ГЛАВА 4

Управление оперативной памятью

Что-то с памятью моей
стало…
Р. Рождественский

Основной ресурс системы, распределением которого занимается ОС, — оперативная память. Поэтому организация памяти оказывает большое влияние на структуру и возможности ОС. Во введении мы видели, что основное различие между двумя большими классами систем, которые в данной книге называются ОС и ДОС, состоит в том, используют эти системы средства трансляции адресов при доступе к памяти (так называемую виртуальную память) или нет. В настоящее время сложилась даже более интересная ситуация — переносимая операционная система UNIX, рассчитанная на машины со страничным диспетчером памяти, произвела жесткий отбор, и теперь практически все машины общего назначения, начиная от x86 и заканчивая суперкомпьютерами или, скажем, процессором Alpha, имеют именно такую организацию адресного пространства.

Самый простой вариант задачи управления памятью — отсутствие диспетчера памяти, т. е. совпадение физического и логического адресных пространств. В этих условиях система не может контролировать доступ пользовательских программ к памяти; если мы определяем процесс как единицу исполнения, обладающую собственным адресным пространством, то вся система вместе с пользовательскими задачами представляет собой единый процесс. Управление памятью со стороны ОС ограничивается только решением вопроса, какая память занята кодом и данными программы, а какая — свободна. Даже в этих условиях управление памятью представляет собой непростую задачу, для решения которой придумано много различных (и, в ряде отношений, не совсем удачных) решений.

В системах с виртуальной памятью мы сталкиваемся с вариантом той же самой задачи при управлении виртуальным адресным пространством в пределах процесса. Действительно, наличие диспетчера памяти дает довольно большую свободу во время отображения адресного пространства на физическую память, но при распределении самого адресного пространства мы сталкиваемся с той же самой задачей, что и при управлении открытой памятью: необходимо гарантировать, что ни один вновь создаваемый объект не будет конфликтовать по [виртуальным] адресам с уже существовавшими объектами. Поэтому многие из алгоритмов, первоначально придуманных для управления открытой физической памятью, находят применение при управлении виртуальным адресным пространством в пределах задачи.

4.1. Открытая память

Рассмотрим простейшую однозадачную ОС, в которой нет никакого диспетчера памяти и допускается работа только одной задачи. Именно так работают CP/M и RT-11 SJ (Single-Job, однозадачная). В этих системах программы загружаются с фиксированного адреса `PROG_START`. В CP/M это 0x100; в RT-11 — 01000. По адресам от 0 до начала программы находятся векторы прерываний, а в RT-11 — также и стек программы. Операционная система размещается в старших адресах памяти. Адрес `SYS_START`, с которого она начинается, зависит от количества памяти у машины и от конфигурации ОС (рис. 4.1).

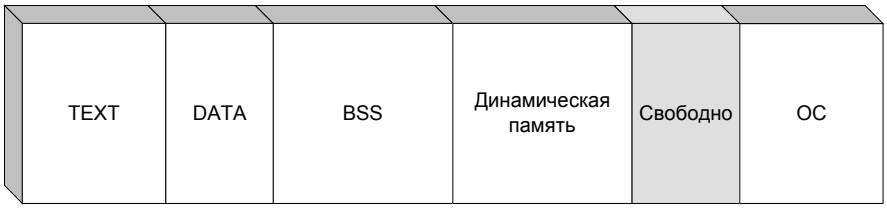

Рис. 4.1. Управление памятью в однопроцессной ОС с открытой памятью

В этом случае управление памятью со стороны системы состоит в том, что загрузчик проверяет, поместится ли загружаемый модуль в пространство от `PROG_START` до `SYS_START`. Если объем памяти, который использует программа, не будет меняться во время ее исполнения, то на этом все управление и заканчивается.

Однако программа может использовать динамическое управление памятью, например, функцию `malloc()` или что-то в этом роде. В таком случае уже код `malloc()` должен следить за тем, чтобы не залезть в системные адреса. Как правило, динамическая память начинает размещаться с адреса `PROG_END` =

`PROG_START + PROG_SIZE`. `PROG_SIZE` в данном случае обозначает полный размер программы, т. е. размер ее кода, статических данных и области, выделенной под стек.

Функция `malloc()` поддерживает некоторую структуру данных, следящую за тем, какие блоки памяти из уже выделенных были освобождены, так называемый *пул* (pool, от англ. лужа, бассейн) или *кучу* (heap). При каждом новом запросе она сначала ищет блок подходящего размера в пуле, и только когда этот поиск завершится неудачей, просит новый участок памяти у системы. Для этого используется переменная, которая в библиотеке языка С называется `brk_addr` (рис. 4.2).

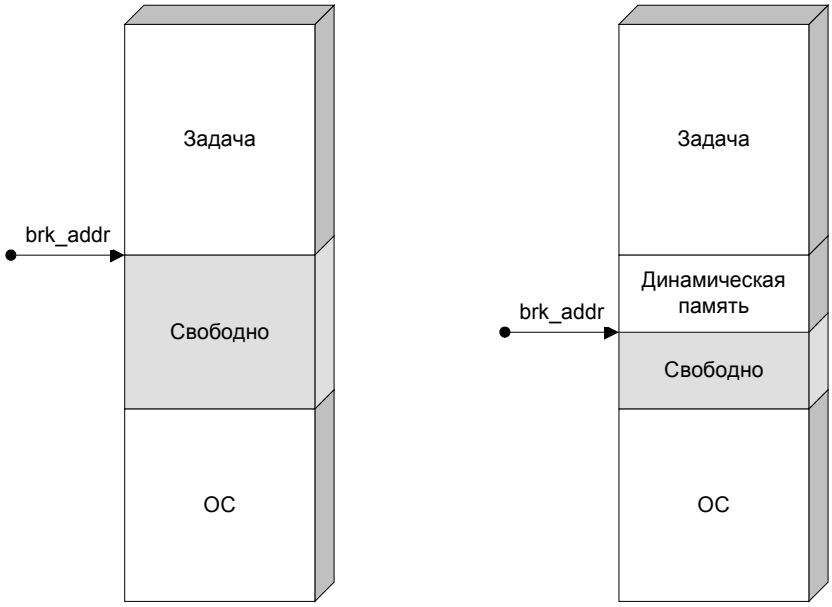

Рис. 4.2. Загруженная программа и `brk_addr`

Изначально эта переменная равна `PROG_END`, ее значение увеличивается при выделении новых блоков, но в некоторых случаях может и уменьшаться. Это происходит, когда программа освобождает блок, который заканчивается на текущем значении `brklevel`.

Аналогичным образом происходит выделение адресного пространства в многозадачных ОС с виртуальной памятью. Задача обычно занимает начальные адреса, но не с самого начала — например, в Unix первые 8 Мбайт виртуального адресного пространства заняты так называемой *сторожевой зоной* (guard area), доступ к которой запрещен. Затем идут код и статические данные задачи, затем ее динамическая память. Окончание динамической памяти

отмечается значением `brk_addr`. Как и в простых системах с открытой памятью, задача просит у ОС дополнительной памяти, вызывая системный вызов `sbrk` (пример 4.1).

Пример 4.1. Выделение дополнительной памяти в GNU LibC для Linux

morecore.c:

```
/* Copyright (C) 1991, 1992 Free Software Foundation, Inc.
Этот файл является частью библиотеки C проекта GNU (GNU C Library).

GNU C Library является свободным программным обеспечением;
вы можете передавать и/или модифицировать ее в соответствии
с положениями GNU General Public License версии 2 или (по вашему выбору)
любой более поздней версии.

Библиотека GNU C распространяется в надежде, что она будет полезна, но
БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ; даже без неявно предполагаемых гарантий

КОММЕРЧЕСКОЙ ЦЕННОСТИ или ПРИГОДНОСТИ ДЛЯ КОНКРЕТНОЙ ЦЕЛИ.
Подробнее см. GNU General Public License.

Вы должны были получить копию GNU General Public License вместе с
GNU C Library; см. файл COPYING. Если вы ее не получили, напишите по
адресу: Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.
*/

#ifndef        _MALLOC_INTERNAL
#define        _MALLOC_INTERNAL
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/syscall.h>
#endif

#ifndef        __GNU_LIBRARY__
#define        __sbrk        sbrk
#endif

extern void * ___brk_addr;

extern __ptr_t __sbrk __P ((int increment));
```

```
extern int __init_brk __P ((void));

#ifndef NULL
#define NULL 0
#endif

/* Выделить еще INCREMENT байтов пространства данных
   и возвратить начало пространства данных или NULL при ошибках.
   Если INCREMENT отрицателен, сжать пространство данных.  */

__ptr_t
__default_morecore (ptrdiff_t increment)
{
  __ptr_t result = __sbrk ((int) increment);
  if (result == (__ptr_t) -1)
    return NULL;
  return result;
}

/* Эта функция почти полностью аналогична __default_morecore ().
 * Но она вызывается только однажды через __morecore.

 */

__ptr_t
__default_morecore_init (ptrdiff_t increment)
{
  __ptr_t result;

  if ( __init_brk() != 0)
    return NULL;

  if (__morecore == __default_morecore_init)
       _morecore = __default_morecore;

  result = __sbrk ((int) increment);
  if (result == (__ptr_t) -1)
    return NULL;
  return result;
}
```

В простейшем случае, все остальное адресное пространство доступно задаче; ОС при этом находится в отдельном адресном пространстве и задача ее не видит. Благодаря этому, с одной стороны, задача не может повредить код и данные ОС, и, с другой стороны, задача может использовать больше памяти.

В более сложных случаях код и данные задачи занимают несколько несмежных областей в ОЗУ (виртуальном или физическом). Это бывает в многозадачных системах с открытой памятью (Win16, старые версии Mac OS, Novell Netware и др.); в системах с виртуальной памятью это получается из-за использования разделяемых библиотек, разделяемой памяти и отображения файлов на память. В таких системах пул динамической памяти обычно оказывается разбит на несмежные области. В последующих разделах данной главы мы увидим, что большинство стратегий управления пулом не требуют, чтобы пул был непрерывен.

4.2. Алгоритмы динамического управления памятью

> Герой имел привычку складывать окурки в кожаный кисет и употреблять их для изготовления новых самокруток. Таким образом, согласно велению неумолимого закона средних чисел, какую-то часть этого табака он курил в течение многих лет.
>
> *Т. Пратчетт*

При *динамическом выделении памяти* запросы на выделение памяти формируются во время исполнения задачи. Динамическое выделение, таким образом, противопоставляется *статическому*, когда запросы формируются на этапе компиляции программы. В конечном итоге, и те, и другие запросы нередко обрабатываются одним и тем же алгоритмом выделения памяти в ядре ОС. Но во многих случаях статическое выделение можно реализовать намного более простыми способами, чем динамическое. Главное отличие здесь в том, что при статическом выделении кажется неестественной — и поэтому редко требуется — возможность отказаться от ранее выделенной памяти. При динамическом же распределении часто требуется предоставить возможность отказываться от запрошенных блоков так, чтобы освобожденная память могла использоваться для удовлетворения последующих запросов. Таким образом, динамический распределитель (*аллокатор*, от англ. allocator) вместо простой границы между занятой и свободной памятью (которой достаточно в простых случаях статического распределения) вынужден хранить список возможно несвязных областей свободной памяти, называемый *пулом* (pool) или *кучей* (heap).

Многие последовательности запросов памяти и отказов от нее могут привести к тому, что вся доступная память будет разбита на блоки маленького размера, и попытка выделения большого блока завершится неудачей, даже если сумма длин доступных маленьких блоков намного больше требуемой. Это явление называется *фрагментацией памяти* (рис. 4.3). Иногда используют более точный термин — *внешняя фрагментация* (что такое внутренняя фрагментация, будет рассказано далее). Кроме того, большое количество блоков требует длительного поиска. Существует также много мелких трудностей разного рода. К счастью, человечество занимается проблемой распределения памяти уже давно и найдено много хороших или приемлемых решений.

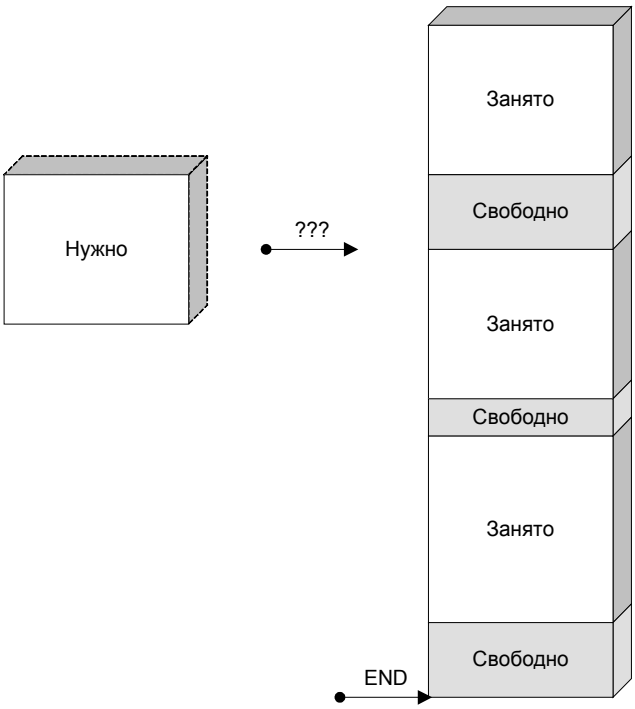

Рис. 4.3. Внешняя фрагментация

В зависимости от решаемой задачи используются различные стратегии поиска свободных блоков памяти. Например, программа может выделять блоки одинакового размера или нескольких фиксированных размеров. Это сильно облегчает решение задач дефрагментации и поиска свободных участков ОЗУ.

Возможны ситуации, когда блоки освобождаются в порядке, обратном тому, в котором они выделялись. Это позволяет свести выделение памяти к стеко-

вой структуре, т. е. фактически вернуться к простому запоминанию границы между занятой и свободной памятью.

Возможны также ситуации, когда некоторые из занятых блоков можно переместить по памяти — тогда есть возможность проводить *дефрагментацию памяти*, перемещение занятых блоков памяти с целью объединить свободные участки. Например, функцию `realloc()` в ранних реализациях системы UNIX можно было использовать именно для этой цели.

В стандартных библиотечных функциях языков высокого уровня, таких как `malloc/free/realloc` в C, `new/dispose` в Pascal и т. д., как правило, используются алгоритмы, рассчитанные на наиболее общий случай: программа запрашивает блоки случайного размера в случайном порядке и освобождает их также случайным образом.

