Глава 12
Выделение памяти ядром

12.1. Введение

Операционная система управляет всей доступной физической памятью машины и производит ее выделение для остальных подсистем ядра и прикладных задач. При изначальной загрузке системы ядро резервирует часть оперативной памяти под собственные коды и неизменяемые структуры данных. Эта часть ядра никогда не выгружается из памяти и не может быть использована для каких-то других целей[1]. Оставшейся памятью системы ядро управляет динамически, выделяя ее для различных клиентов (процессов и подсистем ядра). Такая память обычно освобождается при необходимости.

В UNIX оперативная память системы делится на порции фиксированного размера, также называемые *страницами*. Размер страницы памяти является некоторой степенью числа 2, в большинстве случаев ее объем составляет 4 Кбайт. UNIX является системой виртуальной памяти, поэтому логически последовательные страницы памяти не всегда имеют тот же порядок размещения в физической памяти компьютера. Следующие три главы книги посвящены описанию виртуальной памяти. Подсистема управления памятью отвечает за соответствие логических (или виртуальных) страниц с реальным расположением данных в физической памяти. В ответ на запрос о выделении блока логически смежных страниц памяти подсистема может предоставить несколько физически непоследовательных страниц.

Такой подход упрощает задачу выделения памяти. С этой целью ядро поддерживает связанный список свободных страниц. Если процессу необходимо получить некоторое количество страниц, ядро отберет их из числа свободных. При освобождении страниц ядро снова возвратит их в список. При этом фактическое расположение страниц в памяти компьютера не важно. Работа с памятью на уровне страниц реализована в системе 4.3BSD путем применения процедур memall() и memfree(), в системе SVR4 для этой цели применяются процедуры get_page() и freepage().

---

[1] Во многих современных системах UNIX (например, в AIX) часть ядра может находиться в страничной памяти.

Распределитель памяти страничного уровня (page-level allocator) имеет два принципиально важных клиента (см. рис. 12.1). Один из них называется *страничной подсистемой* (paging system) и представляет собой часть виртуальной системы памяти. Он производит выделение памяти для прикладных процессов, размещая страницы в их адресном пространстве. Во многих системах UNIX страничная система также управляет страницами, используемыми под дисковые буферы блоков. Другим клиентом является *распределитель памяти ядра* (kernel memory allocator), предоставляющий буферы памяти разного размера для множества подсистем ядра. Чаще всего ядру необходимы области памяти различной длины на короткие промежутки времени.

Ниже перечислены несколько задач, использующих память ядра.

- Процедура преобразования полного имени может запрашивать буфер (обычно размером 1024 байта) в целью копирования полного имени из пользовательского адресного пространства.
- Процедура allocb() выделяет буферы STREAMS произвольных размеров.
- Во многих реализациях системы UNIX в памяти размещаются структуры zombie, в которых находится информация о статусе выхода и использовании ресурсов завершенных процессов.
- В SVR4 ядро инициирует многие объекты (структуры proc, объекты vnode, блоки дескрипторов файлов и т. д.) динамически по мере необходимости.

Большинство таких запросов требуют памяти в меньшем объеме, чем размер одной страницы, следовательно, применение распределителя памяти ядра для таких задач является неподходящим. Для выделения памяти более мелкими порциями необходимо использовать другой механизм. Простейшим решением проблемы является запрещение динамического выделения памяти [1]. В ранних реализациях UNIX использовались таблицы фиксированного размера для объектов vnode, структур proc и т. д. Если в таких системах запрашивалась память для временного хранения полных имен или сетевых сообщений, ядро выделяло для них буферы из буферного кэша. Кроме этого, в отдельных ситуациях применялись *непредусмотренные изначально* (ad hoc) схемы выделения памяти, например в драйверах терминалов использовались *clists*.

Описанный подход имеет несколько недостатков. Технология является абсолютно негибкой, так как размеры всех таблиц и кэшей определяются на момент загрузки системы (чаще всего даже на стадии компиляции системы) и не могут быть изменены при необходимости. Размеры таких таблиц, принятые по умолчанию, выбираются разработчиками систем на основе анализа их использования в обычных рабочих группах. Хотя системные администраторы, как правило, обладают возможностью настройки размеров таблиц, чаще всего такие действия реализуются методом проб и ошибок. Если задать слишком маленький размер таблиц, это приведет к их переполнению и воз-

можному отказу системы без предупреждения. Если размер таблиц слишком велик, теряются впустую большие объемы памяти, а приложениям становится доступно меньшее ее количество. Это приводит к снижению общей производительности системы.

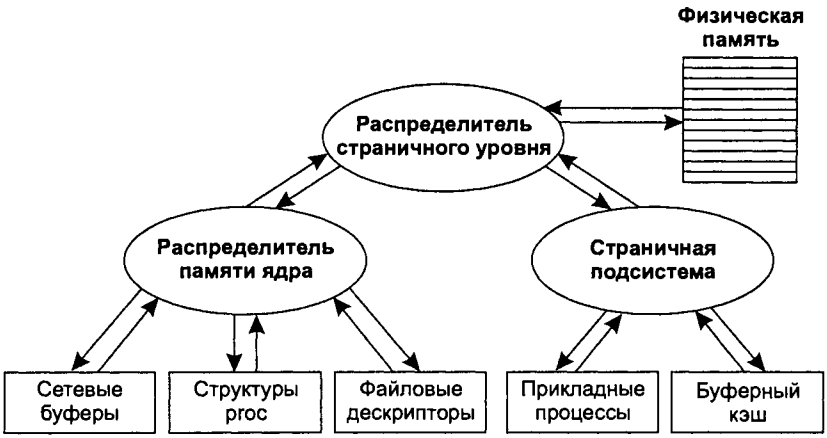

Рис. 12.1. Распределители памяти в ядре системы

Становится очевидным, что ядру необходимо общецелевое средство выделения памяти, которое умеет эффективно обрабатывать как объемные, так и небольшие области памяти. В следующем разделе будут кратко описаны требования, предъявляемые к распределителю памяти, а также критерии, по которым можно судить о его различных реализациях. Затем вы увидите описание и анализ некоторых распределителей памяти, применяемых в современных версиях UNIX.

12.2. Требования к функциональности

Распределитель памяти ядра (kernel memory allocator или KMA) обслуживает запросы на выделение динамической памяти от различных клиентов, таких как анализатор полных имен, STREAMS или средства взаимодействия процессов. KMA не занимается запросами на обслуживание страничной памятью прикладных процессов, за это отвечает страничная подсистема.

При загрузке системы ядро сначала производит резервирование памяти для собственных кодов и статических структур данных, а также некоторых встроенных областей, таких как буферный кэш. Распределитель страничного уровня осуществляет управление остальной частью физической памяти, которая может быть использована для ее динамического выделения в результате как запросов ядра, так и прикладных процессов.

Распределитель страничного уровня изначально запрашивает часть пространства у KMA, который должен эффективно использовать этот пул (pool) памяти. Некоторые реализации не позволяют изменять общее количество памяти, зарезервированное KMA. Другие разрешают распределителю памяти ядра забирать дополнительные объемы памяти у страничной подсистемы. В некоторых системах допускается даже двунаправленный обмен, разрешающий KMA передавать обратно освободившуюся память, ранее запрошенную у страничной подсистемы.

Если распределитель KMA сталкивается с отсутствием свободной памяти, то он блокирует вызывающий процесс до тех пор, пока некоторый объем памяти не освободится. Вызывающий процесс может передавать KMA флаг, указывающий на необходимость возврата с ошибкой (обычно NULL) вместо блокировки. Такая операция чаще всего применяется обработчиками прерываний, которые могут предпринять некоторые корректирующие действия при неудачном выполнении запроса. Например, если сетевое прерывание не может получить память для хранения входящего пакета, оно вправе просто пропустить такой пакет, надеясь на то, что отправитель повторно перешлет его позже.

Распределитель памяти ядра должен отслеживать, какие части пула в данный момент заняты или свободны. Освободившаяся часть памяти должна быть доступна для других запросов. В идеале обработка запроса на выделение памяти может не завершиться успешно только в том случае, если память действительно вся занята, то есть только когда общее количество свободной памяти, доступной распределителю, меньше, чем объем, указанный в запросе. В реальности сбой выделения происходит чаще в результате *фрагментации* памяти: даже если ее общего количества достаточно для удовлетворения запроса, доступная память чаще всего представляет собой непоследовательные фрагменты.

12.2.1. Критерии оценки

Важным критерием оценки распределителя памяти является возможность минимизировать ее потери. Объем физической памяти ограничен, следовательно, распределитель должен уметь эффективно использовать доступное пространство. Одной из величин, измеряющих эффективность, является *фактор использования* (utilization factor), являющийся соотношением общего объема запрашиваемой памяти к объему, позволяющему удовлетворить эти запросы. Идеальный распределитель памяти будет иметь 100%-ное использование, однако на практике приемлемой величиной является 50% [9]. Основная причина потерь во фрагментации памяти такова: обычно свободная память разбита на множество участков, которые слишком малы, чтобы быть востребуемыми. Распределитель убавляет фрагментацию путем слияния смежных участков свободной памяти в единую порцию.

Распределитель KMA должен быть быстрым, так как он интенсивно используется различными подсистемами ядра (в том числе обработчиками прерываний, чья производительность критична для системы в целом). Также важны показатели среднестатистической и максимальной задержки. Стеки ядра невелики по объему, поэтому оно прибегает к динамическому размещению в тех случаях, когда обычный процесс просто запрашивает объект из собственного стека. Такой подход требует высокой скорости выделения стека ядра. Медленно работающий распределитель оказывает отрицательное влияние на производительность системы в целом.

Распределитель должен обладать простым программным интерфейсом, подходящим под нужды различных клиентов. Одним из подходов является реализация интерфейса в виде функций, сходных с malloc() и free(), вызываемых прикладными процессам и предлагаемых в стандартной библиотеке системы:

```
void* malloc (size_t nbytes);
void free (void *ptr);
```

Важным преимуществом такого интерфейса является то, что при вызове процедуры free() не нужно знать размеры освобождаемого сегмента памяти. Часто одна из функций ядра запрашивает некоторое количество памяти и передает его другой подсистеме, которая позже и осуществляет ее освобождение. Например, сетевой драйвер может запросить буфер для хранения входящего сообщения и направить его модулю более высокого уровня для обработки данных и последующего освобождения буфера. Обладание информацией о размере запрашиваемого объекта для такого модуля не принципиально. Если задачу слежения за подобной информацией будет выполнять KMA, это сильно упростит работу, выполняемую его клиентами.

Еще одним полезным средством является разрешение на освобождение лишь части выделенной клиенту ранее области памяти. Если клиенту нужно освободить лишь часть занимаемой памяти, распределитель должен уметь корректно обрабатывать такую ситуацию. Интерфейс malloc()/free() не поддерживает эту возможность. Процедура free() освобождает область целиком и возвращает ошибку, если при ее вызове указать адрес, не совпадающий с указанным malloc(). Возможность увеличения буфера прикладных процессов (например, путем применения функции realloc()) также является весьма полезной.

Выделенная для использования память должна быть соответствующим образом упорядочена для быстрого обращения к ней. Для многих реализаций архитектуры RISC это является обязательным требованием. В большинстве систем достаточно упорядочивания в виде *длинных слов* (longword), однако в 64-разрядных машинах, таких как DEC Alpha AXP [6], может потребоваться выравнивание по 8-разрядной границе. Еще одним параметром, связанным с вопросами выделения областей памяти, является минимальный ее размер, который обычно равняется 16 или 8 байтам.

