SLAB allocation

Архитектура, история и применение

Мочеков Семён 409184, ИТМО

Проблематика управления памятью

Управление памятью – одна из ключевых задач операционной системы. Обеспечение эффективного распределения памяти и грамотное использование ограниченных ресурсов – то, что требуется от распределителя памяти. Некоторые методы распределения памяти, такие как *Buddy allocation*, сталкиваются с проблемами фрагментации и высокой нагрузкой на процессор при частом выделении и освобождении памяти. С увеличением масштабов вычислений и усложнением системного ПО возникла потребность в более гибких и производительных механизмах работы с памятью.

SLAB-аллокатор был предложен как решение этих проблем, обеспечивая быструю работу с памятью и снижая фрагментацию. Его основной принцип заключается в создании пулов заранее инициализированных объектов, что позволяет избежать издержек на частую инициализацию и освобождение памяти.

Использование данной реализации аллокатора особенно эффективно в средах, где часто требуется создание и удаление большого количества однотипных объектов, таких как структуры данных ядра или элементы сетевых буферов. Этот подход значительно повышает производительность операционных систем, минимизируя задержки при управлении памятью.

Внедрение SLAB, конечно, не лишено недостатков. Сложность реализации и необходимость ручной настройки параметров могут привести к избыточному потреблению памяти или неэффективному распределению ресурсов.

Таким образом, совершенствование методов управления памятью, включая развитие SLAB-аллокатора, остаётся важным направлением для повышения эффективности операционных систем и решения современных задач в условиях роста вычислительных нагрузок.

Идеи SLAB используются в операционных системах до сих пор, так что детальное рассмотрение данной концепции крайне полезно в изучении устройства операционных систем.

История SLAB

SLAB-аллокатор был впервые предложен Джеффом Бонвиком (*Jeff Bonwick*) в 1994 году для операционной системы *SunOS* (*Solaris)*. Его цель заключалась в решении проблем, связанных с фрагментацией памяти и высокой стоимостью выделения и освобождения объектов. В отличие от имеющихся тогда методов, SLAB-аллокатор использует кэширование объектов, что значительно снижает накладные расходы и улучшает производительность.

Позже SLAB-аллокатор был адаптирован для других UNIX-подобных операционных систем. Однако со временем были выявлены некоторые недостатки оригинальной реализации, такие как высокая сложность кода и избыточное потребление памяти.

В результате появились логические продолжения:

- **SLUB (The Unqueued Slab Allocator)** — предложенный для упрощения структуры SLAB и снижения накладных расходов. В ядре Linux по умолчанию используется именно SLUB с версии 2.6.23.
- **SLOB (Simple List of Blocks)** — облегчённый аллокатор для систем с ограниченными ресурсами.

Эти аллокаторы развивали идеи SLAB, адаптируя их под различные сценарии использования, балансируя между производительностью, простотой реализации и экономией памяти.

Основная единица такой системы – slab. В дальнейшем будем называть этим словом непрерывную область памяти, которой оперирует SLAB-аллокатор.

Принципы работы SLAB

SLAB-аллокатор основан на идее предварительного выделения и кэширования объектов для оптимизации операций выделения и освобождения памяти. Такой подход позволяет значительно сократить издержки на частую инициализацию объектов и минимизировать фрагментацию памяти.