Впрочем, случайные запросы — далеко не худший вариант. Даже не зная деталей стратегии управления кучей, довольно легко построить программу, которая "испортит жизнь" многим распространенным алгоритмам (пример 4.2). Доказано [Кнут 2000], что для любого алгоритма динамического выделения памяти, который не допускает перемещения выделенных блоков, можно построить такую последовательность запросов на выделение и освобождение памяти, которая приведет к блокировке алгоритма — невозможности удовлетворить запрос, несмотря на то, что общий объем доступной памяти превышает запрашиваемый.

Пример 4.2. Пример последовательности запросов памяти

```
while(TRUE) {
 void * b1 = malloc(random(10));
 /* Случайный размер от 0 до 10 байт */
 void * b2 = malloc(random(10)+10);
 /* ........  от 10 до 20 байт */

 if(b1 == NULL && b2 == NULL) /* Если памяти нет */
  break;     /* Выйти из цикла */

 free(b1);
}
void * b3 = malloc(150);

/* Скорее всего, память не будет выделена */
```

В результате исполнения такой программы вся доступная память будет "порезана на лапшу": между любыми двумя свободными блоками будет размещен занятый блок меньшего размера (рис. 4.4).

К счастью, пример 4.2 имеет искусственный характер. В реальных программах такая ситуация встречается редко, и часто оказывается проще исправить программу, чем вносить изменения в универсальный алгоритм управления кучей.

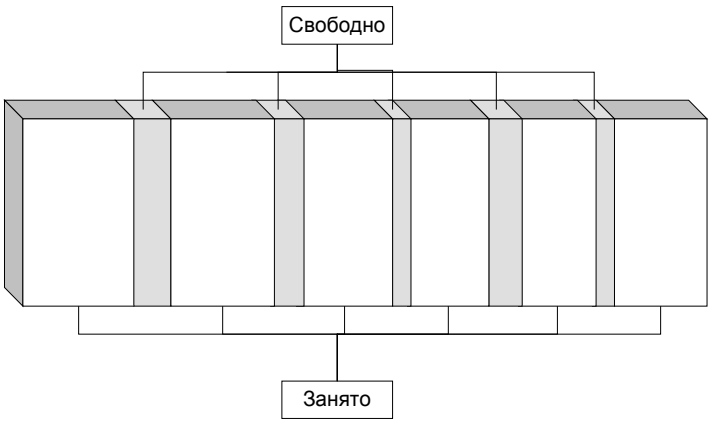

Рис. 4.4. Результат работы программы примера 4.2

Впрочем, необходимо еще раз подчеркнуть, что задача динамического управления памятью (как физическим ОЗУ, так и виртуальным адресным пространством) в общем случае неразрешима: если мы требуем, чтобы выделяемые блоки памяти занимали непрерывный диапазон и не можем перемещать занятые блоки, мы всегда можем построить последовательность запросов, которая заведет наш аллокатор в тупик. При этом оба допущения (непрерывность блоков и невозможность их перемещения) вполне реалистичны. Так, стандартная схема индексации в массивах (сложение индекса и начального адреса) предполагает, что массив непрерывен. Эта стандартная схема привлекательна тем, что обращение к элементам массива происходит за фиксированное (и весьма малое) время. Существуют различные способы построения динамических массивов, способных занимать несмежные участки памяти, но время обращения к элементу такого массива зависит от количества фрагментов, на которые разбит массив, а это далеко не всегда приемлемо.

Далее, выделение памяти всегда предполагает последующее обращение к выделенной памяти, т. е. сохранение указателей на нее. В большинстве языков программирования, в том числе в C/C++ и Pascal, невозможно проследить дальнейшую судьбу этих указателей; даже в тех языках, где указатели во вре-

мя исполнения отличаются от скалярных данных (например, в Java/C#), такое прослеживание оказывается недопустимо дорогой операцией, даже при сборке мусора *(см. разд 4.3)* не собирается всей необходимой для этого информации. Если выделенный блок памяти используется для хранения кода, то код в момент загрузки будет настроен на эти адреса; большинство загрузчиков отбрасывают таблицу перемещений после загрузки и не допускают последующего перемещения однажды настроенного кода.

Пример 4.2 построен на том предположении, что система выделяет нам блоки памяти, размер которых соответствует запрошенному с точностью до байта. Если же минимальная единица выделения равна 32 байтам, никакой внешней фрагментации наш пример не вызовет: на каждый запрос будет выделяться один блок. Но при этом мы столкнемся с обратной проблемой, которая называется *внутренней фрагментацией*: если система умеет выделять только блоки, кратные 32 байтам, а нам реально нужно 15 или 47 байт, то 17 байт на блок окажутся потеряны (рис. 4.5).

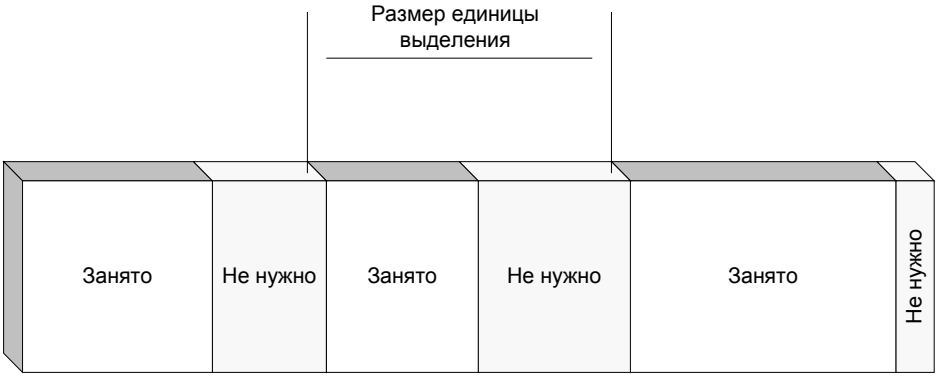

Рис. 4.5. Внутренняя фрагментация

Чем больше размер единицы выделения, тем меньше нам грозит фрагментация внешняя, и тем большие потери обеспечивает фрагментация внутренняя. Величина потерь зависит от среднего размера запрашиваемого блока. Грубая оценка свидетельствует о том, что в каждом блоке в среднем теряется половина единицы выделения, т. е. отношение занятой памяти к потерянной будет $\frac{1}{2}\frac{Ns}{N\bar{l}}$, где N — количество выделенных блоков, s — размер единицы выделения, а $\bar{l}$ — средний размер блока. Упростив эту формулу, мы получим вы-

ражение для величины потерь: $\frac{s}{2\bar{l}}$, т. е. потери линейно растут с увеличением размера единицы выделения.

Если средний размер блока сравним с единицей выделения, наша формула теряет точность, но все равно дает хорошую оценку порядка величины потерь. Так, если $s = \bar{l}$, наша формула дает 50% потерь, что вполне согласуется со здравым смыслом: если запрашиваемый блок чуть короче минимально возможного, теряется только это "чуть"; зато если он чуть длиннее, то для него отводится два минимальных блока, один из которых теряется почти весь. Точная величина потерь определяется распределением запрашиваемых блоков по длине (для ее вычисления требуется брать интеграл от функции этого распределения), но я предпочитаю оставить вывод точной формулы любопытному читателю.

Все системы управления памятью (не только оперативной, но и дисковой), которые в той или иной форме допускают размещение объектов в несмежных областях памяти и их перемещение (именно это делают сегментные и страничные диспетчеры памяти, файловые системы и некоторые другие системы управления памятью, которые не будут рассматриваться в этой книге, например, алгоритмы размещения таблиц и индексов реляционных СУБД), также в той или иной форме предполагают наличие минимального блока памяти, который подвергается переадресации и/или перемещению. Поэтому ни одно известное средство борьбы с внешней фрагментацией не может избавиться от внутренней фрагментации.

Варианты алгоритмов распределения памяти исследовались еще в 50-е годы XX века. Итоги многолетнего изучения этой проблемы приведены в [Кнут 2000] и многих других учебниках, и они в определенном смысле неутешительны: хотя есть стратегии управления памятью, которые в каких-то отношениях лучше других, но универсального алгоритма, который не страдал бы от того или иного вида фрагментации и был предсказуем в других отношениях, не существует. Как уже отмечалось, во многих случаях оказывается проще исправить программу (привести ее в соответствие с ожиданиями алгоритма распределения памяти), чем подбирать "правильный" аллокатор. Это один из примеров "самосбывающейся" статистики, когда ориентация ОС или стандартных библиотек языков программирования на определенную модель поведения пользовательских программ приводит к распространению программ, более или менее соответствующих этой статистике.

Обычно все свободные блоки памяти объединяются в двунаправленный связанный список. Список должен быть двунаправленным для того, чтобы из него в любой момент можно было извлечь любой блок. Впрочем, если все

действия по извлечению блока производятся после поиска, то можно слегка усложнить процедуру поиска и всегда сохранять указатель на предыдущий блок. Это решает проблему извлечения и можно ограничиться однонаправленным списком. Беда только в том, что многие алгоритмы при объединении свободных блоков извлекают их из списка в соответствии с адресом, поэтому для таких алгоритмов двунаправленный список остро необходим.

Поиск в списке может вестись тремя способами: до нахождения *первого подходящего* (first fit) блока, до блока, размер которого ближе всего к заданному — *наиболее подходящего* (best fit), и, наконец, до нахождения самого большого блока, *наименее подходящего* (worst fit).

Использование стратегии worst fit имеет смысл разве что в сочетании с сортировкой списка по убыванию размера. Это может ускорить выделение памяти (всегда берется первый блок, а если он недостаточно велик, мы с чистой совестью можем сообщить, что свободной памяти нет), но создает проблемы при освобождении блоков: время вставки в отсортированный список пропорционально $O(n)$, где n — размер списка.

Помещать блоки в отсортированный массив еще хуже — время вставки становится $O(n + \log(n))$ и появляется ограничение на количество блоков. Использование хэш-таблиц или двоичных деревьев требует накладных расходов и усложнений программы, которые себя в итоге не оправдывают.

В действительности, можно использовать для организации кучи своеобразную структуру данных, которая также называется кучей (heap), — это своего рода дерево, в котором самые большие элементы размещаются не в самой правой ветви (как в сортированных деревьях), а ближе к корню дерева. Такие структуры данных используются в качестве промежуточных при сортировке. Не исключено, что когда-то аллокаторы были устроены именно так; возможно даже, что название "куча" происходит именно от использования таких структур. На практике стратегия worst fit используется при размещении пространства в файловых системах, например в HPFS, но ни одного примера ее использования для распределения оперативной памяти в современных ОС и средах исполнения ЯВУ мне не известно.

Чаще всего применяют несортированный список. Для нахождения наиболее подходящего мы обязаны просматривать весь список, в то время как первый подходящий может оказаться в любом месте, и среднее время поиска будет меньше. Насколько меньше — зависит от отношения количества подходящих блоков к общему количеству. (Читатели, знакомые с теорией вероятности, могут самостоятельно вычислить эту зависимость.)

В общем случае best fit увеличивает фрагментацию памяти. Действительно, если мы нашли блок с размером больше заданного, мы должны отделить "хвост" и пометить его как новый свободный блок. Понятно, что в случае best fit средний размер этого "хвоста" будет маленьким, и мы в итоге получим

большое количество мелких блоков, которые невозможно объединить, т. к. пространство между ними занято.

В тех ситуациях, когда мы размещаем блоки нескольких фиксированных размеров, этот недостаток роли не играет и стратегия best fit может оказаться оправданной. Однако библиотеки распределения памяти рассчитывают на общий случай, и в них обычно используются алгоритмы first fit.

При использовании first fit с линейным двунаправленным списком возникает специфическая проблема. Если каждый раз просматривать список с одного и того же места, то большие блоки, расположенные ближе к началу, будут чаще удаляться. Соответственно, мелкие блоки будут скапливаться в начале списка, что увеличит среднее время поиска (рис. 4.6). Простой способ борьбы с этим явлением состоит в том, чтобы просматривать список то в одном направлении, то в другом. Более радикальный и еще более простой метод заключается в следующем: список делается кольцевым, и каждый поиск начинается с того места, где мы остановились в прошлый раз. В это же место добавляются освободившиеся блоки. В результате список очень эффективно перемешивается и никакой "антисортировки" не возникает.

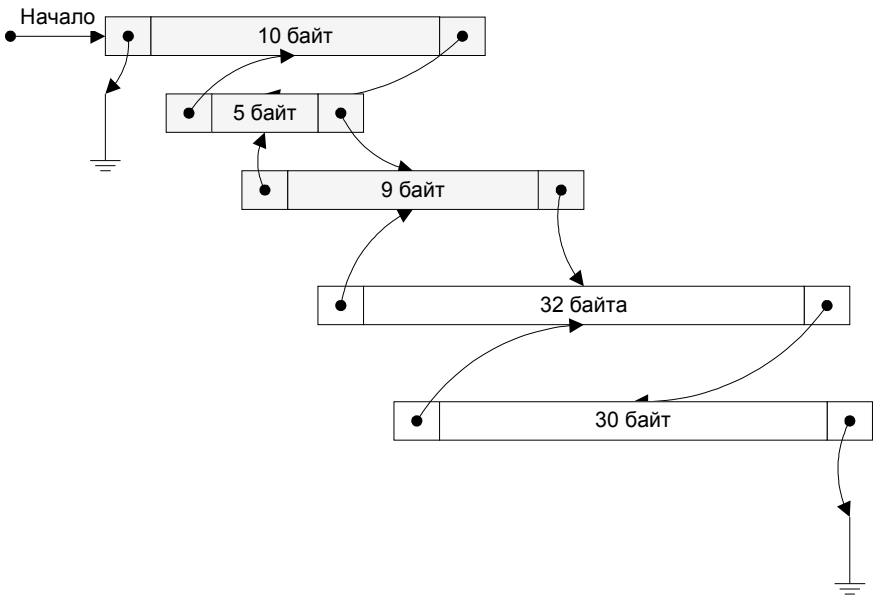

Рис. 4.6. Антисортировка

Разработчик программы динамического распределения памяти обязан решить еще одну важную проблему, а именно — объединение свободных блоков. Действительно, обидно, если мы имеем сто свободных блоков по одному ки-

лобайту и не можем сделать из них один блок в сто килобайт. Но если все эти блоки расположены в памяти один за другим, а мы не можем их при этом объединить, — это просто унизительно.

Кроме того, если мы умеем объединять блоки и видим, что объединенный блок ограничен сверху значением `brklevel`, то мы можем, вместо помещения этого блока в список, просто уменьшить значение `brklevel` и, таким образом, вернуть ненужную память системе.