Многие коммерческие продукты используют память в круговом режиме. Например, машина в течение дня обслуживает запросы к базе данных и обработку транзакций, а по ночам занимается резервным копированием и реорганизацией структуры базы. Перечисленные задачи будут иметь совершенно разные требования к памяти. Обработка транзакций, скорее всего, потребует нескольких небольших участков памяти ядра для реализации блокировки базы данных, в то время как операция резервного копирования будет нуждаться в максимальном количестве свободной памяти для своих действий.

Многие распределители памяти разбивают доступный им пул на отдельные участки или *сегменты* (buckets) для запросов различных типов. Например, сегменты одного типа могут быть только 16-битовыми, в то время как другого — 64-битовыми. Такие распределители должны уметь противостоять неравномерному или круговому использованию, описанному выше. В некоторых распределителях после того, как вся память была отдана одному сегменту, она не может быть использована в дальнейшем для запросов областей иных размеров. Это чревато дисбалансом — большим объемом неиспользуемой памяти в одном сегменте и недостаточностью ресурсов памяти для других сегментов системы. Распределитель памяти должен обладать функцией динамического переприсвоения ее участков от одного сегмента другому.

Еще одним важным критерием является взаимодействие KMA со страничной подсистемой. Распределителю нужно уметь забирать часть страничной памяти при исчерпании первоначального объема. Страничная система должна обладать возможностью восстановления неиспользуемой памяти KMA. Такой обмен между двумя подсистемами требуется координировать соответствующим образом, чтобы предупредить устаревание каждой из подсистем.

Рассмотрим несколько методик выделения памяти и проведем их анализ, исходя из критериев оценки, перечисленных в этом разделе.

12.3. Распределитель карты ресурсов

Карта ресурсов (resource map) — это набор пар <base, size> (<базовый адрес, размер>), используемый для отслеживания свободных областей памяти (см. рис. 12.2). Изначально область памяти описывается при помощи единственного вхождения карты, в котором указатель равен стартовому адресу области, а размер равен ее общему объему памяти (рис. 12.2, *а*). После этого клиенты начинают запрашивать и освобождать участки памяти, вследствие чего область становится фрагментированной. Ядро создает для каждого нового последовательного свободного участка памяти новое вхождение карты. Элементы карты сортируются в порядке возрастания адресов, что упрощает задачу слияния свободных участков.

При помощи карты ресурсов ядро может выполнить запросы на выделение памяти исходя из трех правил.

- **Первый подходящий участок.** Выделение памяти из первой по счету свободной области, имеющей достаточный для удовлетворения запроса объем. Это самый быстрый алгоритм из всех трех, но он не совсем оптимален применительно к соображениям уменьшения фрагментации.
- **Наиболее подходящий участок.** Выделение памяти из области наименьшего размера, достаточного для удовлетворения запроса. Основным недостатком метода является необходимость иногда пропускать некоторое количество сегментов, которые являются слишком мелкими.
- **Наименее подходящий участок.** Выделение области максимального размера до тех пор, пока не будет найден более подходящий вариант. Такой подход кажется не совсем логичным, однако его применение основано на ожидании, что выделенная область будет достаточно велика для использования в последующих запросах.

Ни один из приведенных способов не идеален для всех возможных случаев. В книге [8] предлагается детальный анализ этих, а также других механизмов выделения памяти. В системах UNIX применяется методика первого подходящего участка.

На рис. 12.2 показан простейший пример карты ресурсов, обслуживающий 1024-байтовую область памяти. Область поддерживает две операции:

```
offset_t rmalloc (size); /* возвращает смещение выделенного участка */
void rmfree (base, size);
```

Изначально вся область памяти свободна и описывается единственным вхождением карты ресурсов (рис. 12.2, *а*). Для примера мы произведем два запроса на выделение участков памяти размером 256 и 320 байтов соответственно. После этого мы освободим 128 байтов, начиная со смещения 256. На рис. 12.2, *б* показано состояние карты после завершения перечисленных операций. В этот момент времени карта содержит два свободных участка и, следовательно, два вхождения для их описания.

На следующем этапе происходит освобождение еще 128 байтов, начиная от смещения 128. Распределитель памяти определяет, что этот участок размещен последовательно с другим свободным участком, начинающимся с отметки 256. В этом случае распределитель объединяет их в единый сегмент размером 256 байтов. В результате карта ресурсов выглядит как показано на рис. 12.2, *в*. Последний рисунок описывает ситуацию, сложившуюся через некоторое время после выполнения еще нескольких операций. Необходимо отметить, что, несмотря на то что общий размер свободной памяти равняется 256 байтам, распределитель памяти не может выделить участок длиной, превышающей 128 байтов.

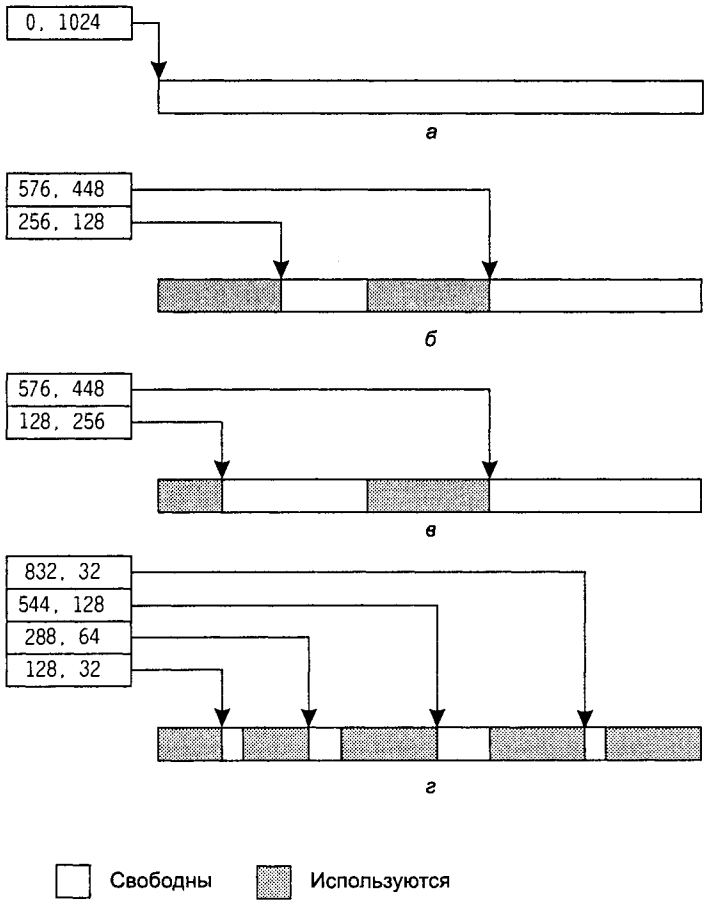

Рис. 12.2. Состояние карты ресурсов: *а* — изначальная конфигурация; *б* — после rmalloc(256), rmalloc(320) и rmfree(256,128); *в* — после rmfree(128,128); *г* — после еще некоторого количества операций

12.3.1. Анализ

Карта ресурсов применяется для простейшего случая распределения памяти. Она обладает несколькими преимуществами:

- ✦ алгоритм прост для практической реализации;
- ✦ карта ресурсов не ограничена в применении только для задач выделения и освобождения памяти. Она может быть использована и для обработки наборов различных других объектов, расположенных в определенном порядке и доступных для выделения и освобождения непрерывными участками (к таким объектам относятся, к примеру, вхождения таблицы страниц и семафоры);

- карта позволяет выделять точное количество байтов, равное запрошенному, без потерь памяти. На практике распределитель памяти всегда округляет количество выделяемой памяти до числа, делящегося на четыре или восемь, с точки зрения простоты и удобства выравнивания;
- от клиента не требуется всегда возвращать участок памяти, равный запрошенному. Как показывает предыдущий пример, клиент может освободить любую часть выделенного ранее участка, при этом распределитель памяти корректно отработает возникшую ситуацию. Такая возможность стала доступна потому, что в качестве аргумента процедуры rmfree() указывается размер освобождаемого участка, а учетная информация (то есть карта ресурсов) поддерживается системой отдельно от самой выделяемой памяти;
- распределитель соединяет последовательные участки памяти в один, что дает возможность выделять в дальнейшем области памяти различной длины.

Распределитель ресурсов имеет и ряд существенных недостатков:

- по истечении какого-то времени работы карта становится сильно фрагментированной. В ней оказывается большое количество участков малого размера. Это приводит к низкой востребованности ресурса. В частности, распределитель карты ресурсов плохо справляется с задачей обслуживания «больших» запросов;
- по мере увеличения фрагментации синхронно наращивается и сама карта ресурсов, так как для размещения данных о каждом новом свободном участке требуется новое вхождение. Если карта настроена на фиксированное количество вхождений, то в некоторый момент времени она может переполниться, а распределитель памяти потерять данные о какой-то доле свободных участков;
- если карта будет расти динамически, то для ее вхождений потребуется собственный распределитель. Эта проблема является «рекурсивной», и ниже вы увидите одно из ее решений;
- для решения задачи объединения свободных смежных областей памяти распределитель должен поддерживать карту, упорядоченную в порядке увеличения смещения от базового адреса. Операция сортировки весьма затратна, более того, она должна производиться по месту в том случае, если карта реализована в виде массива фиксированного размера. Нагрузка на систему, возникающая при сортировке, является весьма ощутимой даже в том случае, если карта размещается в памяти динамически и организована в виде связанного списка;
- часто требуется выполнять операцию последовательного поиска в карте с целью обнаружения достаточно большого для удовлетворения запроса участка. Эта процедура занимает много времени и выполняется медленнее при сильной фрагментации памяти;

- несмотря на наличие возможности возврата свободных участков памяти в хвост пула страничной подсистемы, алгоритм выделения и освобождения памяти не приспособлен для такой операции. На практике распределитель никогда не стремится достичь большей непрерывности вверенных ему областей памяти.

Карты ресурсов являются низкопроизводительными. Именно по этой причине они не подходят для применения в роли универсального распределителя памяти ядра. Средства взаимодействия процессов System V используют карты ресурсов для выделения наборов семафоров и областей данных сообщений. Подсистема виртуальной памяти 4.3BSD применяет методику для обработки вхождений таблицы страниц, указывающих на таблицы страниц прикладных процессов (см. подробнее в разделе 13.4.2).

Управление картами ресурсов может быть изменено в лучшую сторону сразу же в нескольких направлениях. Вхождение карты чаще всего удобнее размещать в первых нескольких байтах свободного участка памяти. Такой подход не требует выделения дополнительной памяти для хранения самой карты и ее новых вхождений. На первый свободный участок может указывать единственная глобальная переменная, в то время как внутри каждого свободного участка может храниться информация о его размере и указатель на следующий свободный участок в области памяти. В этом случае минимальный размер участка должен равняться по крайней мере двум словам, в одном из которых будет храниться размер, а во втором — указатель. Такое требование можно удовлетворить путем принудительного выделения или освобождения участков, длина которых кратна двум (словам). В файловой системе FFS (см. раздел 9.5) используется немного другой вариант такого алгоритма для обработки свободного места в блоках каталогов.

