
- для обычных приложений:
  - через интерфейс ОС;
  - не редко ОС позволяет уменьшить/расширить границы памяти;
  - например, в Unix-like системах есть вызовы *brk*/*sbrk*, а так же *mmap*/*munmap*.

# Q&A