Представим себе для начала, что все, что мы знаем о блоке, — это его начальный адрес и размер. Легко понять, что это очень плохая ситуация. Действительно, для объединения блока с соседями мы должны найти их в списке свободных или же убедиться, что там их нет. Для этого мы должны просмотреть весь список. Как одну из идей мозгового штурма можно выдвинуть предложение сортировать список свободных блоков по адресу.

Гораздо проще запоминать в дескрипторе блока указатели на дескрипторы соседних блоков. Немного развив эту идею, мы приходим к методу, который называется *алгоритмом парных меток* и состоит в том, что мы добавляем к каждому блоку по два слова памяти. Именно слова, а не байта. Дело в том, что требуется добавить достаточно места, чтобы хранить в нем размер блока в байтах или словах. Обычно такое число занимает столько же места, сколько и адрес, а размер слова обычно равен размеру адреса. На x86 в реальном режиме это не так, но это вообще довольно странный процессор.

Итак, мы добавляем к блоку два слова — одно перед ним, другое после него. В оба слова мы записываем размер блока. Получается своеобразный дескриптор, который окружает блок. При этом мы устанавливаем, что значения длин будут положительными, если блок свободен, и отрицательными, если блок занят. Можно сказать и наоборот, важно только потом соблюдать это соглашение (рис. 4.7).

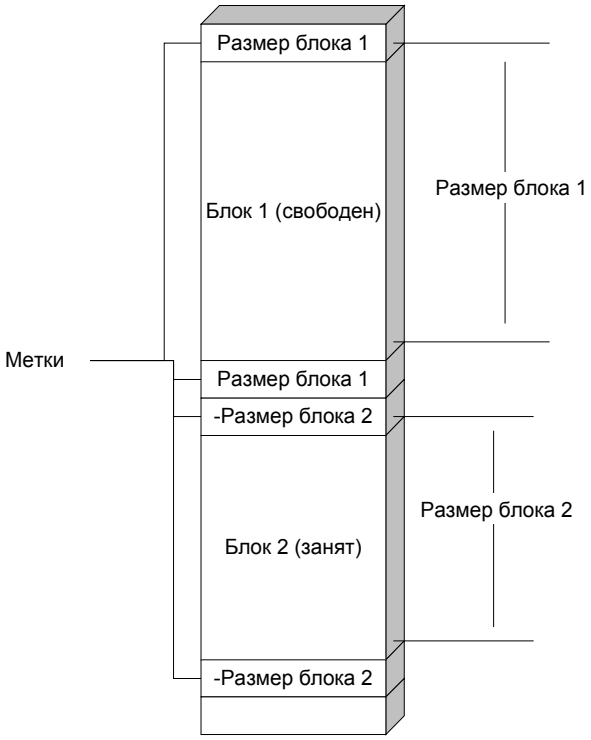

Рис. 4.7. Парные метки

Представим, что мы освобождаем блок с адресом `addr`. Считаем, что `addr` имеет тип `word *`, и при добавлении к нему целых чисел результирующий адрес будет отсчитываться в словах, как в языке C. Для того чтобы проверить, свободен ли сосед перед ним, мы должны посмотреть слово с адресом `addr-2`. Если оно отрицательно, то сосед занят, и мы должны оставить его в покое (рис. 4.8). Если же оно положительно, то мы можем легко определить адрес начала этого блока как `addr-addr[-2]`.

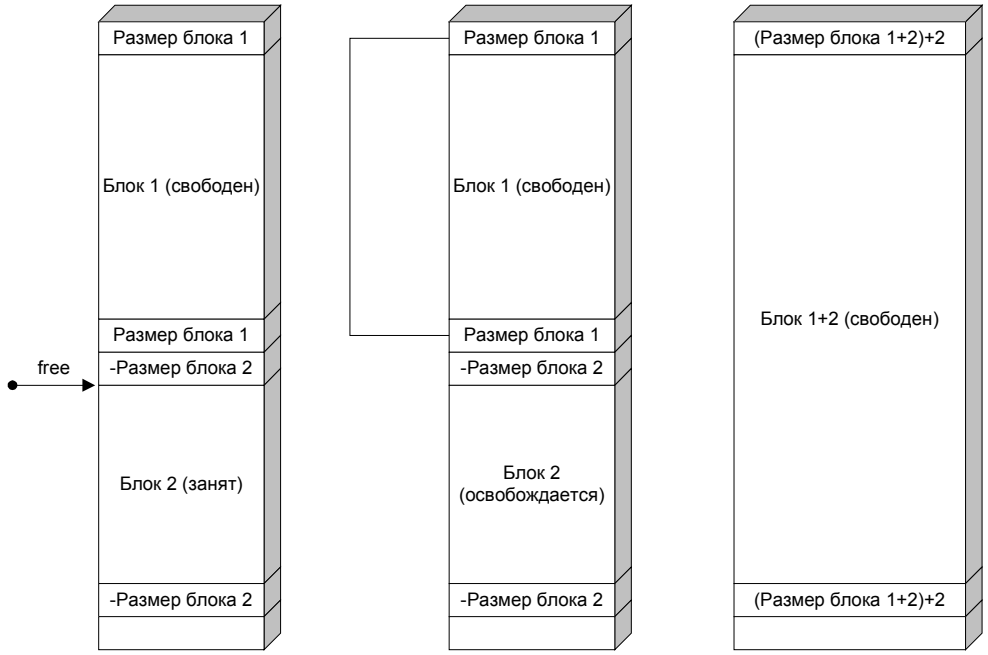

Рис. 4.8. Объединение с использованием парных меток

Определив адрес начала блока, мы можем легко объединить этот блок с блоком `addr`, нам нужно только сложить значения меток-дескрипторов и записать их в дескрипторы нового большого блока. Нам даже не нужно будет добавлять освобождаемый блок в список и извлекать оттуда его соседа!

Похожим образом присоединяется и сосед, стоящий после него. Единственное отличие состоит в том, что этого соседа все-таки нужно извлекать из списка свободных блоков.

Фактически, парные метки можно рассматривать как способ реализации решения, предложенного нами как одна из идей мозгового штурма: двунаправленного списка, включающего в себя как занятые, так и свободные блоки, и отсортированного по адресу. Дополнительное преимущество приведенного алгоритма состоит в том, что мы можем отслеживать такие ошибки, как многократное освобождение одного блока, запись в память за границей блока и иногда даже обращение к уже освобожденному блоку. Действительно, мы в любой момент можем проверить всю цепочку блоков памяти и убедиться в том, что все свободные блоки стоят в списке, что в нем стоят только свободные блоки, что сами цепочка и список не испорчены и т. д.

Примечание

Это действительно большое преимущество, т. к. оно значительно облегчает выявление ошибок работы с указателями, о которых в руководстве по Zortech C/C ++ сказано, что "опытные программисты, услышав это слово [ошибка работы с указателями — прим. авт.], бледнеют и прячутся под стол" ([Zortech v3.x]).

Итак, одним из лучших универсальных алгоритмов динамического распределения памяти является *алгоритм парных меток* с объединением свободных блоков в двунаправленный кольцевой список и поиском по принципу first fit. Этот алгоритм обеспечивает приемлемую производительность почти для всех стратегий распределения памяти, используемых в прикладных программах.

Алгоритм парных меток был предложен Дональдом Кнутом в начале 60-х годов XX века.

В третьем издании классической книги [Кнут, 2000] этот алгоритм приводится под названием "освобождения с дескрипторами границ". В современных системах используются и более сложные структуры дескрипторов, но всегда ставится задача обеспечить поиск соседей блока по адресному пространству за фиксированное время. В этом смысле практически все современные подпрограммы динамического выделения памяти (в частности, реализации стандартной библиотеки языка C) используют аналоги алгоритма парных меток или, точнее, отсортированные по адресу двунаправленные списки, в которые включены как свободные, так и занятые блоки. Сортировка по адресу обеспечивает, что блоки, размещенные в памяти рядом, оказываются соседями в этом списке, поэтому поиск соседей освобождаемого блока происходит за константное и, на практике, за очень малое время. Другие известные подходы либо просто хуже, чем этот, либо проявляют свои преимущества только в специальных случаях.

Реализация `malloc` в библиотеке GNU LibC

GNU LibC — это реализация стандартной библиотеки языка C в рамках freeware-проекта GNU Not Unix. Реализация функций `malloc/free` этой библиотеки (пример 4.3) использует смешанную стратегию: блоки размером более 4096 байт выделяются стратегией first fit из двусвязного кольцевого списка с использованием циклического просмотра, а освобождаются с помощью метода, который в указанном ранее смысле похож на алгоритм парных меток. Все выделяемые таким образом блоки будут иметь размер, кратный 4096 байтам.

На каждые 4096 байт пула памяти GNU LibC выделяет специальную запись-дескриптор. Эти записи объединяются в динамический массив `_heapinfo`; поскольку сам этот массив выделяется из той же области памяти, что и пул, управление этим массивом представляет собой довольно сложную задачу, иногда напоминающую вытаскивание себя за волосы из болота в духе барона Мюнхаузена; если читатель желает самостоятельно разобраться в том, как это происходит, я рекомендую самостоятельно проанализировать функцию `morecore` из примера 4.3 и комментарии к нему.

Размещение дескрипторов в `_heapinfo` в точности соответствует размещению соответствующих страниц в пуле. Пересчет указателя в пуле в индекс его дескриптора в примере 4.3 осуществляется макроопределением `BLOCK` по достаточно очевидному алгоритму. Понимание кода несколько затрудняется тем, что слово block в идентификаторах и комментариях используется в двух смыслах: для обозначения блоков размером 4096 байт и для обозначения непрерывных областей памяти, образованных такими блоками.

Элементом массива является структура `_malloc_info`, которая требует отдельного обсуждения. Для нефрагментированных блоков эта структура описывает область непрерывной памяти, к которой принадлежит блок: начало и размер этой области и ее статус (занята она или свободна). LibC поддерживает в актуальном состоянии описатели только для первого блока каждой непрерывной области памяти (свободной или занятой). Остальные дескрипторы области используются только как заполнители; впрочем, если область будет поделена на части, то первые из дескрипторов частей будут использоваться по назначению.

Головные дескрипторы всех областей связаны в два двунаправленных списка. В одном списке содержатся все области в пуле (и свободные, и занятые, и фрагментированные). Этот список отсортирован по адресу блока и полностью аналогичен списку, образованному парными метками в классическом алгоритме парных меток; он никогда не просматривается полностью, но используется для поиска соседей блока при его освобождении. Блоки вставляются в этот список при добавлении памяти функцией `morecore` или при делении крупного блока на части, а удаляются — при объединении нескольких соседних свободных блоков.

Второй список кольцевой, несортированный и содержит только свободные блоки. По этому списку при выделении памяти осуществляется поиск по принципу первого подходящего; выделяемые блоки исключаются из этого списка, но если при выделении оказалось необходимо отрезать от блока "хвост", то этот хвост возвращается в список в качестве самостоятельного блока.

Блоки меньшего размера объединяются в очереди с размерами, пропорциональными степеням двойки, как в описанном далее алгоритме близнецов. Элементы этих очередей называются фрагментами (рис. 4.9). В отличие от алгоритма близнецов, мы не объединяем при освобождении парные фрагменты. Вместо этого мы разбиваем 4-килобайтовый блок на фрагменты одинакового размера. Если, например, наша программа сделает запросы на 514 и 296 байт памяти, ей будут переданы фрагменты в 1024 и 512 байт соответственно. Под эти фрагменты будут выделены полные блоки в 4 Кбайта, и внутри них будет выделено по одному фрагменту. При последующих запросах на фрагменты такого же размера будут использоваться свободные фрагменты этих блоков. Пока хотя бы один фрагмент блока занят, весь блок считается занятым. Когда же освобождается последний фрагмент, блок возвращается в пул.

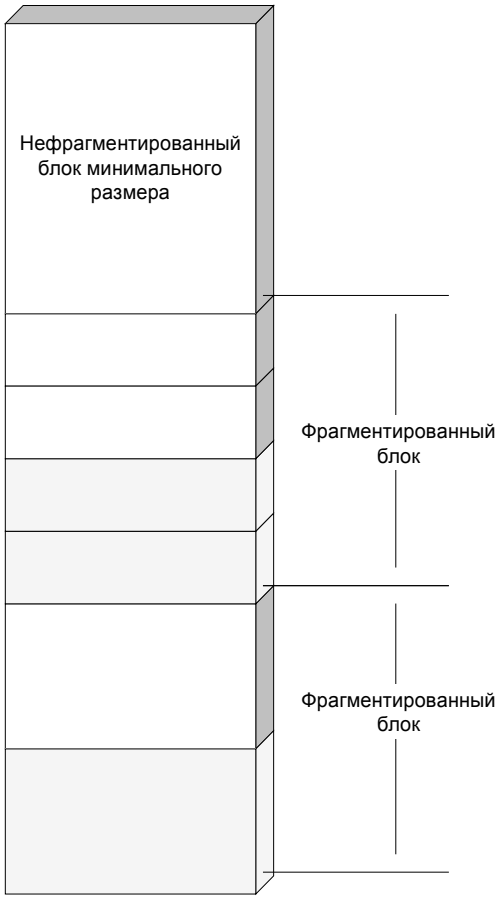

Рис. 4.9. Фрагменты в реализации `malloc` из GNU LibC

Описатели блоков хранятся не вместе с самими блоками, а в отдельном динамическом массиве `_heapinfo`. Описатель заводится не на непрерывную последовательность свободных байтов, а на каждые 4096 байт памяти (в примере 4.3 именно это значение принимает константа `BLOCKSIZE`). Благодаря этому мы можем вычислить индекс описателя в `_heapinfo`, просто разделив на 4096 смещение освобождаемого блока от начала пула.

Для нефрагментированных блоков описатель хранит состояние (занят-свободен) и размер непрерывного участка, к которому принадлежит блок. Благодаря этому, как и в алгоритме парных меток, мы легко можем найти соседей освобождаемого участка памяти и объединить их в большой непрерывный участок.

Для фрагментированных блоков описатель хранит размер фрагмента, счетчик занятых фрагментов и список свободных. Кроме того, все свободные фрагменты одного размера объединены в общий список — заголовки этих списков собраны в массив `_fraghead`.

Используемая структура данных занимает больше места, чем применяемая в классическом алгоритме парных меток, но сокращает объем списка свободных блоков и поэтому имеет более высокую производительность. Средний объем блока, выделяемого современными программами для ОС общего назначения, измеряется многими килобайтами, поэтому в большинстве случаев повышение накладных расходов памяти оказывается терпимо.