Структура

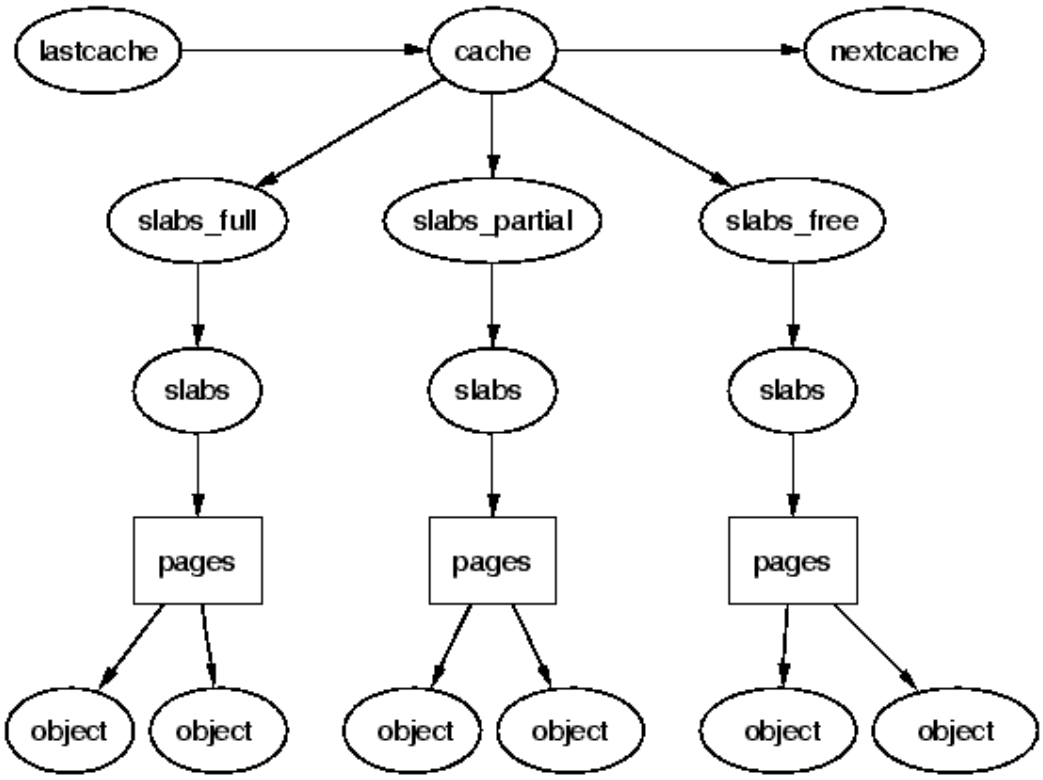
SLAB-аллокатор организован в виде трёхуровневой структуры:

1) **Кэш (Cache):**
   В контексте этого аллокатора это понимается ещё как менеджер slab'ов, содержащих объекты определённого типа. Все это менеджеры объединены в cache chain – двусвязный цикличный лист. Для каждого типа объекта в системе создаётся отдельный кэш.

2) **SLAB:**
   Это блок виртуальной памяти, выделенный под множество однотипных объектов. Slab может находиться в одном из трёх состояний:

   - **Свободный (Free):** все объекты пусты.
   - **Частично заполненный (Partial):** некоторые объекты заняты, а некоторые свободны.
   - **Полный (Full):** все объекты заняты.

3) **Объекты (Objects):**
   Это экземпляры структур или данных, которые аллокатор выдаёт по запросу. Объекты располагаются в slab'ах, и их выделение происходит без необходимости обращаться к системе низкоуровневого выделения памяти.



Изображение 1. Схема SLAB

Алгоритм работы

1. **Выделение памяти:**
   Когда процесс запрашивает память для объекта, SLAB-аллокатор ищет свободный объект в частично заполненных slab’ах. Если таких нет, он использует пустой slab или создаёт новый.

2. **Освобождение памяти:**
   При освобождении объекта он возвращается в пул свободных объектов. Если весь SLAB становится пустым, он может быть возвращён в систему или сохранён для дальнейшего использования.

3. **Кэширование объектов:**
   SLAB-аллокатор сохраняет выделенные, но неиспользуемые объекты в инициализированном состоянии, что позволяет быстро повторно использовать их без дополнительной инициализации.

Кэширование

Кэширование объектов — это техника работы с объектами, которые часто выделяются и освобождаются. Идея состоит в том, чтобы сохранить неизменную часть начального состояния объекта - его состояние завершённого конструктора - между использованиями, так что не нужно постоянно использовать деструкторы и конструкторы, каждый раз, когда объект не нужен или необходим.