Несмотря на то, что перечисленные изменения подходят для обычного распределителя памяти, они не могут быть реализованы в специализированных случаях применения карты ресурсов, в которых не остается свободного места для хранения данных о вхождениях, например, при использовании карты для семафоров или вхождений таблицы страниц.

12.4. Простые списки, основанные на степени двойки

Списки свободной памяти, основанные на степени числа 2, чаще всего применяются для реализации процедур malloc() и free() в библиотеке С прикладного уровня. Методика использует набор списков свободной памяти. В каждом списке хранятся буферы определенного размера. Размер буфера всегда кратен степени числа 2. На рис. 12.3 показан пример шести списков, содержащих буферы размером 32, 64, 128, 256, 512 и 1024 байта соответственно.

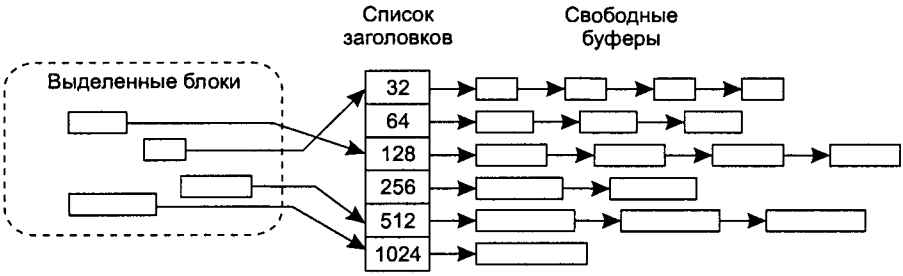

Рис. 12.3. Список свободных буферов с длиной, кратной степени числа 2

Каждый буфер имеет заголовок длиной в одно слово, этим фактом ограничивая возможности использования соотносимой с ним области памяти. Если буфер свободен, в его заголовке хранится указатель на следующий свободный буфер. В другом случае в заголовок буфера помещается указатель на список, в который он должен быть возвращен при освобождении. В некоторых реализациях заголовок содержит вместо этой информации размер выделенной области. Это позволяет обнаружить некоторые «баги», однако требует от процедуры free() вычисления местонахождения списка исходя из данных о размерах буферов.

Для выделения памяти клиент вызывает malloc(). В качестве входного аргумента функции передается желаемая величина участка. Распределитель вычисляет минимальный размер буфера, подходящий для удовлетворения запроса. Для этого необходимо прибавить к заданной величине слово, в котором будет размещен заголовок, и округлить полученное значение кверху до числа, являющегося степенью двойки. Буферы размером 32 байта подходят для выделения памяти объемом 0–28 байтов, 64-байтовые буферы — для объемов 29–60 байтов и т. д. После вычисления распределитель извлекает буфер из соответствующего списка и оставляет в заголовке указатель на список свободных участков памяти. Процесс, запрашивающий участок памяти, получает в ответ указатель на следующий после заголовка байт.

Если клиент желает освободить буфер, то он вызывает для этой цели процедуру free(), входным аргументом которой является указатель, возвращенный malloc(). При этом нет необходимости уточнять размер буфера. Очевидно, что результатом выполнения операции free() станет освобождение буфера целиком. Технология не поддерживает возможности частичного освобождения участка памяти. Процедура free() производит обращение к заголовку буфера, откуда извлекает указатель на список свободных буферов и затем переносит его в этот список.

Распределитель памяти может быть проинициализирован путем предварительного выделения ему определенного количества буферов в каждом списке. Другой вариант предполагает создание изначально свободного списка, для заполнения которого вызывается распределитель страничного уровня. Если список становится пустым, обработка последующего запроса malloc()

для выделения участка памяти определенного размера может быть произведена одним из перечисленных способов:

- запрос блокируется до освобождения буфера подходящей длины;
- запрос удовлетворяется путем выделения буфера большего размера. Поиск свободного буфера начинается со следующего списка (по возрастанию размеров буферов) и продолжается до тех пор, пока не будет обнаружен непустой список;
- происходит запрос дополнительного объема памяти от распределителя страничного уровня. При этом создается нужное количество буферов заданного размера.

Каждый метод обладает определенными достоинствами и недостатками. Наиболее подходящий вариант действий зависит от ситуации. Например, реализация алгоритма на уровне ядра может использовать в запросах на выделение памяти дополнительный аргумент приоритетности. В таком случае распределитель вправе блокировать низкоприоритетные запросы, если в списке не имеется ни одного свободного буфера указанного размера, при этом используя два последних метода для немедленной обработки высокоприоритетных запросов.

12.4.1. Анализ

Описанный алгоритм выделения памяти является весьма простым и быстрым. Его основным отличием от предыдущего является отсутствие необходимости длительного поиска в карте ресурсов и решение проблемы фрагментации области памяти. При использовании буферов нижний предел производительности всегда ограничен. Распределитель памяти предоставляет понятный программный интерфейс, важнейшим преимуществом которого является процедура free(), в качестве входного аргумента которой больше не нужно задавать размер буфера. В результате этого выделенный буфер может передаваться другим функциям или подсистемам, а также освобождаться, и для этого достаточно использовать всего указатель на него. С другой стороны, интерфейс не позволяет клиенту освобождать буферы частично.

В алгоритме имеется несколько значимых недостатков. Округление количества запрошенной памяти кверху до следующей степени двойки часто приводит к реальному использованию лишь части буфера, что в результате влечет неэкономное распределение памяти. Проблема становится еще более очевидной из-за необходимости хранения заголовков буферов внутри самих выделенных буферов. Многие запросы требуют объемов памяти, уже равных степени числа 2. Удовлетворение таких запросов имеет следствием почти 100%-ную потерю выделяемых участков памяти, так как чтобы поместить заголовок в буфер (а это всего 4 байта), распределитель округлит необходимое количество байтов до следующей степени двойки. Например, в ответ на просьбу о предоставлении 512 байтов памяти будет выделено уже 1024 байта.

Технология не поддерживает слияния смежных буферов, что позволило бы удовлетворить запросы на выделение областей памяти большей протяженности. Размер каждого буфера остается неизменным в течение его периода жизни. Большие буферы иногда можно использовать для удовлетворения запросов на области памяти малой длины. Несмотря на то, что некоторые реализации алгоритма позволяют «заимствовать» часть памяти у страничной системы, не существует механизма обратной передачи таких буферов после освобождения.

И хотя алгоритм степени двойки намного быстрее работы с картой ресурсов, его производительность может быть поднята еще больше. В частности, реализация перебора, показанная в листинге 12.1, является слишком медленной и неэффективной.

Листинг 12.1. Грубая реализация процедуры malloc()

```
void*malloc (size)
{
   int ndx = 0; /* индекс списка свободных буферов */
   int bufsize = 1 << MINPOWER; /* минимальный размер буфера */
   size+=4; /* выделение байтов для заголовка */
   assert (size <= MAXBUFSIZE);
   while (bufsize < size) {
      ndx++;
      bufsize <<= 1;
   }
   ... /* на этом этапе ndx является индексом к соответствующему списку
      свободных буферов */
}
```

В следующем разделе будет описан более совершенный алгоритм, в котором решены многие перечисленные проблемы.

12.5. Распределитель Мак-Кьюзика—Кэрелса

Маршалл Кирк Мак-Кьюзик и Майкл Дж. Кэрелс разработали усовершенствованный метод выделения памяти [12], который был реализован во многих вариантах системы UNIX, в том числе 4.4BSD и Digital UNIX. Методика позволяет избавиться от потерь в тех случаях, когда размер запрашиваемого участка памяти равен некоторой степени двойки. В нем также была произведена оптимизация перебора в цикле (см. листинг 12.1). Такие действия теперь нужно производить только в том случае, если на момент компиляции неизвестен размер выделенного участка.

Алгоритм подразумевает, что память разбита на набор последовательных страниц, и все буферы, относящиеся к одной странице, должны иметь одинаковый размер (являющийся некоторой степенью числа 2). Для управления

страницами распределитель использует дополнительный массив kmemsizes[]. Каждая страница может находиться в одном из трех перечисленных состояний.

- Быть свободной. В этом случае соответствующий элемент массива kmemsizes[] содержит указатель на элемент, описывающий следующую свободную страницу.
- Быть разбитой на буферы определенного размера. Элемент массива содержит размер буфера.
- Являться частью буфера, объединяющего сразу несколько страниц. Элемент массива указывает на первую страницу буфера, в которой находятся данные о его длине.

На рис. 12.4 показан простейший пример организации страницы размером 1024 байта. Массив freelistarr[] содержит заголовки всех буферов, имеющих размер меньше одной страницы.

Так как длина всех буферов одной страницы одинакова, нет нужды хранить в заголовках выделенных буферов указатель на список свободных буферов. Процедура free() находит страницу путем маскирования младших разрядов адреса буфера и обнаружения размера буфера в соответствующем элементе массива kmemsizes[]. Отсутствие заголовка в выделенных буферах позволяет экономить память при удовлетворении запросов с потребностью в памяти, кратной некоторой степени числа 2.

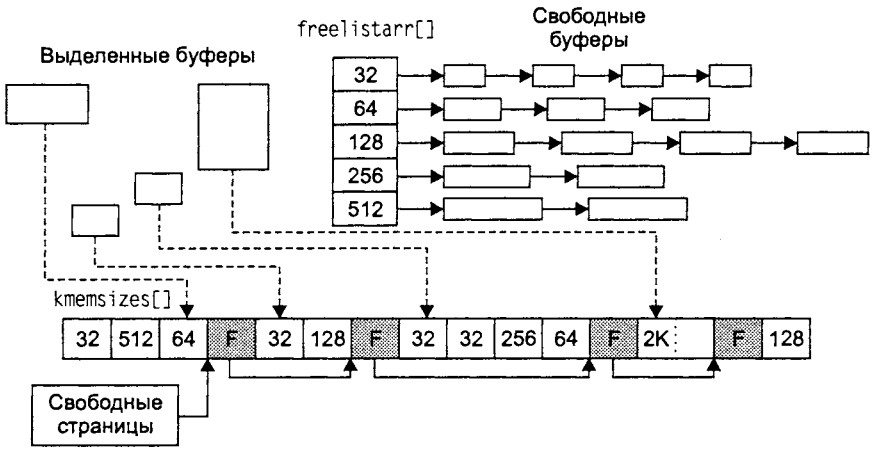

Рис. 12.4. Распределитель Мак-Кьюзика—Кэрелса

Вызов процедуры malloc() заменен макроопределением, которое производит округление значения длины запрашиваемого участка вверх до достижения числа, являющегося степенью двойки (при этом не нужно прибавлять какие-либо дополнительные байты на заголовок) и удаляет буфер из соответствующего списка свободных буферов. Макрос вызывает функцию malloc() для запроса одной или нескольких страниц тогда, когда список свободных буферов

необходимого размера пуст. В этом случае malloc() вызывает процедуру, которая берет свободную страницу и разделяет ее на буферы необходимого размера. Здесь цикл заменен на схему вычислений по условию. Реализация для области памяти, схематично изображенной на рис. 12.4, показана в листинге 12.2.