Пример 4.3. Реализация `malloc/free` **в GNU LibC. Функция** `__default_morecore` **приведена в примере 4.1**

```
malloc.c
/* Распределитель памяти 'malloc'.
   Copyright 1990, 1991, 1992 Free Software Foundation
                  Написана в мае 1989 Mike Haertel.

GNU C Library является свободным программным обеспечением;
вы можете передавать другим лицам и/или модифицировать ее в соответствии
с положениями GNU General Public License версии 2 или (по вашему выбору)
любой более поздней версии.

Библиотека GNU C распространяется в надежде, что она будет полезна, но
БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ; даже без неявно предполагаемых гарантий

КОММЕРЧЕСКОЙ ЦЕННОСТИ или ПРИГОДНОСТИ ДЛЯ КОНКРЕТНОЙ ЦЕЛИ.
Подробнее см. GNU General Public License.

Вы должны были получить копию GNU General Public License вместе с

GNU C Library; см. файл COPYING. Если вы ее не получили, напишите по
адресу: Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.


   С автором можно связаться по электронной почте по адресу
   mike@ai.mit.edu, или Mike Haertel c/o Free Software Foundation.  */

#ifndef        _MALLOC_INTERNAL
#define _MALLOC_INTERNAL
#include <malloc.h>
#endif

#ifdef __ELF__
```

```
#pragma weak malloc = __libc_malloc
#endif

/* Как действительно получить дополнительную память. */
__ptr_t (*__morecore) __P ((ptrdiff_t __size))
        = __default_morecore_init;

/* Предоставляемая пользователем отладочная функция (hook) для 'malloc'.
*/
void (*__malloc_initialize_hook) __P ((void));
__ptr_t (*__malloc_hook) __P ((size_t __size));

/* Указатель на основание первого блока. */
char *_heapbase;

/* Таблица информационных записей о блоках.  Размещается

через align/__free (не malloc/free).  */
malloc_info *_heapinfo;

/* Количество информационных записей. */
static size_t heapsize;

/* Индекс поиска в информационной таблице.  */
size_t _heapindex;

/* Ограничитель допустимых индексов в информационной таблице */
size_t _heaplimit;

#if 1        /* Adapted from Mike */
/* Счетчик больших блоков, размещенных для каждого из размеров
фрагментов. */
int _fragblocks[BLOCKLOG];
#endif

/* Списки свободных фрагментов по размерам.  */
struct list _fraghead[BLOCKLOG];

/* Инструментальные переменные.  */
size_t _chunks_used;
size_t _bytes_used;
size_t _chunks_free;
```

```
size_t _bytes_free;

/* Имеем ли мы опыт? */
int __malloc_initialized;

/* Выровненное размещение.  */
static __ptr_t align __P ((size_t));
static __ptr_t
align (size)
     size_t size;
{
  __ptr_t result;
  unsigned long int adj;

  result = (*__morecore) (size);
  adj = (unsigned long int) ((unsigned long int) ((char *) result -
                                                  (char *) NULL)) % BLOCKSIZE;
  if (adj != 0)
    {
      adj = BLOCKSIZE - adj;
      (void) (*__morecore) (adj);
      result = (char *) result + adj;
    }
  return result;
}

/* Настроить все и запомнить, что у нас есть. */
static int initialize __P ((void));
static int
initialize ()
{
  if (__malloc_initialize_hook)
     (*__malloc_initialize_hook) ();

  heapsize = HEAP / BLOCKSIZE;
  _heapinfo = (malloc_info *) align (heapsize * sizeof (malloc_info));
  if (_heapinfo == NULL)
    return 0;
  memset (_heapinfo, 0, heapsize * sizeof (malloc_info));
  _heapinfo[0].free.size = 0;
```

```
  _heapinfo[0].free.next = _heapinfo[0].free.prev = 0;
  _heapindex = 0;
  _heapbase = (char *) _heapinfo;
  __malloc_initialized = 1;
  return 1;
}

/* Получить выровненную память, инициализируя
   или наращивая таблицу описателей кучи по мере необходимости. */
static __ptr_t morecore __P ((size_t));
static __ptr_t
morecore (size)
     size_t size;
{
  __ptr_t result;
  malloc_info *newinfo, *oldinfo;
  size_t newsize;

  result = align (size);
  if (result == NULL)
    return NULL;

  /* Проверить, нужно ли нам увеличить таблицу описателей. */
  if ((size_t) BLOCK ((char *) result + size) > heapsize)
    {
      newsize = heapsize;
      while ((size_t) BLOCK ((char *) result + size) > newsize)
        newsize *= 2;
      newinfo = (malloc_info *) align (newsize * sizeof (malloc_info));
      if (newinfo == NULL)
        {
          (*__morecore) (-size);
          return NULL;
        }
      memset (newinfo, 0, newsize * sizeof (malloc_info));
      memcpy (newinfo, _heapinfo, heapsize * sizeof (malloc_info));
      oldinfo = _heapinfo;
      newinfo[BLOCK (oldinfo)].busy.type = 0;
      newinfo[BLOCK (oldinfo)].busy.info.size
        = BLOCKIFY (heapsize * sizeof (malloc_info));
```

```
      _heapinfo = newinfo;
      _free_internal (oldinfo);
      heapsize = newsize;
    }

  _heaplimit = BLOCK ((char *) result + size);
  return result;
}

/* Выделить память из кучи. */
__ptr_t
__libc_malloc (size)
     size_t size;
{
  __ptr_t result;
  size_t block, blocks, lastblocks, start;
  register size_t i;
  struct list *next;

  /* Некоторые программы вызывают malloc (0). Мы это допускаем. */
#if 0
  if (size == 0)
    return NULL;
#endif

  if (!__malloc_initialized)
    if (!initialize ())
      return NULL;

  if (__malloc_hook != NULL)
    return (*__malloc_hook) (size);

  if (size < sizeof (struct list))
      size = sizeof (struct list);

  /* Определить политику размещения на основании размера запроса. */
  if (size <= BLOCKSIZE / 2)
    {
      /* Маленькие запросы получают фрагмент блока.
         Определяем двоичный логарифм размера фрагмента. */
```

```
      register size_t log = 1;
      --size;
      while ((size /= 2) != 0)
        ++log;

      /* Просмотреть списки фрагментов на предмет свободного
         фрагмента желаемого размера. */
      next = _fraghead[log].next;
      if (next != NULL)
        {
          /* Найдены свободные фрагменты этого размера.
             Вытолкнуть фрагмент из списка фрагментов и вернуть его.
             Обновить счетчики блока nfree и first. */
          result = (__ptr_t) next;
          next->prev->next = next->next;
          if (next->next != NULL)
            next->next->prev = next->prev;
          block = BLOCK (result);
          if (--_heapinfo[block].busy.info.frag.nfree != 0)
            _heapinfo[block].busy.info.frag.first = (unsigned long int)
              ((unsigned long int) ((char *) next->next — (char *) NULL)
               % BLOCKSIZE) >> log;

          /* Обновить статистику.  */
          ++_chunks_used;
          _bytes_used += 1 << log;
          --_chunks_free;
          _bytes_free -= 1 << log;
        }
      else
        {
          /* Нет свободных фрагментов желаемого размера.  Следует взять
             новый блок, поделить его на фрагменты и вернуть первый. */
          result = __libc_malloc (BLOCKSIZE);
          if (result == NULL)
            return NULL;
#if 1       /* Adapted from Mike */
          ++_fragblocks[log];
#endif
          /* Связать все фрагменты, кроме первого, в список свободных. */
```

```
          for (i = 1; i < (size_t) (BLOCKSIZE >> log); ++i)
            {
              next = (struct list *) ((char *) result + (i << log));
              next->next = _fraghead[log].next;
              next->prev = &_fraghead[log];
              next->prev->next = next;
              if (next->next != NULL)
                next->next->prev = next;
            }

          /* Инициализировать счетчики nfree и first для этого блока. */
          block = BLOCK (result);
          _heapinfo[block].busy.type = log;
          _heapinfo[block].busy.info.frag.nfree = i - 1;
          _heapinfo[block].busy.info.frag.first = i - 1;

          _chunks_free += (BLOCKSIZE >> log) - 1;
          _bytes_free += BLOCKSIZE - (1 << log);
          _bytes_used -= BLOCKSIZE - (1 << log);
        }
    }
  else
    {
      /* Большие запросы получают один или больше блоков.
         Просмотреть список свободных циклически, начиная с точки, где мы
         были в последний раз.  Если мы пройдем полный круг, не обнаружив
         достаточно большого блока, мы должны будем запросить еще память
         у системы. */
      blocks = BLOCKIFY (size);
      start = block = _heapindex;
      while (_heapinfo[block].free.size < blocks)
        {
          block = _heapinfo[block].free.next;
          if (block == start)
            {
              /* Необходимо взять больше [памяти] у системы.
                 Проверить, не будет ли новая память продолжением
                 последнего свободного блока; если это так, нам не
                 надо будет запрашивать так много.  */
              block = _heapinfo[0].free.prev;
```

```
              lastblocks = _heapinfo[block].free.size;
              if (_heaplimit != 0 && block + lastblocks == _heaplimit &&
                  (*__morecore) (0) == ADDRESS (block + lastblocks) &&
                  (morecore ((blocks — lastblocks) * BLOCKSIZE)) != NULL)
                {
#if 1         /* Adapted from Mike */
                  /* Обратите внимание, что morecore() может изменить
                     положение последнего блока, если она перемещает
                     таблицу дескрипторов и старая копия таблицы
                     сливается с последним блоком. */
                  block = _heapinfo[0].free.prev;
                  _heapinfo[block].free.size += blocks — lastblocks;
#else
                  _heapinfo[block].free.size = blocks;
#endif
                  _bytes_free += (blocks — lastblocks) * BLOCKSIZE;
                  continue;
                }
              result = morecore (blocks * BLOCKSIZE);
              if (result == NULL)
                return NULL;
              block = BLOCK (result);
              _heapinfo[block].busy.type = 0;
              _heapinfo[block].busy.info.size = blocks;
              ++_chunks_used;
              _bytes_used += blocks * BLOCKSIZE;
              return result;
            }
        }

      /* В этой точке мы [так или иначе] нашли подходящую запись
         в списке свободных.  Понять, как удалить то, что нам нужно,
         из списка. */
      result = ADDRESS (block);
      if (_heapinfo[block].free.size > blocks)
        {
          /* Блок, который мы нашли, имеет небольшой остаток,
             так что присоединить его задний конец к списку свободных. */
          _heapinfo[block + blocks].free.size
            = _heapinfo[block].free.size — blocks;
```

```
          _heapinfo[block + blocks].free.next
            = _heapinfo[block].free.next;
          _heapinfo[block + blocks].free.prev
            = _heapinfo[block].free.prev;
          _heapinfo[_heapinfo[block].free.prev].free.next
            = _heapinfo[_heapinfo[block].free.next].free.prev
            = _heapindex = block + blocks;
        }
      else
        {
          /* Блок точно соответствует нашему запросу, поэтому
             просто удалить его из списка. */
          _heapinfo[_heapinfo[block].free.next].free.prev
            = _heapinfo[block].free.prev;
          _heapinfo[_heapinfo[block].free.prev].free.next
            = _heapindex = _heapinfo[block].free.next;
          --_chunks_free;
        }

      _heapinfo[block].busy.type = 0;
      _heapinfo[block].busy.info.size = blocks;
      ++_chunks_used;
      _bytes_used += blocks * BLOCKSIZE;
      _bytes_free -= blocks * BLOCKSIZE;
    }

  return result;
}
```