Так же для повышения производительности SLAB-аллокатор использует кэш-локальность процессора. Часто используемые объекты располагаются таким образом, чтобы максимально задействовать кеши CPU, минимизируя задержки доступа к памяти. Что ещё делает SLAB? Такие оптимизации как:

- Фокусирование на удержании данных на одном процессоре.
- Разделения объектов по узлам NUMA (Non-Uniform Memory Access) для оптимизации в случае многопроцессорных систем.

В многопоточных и многопроцессорных средах SLAB-аллокатор минимизирует блокировки за счёт использования локализации кэшей на каждом процессоре. Это снижает конкуренцию за ресурсы и увеличивает масштабируемость системы. Локальные кэши позволяют процессору выделять и освобождать объекты без взаимодействия с другими ядрами, что критически важно для высокопроизводительных систем.

Stream Head Allocation + Free Costs (μsec)			
allocator	construction + destruction	memory allocation	other init.
old	23.6	9.4	1.9
new	0.0	3.8	1.9

Изображение 2. Влияние кэширования на временную сложность конструкторов и деструкторов

Поддержка конструкторов и деструкторов

Чуть подробнее про данное понятие. SLAB-аллокатор поддерживает функции инициализации (**конструктор**) и деинициализации (**деструктор**) объектов и оптимизирует их, за счёт кэширования. Это особенно полезно в ядре ОС для управления сложными структурами данных.

- **Конструктор** вызывается при создании нового объекта в SLAB и выполняет начальную настройку.
- **Деструктор** вызывается перед тем, как объект будет возвращён в систему, что позволяет корректно освободить ресурсы.

Выделение и освобождение slab’ов

SLAB-аллокатор работает поверх низкоуровневых механизмов выделения памяти (в Linux, например – *Buddy allocation*). Когда необходимо создать новый slab, аллокатор запрашивает память у системного аллокатора и разбивает её на объекты. Размер slab’а обычно кратен размеру страницы памяти, что упрощает управление и минимизирует фрагментацию.

Работа с фрагментацией

SLAB-аллокатор борется с двумя видами фрагментации – внутренней и внешней. Уделим этому моменту внимание далее.

Механизмы управления памятью в SLAB

SLAB-аллокатор разработан для оптимизации выделения и освобождения памяти, минимизации фрагментации и увеличения производительности системы, как было описано раньше. Его механизмы управления памятью построены на эффективной организации пулов объектов и автоматическом управлении этими пулами в зависимости от нагрузки.

Рассмотрим подробнее принцип связи SLAB и памяти.

Выделение памяти

Процесс выделения памяти в SLAB-аллокаторе происходит в несколько этапов:

1. **Поиск свободного объекта:**
   При запросе памяти аллокатор сначала проверяет существующие slab’ы в состоянии *Partial*, чтобы найти свободный объект. Это минимизирует необходимость выделения новых slab’ов.
2. **Использование пустых slab’ов:**
   Если частично заполненных slab’ов нет, проверяются пустые slab’ы. Это позволяет избежать затрат на инициализацию новых slab’ов.
3. **Выделение нового slab:**
   Если свободных объектов нет, создаётся новый slab. Аллокатор запрашивает у системы блок памяти и делит его на объекты фиксированного размера.
4. **Инициализация объектов:**
   При необходимости к объекту может быть применён конструктор.

Освобождение памяти

Процесс освобождения памяти также оптимизирован:

1. **Возврат объекта в кэш:**
   Освобождённый объект возвращается в пул свободных объектов внутри SLAB, что позволяет избежать дорогостоящих операций освобождения на уровне системы.
2. **Изменение статуса SLAB'а:**
   Если освобождение объекта делает SLAB полностью пустым, этот SLAB может быть возвращён в систему или сохранён для будущих запросов.
3. **Деинициализация объекта:**
   Аналогично, может быть применён деструктор.

Защита от фрагментации

SLAB-аллокатор эффективно борется с внутренней фрагментацией благодаря фиксированному размеру объектов в каждом slab'е. Однако внешняя фрагментация может возникнуть при неэффективном управлении slab'ами, особенно когда объекты динамически создаются и уничтожаются. Для снижения внешней фрагментации используются следующие приёмы:

- Поддержка различных размеров slab'ов.
- Разделение объектов по частоте использования.
- Адаптивное масштабирование кэшей.