Листинг 12.2. Применение макроопределения, увеличивающего скорость работы malloc()

```
#define NDX(size) \
   (size) > 128 \
      ? (size) > 256 ? 4 : 3 \
      : (size) > 64 \
         ? 2\
         : (size) > 32 ? 1 : 0
#define MALLOC (space, cast, size, flags) \
   {   \
      register struct freelisthdr* flh; \
      if (size <= 512 &&\
      (flh = freelistarr [NDX(size)]!= NULL) { \
         space = (cast)flh->next; \
         flh->next = *(caddr_t *)space; \
      } else \
         space = (cast)malloc (size, flags);\
   }
```

Основным преимуществом такого решения является тот факт, что если выделяемый размер известен в момент компиляции, определение NDX() сжимается до константы времени компиляции, что позволяет сократить значительное количество инструкций. Второй макрос применяется для простейших ситуаций освобождения буфера, вызов самой функции осуществляется в редких случаях, например, когда требуется освободить буфер большого размера.

12.5.1. Анализ

Приведенный алгоритм значительно улучшает методику распределения памяти на основе степени числа 2, описанную в разделе 12.4. Он работает намного быстрее, потери памяти при его применении сильно сокращаются. Алгоритм позволяет эффективно обрабатывать запросы на выделения как малых, так и больших участков памяти. Однако описанная методика обладает и некоторыми недостатками, связанными с необходимостью использования участков, равных некоторой степени числа 2. Не существует какого-либо способа перемещения участков из одного списка в другой. Это делает распределитель не совсем подходящим средством при неравномерном использовании памяти, например, если системе необходимо много буферов малого размера на короткий промежуток времени. Технология также не дает возможности возвращать участки памяти, запрошенные ранее у страничной системы.

12.6. Метод близнецов

Метод близнецов (или buddy system) является схемой выделения памяти, сочетающей в себе возможность слияния буферов и распределитель по степени числа 2[1] [13]. В основе метода лежит создание буферов малого размера путем деления пополам больших буферов и слияния смежных буферов по мере возможности. При разделении буфера на два каждая половина называется *близнецом* (buddy) второй.

Рассмотрим метод близнецов на простом примере (см. рис. 12.5), в котором алгоритм применяется для обработки 1024-байтового блока с минимальным размером участков в 32 байта. Распределитель использует битовую карту для отслеживания каждой 32-битовой порции блока: если бит установлен, то соответствующий участок занят. Она также поддерживает список свободных буферов любого допустимого размера (по степеням двойки в диапазоне от 32 до 512). Изначально блок представляет собой единый буфер. Представим, что произойдет с ним при поступлении некоей последовательности запросов на выделение памяти и ответную реакцию распределителя.

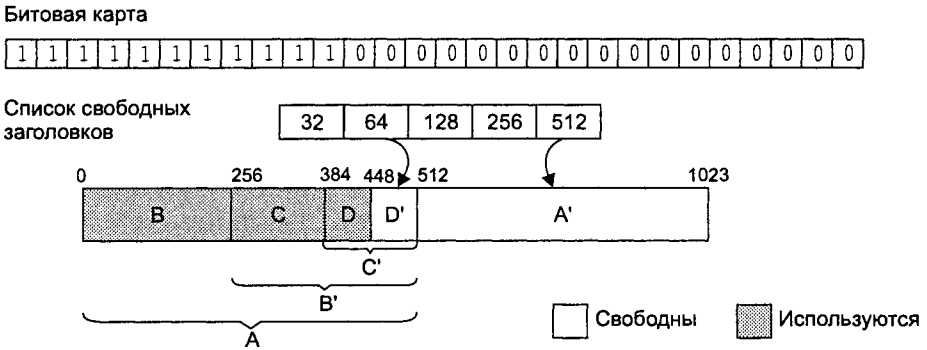

Рис. 12.5. Метод близнецов

1. allocate(256). Блок делится на два близнеца, A и A'; блок A' поступает в список свободных 512-байтовых буферов. Затем буфер A разбивается на B и B'. B' заносится в список свободных 256-байтных буферов, а буфер B передается клиенту.
2. allocate(128). Распределитель обнаруживает, что список свободных 128-байтовых буферов пуст. Тогда он проверяет список 256-байтовых буферов, изымает оттуда B' и разделяет его на C и C'. После этого буфер C' помещается в список свободных 128-байтовых буферов, а буфер B возвращается клиенту.

[1] Здесь описывается двоичная методика близнецов, которая является наиболее простой и популярной из систем близнецов. На самом деле можно реализовать и другие варианты алгоритма, разделяющие буферы на четыре, восемь и более частей.

3. allocate(64). Распределитель выясняет, что список свободных 64-байтовых буферов пуст. Тогда он обходит список 128-байтовых буферов, удаляет оттуда C′ и дробит его на D и D′. Последний передается в список свободных 64-байтовых буферов, а буфер D возвращается клиенту. Ситуация на этот момент времени показана на рис. 12.5.
4. allocate(128). Распределитель узнает, что списки свободных 128- и 256-байтовых буферов пусты. Он проверяет список свободных 512-байтовых буферов и удаляет оттуда буфер A′. Затем A′ разделяется на два близнеца E и E′, а буфер E — на F и F′. После этого буфер E′ переносится в список свободных буферов размером 256 байтов, а буфер F′ — в список 128-байтовых буферов. Буфер F передается клиенту.
5. release(C, 128). Буфер C возвращается в список свободных 128-байтовых буферов. Это приводит к состоянию, продемонстрированному на рис. 12.6.

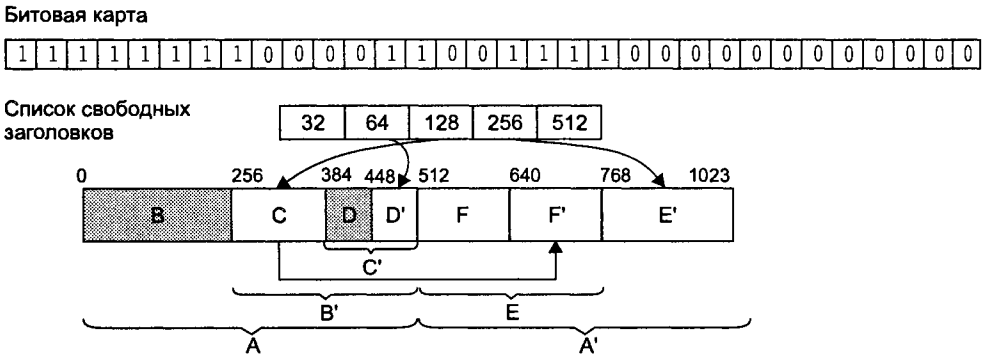

Рис. 12.6. Метод близнецов, стадия 2

До этого момента нам ни разу не потребовалось совмещать смежные буферы. Однако представьте, что следующий запрос к распределителю будет таков:

6. release(D,64). Обработав такой запрос, распределитель увидит, что буфер D′ также свободен. Тогда он совместит его со своим близнецом для получения буфера C′. Но в этот момент буфер не занят, поэтому распределитель объединит его с C′ для образования B′. После буфер B′ будет помещен в список свободных 256-байтовых буферов. Состояние рассматриваемого блока памяти на этом этапе продемонстрировано на рис. 12.7.

Описывая принцип работы метода близнецов, необходимо упомянуть следующие интересные моменты:

- размер запрашиваемой памяти обычно округляется до следующей степени числа 2;

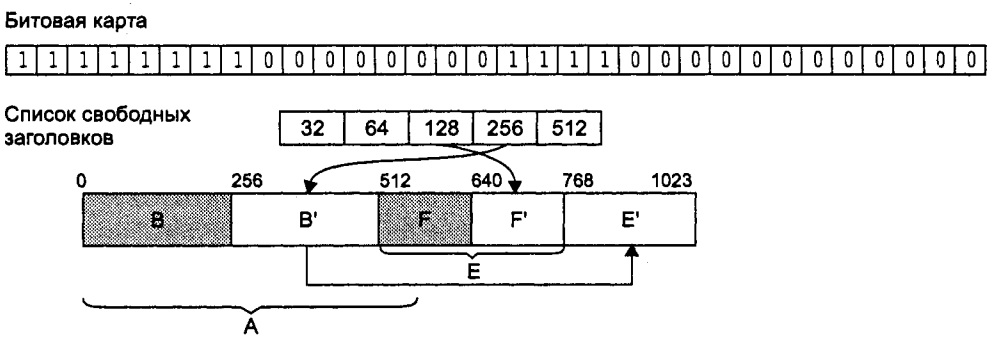

Рис. 12.7. Метод близнецов, стадия 3

- в каждом запросе рассматриваемого примера соответствующий список свободных буферов оказывался пустым. В реальности это чаще всего не так. Если в списке имеется буфер необходимого размера, распределитель будет использовать именно его, поэтому совмещение буферов становится ненужным;
- адрес и размер буфера есть в совокупности достаточная информация для обнаружения близнеца. Это происходит потому, что алгоритм размещает буфер, руководствуясь его длиной. Например, у 128-байтового буфера, имеющего смещение 256, близнец находится по адресу 384, в то время как 256-байтовый буфер с таким же смещением обладает близнецом, расположенным в точке 0;
- при обработке каждого запроса происходит обновление битовой карты с целью отражения в ней нового состояния буфера. При слиянии распределитель узнает из этой карты, свободен ли необходимый ему буфер-близнец;
- в нашем примере распределитель обрабатывает единственную страницу памяти размером 1024 байта. На самом деле он может обслуживать одновременно несколько различных страниц памяти. Единый набор заголовков свободных буферов может хранить данные о буферах всех страниц сразу. При этом слияние будет происходить по ранее приведенному алгоритму, так как близнец определяется исходя из смещения буфера на конкретной странице. В этом случае, однако, распределитель будет поддерживать отдельные битовые карты для каждой страницы памяти.

12.6.1. Анализ

Методика близнецов обладает замечательным свойством соединения смежных свободных буферов. Возможность изменения размеров участков памяти и их повторного использования в преобразованном виде придает алгоритму большую степень гибкости. Методика позволяет легко обмениваться памятью

между распределителем и страничной системой. Если распределителю в какой-то момент времени понадобился больший объем памяти, он вправе запросить новую страницу и затем при необходимости поделить ее на буферы. После того как процедура освобождения соберет страницу в единый блок, распределитель может вернуть ее обратно страничной системе.

Главным недостатком алгоритма является его производительность. Каждый раз при освобождении буфера распределитель пытается соединить вместе как можно больше участков памяти. При чередовании запросов на выделение и освобождение алгоритм будет объединять только что разделенные буферы. Операция объединения рекурсивна, что дает в результате замедление ее работы. В следующем разделе будет рассмотрена реализация усовершенствованного алгоритма близнецов в SVR4, преодолевающая это узкое место.

Еще одним недостатком методики является ее программный интерфейс. Процедура освобождения должна получать в качестве аргумента адрес и размер буфера. Более того, распределитель памяти поддерживает освобождение только целого буфера. Частичное освобождение является неэффективным, так как в этом случае неполный буфер не будет иметь близнеца.

12.7. Алгоритм отложенного слияния в SVR4

Основной проблемой простого метода близнецов являлась низкая производительность работы из-за необходимости постоянного разделения и слияния буферов. Обычно распределители памяти находятся в некотором равновесном состоянии, то есть количество используемых буферов каждого размера остается в примерно известном диапазоне. При таких условиях совмещение буферов не дает никаких преимуществ, время частично тратится впустую. Слияние необходимо только при постоянно меняющихся условиях использования буферов памяти.