free.c:

```
/* Освободить блок памяти, выделенный 'malloc'.
   Copyright 1990, 1991, 1992 Free Software Foundation
                  Написано в мае 1989 Mike Haertel.

GNU C Library является свободным программным обеспечением;
вы можете перераспространять ее и/или модифицировать ее в соответствии
с положениями GNU General Public License версии 2 или (по вашему выбору)
любой более поздней версии.

Библиотека GNU C распространяется в надежде, что она будет полезна, но
```

```
БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ; даже без неявно предполагаемых гарантий

КОММЕРЧЕСКОЙ ЦЕННОСТИ или ПРИГОДНОСТИ ДЛЯ КОНКРЕТНОЙ ЦЕЛИ.
Подробнее см. GNU General Public License.

   С автором можно связаться по электронной почте по адресу
   mike@ai.mit.edu, или Mike Haertel c/o Free Software Foundation.   */

#ifndef        _MALLOC_INTERNAL
#define _MALLOC_INTERNAL
#include <malloc.h>
#endif

#ifdef __ELF__
#pragma weak free = __libc_free
#endif

/*Предоставляемая пользователем отладочная функция (hook) для 'free'. */
void (*__free_hook) __P ((__ptr_t __ptr));

/* Список блоков, выделенных memalign. */
struct alignlist *_aligned_blocks = NULL;

/* Вернуть память в кучу.  Аналогична 'free', но не вызывает
   __free_hook, даже если он определен. */
void
_free_internal (ptr)
     __ptr_t ptr;
{
  int type;
  size_t block, blocks;
  register size_t i;
  struct list *prev, *next;

  block = BLOCK (ptr);

  type = _heapinfo[block].busy.type;
  switch (type)
    {
    case 0:
```

```
/* Собрать как можно больше статистики как можно раньше. */
--_chunks_used;
_bytes_used -= _heapinfo[block].busy.info.size * BLOCKSIZE;
_bytes_free += _heapinfo[block].busy.info.size * BLOCKSIZE;

/* Найти свободный кластер, предшествующий этому в списке
   свободных.
   Начать поиск с последнего блока, к которому было обращение.
   Это может быть преимуществом для программ, в которых выделение
   локальное. */
i = _heapindex;
if (i > block)
  while (i > block)
    i = _heapinfo[i].free.prev;
else
  {
    do
      i = _heapinfo[i].free.next;
    while (i > 0 && i < block);
    i = _heapinfo[i].free.prev;
  }

/* Определить, как включить этот блок в список свободных. */
if (block == i + _heapinfo[i].free.size)
  {
    /* Слить этот блок с его предшественником. */
    _heapinfo[i].free.size += _heapinfo[block].busy.info.size;
    block = i;
  }
else
  {
    /* Действительно включить этот блок в список свободных. */
    _heapinfo[block].free.size = _heapinfo[block].busy.info.size;
    _heapinfo[block].free.next = _heapinfo[i].free.next;
    _heapinfo[block].free.prev = i;
    _heapinfo[i].free.next = block;
    _heapinfo[_heapinfo[block].free.next].free.prev = block;
    ++_chunks_free;
  }

/* Теперь, поскольку блок включен, проверить, не можем ли мы
```

```
         слить его с его последователем (исключая его последователя из
         списка и складывая размеры). */

      if (block + _heapinfo[block].free.size ==
_heapinfo[block].free.next)
        {
          _heapinfo[block].free.size
            += _heapinfo[_heapinfo[block].free.next].free.size;
          _heapinfo[block].free.next
            = _heapinfo[_heapinfo[block].free.next].free.next;
          _heapinfo[_heapinfo[block].free.next].free.prev = block;
          --_chunks_free;
        }

      /* Проверить, не можем ли мы вернуть память системе. */

      blocks = _heapinfo[block].free.size;
      if (blocks >= FINAL_FREE_BLOCKS && block + blocks == _heaplimit
          && (*__morecore) (0) == ADDRESS (block + blocks))
        {
          register size_t bytes = blocks * BLOCKSIZE;
          _heaplimit -= blocks;
          (*__morecore) (-bytes);
          _heapinfo[_heapinfo[block].free.prev].free.next
            = _heapinfo[block].free.next;
          _heapinfo[_heapinfo[block].free.next].free.prev
            = _heapinfo[block].free.prev;
          block = _heapinfo[block].free.prev;
          --_chunks_free;
          _bytes_free -= bytes;
        }

      /* Установить следующий поиск, стартовать с этого блока. */

      _heapindex = block;
      break;

    default:
      /* Собрать некоторую статистику. */
      --_chunks_used;
      _bytes_used -= 1 << type;
```

```
      ++_chunks_free;
      _bytes_free += 1 << type;

      /* Получить адрес первого свободного фрагмента в этом блоке. */
      prev = (struct list *) ((char *) ADDRESS (block) +
                              (_heapinfo[block].busy.info.frag.first << type));

#if 1        /* Adapted from Mike */
     if (_heapinfo[block].busy.info.frag.nfree == (BLOCKSIZE >> type) - 1
       && _fragblocks[type] > 1)
#else
    if (_heapinfo[block].busy.info.frag.nfree == (BLOCKSIZE >> type) - 1)
#endif
        {
          /* Если все фрагменты этого блока свободны, удалить их из
             списка фрагментов и освободить полный блок. */
#if 1        /* Adapted from Mike */
          --_fragblocks[type];
#endif
          next = prev;
          for (i = 1; i < (size_t) (BLOCKSIZE >> type); ++i)
            next = next->next;
          prev->prev->next = next;
          if (next != NULL)
            next->prev = prev->prev;
          _heapinfo[block].busy.type = 0;
          _heapinfo[block].busy.info.size = 1;

          /* Следим за точностью статистики. */
          ++_chunks_used;
          _bytes_used += BLOCKSIZE;
          _chunks_free -= BLOCKSIZE >> type;
          _bytes_free -= BLOCKSIZE;

          __libc_free (ADDRESS (block));
        }
      else if (_heapinfo[block].busy.info.frag.nfree != 0)
        {
          /* Если некоторые фрагменты этого блока свободны, включить
             этот фрагмент в список фрагментов после первого свободного
             фрагмента этого блока. */
```

```
          next = (struct list *) ptr;
          next->next = prev->next;
          next->prev = prev;
          prev->next = next;
          if (next->next != NULL)
            next->next->prev = next;
          ++_heapinfo[block].busy.info.frag.nfree;
        }
      else
        {
          /* В этом блоке нет свободных фрагментов, поэтому включить
             этот фрагмент в список фрагментов и анонсировать, что это
             первый свободный фрагмент в этом блоке. */
          prev = (struct list *) ptr;
          _heapinfo[block].busy.info.frag.nfree = 1;
          _heapinfo[block].busy.info.frag.first = (unsigned long int)
            ((unsigned long int) ((char *) ptr — (char *) NULL)
             % BLOCKSIZE >> type);
          prev->next = _fraghead[type].next;
          prev->prev = &_fraghead[type];
          prev->prev->next = prev;
          if (prev->next != NULL)
            prev->next->prev = prev;
        }
      break;
    }
}

/* Вернуть память в кучу. */
void
__libc_free (ptr)
     __ptr_t ptr;
{
  register struct alignlist *l;

  if (ptr == NULL)
    return;

  for (l = _aligned_blocks; l != NULL; l = l->next)
    if (l->aligned == ptr)
```

```
      {
        l->aligned = NULL;
        /* Пометить элемент списка как свободный. */
        ptr = l->exact;
        break;
      }

  if (__free_hook != NULL)
    (*__free_hook) (ptr);
  else
    _free_internal (ptr);
}

#include <gnu-stabs.h>
#ifdef elf_alias
elf_alias (free, cfree);
#endif
```

К основным недостаткам алгоритма first fit относится невозможность оценки времени поиска подходящего блока, что делает его неприемлемым для задач реального времени. Для этих задач требуется алгоритм, который способен за фиксированное (желательно, небольшое) время либо найти подходящий блок памяти, либо дать обоснованный ответ о том, что подходящего блока не существует.

Проще всего решить эту задачу, если нам требуются блоки нескольких фиксированных размеров (рис. 4.10). Мы объединяем блоки каждого размера в свой список. Если в списке блоков требуемого размера ничего нет, мы смотрим в список блоков большего размера. Если там что-то есть, мы разрезаем этот блок на части, одну отдаем запрашивающей программе, а вторую... Правда, если размеры требуемых блоков не кратны друг другу, что мы будем делать с остатком?

Для решения этой проблемы нам необходимо ввести какое-либо ограничение на размеры выделяемых блоков. Например, можно потребовать, чтобы эти размеры равнялись числам Фибоначчи (последовательность целых чисел, в которой $F_{i+1} = F_i + F_{i-1}$). В этом случае, если нам нужно F_i байт, а в наличии есть только блок размера F_{i+1}, мы легко можем получить два блока — один требуемого размера, а другой — F_{i-1}, который тоже не пропадет. Да, любое ограничение на размер приведет к внутренней фрагментации, но так ли велика эта плата за гарантированное время поиска блока?

На практике числа Фибоначчи не используются. Одной из причин, по-видимому, является относительная сложность вычисления такого F_i, которое не меньше требуемого размера блока. Другая причина — сложность объединения свободных блоков со смежными адресами в блок большего размера. Зато широкое применение нашел алгоритм, который ограничивает последовательные размеры блоков более простой зависимостью — степенями числа 2: 512 байт, 1 Кбайт, 2 Кбайт и т. д. Такая стратегия называется *алгоритмом близнецов* (рис. 4.11).

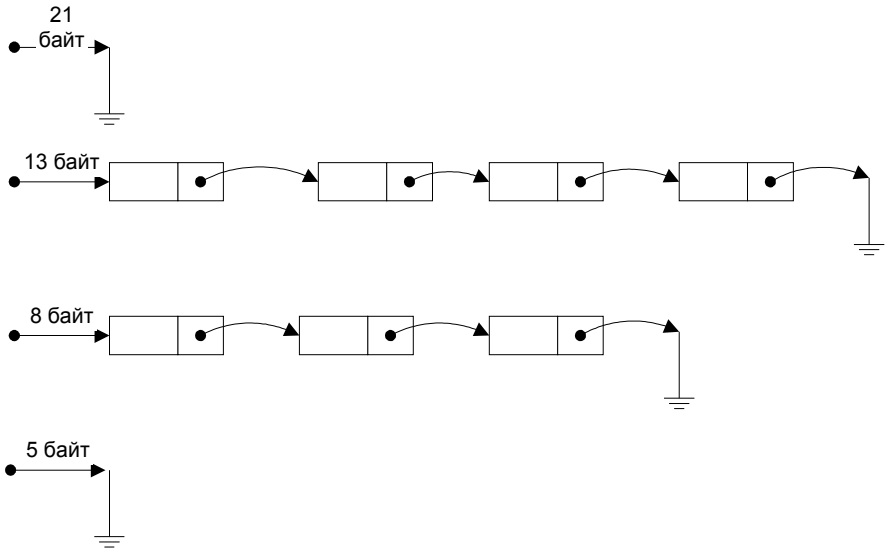

Рис. 4.10. Выделение блоков фиксированных размеров

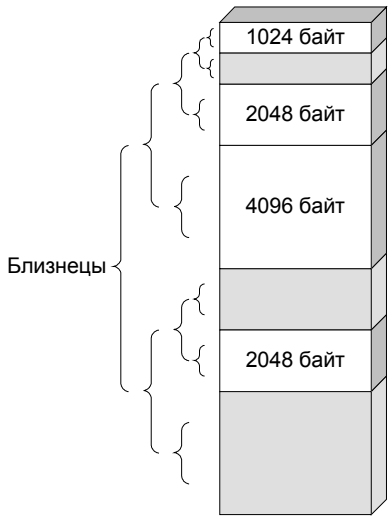

Рис. 4.11. Алгоритм близнецов

Одно из преимуществ этого метода состоит в простоте объединения блоков при их освобождении. Адрес блока-близнеца получается простым инвертированием соответствующего бита в адресе нашего блока. Нужно только проверить, свободен ли этот близнец. Если он свободен, то мы объединяем братьев в блок вдвое большего размера и т. д. Даже в наихудшем случае время поиска не превышает O ($\log(S_{max}) - \log(S_{min})$), где S_{max} и S_{min} обозначают, соответственно, максимальный и минимальный размеры используемых блоков. Это делает алгоритм близнецов трудно заменимым для ситуаций, в которых необходимо гарантированное время реакции — например, для задач реального времени. Часто этот алгоритм или его варианты используются для выделения памяти внутри ядра ОС. Например, именно таким способом выделяет память функция `kmalloc` в ядре Linux.

Существуют и более сложные варианты применения описанного ранее подхода. Например, пул свободной памяти Novell Netware состоит из 4 очередей с шагом 16 байт (для блоков размерами 16, 32, 48, 64 байта), 3 очередей с шагом 64 байта (для блоков размерами 128, 192, 256 байт) и 15 очередей с шагом 256 байт (от 512 байт до 4 Кбайт). При запросах большего размера выделяется целиком страница. Любопытно, что возможности работы в режиме реального времени, присущие этой изощренной стратегии, в Netware практически не используются.

Например, если драйвер сетевого интерфейса при получении очередного пакета данных обнаруживает, что у него нет свободных буферов для его приема, он не пытается выделить новый буфер стандартным алгоритмом. Вместо

этого, драйвер просто игнорирует пришедшие данные, лишь увеличивая счетчик потерянных пакетов. Отдельный системный процесс следит за состоянием этого счетчика и только при превышении им некоторого порога за некоторый интервал времени выделяет драйверу новый буфер.

Подобный подход к пользовательским данным может показаться циничным, но надо вспомнить, что при передаче данных по сети возможны и другие причины потери пакетов, например порча данных из-за электромагнитных помех. Поэтому все сетевые протоколы высокого уровня предусматривают средства пересылки пакетов в случае их потери, какими бы причинами эта потеря ни была вызвана. С другой стороны, в системах реального времени игнорирование данных, которые мы все равно не в состоянии принять и обработать, — довольно часто используемая, хотя и не всегда приемлемая стратегия.

Существует еще один способ обеспечить перераспределение памяти между пулами для блоков разного размера — так называемые слабовые аллокаторы [Bonwick 1994]. При их использовании пул разбивают на большие блоки, называемые слабами (slab — англ. брусок, плита), каждый из которых содержит некоторое количество записей требуемого размера. Когда программа запрашивает память, аллокатор пытается найти свободную запись в уже частично занятых слабах; если это невозможно, выделяются новые слабы. Допускается возможность возврата полностью свободных слабов системе, в том числе и их повторное использование для слабов других типов.

Такая стратегия особенно эффективна в условиях, когда программа создает большое число объектов небольшого размера, и при этом размер объекта не кратен степени двойки. Типичными примерами являются буферы под сетевые пакеты (у Ethernet максимальный размер пакета составляет 1500 байт) и другие структуры данных ядра — дескрипторы открытых файлов, дескрипторы процессов и т. д.

Слабовый аллокатор в ядре Linux

Слабы появились в ядре Linux 2.4, несомненно, под впечатлением от результатов, которые дал переход к этой схеме управления памятью в ядре Solaris. Слабовый аллокатор управляет памятью с помощью именованных кэшей (cache), каждый из которых представляет пул блоков памяти определенного типа. Пул создается функцией

```
kmem_cache_t * kmem_cache_create (const char * name, size_t size,
size_t offset, unsigned long flags, void (*ctor) (void*,
kmem_cache_t *, unsigned long), void (*dtor) (void*, kmem_cache_t
*, unsigned long));
```

Смысл большинства параметров очевиден из их названий:

`name` — имя кэша;

`size` — размер элемента в байтах;

`offset` — смещение слаба относительно границы страницы;

`flags` — дополнительные параметры (флаги);

`ctor` и `dtor` — указатели на функции — конструктор и деструктор, которые надо вызвать для каждого элемента слаба. Они вызываются не в момент выделения памяти, а в момент разметки слаба (`ctor`) и в момент возврата памяти из-под свободного слаба системе (`dtor`).

Для выделения памяти из кэша используется функция:

```
void * kmem_cache_alloc (kmem_cache_t * cachep, int flags);
```

Параметр `flags` представляет собой комбинацию нескольких битовых полей, которые позволяют задать приоритет обращения к памяти. Основные уровни приоритета — это `GFP_BUFFER`, `GFP_KERNEL` и `GFP_ATOMIC`. Наиболее высокий уровень приоритета — это `GFP_ATOMIC`, при котором аллокатор возвращает только блоки из уже размеченных слабов и не пытается увеличивать размер кэша. Запросы к памяти с этим приоритетом с довольно высокой вероятностью завершаются неудачей (хотя система стремится поддерживать определенное количество памяти, доступной для таких запросов), но завершаются за фиксированное время и, благодаря этому, могут вызываться из обработчиков прерываний.

Кэш представляет собой список слабов, каждый из которых разбит на элементарные блоки указанного размера. В зависимости от размера блока, используются две схемы разметки: on-slab (когда метаинформация о том, какие блоки в слабе свободны, а какие — нет, хранится в самом слабе) и off-slab, когда эта информация хранится в отдельных областях памяти (рис. 4.12).