Slab coloring

Чтобы уменьшить вероятность потенциальных конфликтов обращения к кэш памяти, улучшить её использование применяется техника *Slab coloring*. Если в объекте осталось некоторое не используемое в slab место, то оно отводится под так называемую покраску. Раскраска slab'ов — это схема, которая призвана принудить объекты в разных slab'ах использовать разные строки в кэше. Размещая объекты с разным начальным смещением в пределах одного slab, можно добиться того, что объекты будут использовать разные строки в кэше процессора, что гарантирует, что объекты из одного кэша slab'а уменьшат вероятность затереть друг друга. При такой схеме пространство, которое в противном случае было бы потрачено впустую, выполняет новую функцию.

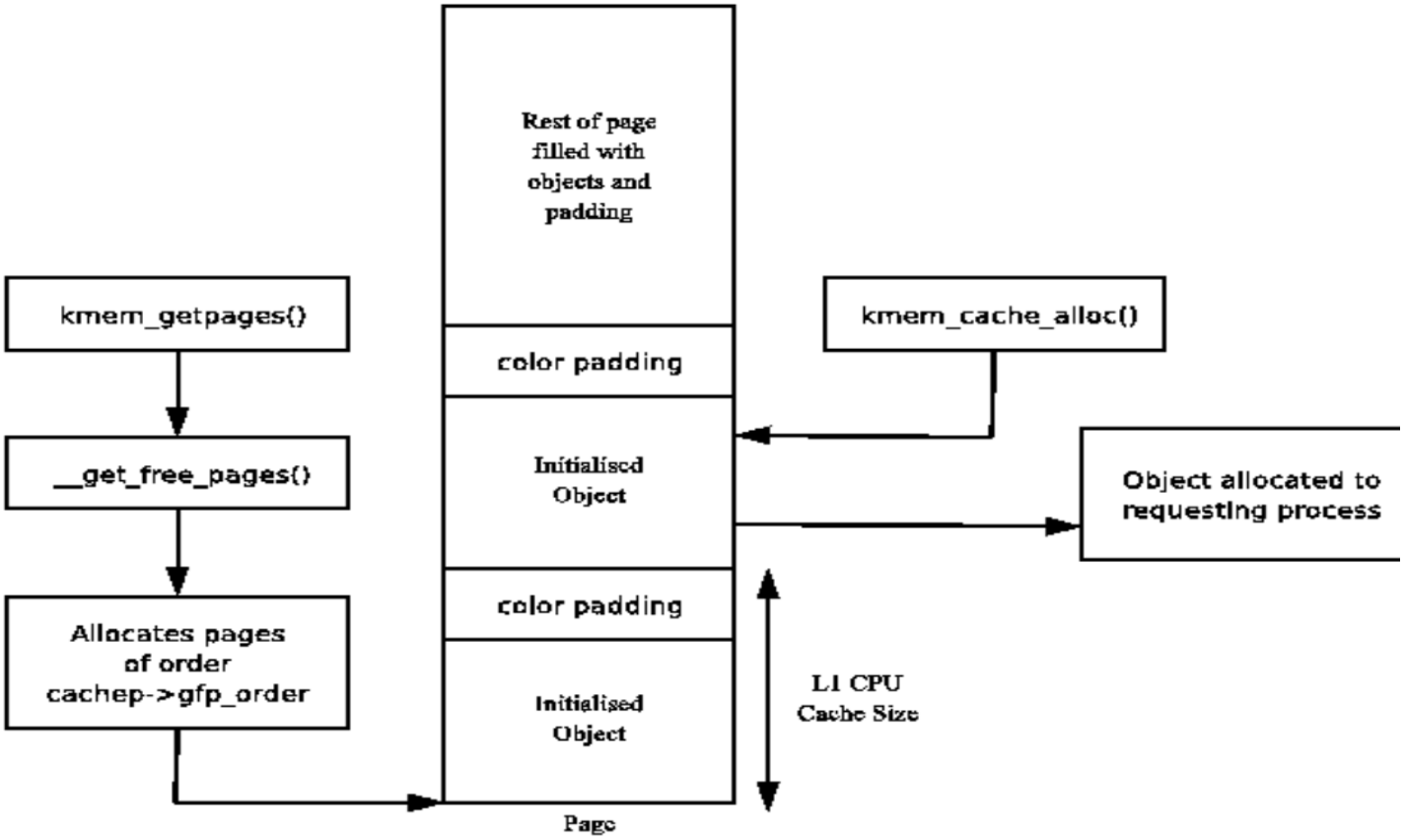


Рисунок 3. Схема Slab с объектами, выравненными по L1 кэшу процессора

Эффективность SLAB

Для эффективного управления памятью в операционных системах и приложениях инженеры совершенствуют систему управления памятью. Будь то модифицирование имеющихся реализаций и разработка новых концепций. В этом разделе рассмотрим какой вклад внёс SLAB, а также что из себя представляет его преемник, который используется в операционных системах до сих пор – SLUB.

Для начала упомянем ещё один менеджер памяти – Buddy allocator. Он всё ещё находит себе применение и используется в паре со SLAB-подобными аллокаторами для работы с физической памятью, как приводилось в пример с Linux ранее. **Buddy allocator**— это система управления памятью, которая делит память на блоки степеней двойки и рекурсивно объединяет их при освобождении. Не будем вдаваться в подробности реализации, однако отметим преимущества связки со SLAB:

- SLAB эффективен для мелких и часто используемых объектов за счёт кэширования и отсутствия перерасхода памяти на выравнивание.
- Buddy allocator хорошо справляется с крупными и динамическими аллокациями благодаря гибкости блоков.