Задержка слияния определяется как время, необходимое либо для соединения буфера со своим близнецом, либо для определения того факта, что близнец занят. Объединение приводит к появлению буфера размера увеличенной степени числа 2, процесс повторяется до тех пор, пока распределитель перестанет находить свободных близнецов (то есть операция является рекурсивной). В алгоритме близнецов каждая операция освобождения приводит, по крайней мере, к однократной задержке слияния, а чаще всего, не только к одной.

Интуитивно-логичное решение проблемы заключается в откладывании операции слияния до того момента, пока она не будет необходима, — вот тогда и нужно производить сращивание как можно большего числа буферов. Несмотря на то, что такой подход уменьшит время, затрачиваемое в среднем

для выделения и освобождения буферов, то небольшое количество запросов, которое требует вызова операции объединения, будет обслуживаться слишком медленно. Распределитель памяти может быть вызван критичными к скорости выполнения функциями, такими как обработчики прерываний, отсюда становится очевидной необходимость управления таким наихудшим вариантом. Необходимо создать такое средство, которое бы задерживало операции слияния буферов, кроме критичных случаев, а также умело распределять операции объединения на несколько запросов (что не так сильно увеличит время реакции на запрос, повлекший за собой необходимость вызова этой операции). В работе [10] описывается решение проблемы, основанное на ассоциации с каждым классом буферов *«водяных знаков»* (watermarks). В системе SVR [2] используется та же идея, но с большей эффективностью (см. следующие разделы).

12.7.1. Отложенное слияние

Освобождение буфера памяти производится в два этапа. Сначала буфер помещается в список свободных буферов, что делает его доступным для последующих запросов на выделение. После этого буфер помечается в битовой карте как свободный и по возможности присоединяется к смежным буферам (операция слияния). В обычной системе близнецов при каждой операции освобождения выполняются обе перечисленные стадии.

Методика отложенного слияния подразумевает выполнение первой ступени, в результате которой буфер становится *свободным локально* (доступным для выделения внутри своего класса, но не для слияния). Необходимость выполнения второго шага зависит от состояния класса буферов. В любой момент времени класс содержит N буферов, среди которых A активных буферов, L локально свободных буферов и G глобально свободных буферов (то есть помеченных свободными в битовой карте и доступных для слияния). Следовательно,

```
N = A + L + G
```

В зависимости от величины этих параметров класс может находиться в одном из трех состояний:

- **«безмятежном»** (lazy). Буферы расходуются сбалансированно (количество запросов на выделение и освобождение является примерно равным), слияние не является необходимым;
- **требующем восстановления равновесия** (reclaiming). Расход буферов близок к пограничному, операция слияния становится необходимой;
- **требующем ускоренного выполнения** (accelerated). Расход буферов неравновесный, требуется как можно более быстрое проведение слияния.

Состояние определяется параметром, называемым *допуском* (slack) и определяемым по следующей формуле:

```
slack = N - 2L - G
```

Система находится в «ленивом» состоянии, если допуск не меньше двух, состояние восстановления наступает при достижении значения единицы. Состояние разгона возникает при значении допуска 0. Алгоритм подразумевает, что величина допуска никогда не может стать отрицательной. В работе [2] предлагается детальное описание, объясняющее, почему допуск является эффективным средством оценки состояния класса буферов.

При освобождении буфера распределитель памяти SVR4 переносит его в список свободных буферов и проверяет состояние класса. Если список находится в безразличном состоянии, на этом операция завершается. Буфер не помечается свободным в битовой карте. Такой буфер является отложенным, что фиксируется определенным флагом в его заголовке (который имеется только у буферов, находящихся в списке свободных буферов). Он доступен для удовлетворения запросов на участки памяти совпадающих размеров, но не может быть сцеплен со смежными буферами.

Если список находится в состоянии, требующем восстановления равновесия, распределитель помечает буфер как свободный в битовой карте и производит по возможности его слияние. Если список нуждается в скорейшей реорганизации, распределитель совмещает два буфера — только что освобожденный и дополнительно задержанный буфер, если таковой имеется. Если буфер, подвергшийся слиянию, оказывается в списке следующего по величине размера (степени 2), распределитель оценивает состояние этого класса на предмет дальнейшего слияния буферов. Каждая такая операция изменяет значение допуска, следовательно, необходима переоценка.

Для эффективной реализации описанного алгоритма буфер двунаправленно связан со списком свободных буферов. Задержанные буферы передаются в начало списка, остальные буферы — в его конец. В этом случае задержанные буферы будут повторно предоставлены для выделения в первую очередь, что является наиболее приемлемым вариантом, так как размещение таких буферов происходит быстрее всего (нет необходимости обновлять карту). Более того, в состоянии дисбаланса дополнительный отсроченный буфер может быть быстро проверен и извлечен из головы списка. Если первый буфер окажется не задержанным, можно сказать, что в списке таких буферов больше нет.

Методика отложенного слияния существенно усовершенствовала технологию близнецов. В равновесном режиме все списки находятся в безразличном состоянии, поэтому на операции слияния время не тратится. Даже в том случае, если список кренится в сторону недостатка либо избытка буферов, распределитель памяти производит слияние по крайней мере двух буферов при обработке каждого запроса. Следовательно, в худшем варианте мы имеем двойную задержку слияния на класс, что не так плохо по сравнению с простой моделью.

В работе [2] анализируется производительность простого и модифицированного алгоритма близнецов при сравнении в различных тестовых рабочих средах. Из результатов измерений видно, что последний метод показывает улучшение от 10 до 32% по сравнению с обычной схемой. Однако, как этого и следовало ожидать, методика «ленивого» слияния обладает большим значением характеристики разброса и с натугой справляется с освобождением памяти в трудных условиях.

12.7.2. Особенности реализации алгоритма в SVR4

В системе SVR4 применяются две модели пулов памяти — малых (small) и больших (large). Каждый малый пул начинается с блока длиной 4096 байтов, поделенного на 256-байтовые буферы. Первые два буфера используются для хранения структур данных блока (таких как битовая карта), остальные буферы доступны для выделения или разделения. Малые пулы позволяют создавать буферы размером от 8 до 256 байтов, кратным степени числа 2. Большой пул памяти берет начало с единственного блока размером 16 Кбайт и используется для выделения буферов размером от 512 байтов до 16 Кбайт. В нормальном состоянии оба типа пулов памяти содержат большое количество активных буферов.

Распределитель может обмениваться со страничной подсистемой участками памяти, имеющими размер пула. При необходимости получения некоторого объема памяти он запрашивает от страничного распределителя область больше или меньше. После слияния буферов в пуле распределитель возвращает его страничной подсистеме.

Буферы большого пула подвергаются объединению в соответствии с алгоритмом отложенного слияния, так как размер пула соответствует классам буферов большого размера. Для малого пула сращивание буферов осуществляется только до достижения значения 256 байтов. Сборка 256-байтовых буферов пула требует применения отдельной функции. Ее выполнение занимает ощутимый промежуток времени и вследствие этого должно происходить в фоновом режиме. Для проведения слияния и возвращения свободных пулов страничному распределителю служит системный процесс под названием kmdaemon.

12.8. Зональный распределитель в системе Mach-OSF/1

Зональный (zone) распределитель памяти, используемый в системах Mach и OSF/1, обладает возможностью быстрого выделения памяти и сбора мусора в фоновом режиме. Каждому классу динамически размещенных объектов

(таких как структуры proc, удостоверения или заголовки сообщений) выделяется собственная зона, которая является всего лишь набором свободных объектов класса. Даже в том случае, если объекты различных классов имеют одинаковый размер, каждому классу назначается отдельная зона. Например, данные как о *преобразовании портов*, так и о *наборах портов* (см. главу 6) имеют размер 104 байта [7], однако оба класса обладают собственными зонами. Система также поддерживает зоны по степени числа 2, используемые клиентами, для которых не требуется закрытое множество объектов.

Изначально зоны заполняются путем выделения памяти от распределителя страничного уровня, который позже может предоставить дополнительные ресурсы при необходимости. Каждая страница вправе быть использована только для одной зоны; следовательно, все объекты, физически расположенные на одной странице, принадлежат одному и тому же классу. Свободные объекты каждой зоны управляются при помощи связанного списка, в голове которого находится структура zone. Объекты сами по себе выделяются из *зоны над зонами* (zone of zones), каждый элемент которой является структурой zone.

Любая подсистема ядра инициализирует зону, если это необходимо. Этой цели служит функция

```
zinit (size, max, alloc, name);
```

где size указывает на размер каждого объекта, max — максимальный размер зоны в байтах, alloc — объем памяти, добавляемый зоне каждый раз, когда список свободных объектов становится пустым (ядро округляет число до целого количества страниц), name является строкой, описывающей объекты зоны. Функция zinit() выбирает структуру zone из зоны над зонами и записывает в нее значения size, max и alloc. Затем происходит запрос первоначальной области памяти размером alloc байтов у распределителя страничной памяти и деление ее на объекты размером size байтов, которые после помещаются в список свободных объектов. Все активные структуры zone связаны списком, описываемым глобальными переменными first_zone и last_zone (см. рис. 12.8). Первым элементом списка является зона над зонами, из которой инициируется выделение всех остальных элементов.

Процесс выделения и освобождения участков памяти выполняется с очень высокой скоростью и требует всего лишь удаления (или добавления) объектов из списка свободных объектов. Если при попытке выделения список окажется пустым, распределитель запросит у страничной подсистемы дополнительный объем памяти, равный alloc. Если размер пула достигнет значения max, все последующие запросы на выделение завершатся неудачно.

12.8.1. Сбор мусора

Очевидно, что вышеописанная схема требует проведения процедуры сбора мусора, иначе в результате неравномерного использования памяти большая

ее часть станет недоступна. Сбор мусора производится в фоновом режиме, поэтому не замедляет выполнение отдельных операций. Распределитель поддерживает массив, называемый *картой страниц зоны*, где каждый элемент относится к одной из страниц зоны. Вхождение карты содержит два счетчика:

- in_free_list — количество объектов из списка свободных объектов на странице;
- alloc_count — общее количество объектов страницы.

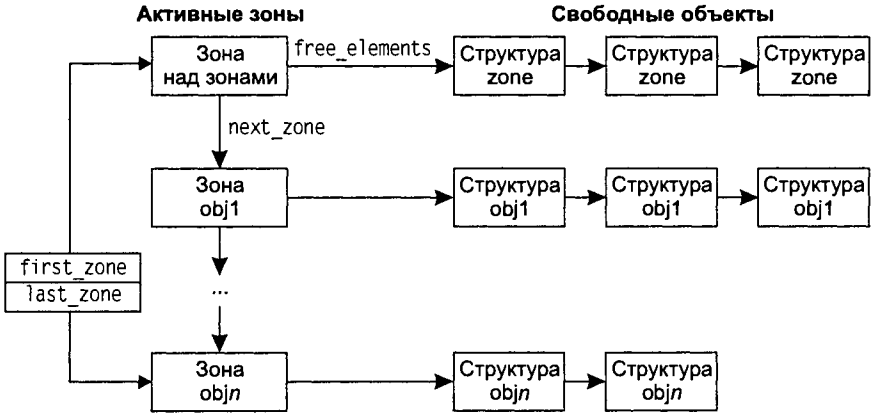

Рис. 12.8. Зональный распределитель

Значение alloc_count устанавливается при запросе страницы зоной от страничного распределителя памяти. Так как размер страницы может не быть равен размеру одного или нескольких объектов, существует возможность занятия одним объектом сразу двух страниц. В этом случае объект учитывается в счетчиках alloc_count обеих страниц. Счетчик in_free_list не изменяется при каждой операции выделения или освобождения. Вычисление его значения происходит при каждом вызове операции сбора мусора. Такой подход позволяет свести до минимума запаздывание обработки отдельных запросов.