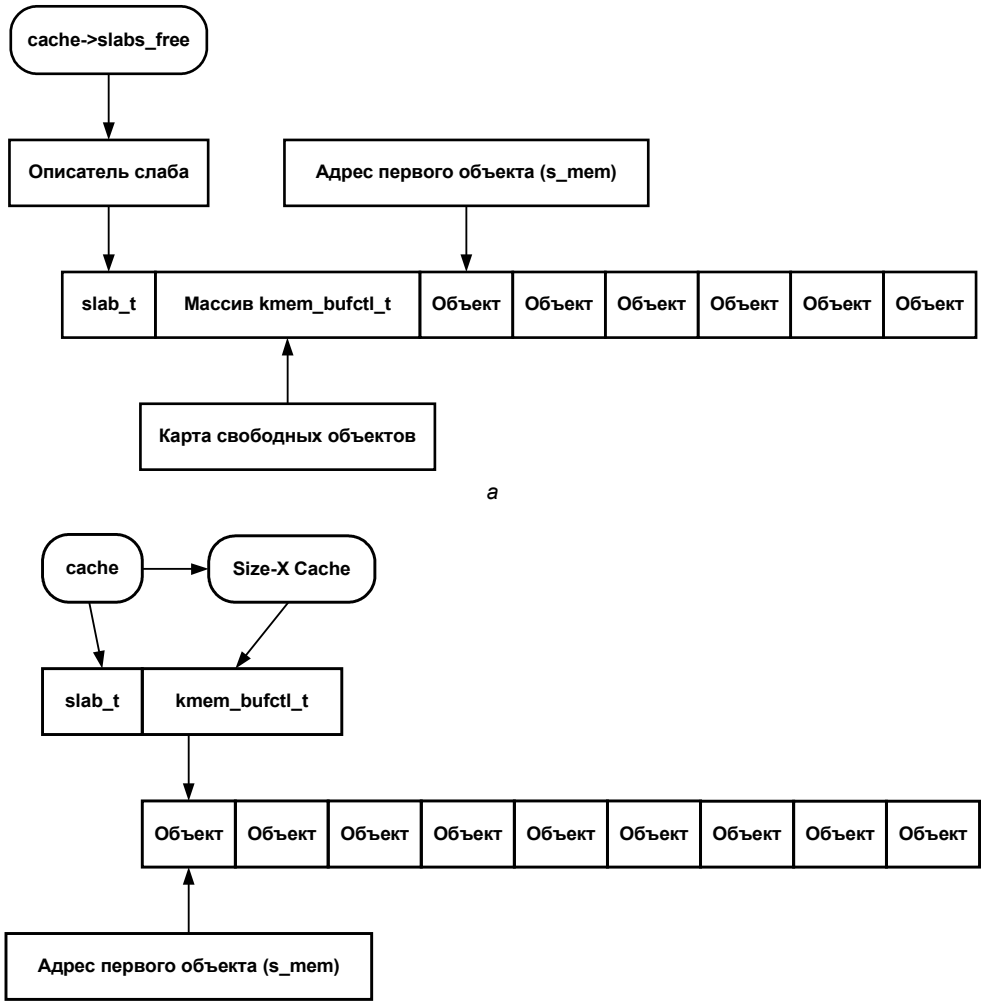

Рис. 4.12. On-slab (*а*) и off-slab (*б*) метаданные в ядре Linux 2.4

Аллокатор поддерживает три списка слабов: полностью занятые, частично свободные и полностью свободные, но размеченные. При запросе на создание объекта аллокатор всегда сначала проверяет доступность частично занятых слабов.

При нехватке свободных блоков в размеченных слабах менеджер кэша размещает новые слабы. Обычно это происходит при запросах с приоритетами `GFP_BUFFER` и `GFP_KERNEL`. При нехватке памяти ядро может потребовать у менеджера памяти сжатия кэша, при котором кэш возвращает системе полностью свободные слабы.

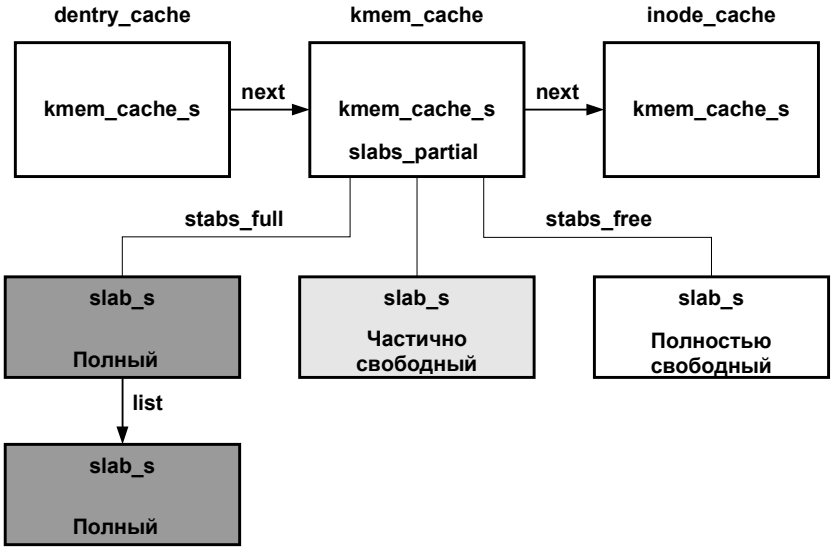

Рис. 4.13. Списки слабов в ядре Linux

4.3. Сборка мусора

— Ладно, — смирился Герера. — Эй, Стелмэн, лучше по-прежнему держись за мое плечо. Не стоит нам разделяться.
— А я и так держусь, — отозвался Стелмэн.
— Слушай, кажется, мне лучше знать, держится кто-нибудь за мое плечо или нет.
— Это твое плечо, Пакстон?
— Нет, — ответил Пакстон.
— Плохо, — сказал Стелмэн очень медленно. — Это совсем плохо.
— Почему?
— Потому что я определенно держусь за чье-то плечо.

Р. Шекли

Явное освобождение динамически выделенной памяти применяется во многих системах программирования и потому привычно для большинства программистов, но оно имеет серьезный недостаток: если программист по какой-то причине не освобождает выделенные блоки, память будет теряться. Эта ошибка, называемая *утечкой памяти* (memory leak), особенно неприятна в программах, которые длительное время работают без перезапуска, — подсистемах ядра ОС, постоянно запущенных сервисах или серверных приложениях. Дополнительная неприятность состоит в том, что медленные утечки могут привести к заметным потерям памяти лишь при многодневной или даже

многомесячной непрерывной эксплуатации системы, поэтому их сложно обнаружить при тестировании.

Напротив, чрезмерное увлечение программиста освобождением объектов может привести к еще более серьезной проблеме — *висячим ссылкам* (dandgling reference). Висячие ссылки возникают, когда мы уничтожаем объект, на который где-то в другом месте сохранена ссылка или — из другой перспективы — когда мы сохраняем ссылки на объект, который уже уничтожен.

Поиск висячих ссылок существенно затрудняется тем фактом, что далеко не всегда память из-под объекта переиспользуется сразу или вскоре после его уничтожения. Поэтому некоторое время чтения по висячей ссылке могут возвращать осмысленные значения, а модификации — проходить без катастрофических последствий. Однако рано или поздно в освобожденной памяти будет создан другой объект. Если старый или новый объект сами содержали указатели на другие объекты, модификации данных по висячей ссылке могут привести к цепному разрушению значительной части или даже всех структур данных, размещенных в памяти нашего процесса. В системах с открытой памятью это обычно приводит к разрушению структур данных, кода других задач и самой ОС, а также к "зависанию" системы. Даже если используется защита памяти, как правило, эта защита срабатывает существенно позже момента, в который была совершена ошибка, поэтому по посмертному дампу программы не всегда можно определить, в какой момент все началось и что вообще произошло.

Главная трудность состоит в индетерминизме — то, насколько рано память будет переиспользована, решающим образом зависит от сценария исполнения программы. Поэтому такие ошибки очень сложно, зачастую невозможно найти при отладке и тестировании программы. Даже когда на такую ошибку натыкается заказчик программы при ее эксплуатации, разработчик далеко не всегда может воспроизвести условия ее возникновения.

Ряд систем программирования предоставляют средства для борьбы с висячими ссылками. Так, отладочная версия стандартной библиотеки Microsoft Visual C предоставляет специальные реализации функций `malloc/free`. При освобождении памяти функция `free` прописывает освобождаемый блок специальным битовым паттерном `0xDD`. В других ситуациях неиспользуемая память прописывается другими паттернами, которые перечислены в табл. 4.1

***Таблица 4.1.** Битовые паттерны, используемые отладочной библиотекой MS VC*

Значение	Расшифровка	Описание
`0xCD`	`Clean Memory`	Память, выделенная `malloc/new`, еще не модифицированная приложением

0xDD	Dead Memory	Память, освобожденная с помощью delete/free. Это может использоваться для обнаружения записи по висячему указателю

Таблица 4.1 *(окончание)*

Значение	Расшифровка	Описание
0xFD	Fence Memory	Известно также как "ничейная земля" ("no mans land"). Используется для ограничения выделенных блоков памяти, как ограждение. Может использоваться для обнаружения выхода индекса за границы массива
0xAB	(Allocated Block?)	Память, выделенная LocalAlloc()
0xBAADF00D	Bad Food	Память, выделенная LocalAlloc() с флагом LMEM_FIXED
0xCC		В коде, компилированном с ключом /GZ, неинициализированные переменные прописываются на байтовом уровне этим шаблоном.

При выделении памяти malloc проверяет целостность паттерна во вновь выделяемом блоке. Разумеется, это сильно снижает производительность, но благодаря этому многие ошибки, связанные с висячими и неинициализированными указателями, можно обнаружить гораздо раньше. Впрочем, все равно обнаружение происходит не непосредственно в момент ошибки, поэтому рассматривать такие средства как панацею нельзя.

Некоторые системы программирования используют специальный метод освобождения динамической памяти, называемый *сборкой мусора* (garbage collection). Этот метод состоит в том, что ненужные блоки памяти не освобождаются явным образом. Вместо этого используется некоторый более или менее изощренный алгоритм, следящий за тем, какие блоки еще нужны, а какие — уже нет.

Это приводит к значительному снижению стоимости разработки — по моему опыту, время разработки приблизительно одинаковых по функциональности программ на C/C++ (языки программирования с явным освобождением памяти) и на Java (язык с неявным освобождением памяти) различается в 2-3 раза. Причем, поскольку Java использует синтаксис, во многом похожий на C/C++, можно не сомневаться, что дело здесь именно в способе управления памятью, а не в синтаксисе. В литературе и на форумах мне встречались оценки, из которых следует, что разница может достигать и 4-5 раз. Разумеется, это сильно зависит от разработчика и от решаемой задачи, но, так или иначе, разница

впечатляющая. К сожалению, как мы увидим далее в этом разделе, это преимущество достигается вовсе не бесплатно.

Все методы сборки мусора так или иначе сводятся к поддержанию базы данных о том, какие объекты на кого ссылаются. Поэтому применять сборку мусора возможно практически только в интерпретируемых языках, таких, как Java, Smalltalk или Lisp, где с каждой операцией можно ассоциировать неограниченно большое количество действий. Впрочем, в некоторых компилируемых языках, таких, как C++, в которых можно переопределять операции присваивания и связывать определенный код с операциями создания и уничтожения объектов, при желании также можно реализовать сборку мусора для некоторых классов объектов.

4.3.1. Подсчет ссылок

Самый простой метод отличать используемые блоки от ненужных — считать, что блок, на который есть ссылка, нужен, а блок, на который ни одной ссылки не осталось, — не нужен. Для этого к каждому блоку присоединяют дескриптор, в котором подсчитывают количество ссылок на него. Каждая передача указателя на этот блок приводит к увеличению счетчика ссылок на 1, а каждое уничтожение объекта, содержавшего указатель, — к уменьшению.

"Уходящий последним гасит свет" — когда при уничтожении ссылки счетчик оказывается равен нулю, блок памяти удаляется. В объектно-ориентированных языках перед этим может быть вызван деструктор соответствующего объекта. Нередко этот способ освобождения памяти не называют сборкой мусора; для него есть специальное название — *подсчет ссылок* (reference counting).

Необходимо отметить важное преимущество подсчета ссылок — при нем удаление объектов и, если это необходимо, вызов деструкторов происходит точно в момент удаления последней ссылки на такой объект, поэтому данный метод используется для управления не только памятью, но и другими дорогостоящими ресурсами. Например, если объект содержал описатель окна пользовательского интерфейса, то при его уничтожении окно можно закрыть — оно точно уже не понадобится программе.

Сборка мусора подсчетом ссылок используется в диалектах языка Basic (в Visual Basic вплоть до версии 6; в VB 7 перешли к просмотру ссылок, который рассматривается далее в этом разделе), в языково-независимом объектном стандарте COM (Common Object Model — общая объектная модель) и в ряде других интерпретируемых языков программирования. Существуют также реализации подсчета ссылок на C++, так называемые смартпойнтеры (smart pointer, дословно "умный указатель"); есть ряд библиотек шаблонов,

которые реализуют подсчет ссылок для произвольных объектов. В частности, такой шаблон включен в ATL (Advanced Template Library — продвинутая библиотека шаблонов) компании Microsoft; эта библиотека, помимо прочего, используется для работы с объектами COM из языка *C*.

Далее в этой книге мы увидим несколько примеров использования подсчета ссылок для целей, отличающихся от задач управления оперативной памятью. Например, в файловых системах ОС семейства Unix файл может иметь несколько имен (так называемых жестких ссылок, hard link). Операции удаления файла как таковой не предоставляется, есть только операция удаления имени. При удалении последнего имени файл действительно удаляется.

Вообще, подсчет ссылок широко применяется в ядрах ОС для управления системными объектами, особенно такими, которые могут разделяться несколькими задачами — например, сегментами разделяемой памяти, особенно разделяемыми библиотеками, дескрипторами открытых файлов в Unix, средствами межпроцессного взаимодействия, объектами графического пользовательского интерфейса и т. д. Необходимость неявного удаления таких объектов продиктована тем, что задача может аварийно завершиться, не освободив некоторые из своих ресурсов; удалять же все разделяемые объекты при завершении одной из задач, очевидно, недопустимо. Часто такие системы указывают в описании системного API, что ресурсы, которые программа не освободит при своем завершении (как штатном, так и аварийном), будут освобождены неявно.

Впрочем, подсчет ссылок обладает одним практически важным недостатком — если у нас есть циклический список, на который нет ни одной ссылки извне, то все объекты в нем будут считаться используемыми, хотя они и являются мусором. Если мы по тем или иным причинам уверены, что кольца не возникают, метод подсчета ссылок вполне приемлем; если же мы используем графы произвольного вида, необходим более умный алгоритм.

4.3.2. Просмотр ссылок

Наиболее распространенной альтернативой подсчету ссылок является периодический просмотр всех ссылок, которые мы считаем "существующими" (что бы под этим ни подразумевалось) (рис. 4.14). В англоязычной литературе такой алгоритм называют mark and sweep (пометить и стереть).