Что касается современной используемой модификации SLAB – **SLUB (The Unqueued SLAB)**, то SLUB — упрощённая версия SLAB, устранившая очереди и использующая упрощённую структуру для уменьшения накладных расходов. Проблема SLAB заключалась в том, что он может поддерживать множество очередей в его каждом кэш-узле, в каждом CPU для выделения объектов. Они хоть и ускоряют работу, но с расширением системы могут приносить сложность в их хранении. Кроме того, затраты на использование памяти будут расти вместе с размером системы, что может быть чревато тем, что в один день вся память машины может быть занята очередями выделения.

В SLUB slab — это просто группа из одной или нескольких страниц, заполненных объектами заданного размера. В самом slab нет никаких метаданных, за исключением того, что свободные объекты компонуются в простой связный список. Когда выполняется запрос на выделение, берётся первый свободный объект, удаляется из списка и возвращается по вызову.

Учитывая отсутствие метаданных для каждого слаба, можно задаться вопросом, как именно находится первый свободный объект. Ответ заключается в том, что SLUB помещает соответствующую информацию в карту памяти системы - страничные структуры, связанные со страницами, составляющими один slab. *struct page*:

```
void *freelist;
short unsigned int inuse;
short unsigned int offset;
```

Так же интересной особенностью аллокатора SLUB является то, что он может объединять slab'ы с похожими размерами и параметрами объектов. В результате в системе уменьшается количество кэшей для slab'ов, улучшается локальность выделения slab'ов и уменьшается фрагментация памяти.

Разобравшись в тем, как SLAB добивается эффективной работы сейчас, перейдём к цифрам, с которых эта эффективность и началась.

Следующие сравнения проделаны самим Джеффом Бонвиком, в актуальный момент внедрения SLAB, когда на рынке имели распространение аллокаторы систем:

- SVr4 – основанный на Buddy allocation.
- SunOS 4.1.3 - Sequential-fit allocation.
- 4.4BSD – Segregate-storage allocation.