Процедура сбора мусора zone_gc() запускается задачей swapper при каждом ее выполнении. Процедура просматривает все списки зон и для каждой из них дважды обходит список свободных объектов. При первом просмотре происходит выявление всех свободных элементов и инкрементирование счетчика in_free_count той страницы, к которой относится такой объект. Если к моменту завершения сканирования значения in_free_list и alloc_count окажутся равными, то это означает, что все объекты страницы свободны и страница может быть использована вновь. На втором этапе просмотра zone_gc() удаляет все такие объекты из списка. После этого происходит вызов kmemfree(), выполняющий передачу свободных страниц распределителю страничного уровня.

12.8.2. Анализ

Зональный распределитель является быстрым и эффективным. Он обладает простым программным интерфейсом. Объекты размещаются при помощи

```
obj = void* zalloc (struct zone* z);
```

где z указывает на зону для данного класса объектов, инициализированного ранее путем вызова zinit(). Объекты освобождаются посредством вызова

```
void zfree (struct zone* яб void* obj);
```

Процедура требует, чтобы клиенты освобождали объекты целиком, а также знали, какой зоне такие объекты принадлежат. Методика не предоставляет возможности освобождения лишь части выделенного ранее объекта.

Одним из интересных свойств методики является тот факт, что зональный распределитель использует самого себя для размещения структур zone свежесозданных зон. Это приводит к возникновению дилеммы «курицы и яйца» при первоначальной загрузке системы. Системе управления памятью требуются зоны для размещения собственных структур данных, в то время как зональной подсистеме необходим распределитель памяти страничного уровня, который является частью подсистемы управления памятью. Эта проблема решена путем использования статически сконфигурированного небольшого участка памяти, в котором создается и заполняется *зона над зонами.*

Объекты зон имеют размер, равный запрошенному, что избавляет от потерь памяти, присущих методам степени двойки. Процедура сбора мусора предоставляет механизм повторного использования памяти, свободные страницы могут быть возвращены страничной системе и позже запрошены для размещения других зон.

Основным спорным фактором методики является эффективность работы процедуры сбора мусора. Процедура выполняется в фоновом режиме, поэтому она не влияет напрямую на производительность отдельных запросов на выделение или освобождение памяти. Алгоритм сбора мусора является медленным и требует последовательного просмотра сначала всех свободных объектов, а затем — всех страниц зоны. Это влияет на общую скорость реакции системы, так как процесс сбора мусора занимает процессор до тех пор, пока не завершит свою работу.

В работе [14] утверждается, что процесс сбора мусора не сильно влияет на скорость параллельной компиляции. Однако не существует опубликованных где-либо оценок издержек процесса сбора мусора и трудно оценить его влияние на общую производительность системы. Алгоритм сбора мусора сложен и неэффективен, его стоит сравнить с составным распределителем (описываемым в разделе 12.10), который является более простым и быстрым альтернативным вариантом решения этой задачи.

12.9. Иерархический распределитель памяти для многопроцессорных систем

Выделение памяти на многопроцессорных системах разделения времени требует учета некоторых дополнительных факторов. Структуры данных, такие как списки свободных буферов или карты размещения, используемые в традиционных системах, в условиях выполнения на многопроцессорных машинах нуждаются в защите от доступа при помощи блокировки. В крупных параллельных системах это приводит к сложным условиям такой блокировки, в результате чего процессоры часто оказываются в режиме ожидания ее снятия.

Одно из решений этой проблемы реализовано в Dynix, многопроцессорном варианте системы UNIX для машин Sequent S2000 [11]. Здесь применена иерархическая схема выделения памяти, поддерживающая программный интерфейс System V. Многопроцессорные машины Sequent используются для крупных интерактивных сред обработки транзакций. Распределитель памяти при такой загрузке демонстрирует высокую производительность работы.

Архитектура распределителя приведена на рис. 12.9. Нижний *процессорный* уровень (per-CPU) производит наиболее быстрые операции, в то время как верхний уровень *слияния в страницу* (coalesce-to-page) используется для медленного процесса слияния. Есть также (на рис. не показан) еще один уровень распределителя, *слияния в vmblock* (coalesce-to-vmblock), который управляет выделением страниц внутри больших (4 Мбайт) областей памяти.

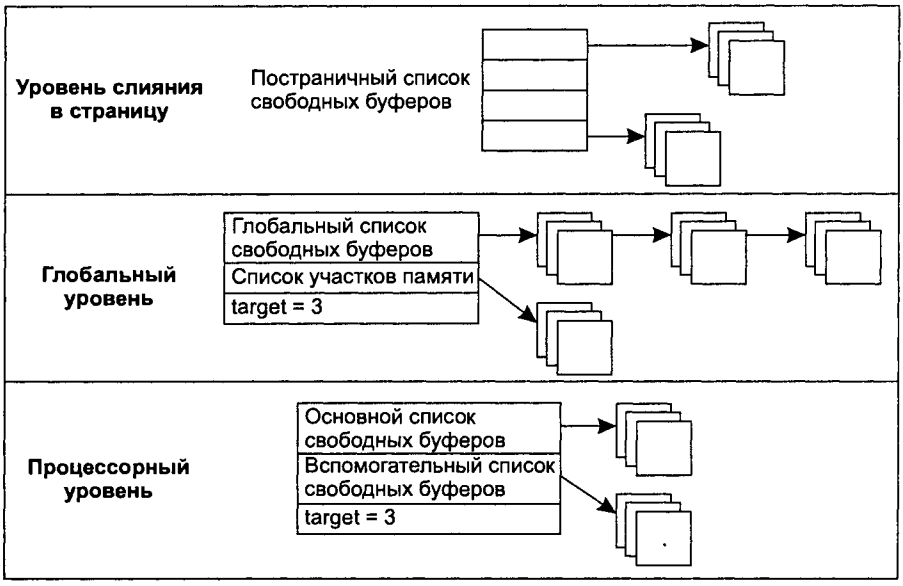

Рис. 12.9. Иерархический распределитель памяти для многопроцессорных систем

Процессорный уровень управляет одним набором пулов степени двойки для каждого процессора. Такие области изолированы от других процессоров и, следовательно, могут быть доступны без глобальной блокировки. Выделение и освобождение памяти происходит в большинстве случаев весьма быстро, так как в этом процессе участвует только локальный список свободных блоков.

Если список становится пустым, он может быть заполнен заново на *глобальном уровне* (global layer), поддерживающем собственные пулы памяти размером степени двойки. Освободившиеся буферы в кэше процессорного уровня могут быть переданы обратно на глобальный уровень. Для оптимизации обмена между этими двумя уровнями буферы группируются по значению target (например, на рис. 12.9 за один проход перемещаются три буфера), что позволяет защититься от ненужных операций со связанными списками.

Для реализации такой возможности на процессорном уровне поддерживается два списка свободных буферов — *основной* (main) и *вторичный* (aux). При выделении и освобождении буферов напрямую используется первичный список. Если он становится пустым, в него перемещаются буферы из дополнительного списка, который пополняется на глобальном уровне. Если основной список переполнится (то есть превысит target), то они будут отправлены во вторичный список, буферы которого будут возвращены на глобальный уровень. В этом случае доступ к глобальному уровню будет осуществлен по крайней мере target раз. Значение переменной target является настраиваемым. Увеличение значения target уменьшает количество операций доступа к глобальному уровню, но при этом одновременно удерживается большее количество буферов в кэшах процессорного уровня.

На глобальном уровне поддерживаются общие списки свободных буферов степени двойки. Каждый список поделен на группы буферов в соответствии с target. Случается, что нужно передать на глобальный уровень некоторое число блоков сверх значения этой переменной. Такие блоки добавляются в отдельный список сегментов памяти, являющийся промежуточной областью перед переносом в общий список свободных буферов.

Если объем общего списка превысит значение глобальной переменной target, «лишние» буферы будут возвращены на уровень *слияния в страницу*, на котором хранятся постраничные списки свободных буферов (все буферы одной страницы имеют одинаковый размер). На этом уровне буферы помещаются в список свободных буферов, к которому они относятся, и увеличивается счетчик свободных объектов страницы. Когда все буферы страницы будут возвращены в список, ее можно передавать обратно страничной системе. С другой стороны, уровень слияния в страницу может запрашивать дополнительную память от страничной системы с целью создания новых буферов.

На уровне слияния в страницу происходит сортировка всех списков по количеству свободных блоков на каждой из них. Выделение буферов выполняется из страницы, имеющей наименьшее количество свободных блоков.

Страницам с большим количеством свободных блоков дается дополнительный шанс на освобождение остальных свободных блоков, что увеличивает вероятность их возврата страничной системе. Результатом таких действий является высокая степень эффективности слияния.

12.9.1. Анализ

Алгоритм Dynix позволяет эффективно выделять память для многопроцессорных систем разделения памяти. Он поддерживает стандартный интерфейс System V и обеспечивает обмен участками памяти между распределителем и страничной системой. Кэширование на уровне процессоров уменьшает состязательность при глобальной блокировке, а двунаправленные связанные списки свободных объектов способствуют быстрому обмену буферами между процессорным и глобальным уровнем.

Интересно сравнить технологию слияния системы Dynix с зональным распределителем ОС Mach. Алгоритм Mach использует методику «пометки-и-подметания» (mark-and-sweep), где каждый раз происходит последовательное сканирование всего пула памяти. Это действие требует большого количества вычислений и вследствие этого производится в фоновом режиме. В системе Dynix всякий раз при освобождении блоков на уровне слияния на страницу изменяются и страничные структуры данных. Когда все буферы страницы освобождаются, страница может быть возвращена страничной системе. Это происходит в фоновом режиме как часть операции освобождения, что дает лишь небольшую прибавку ко времени ее выполнения, следовательно технология Dynix защищена от уменьшения производительности при наихудших условиях.

Результаты измерений [11] показали, что на однопроцессорной машине алгоритм Dynix быстрее алгоритма Мак-Кьюзика—Кэрелса в 3–5 раз. Увеличение быстродействия еще более заметно на многопроцессорных машинах (сотни и даже тысячи раз на 25 процессорах). Однако эти оценки были получены по сценарию, наиболее «легкому» для системы, при котором выделение памяти происходит из кэша аппаратного уровня. Измерения не проводились для более общего случая.

12.10. Составной распределитель системы Solaris 2.4

Составной (от англ. slab — кусок, «плита») распределитель был представлен в системе Solaris 2.4. Он позволил избавиться от многих проблем производительности, проигнорированных создателями других алгоритмов, описанных в этой главе. В результате методика показала наилучшую производительность и оптимальное использование памяти по сравнению со всеми другими реализациями. При разработке распределителя в первую очередь внимание было

обращено на три аспекта: повторное использование объектов, применение аппаратных кэшей и рабочую площадку распределителя.

12.10.1. Повторное использование объектов

Ядро использует распределитель для создания различных временных объектов, таких как индексные дескрипторы, структуры proc и сетевые буферы. Последовательность действий над объектом сводится к нижеперечисленным операциям.

1. Выделение памяти.
2. Создание (инициализация) объекта.
3. Использование объекта.
4. Уничтожение объекта.
5. Освобождение памяти.