Обычно просмотр начинается с именованных переменных и параметров функций. Если некоторые из указуемых объектов сами по себе могут содержать ссылки, мы вынуждены осуществлять просмотр рекурсивно. Проведя эту рекурсию до конца, мы можем быть уверены, что то и только то, что мы просмотрели, является нужными данными, и с чистой совестью можем объявить все остальное мусором. Эта стратегия решает проблему кольцевых

списков, но требует остановки всей остальной деятельности, которая может сопровождаться созданием или уничтожением ссылок.

Сборка мусора просмотром ссылок (иногда ее называют просто сборкой мусора без каких-либо дополнительных уточнений) является очень дорогой операцией, т. к. требует остановки всех действий, связанных с передачей указателей (присваивания, передача параметров процедурам и т. д.). По той же причине эту операцию нельзя запускать отдельной низкоприоритетной нитью; если сборка мусора делается в многопоточной программе, на время этой операции необходимо остановить все нити.

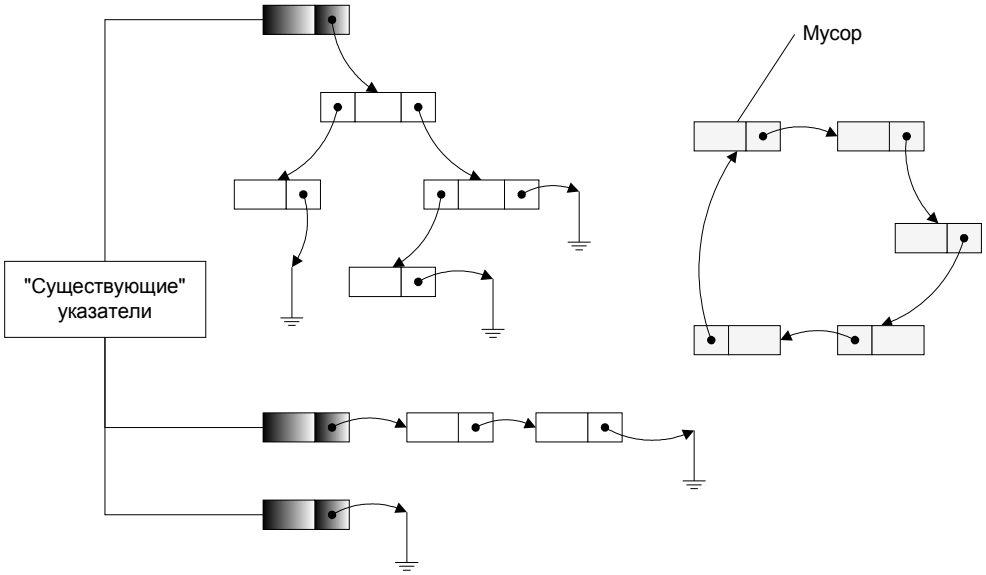

Рис. 4.14. Сборка мусора просмотром ссылок

Реальные системы, использующие сборку мусора, вызывают сборщик через определенные интервалы времени либо при достижении некоторого порога занятой памяти, что произойдет раньше. Ряд распространенных систем программирования, в том числе виртуальные машины Java, позволяют программисту вызвать сборщик мусора явно.

Но все эти методы, разумеется, не позволяют программисту гарантировать, что объект будет уничтожен к определенному моменту времени. В результате, потребности таких систем в оперативной памяти очень велики, и на большинстве задач в несколько раз превосходят требования эквивалентных программ, разработанных на основе явного удаления объектов или подсчета ссылок.

Кроме того, если объект занимает не только память, но и какие-то дорогостоящие внешние ресурсы — элементы графического пользовательского интерфейса, открытые файлы, сетевые соединения — эти ресурсы также будут освобождаться только по мере проходов сборщика мусора, т. е. в труднопредсказуемые моменты времени. На практике это приводит к чрезмерному использованию (или, скорее, к чрезмерному резервированию) таких ресурсов и опять-таки значительному повышению ресурсоемкости приложений.

Таким образом, хотя неявное удаление объектов резко снижает стоимость разработки программ, оно столь же резко повышает стоимость их исполнения. Для широкого класса задач — например, для систем поддержки бизнеса в небольших и средних предприятиях, которые разрабатываются на заказ и в которых переделка кода требуется довольно часто из-за меняющихся потребностей бизнеса, — оказывается выгодно минимизировать стоимость разработки и мириться с относительно высокой стоимостью исполнения.

В программах, распространяющихся большими тиражами, стоимость исполнения оказывается более важна — но тут вступают в игру более сложные факторы; например, необходимо учитывать возможность резкого изменения рыночной конъюнктуры (коммерческие риски) и тот, очевидный для экономистов, но не всегда понятный техническим специалистам факт, что деньги сегодня дороже денег завтра.

Говоря проще, если мы нарисовали бизнес-план, предполагающий миллионы запусков нашей программы, и, исходя из этого, оценили, что ее целесообразно писать на C++, мы сталкиваемся с риском, что кто-то напишет аналогичную программу раньше нас и наш проект закончится коммерческим провалом. При этом количество реальных запусков нашей программы будет измеряться единицами, а вовсе не запланированными миллионами.

Необходимо отметить, что бòльшие трудозатраты при разработке на C++ приводят к увеличению сроков разработки и повышают этот риск. Несмотря на все эти соображения, тиражируемые продукты до сих пор преимущественно разрабатываются на языках с явным освобождением памяти.

Существует также ряд сегментов рынка, в которых стоимость исполнения однозначно важнее стоимости разработки. Нередко случается, что одной только неспособности исполнить программу в определенных ресурсных рамках достаточно для полного провала проекта. Важный пример — это встраиваемые приложения, особенно те, где требуется автономное питание — носимые и портативные устройства, бортовые компьютеры транспортных средств и космических аппаратов, многие военные приложения. В этих приложениях решающей часто оказывается не стоимость ресурсов самих по себе, а их энергопотребление. Добавление динамической оперативной памяти приводит к увеличению потребляемой мощности. У бортового компьютера это может вы-

звать перегрузку бортовой электросети, в портативном устройстве приведет к снижению срока автономной работы и/или увеличению габаритов и цены устройства за счет более мощной батареи и т. д. Для статического ОЗУ или ЭСППЗУ это не всегда так, но такая память намного дороже динамического ОЗУ и может быть просто не доступна в требуемых объемах.

4.3.3. Генерационная сборка мусора

Важным недостатком сборщиков мусора с просмотром ссылок является тот факт, что на время работы сборщика необходимо останавливать работу системы или, точнее, всю деятельность, которая может привести к созданию и уничтожению ссылок на объекты. Поэтому сборка мусора в том виде, в каком она описана в разд. 4.3.2, не может осуществляться фоновой нитью. Кроме того, время работы сборщика мусора растет в линейной зависимости от количества объектов. В больших приложениях оно может составлять десятки процентов от общего времени исполнения программы.

Особенно большую проблему эти недостатки представляют в таких языках, как Java/C#, в которых все объекты (кроме скалярных переменных) выделяются и уничтожаются сборщиком мусора. Действительно, если в языке C++ мы можем описать локальную переменную в функции или методе, память под нее будет выделена в момент вызова функции в стеке. При выходе из функции или, точнее, из блока, в котором был описан объект, эта память будет автоматически освобождена. Напротив, в Java в стеке создается только ссылка на объект; сам объект выделяется в куче. После выхода из блока уничтожается указатель, но не объект; для уничтожения объекта необходим проход сборщика мусора. Таким образом, нормальная работа программы на Java сопровождается созданием большого количества короткоживущих объектов, которые, однако, сами по себе не уничтожаются.

Генерационный сборщик требует перемещения объектов по памяти, поэтому он несовместим с языками, использующими указатели, такими как Pascal и C/C++. Для работы генерационного сборщика система должна реализовать "ручки" (handle) — промежуточные объекты, через которые проходят все обращения к указуемому объекту. Язык высокого уровня делает использование "ручки" прозрачным для программиста и не требует явных преобразований "ручки" в указатель.

В простейшей форме генерационного сборщика мусора (generational garbage collection или generational scavenging) [Ungar 1984] объектный пул разбит на две части одинакового размера. Все вновь создаваемые объекты создаются в одной из частей, называемой "эдемом". Вторая часть пула в это время не используется (рис. 4.15).

При заполнении "эдема" система останавливается и начинается сборка мусора. Сборщик просматривает все объекты и — рекурсивно — ссылки из них на другие объекты, однако вместо простой отметки объекта он копирует каждый из обнаруженных объектов в неиспользуемую часть пула. Когда просмотр завершен, бывший "эдем" вместе со всем его содержимым объявляется мусором. Неиспользовавшаяся половина пула объявляется новым "эдемом" и работа системы продолжается. Такой сборщик мусора также называется *копирующим* (copying).

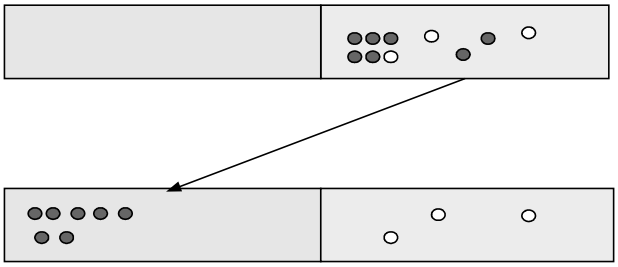

Рис. 4.15. Копирующий сборщик мусора

В системах, в которых объекты имеют финализационные методы, — как в Java/C# — разумеется, нельзя просто отбросить старый "эдем". Необходимо просмотреть его и выполнить финализационные методы для каждого из отброшенных объектов, но это может быть сделано фоновым процессом.

Важным преимуществом такого сборщика является тот факт, что сборка мусора оказывается совмещена с дефрагментацией пула. Поэтому можно значительно ускорить работу аллокатора — вместо просмотра списка свободных блоков памяти аллокатор может просто запоминать границу между свободной и занятой частями "эдема" и выделять память под новые объекты, сдвигая эту границу. Впрочем, основной недостаток сборщиков мусора — необходимость останавливать систему на время сборки — такой подход не устраняет.

Идея более сложных версий генерационных сборщиков основана на гипотезе, что срок жизни большинства объектов невелик и вероятность уничтожения вновь созданного объекта существенно выше, чем вероятность уничтожения давно существовавших объектов. Кроме того, с высокой вероятностью вновь создаваемые объекты будут ссылаться на старые объекты и друг на друга, но старые объекты вряд ли будут ссылаться на короткоживущие.

Вместо двух поколений объектов пул разбивается на несколько поколений разного размера с разными ожидаемыми сроками жизни. Сборка мусора состоит в переносе объектов из младших поколений в старшие. Сборка мусора,

которая охватывает только "эдем" и младшее поколение, называется "*малой*" (minor). Заполнение пула старшего поколения инициирует "б*ольшую*" (major) сборку мусора, которая охватывает весь пул.

При таком подходе серьезную проблему представляют ссылки из объектов старших поколений на объекты младших поколений. Гипотеза, что такие ссылки редки, весьма убедительна и подтверждается практикой, но практика же показывает, что такие ссылки иногда все же возникают. Кроме того, для корректной работы системы такие ссылки необходимо обрабатывать, как бы ни была низка вероятность их появления.

Чтобы решить эту проблему, система отслеживает все передачи ссылок между объектами и запоминает все ссылки из объектов старшего поколения на младшие в специальном списке, который так и называется — remembered set (запомненное множество; общепринятого русскоязычного перевода этого названия нет). Таким образом, при малой сборке мусора необходимо просмотреть ссылки из именованных переменных, remembered set и — рекурсивно — из объектов младшего поколения, на которые эти ссылки указывают. Попадание по ссылке на объект старшего поколения означает прекращение рекурсии.

Этот подход обеспечивает приемлемую производительность только при небольшом объеме remembered set. Статистика показывает, что на практике этот объем действительно оказывается небольшим и, при удачном подборе параметров сборщика, иногда вообще нулевым.

Ориентация на среднюю производительность делает генерационную сборку мусора неприемлемой для задач реального времени; впрочем, все варианты сборки мусора с просмотром не обеспечивают гарантированного поведения и потому для задач реального времени непригодны.

Ключевым параметром, который определяет поведение системы под данной конкретной загрузкой, оказывается относительный размер пулов разных поколений. При правильном подборе этого параметра большинство объектов уничтожаются при малых сборках мусора. Большие сборки при этом происходят относительно редко. Поэтому, хотя работа сборщика по-прежнему предполагает остановку системы, среднее время такой остановки оказывается невелико и наблюдаемая производительность системы значительно возрастает.

Дополнительное преимущество генерационного сборщика состоит в том, что часто используемые объекты каждого из поколений накапливаются в начале пула своего поколения. Таким образом, все часто используемые объекты собираются в нескольких относительно небольших областях памяти — благодаря этому значительно повышается эффективность работы кэшей центральных процессоров и страничной виртуальной памяти.

Сборка мусора в Sun Java HotSpot VM

Виртуальная машина Java HotSpot была реализована в версии Java 1.2.2 и стала стандартной в Java 1.3. В этой VM используется генерационный сборщик мусора, предполагающий разбиение объектов на четыре поколения: перманентные, старые (old), молодые (new/young) и вновь создаваемые (eden) [Gottry 2002]. Структура пула Java Hotspot изображена на рис. 4.16.

Перманентные объекты — это системные объекты JVM, срок жизни которых всегда равен времени исполнения программы. При нормальной работе системы они не подвергаются сборке мусора.