Временные затраты на выделение и освобождение

Memory Allocation + Free Costs		
allocator	time (μsec)	interface
slab	3.8	kmem_cache_alloc
4.4BSD	4.1	kmem_alloc
slab	4.7	kmem_alloc
SVr4	9.4	kmem_alloc
SunOS 4.1.3	25.0	kmem_alloc

Изображение 4. Стоимость аллоцирования и освобождения памяти

Заметен прирост, за счёт применяемых SLAB методов кэширования.

Ещё более наглядно это видно в работе с объектами и разными типами аллокации:

Effect of Object Caching			
allocation type	without caching	with caching	improve-ment
allocb	8.3	6.0	1.4x
dupb	13.4	8.7	1.5x
shalloc	29.3	5.7	5.1x
allocq	40.0	10.9	3.7x
anonmap_alloc	16.3	10.1	1.6x
makepipe	126.0	98.0	1.3x

Изображение 5. Влияние кэширования объектов

Работа с памятью

Одним из преимуществ SLAB мы отмечали уменьшение фрагментации памяти, рассмотрим это в цифрах.

Четыре бенчмарка позволяют посмотреть на эффективность в разных сценариях:

- **boot** – использование памяти в момент ребута.
- **spike –** нагружает требованием структур ядра, создавая множество процессов, которые требуют socket().
- **find** – схож со spike: $ *find /usr -mount -exec file {} \;*.
- **kenbus –** создаёт множество конкурентных запросов, требующих памяти для пространства пользователя и ядра.

	Total Fragmentation (waste)				
allocator	boot	spike	find	kenbus	s/m
slab	11%	13%	14%	14%	233
SunOS 4.1.3	7%	19%	19%	27%	210
4.4BSD	20%	43%	43%	45%	205
SVr4	23%	45%	45%	46%	199

Рисунок 6. Итоговая фрагментация (потери)

Дополнительно так же можно привести результаты на более длинной дистанции, в виде использования аллокаторов в течение недели для базовых нужд (нетсёрфинг, использование редакторов, подключение к серверам, использование простых скриптов):

Effect of One Week of Light Desktop Use		
allocator	kernel heap	fragmen-tation
slab	6.0 MB	9%
SunOS 4.1.3	6.7 MB	17%
SVr4	8.5 MB	35%
4.4BSD	9.0 MB	38%

Рисунок 7. Влияние использования ПК в лёгком сценарии в течение недели

Прирост в SunOS

В заключение отметим, как переход в своё время на SLAB в SunOS, благодаря Джеффу Бонвику, сказался на эффективности этой системы.

System Performance Improvement with Slab Allocator		
workload	gain	what it measures
DeskBench	12%	window system
kenbus	17%	timesharing
TPC-B	4%	database
LADDIS	3%	NFS service
parallel make	5%	parallel compilation
terminal server	5%	many-user typing

Рисунок 8. Рост производительности за счёт SLAB

Как итог, можно заключить, что данная система менеджмента памяти показывает рост производительности по множеству направлений функциональности операционной системы. Благодаря этому становится ясна значимость этой технологии.

Литература

1. **Bonwick J.** The Slab Allocator: An Object-Caching Kernel Memory Allocator [Электронный ресурс].
   – Режим доступа: https://people.eecs.berkeley.edu/~kubitron/courses/cs194-24-S14/hand-outs/bonwick_slab.pdf
2. **Gorman M.** Understanding the Linux Virtual Memory Manager. Chapter 8. Slab Allocator [Электронный ресурс].
   – Режим доступа:
   https://www.kernel.org/doc/gorman/html/understand/understand011.html
3. **Corbet J.** SLUB: The Unqueued Slab Allocator [Электронный ресурс].
   – Режим доступа: https://lwn.net/Articles/229984/
4. **Debian Kernel Mailing List.** SLUB Allocator Discussion [Электронный ресурс].
   – Режим доступа: https://lists.debian.org/debian-kernel/2012/03/msg00944.html
5. **Christoph L.** NUMA aware slab allocator V2. [Электронный ресурс].
   – Режим доступа: https://lwn.net/Articles/135444/
6. Kernel Memory Management Presentation [Электронный ресурс].
   – Режим доступа: https://www.cs.nmsu.edu/~ekerriga/presentation/index2.html