Объекты ядра обычно являются достаточно сложными и содержат в себе дополнительные объекты, такие как счетчики ссылок, заголовки связанных списков, взаимные исключения и условные переменные. В момент создания объектов их поля устанавливаются в некое фиксированное, *начальное* состояние. При уничтожении объектов распределитель снова обращается к этим объектам и, как правило, перед освобождением памяти восстанавливает начальное состояние.

Приведем пример. Объект vnode содержит заголовок связанного списка резидентных страниц. При инициализации vnode список является пустым. Во многих реализациях UNIX [3] ядро производит сброс vnode только после того, как все его страницы были удалены из памяти. Следовательно, перед освобождением vnode (на стадии уничтожения) его связанный список снова становится пустым.

Если ядро заново использует объект для другого объекта vnode, ему не нужно повторно инициализировать заголовок связанного списка, так как это уже было выполнено на стадии уничтожения. Тот же принцип относится и к другим инициализированным полям. Например, ядро размещает объекты с первоначальным значением счетчика ссылок, равным единице, и удаляет их при освобождении последней ссылки, при этом счетчик ссылок снова становится равным единице. Взаимные исключения первично устанавливаются в *разблокированное* состояние, при освобождении для таких объектов блокировка также должна быть отменена.

Эти примеры показывают преимущества кэширования и повторного использования объектов по сравнению с выделением и инициализацией областей памяти. Кэши объектов эффективно используют доступное пространство, так как методика не прибегает к округлению до следующей степени числа 2. Зональный распределитель (раздел 12.8) также базируется на кэшировании объектов и использует память с высокой эффективностью. Он, однако, не из-

меняет состояние объектов, что не позволяет избавиться от дополнительных действий по их повторной инициализации.

12.10.2. Применение аппаратных кэшей

Традиционные распределители памяти порождают неочевидную, но очень важную проблему при использовании аппаратного кэша. Многие процессоры обладают кэшем данных первого уровня небольшого объема (размер кэша равен некоторой степени числа 2, более подробно см. в разделе 15.13). Адрес ячейки кэша вычисляется в MMU по следующей формуле:

```
cache location = address % cash size; // адрес ячейки кэша = адрес % размер кэша
```

Если в адресе присутствует отсылка на аппаратный кэш, сначала проверяется его адрес ячейки на предмет наличия данных. Если данные отсутствуют, на аппаратном уровне происходит сброс памяти в кэш, тем самым перезаписывается его прежнее содержимое.

Обычные распределители памяти (например, алгоритм Мак-Кьюзика—Кэрелса или методика близнецов) округляют размер выделяемой памяти до следующей по счету степени числа 2 и возвращают объекты этого размера. Более того, большинство объектов ядра обладают некоторыми важными, часто используемыми полями, расположенными ближе к их началу.

Эффект, обусловленный этими двумя факторами, ощутим. Представим, к примеру, реализацию, где индексный дескриптор имеет размер 300 байтов, 48 из которых являются часто востребуемыми. Ядро выделит для этого объекта 512-байтовый буфер, выравненный по 512-байтовой границе. Из них только 48 используются часто, то есть всего 9%.

В результате получается, что часть аппаратного кэша, близкого к 512-байтовой границе, заполнена важным содержимым, в то время как оставшаяся его часть используется редко. В приведенном примере индексные дескрипторы фактически нуждаются лишь в 9% кэша. Похожий расклад свойственен и другим объектам ядра. Такая аномалия в распределении буферных адресов приводит к неэффективному использованию аппаратного кэша, и, следовательно, низкой производительности операций с памятью.

Описанная проблема становится еще более заметной на машинах, где доступ к памяти осуществляется по нескольким главным шинам. Например, SPARCcenter 2000 [5] чередует передачу данных размером по 256 байтов по двум шинам. В случае последнего примера с индексными дескрипторами применительно к такой машине большинство операций доступа оккупирует шину 0, что в результате приведет к несбалансированному использованию шин.

12.10.3. Рабочая площадка распределителя

Рабочая (или опорная, footprint) площадка распределителя — это часть аппаратного кэша и *буфера ассоциативной трансляции* (translation lookaside

buffer, TLB), перезаписываемых самим распределителем. Более подробно о TLB см. в разделе 13.3.1. Распределители памяти, использующие карты ресурсов или алгоритмы близнецов, должны проверять сразу несколько объектов для нахождения подходящего буфера. Такие объекты обычно расположены в различных областях памяти, физически удаленных друг от друга. Следовательно, они чаще всего отсутствуют в кэше и TLB, что уменьшает производительность распределителя. Еще существеннее то, что при осуществлении доступа к памяти происходит перезапись активных вхождений кэша и TLB, что требует их повторного сброса из основной памяти.

Некоторые распределители, такие как Мак-Кьюзика—Кэрелса или зональный, обладают малой опорной площадкой, поскольку способны оценивать корректность области памяти путем простых вычислений и просто удаляют буфер из соответствующего списка свободных буферов. Для управления своим «следом» в кэше составной распределитель применяет такую же методику.

12.10.4. Структура и интерфейс

Составной распределитель является одним из вариантов зонального метода и организован как набор кэшей объектов. Каждый кэш содержит объекты одного типа. Следовательно, в системе существует один кэш для объектов vnode, один — для структур proc и т. д. Обычно ядро размещает объекты в соответствующих им кэшах и после производит их освобождение. Однако технология обладает механизмом, позволяющим предоставлять дополнительную память для кэша и возвращать обратно лишнюю.

Концептуально каждый кэш поделен на две составляющие (см. рис. 12.10) — *внешнюю* (front end) и *внутреннюю* (back end). Первая часть взаимодействует с клиентом памяти. Клиент получает построенные объекты из кэша и возвращает ему разрушенные объекты. Внутренняя часть взаимодействует с распределителем страничного уровня, занимаясь обменом «кусками» памяти при изменении ее конфигурации.

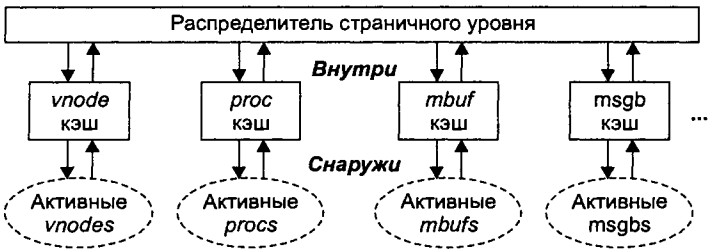

Рис. 12.10. Структура составного распределителя памяти

Подсистема ядра инициализирует кэш для управления объектами определенного типа с помощью функции

```
cachep = kmem_cache_create (name, size, align, ctor, dtor);
```

где name — символьная строка, описывающая объект, size — размер объекта в байтах, align — параметр выравнивания объектов, а ctor и dtor — указатели на функции построения и разрушения объектов соответственно. Функция возвращает указатель на кэш для данного объекта.

После этого ядро системы может заказывать объект в кэше посредством вызова

```
objp = kmem_cache_alloc (cachep, flags);
```

или освобождать его при помощи

```
kmem_cache_free (cachep, objp);
```

Эти команды не создают или уничтожают объект при повторном их использовании. Следовательно, ядро должно восстанавливать объект в *изначальном* состоянии при освобождении. Как уже описывалось в разделе 12.10.1, эти действия чаще всего производятся автоматически и не требуют проведения дополнительных операций.

Если в кэше не остается больше свободной памяти, происходит вызов функции kmem_cache_grow() для запроса области памяти от страничного распределителя и создания в ней объектов. Эта область состоит из нескольких последовательных страниц, составляющих в кэше монолитный участок. Она содержит достаточно памяти для поддержания нескольких копий объекта. Кэш использует небольшую часть области для управления памятью и разделяет свою оставшуюся часть на буферы, равные размеру объекта. После этого он инициализирует объекты путем вызова их конструкторов (операции, указанной в аргументе ctor вызова kmem_cache_create) и добавляет их в кэш.

Если распределителю страничного уровня необходимо вернуть себе часть памяти, для этой цели вызывается операция kmem_cache_reap(). Функция находит области памяти, в которых все объекты свободны, разрушает такие объекты (путем вызова операции-деструктора, указанной в аргументе dtor вызова kmem_cache_create) и затем производит удаление области из кэша.

12.10.5. Реализация алгоритма

Составной распределитель использует для объектов большого и маленького размера разные методики управления. К небольшим объектам относятся те из них, которые помещаются в одну страницу. Распределитель делит каждую область-«плиту» на три части: структуру kmem_slab, набор объектов и некоторое количество неиспользуемого пространства (см. рис. 12.11). Структура kmem_slab занимает 32 байта и располагается в конце этой области. Дополнительные четыре байта в конце каждой плиты служат для хранения указателя на список свободных объектов. Неиспользуемым пространством является некоторое количество памяти, остающееся незадействованным после создания максимально возможного количества объектов в области. Например, если

размер индексного дескриптора равен 300 байтам, область размером 4096 байтов может содержать 13 дескрипторов, 104 байта останутся неиспользованными (вспомним, что некоторое место отводится под структуру kmem_slab и указатели на списки свободных объектов). Это пространство разделяется на две части, о чем более подробно мы расскажем чуть позже.

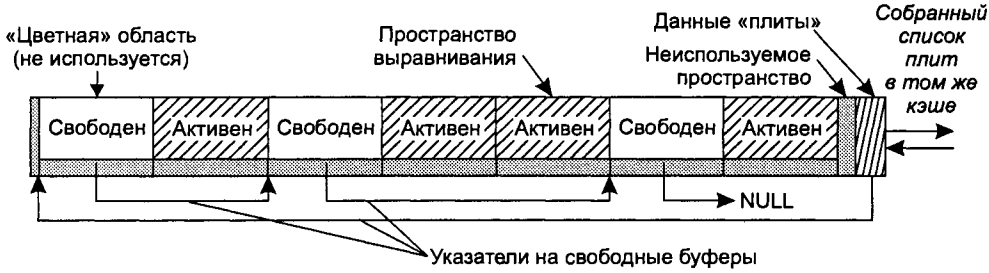

Рис. 12.11. Организация области для размещения объектов малой величины

Структура kmem_slab содержит счетчик используемых объектов, а также указатели на элемент двунаправленного связанного списка областей, расположенных в том же кэше, и указатель на первый свободный объект в области. Каждая область поддерживает собственный связанный список свободных буферов, информация о связи с которым хранится в четырехбайтовых полях, замыкающих каждый объект. Такое поле необходимо только для свободных объектов и должно различаться от самого объекта с целью невозможности его перезаписи.

Неиспользуемое пространство области делится на две части: «цветную» (slab coloring area) в начале и просто остаток непосредственно перед структурой kmem_slab. «Цветные» области получают различные размеры из соображений выравнивания. В примере, приведенном ранее, если индексный дескриптор требует 8-байтового выравнивания, он будет включать в себя 14 «цветных» областей различной длины (от 0 до 104 байтов при 8-байтовом инкременте). Это позволяет более удобно распределить стартовые смещения объектов класса, что приведет к сбалансированному и эффективному использованию аппаратных кэшей и шин памяти.

Ядро производит предоставление объекта посредством удаления первого элемента из списка свободных буферов и последующего увеличения счетчика используемых объектов. При освобождении объекта для идентификации области-плиты необходимо выполнить простейшую математическую операцию:

```
slab address = object address % slab size; // адрес плиты = адрес объекта %
размер плиты
```

Затем происходит возврат объекта в список свободных буферов области и уменьшение счетчика на единицу.

При значении счетчика используемых объектов, равном нулю, область является свободной или готовой к повторному использованию. Кэш соединяет все области в частично отсортированном двунаправленном связанном списке. В голове этого списка хранятся полностью активные области (в которых используются все объекты), в середине — частично используемые, а свободные области располагаются в его конце. Кэш также поддерживает указатель на первую область, имеющую свободный объект. Выделение памяти происходит из нее. Отсюда следует, что кэш не станет выделять объекты из полностью свободных областей до тех пор, пока не заполнятся все частично используемые области. Если страничному распределителю необходимо получить обратно часть памяти, то он произведет проверку конца списка и удалит свободные области.

Области больших размеров

Описанная реализация не позволяет эффективно использовать доступное пространство для объектов большой величины, занимающих сразу несколько страниц памяти. Для таких объектов выделение структур данных управления областью происходит из отдельного участка памяти (другого объекта кэша). Кроме структуры kmem_slab для таких объектов поддерживается еще одна, дополнительная структура kmem_bufctl. Она содержит связи со списком свободных объектов, указатель на kmem_slab и указатель на сам объект. Большая область также поддерживает таблицу хэширования для возможности обратного преобразования в структуру kmem_bufctl.

12.10.6. Анализ

Составной распределитель является прекрасно сконструированным, мощным средством выделения памяти. Он эффективно использует пространство, так как потенциальные потери ограничены структурой kmem_slab, полем связи объектов и неиспользованным участком, размер которого меньше одного объекта области. Большинство запросов обслуживается с максимальной быстротой путем удаления объектов из списка свободных буферов и изменения значения счетчика используемых объектов. Применение схемы «цветных» секций приводит к оптимальному использованию аппаратного кэша и шины памяти, что увеличивает общую производительность системы. Методика также имеет малую рабочую площадку в кэше, так как при обработке большинства запросов происходит доступ только к одной области.

Алгоритм сбора мусора является более простым по сравнению с зональным распределителем памяти, несмотря на то, что обе методики основаны на одних и тех же принципах. Затраты на сбор мусора минимальны, так как распределены по всем запросам: при обработке каждой операции изменяется значение счетчика используемых объектов. Операция возвращения памяти немного увеличивает нагрузку на систему, так как ей приходится производить сканирование различных кэшей с целью обнаружения свободных областей.

Максимально возможное влияние на производительность пропорционально общему количеству кэшей (но не количеству областей).

Среди недостатков составного распределителя следует указать на издержки управления памятью, связанные с тем, что для каждого типа объектов применяется отдельный кэш. Для обычных классов объектов, расположенных в кэше большого объема и часто используемых, издержки являются незначительными. Однако для малых, редко востребуемых объектов, потери часто являются абсолютно неприемлемыми. Вышеописанная проблема относится и к зональному распределителю памяти, используемому в ОС Mach, и решается путем поддержки набора буферов размером, равным некоторой степени числа 2, в которых размещаются нестандартные объекты, не нуждающиеся в собственном кэше.

Составной распределитель не должен отказываться от преимуществ дополнительных процессорных кэшей, наподобие используемых в Dynix. Это упоминается в работе [4] как одно из возможных дополнительных средств для будущих реализаций.

12.11. Заключение

При создании общецелевого распределителя памяти необходимо учитывать множество различных факторов: он должен быть быстрым, простым в использовании и уметь эффективно управлять памятью. В этой главе мы рассмотрели несколько технологий распределения памяти и проанализировали их преимущества и недостатки. Распределитель на основе карты ресурсов является единственным поддерживающим частичное освобождение объекта. Однако используемые в нем методики последовательного поиска приводят к снижению производительности, неприемлемому для большинства приложений. Распределитель Мак-Кьюзика–Кэрелса обладает простым интерфейсом с привлечением стандартных вызовов malloc() и free(). Вместе с тем в нем отсутствует механизм слияния буферов и возврата участков памяти, ранее запрошенных у распределителя страничного уровня. Методика близнецов, наоборот, производит постоянное слияние и деление буферов с целью удовлетворения изменяющихся потребностей к памяти. Производительность ее работы, как правило, низка, особенно при частом вызове операции объединения буферов. Зональный распределитель является весьма быстрым, но обладает недостаточно эффективными механизмами сбора мусора.

Dynix и составной распределитель вобрали в себя достоинства всех перечисленных выше методов. В Dynix применяется методика степеней двойки, сочетающаяся с использованием процессорного кэша и быстрым сбором мусора (на уровне слияния в страницу). Составной распределитель представляет собой модифицированный зональный алгоритм. Увеличение производительности достигается за счет повторного использования объектов и сбалансированного распределения адресов памяти. В этом методе применяется простой

алгоритм сбора мусора, не самым худшим образом влияющий на производительность. Как уже упоминалось ранее, добавление поддержки аппаратных кэшей в алгоритм позволило реализовать на его основе прекрасный распределитель памяти.

В таблице 12.1 показаны результаты ряда экспериментов [4] по сравнению составного распределителя с методикой, реализованной в системе SVR4 и алгоритмом Мак-Кьюзика—Кэрелса. Опыты также показали, что повторное использование объектов уменьшает время, необходимое на их размещение и инициализацию, в 1,3–5,1 раза, в зависимости от типа объекта. Это преимущество является приятным дополнением к улучшенным показателям времени выделения памяти, охарактеризованным таблицей.

Таблица 12.1. Измерения производительности популярных распределителей памяти

	SVR4	Мак-Кьюзика—Кэрелса	Составной распределитель
Среднее время, требуемое для alloc и free, мкс	9,4	4,1	3,8
Общая фрагментация (потери памяти)	46%	45%	14%
Производительность по тесту Kenbus (количество сценариев, выполненное за 1 мин)	199	205	233

Многие методики, рассмотренные в этой главе, годятся для новых реализаций распределителей памяти прикладного уровня. Однако требования к таким программам выдвигаются несколько особые, поэтому хороший распределитель памяти ядра может иметь плохую производительность на прикладном уровне и наоборот. Распределители прикладного уровня обычно управляют большими объемами (виртуальной) памяти, практически ничем не ограничиваемой для большинства приложений. Следовательно, для этого применения памяти слияние буферов и выравнивание адресов не являются настолько критичными моментами по сравнению со скоростью выделения и освобождения памяти. Для прикладного уровня также важны простота и стандартность предлагаемого интерфейса взаимодействия, поскольку таковой обычно используется совершенно отличающимися друг от друга приложениями, созданными разными программистами. Некоторые реализации распределителей памяти прикладного уровня описаны в [9].

12.12. Упражнения

1. Чем отличаются требования, предъявляемые к распределителю памяти ядра и распределителю памяти прикладного уровня?
2. Назовите максимальное количество вхождений карты ресурсов, используемое для обработки ресурса, имеющего `N` элементов.

3. Создайте программу, измеряющую использование памяти и производительность распределителя на основе карт ресурсов (при помощи специально сгенерированной последовательности запросов). Сравните с ее помощью варианты выделения участков: лучшего, худшего и первого подходящего.
4. Предложите свою реализацию функции free() для распределителя Мак-Кьюзика—Кэрелса.
5. Создайте процедуру scavenge(), выполняющую слияние свободных страниц в распределителе Мак-Кьюзика—Кэрелса и освобождение их для распределителя страничного уровня.
6. Реализуйте простейший алгоритм близнецов, управляющий 1024-байтовой областью памяти с минимальным размером участка, равным 16 байтам.
7. Определите проследовательность запросов, которые могут привести к наихудшему варианту поведения простого алгоритма близнецов.
8. Как отразятся на значении переменных N, A, L, G и *допуска* следующие события, происходящие в алгоритме отложенного слияния системы SVR4 (см. раздел 12.7):
   - буфер освобождается при значении допуска больше двух;
   - перераспределяется задержанный буфер;
   - выделяется не задержанный буфер (нет задержанных буферов);
   - буфер освобождается при допуске, равном 1, при этом ни один из свободных буферов не может быть подвергнут слиянию, так как все близнецы в данный момент используются;
   - буфер подвергнут слиянию со своим близнецом.
9. Можно ли преобразовать методики распределения памяти так, чтобы они поддерживали список свободных объектов для каждого процессора в случае использования в многопроцессорных системах (см. технологию Dynix)? Какие из алгоритмов нельзя адаптировать для этой методики и почему?
10. Почему в составном распределителе используются две различные реализации управления большими и малыми объектами?
11. Какой распределитель из описанных в этой главе позволяет клиенту освобождать часть выделенного ранее блока?
12. Какой из распределителей способен отклонить запрос на выделение даже в том случае, если ядро в данный момент обладает блоком памяти, размером достаточным для удовлетворения такого запроса?

12.13. Дополнительная литература

1. Bach, M. J., «The Design of the UNIX Operating System», Prentice-Hall, Englewood Cliffs, NJ, 1986.
2. Barkley, R. E., and Lee, T. P., «A Lazy Buddy System Bound By Two Coalescing Delays per Class», Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, Dec. 1989, pp. 167–176.
3. Barkley, R. E., and Lee, T. P., «A Dynamic File System Inode Allocation and Reclaim Policy», Proceedings of the Winter 1990 USENIX Technical Conference, Jan. 1990, pp. 1–9.
4. Bonwick, J., «The Slab Allocator: An Object-Caching Kernel Memory Allocator», Proceedings of the Summer 1994 USENIX Technical Conference, Jun. 1994, pp. 87–98.
5. Cekleov, M., Frailong, J. M., and Sindhu, P., «Sun–4D Architecture», Revision 1.4, Sun Microsystems, 1992.
6. Digital Equipment Corporation, «Alpha Architecture Handbook», Digital Press, 1992. Digital Equipment Corporation, DEC OSF/1 Internals Overview — Student Workbook, 1993.
7. Knuth, D., «The Art of Computer Programming», Vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, MA, 1973.
8. Korn, D. G., and Vo, K. P., «In Search of a Better Malloc», Proceedings of the Summer 1985 USENIX Technical Conference, Jun. 1985, pp. 489–505.
9. Lee, T. P., and Barkley, R. E., «A Watermark-Based Lazy Buddy System for Kernel Memory Allocation», Proceedings of the Summer 1989 USENIX Technical Conference, Jun. 1989, pp. 1–13.
10. McKenney, P. E., and Slingwine, J., «Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors», Proceedings of the Winter 1993 USENIX Technical Conference, Jan. 1993, pp. 295–305.
11. McKusick, M. K., and Karels, M. J., «Design of a General-Purpose Memory Allocator for the 4.3BSD UNIX Kernel», Proceedings of the Summer 1988 USENIX Technical Conference, Jun. 1988, pp. 295–303.
12. Peterson, J. L., and Norman, T. A., «Buddy Systems», Communications of the ACM, Vol. 20, No. 6, Jun. 1977, pp. 421–431.
13. Sciver, J. V., and Rashid, R. F., «Zone Garbage Collection», Proceedings of the USENIX Mach Workshop, Oct. 1990, pp. 1–15.
14. Stephenson, C. J., «Fast Fits: New Methods for Dynamic Storage Allocation», Proceedings of the Ninth ACM Symposium on Operating Systems Principles, Vol. 17, No. 5, 1983, pp. 30–32.