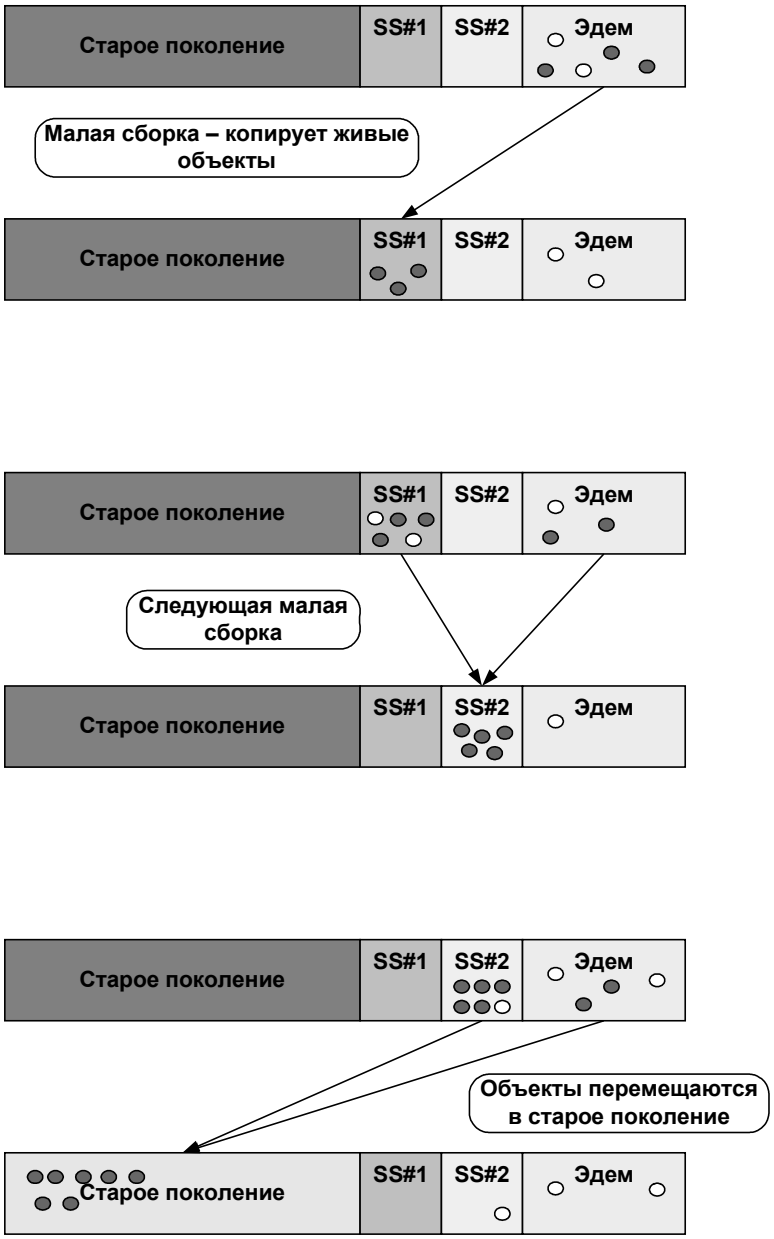

Рис. 4.16. Структура пула в JVM Hotspot

Пул молодых объектов состоит из двух частей, называемых Semispace#1 и SemiSpace#2 (полупространства 1 и 2, сокращенно SS#1 и SS#2). В некоторых статьях SS расшифровывают как Survivor Space (пространство выживших).

Все вновь создаваемые объекты создаются в "эдеме". При заполнении эдема инициируется малая сборка мусора.

При первой малой сборке мусора SS#1 и SS#2 пусты. Все используемые объекты из эдема перемещаются в SS#1; все, что осталось в эдеме, объявляется мусором, эдем очищается.

При второй малой сборке мусора SS#1 занят, но SS#2 пуст. Используемые объекты из SS#1 и эдема перемещаются в SS#2, оставшееся содержимое эдема и SS#1 очищается.

Объект, определенное число раз перемещавшийся между SS#1 и SS#2, признается долгоживущим (tenured) и перемещается в пул старшего поколения. Соответствующий параметр не задается явно; вместо этого система динамически подбирает порог "зрелости" объекта, пытаясь за счет этого обеспечить определенный уровень заполнения пространства выживших.

Вместе с каждым долгоживущим объектом перемещаются и все объекты, на которые он ссылался, даже если эти объекты находились в эдеме. Это делается, чтобы защитить соответствующие объекты от потери при последующих малых сборках мусора.

Данная стратегия обоснована только статистикой исполнения реальных программ и гипотезой, что долгоживущие объекты не ссылаются на короткоживущие, однако практика подтверждает, что обычно она работает довольно хорошо.

При заполнении пула старшего поколения инициируется большая сборка мусора. Эта сборка затрагивает только объекты старшего поколения и производится в два этапа. На первом этапе сборщик помечает используемые объекты, как при обычном mark and sweep. На втором этапе происходит дефрагментация — все используемые объекты копируются в начало пула. В зависимости от настроек JVM, после большой сборки мусора может быть запущена малая.

Таким образом, система имеет следующие параметры настройки:

- `ms` — начальный размер пула, который соответствует сумме размеров эдема и полупространств среднего поколения SS#1 и SS#2;
- `semispaces` или `SurvivorRatio` — размер полупространств среднего поколения; этот параметр может задаваться как абсолютным значением (`semispaces`), так и относительно начального размера пула (`SurvivorRatio`);
- `TargetSurvivorRatio` — доля (в процентах) заполнения пространства выживших;
- размер пространства старшего поколения. В параметрах командной строки вместо этого указывается общий максимальный размер пула `mx`; пространство старшего поколения соответствует разности между минимальным и максимальным размерами пула.

Существует также ряд других способов задавать эти параметры друг относительно друга; чаще всего используется параметр `NewRatio`, задающий отношение объемов пула старого и молодого поколения (суммы объемов эдема и полупространств). Так, при `NewRatio=2` под пул молодого поколения занимается 1/3 памяти, а пул старого, соответственно, 2/3.

Пользователь может подбирать эти параметры самостоятельно, наблюдая за работой приложения, — для этого можно заставить JVM выдавать довольно подробную статистику работы сборщика мусора, в том числе:

- среднее и максимальное время работы сборщика (с разбивкой на малые и большие сборки);
- общий уровень заполнения пула и его разбивка по поколениям;
- порог "зрелости" объекта и т. д.

По умолчанию JVM реализует два режима работы (рис. 4.17): клиентский и серверный.

В клиентском режиме система выделяет 1/9 пула под новое поколение объектов и, соответственно, 8/9 под старое поколение. При этом малые сборки мусора происходят очень часто; клиентские программы обычно работают недолго, и потому до большой сборки мусора может вообще не дойти.

Для серверного режима под новое поколение выделяется 1/3 пула, а под старое — 2/3. Это приводит к значительному перераспределению времени между малыми и большими сборками мусора: малые сборки происходят реже, а большие — чаще, но зато каждая большая сборка занимает меньше времени.

Рис. 4.17. Относительные размеры пулов младшего и старшего поколения в разных версиях HotSpot JVM

Относительно успешная попытка решить проблему фоновой сборки мусора — это *репликационный* или *инкрементальный сборщик мусора*. В действительности, это вариант генерационного сборщика мусора. Основное отличие состоит в том, что при работе (как при просмотре, так и при дефрагментации) репликационный сборщик блокирует не систему в целом, а только нити, которые пытаются модифицировать уже просмотренные объекты. При этом нити, которые только читают значения полей объектов или модифицируют еще не просмотренные объекты, могут продолжать исполнение.

Поскольку объекты — особенно долгоживущие — читаются гораздо чаще, чем модифицируются, это может резко сократить время блокировки и значительно повысить наблюдаемую производительность, хотя, конечно же, полностью исключить блокировку не получается. Наибольший выигрыш при этом достигается при больших сборках, которые затрагивают преимуще-

ственно или исключительно долгоживущие объекты и вносят наибольший вклад в наблюдаемое время работы сборщика.

Особенно это значимо для многопоточных серверных программ.

В анонсе JVM 1.3 инкрементальный сборщик мусора даже был назван работающим без пауз (pauseless) [**java.sun.com** HotSpot]. Впрочем, надо отдать должное сотрудникам компании Sun, писавшим этот документ, — слово pauseless написано в кавычках и с разъяснением, что это означает на самом деле.

Все современные реализации Java, C# и других объектно-ориентированных языков со сборкой мусора (Visual Basic V7, Ocaml и т. д.) в той или иной форме поддерживают репликационную сборку мусора, хотя многие интерпретаторы позволяют ее отключить и вернуться к простому генерационному сборщику — это может быть выгодно для программ с малым числом потоков.

4.4. Открытая память (продолжение)

Описанные ранее алгоритмы распределения памяти используются не операционной системой, а библиотечными функциями, присоединенными к программе или, говоря шире, средой исполнения языка программирования. Однако ОС, которая реализует одновременную загрузку (но не обязательно одновременное исполнение: MS DOS — типичный пример такой системы) нескольких задач, также должна использовать тот или иной алгоритм размещения памяти. Отчасти такие алгоритмы могут быть похожи на работу функции `malloc`. Однако режим работы ОС может вносить существенные упрощения в алгоритм.

Перемещать образ загруженного процесса по памяти невозможно: даже если его код был позиционно-независим и не подвергался перенастройке, сегмент данных может содержать (и почти наверняка содержит) указатели, которые при перемещении необходимо перенастроить. Поэтому при выгрузке задач из памяти перед нами в полный рост встает проблема внешней фрагментации (рис. 4.18).

Управление памятью в OS/360

В этой связи нельзя не вспомнить поучительную историю, связанную с управлением памятью в системах линии IBM System 360. В этих машинах не было аппаратных средств управления памятью, и все программы разделяли общее виртуальное адресное пространство, совпадающее с физическим. Адресные ссылки в программе задавались 12-битовым смещением относительно базового регистра. В качестве базового регистра мог использоваться, в принципе, лю-

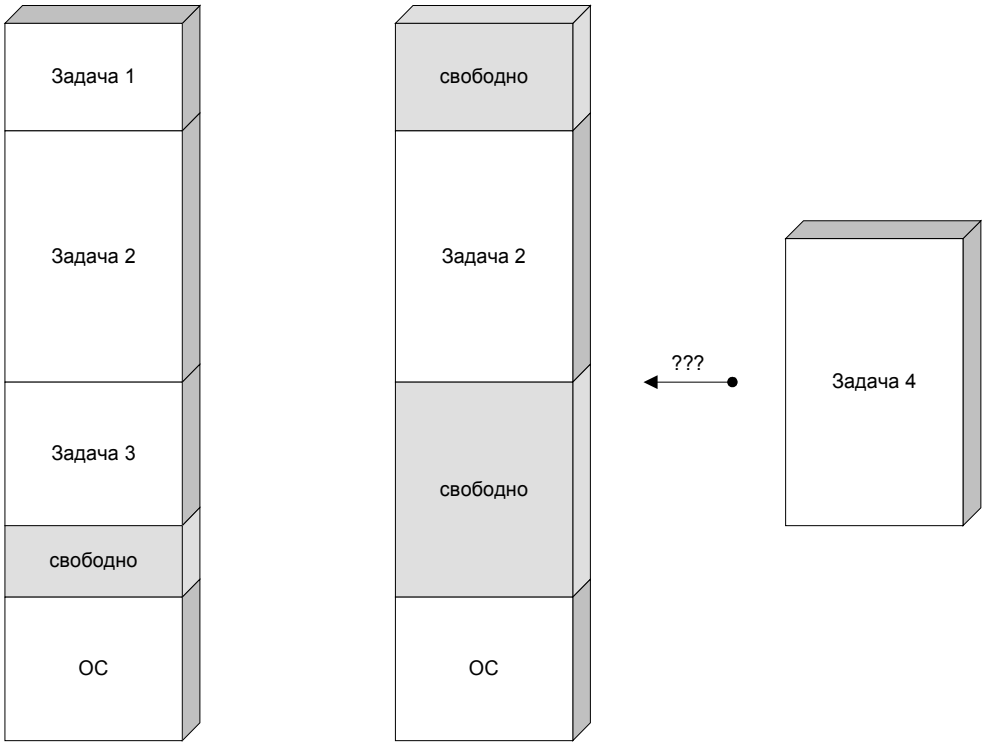

Рис. 4.18. Фрагментация при загрузке и выгрузке задач

бой из 16 32-битовых регистров общего назначения. Предполагалось, что пользовательские программы не модифицируют базовый регистр, поэтому можно загружать их с различных адресов, просто перенастраивая значение этого регистра. Таким образом, была реализована одновременная загрузка многих программ в многозадачной системе OS/360.

Однако после загрузки программу уже нельзя было перемещать по памяти: например, при вызове подпрограммы адрес возврата сохраняется в виде абсолютного 24-битового адреса (в System 360 под адрес отводилось 32-разрядное слово, но использовались только 24 младших бита адреса; в System 370 адрес стал 31-разрядным) и при возврате базовый регистр не применяется. Аналогично, базовый регистр не используется при ссылках на блоки параметров и сами параметры подпрограмм языка FORTRAN (в этом языке все параметры передаются по ссылке), при работе с указателями в PL/I и т. д. Перемещение программы, даже с перенастройкой базового регистра, нарушило бы все такие ссылки.

Разработчики фирмы IBM вскоре осознали пользу перемещения программ после их загрузки и попытались как-то решить эту проблему. Очень любопытный документ "Preparing to Rollin-Rollout Guideline" (Руководство по подготовке [программы] к вкатыванию и выкатыванию; к сожалению, мне не удалось найти полного текста этого документа) описывает действия, которые программа должна была бы предпринять после перемещения. Фактически программа должна была

найти в своем сегменте данных все абсолютные адреса и сама перенастроить их.

Естественно, никто из разработчиков компиляторов и прикладного программного обеспечения не собирался следовать этому руководству. В результате, проблема перемещения программ в OS/360 не была решена вплоть до появления машин System 370 со страничным или странично-сегментным диспетчером памяти и ОС MVS.

В данном случае проблема фрагментации особенно остра, т. к. типичный образ процесса занимает значительную часть всего доступного ОЗУ. Если при выделении небольших блоков мы еще можем рассчитывать на "закон больших чисел" и прочие статистические закономерности, то самый простой сценарий загрузки трех процессов различного размера может привести нас к неразрешимой ситуации (см. рис. 4.16).

Разделы памяти *(см. разд. 3.2)* отчасти позволяют решить проблему внешней фрагментации, устанавливая, что процесс должен либо использовать раздел целиком, либо не использовать его вовсе. Как и все ограничения на размер единицы выделения памяти, это решение загоняет проблему внутрь, переводя внешнюю фрагментацию во внутреннюю. Поэтому некоторые системы предлагают другие способы наложения ограничения на порядок загрузки и выгрузки задач.

Управление памятью в MS DOS

Так, например, процедура управления памятью MS DOS рассчитана на случай, когда программы выгружаются из памяти только в порядке, обратном тому, в каком они туда загружались (на практике они могут выгружаться и в другом порядке, но это явно запрещено в документации и часто приводит к проблемам). Это позволяет свести управление памятью к стековой дисциплине.

Каждой программе в MS DOS отводится блок памяти. С каждым таким блоком ассоциирован дескриптор, называемый *MCB — Memory Control Block* (рис. 4.19). Этот дескриптор содержит размер блока, идентификатор программы, которой принадлежит этот блок, и признак того, является ли данный блок последним в цепочке. Нужно отметить, что программе всегда принадлежит несколько блоков, но это уже несущественные детали. Другая малосущественная деталь та, что размер сегментов и их адреса отсчитываются в параграфах размером 16 байт. Знакомые с архитектурой процессора 8086 должны вспомнить, что адрес MCB в этом случае будет состоять только из сегментной части с нулевым смещением.

После запуска COM-файл получает сегмент размером 64 Кбайт, а EXE — всю доступную память. Обычно EXE-модули сразу после запуска освобождают ненужную им память и устанавливают `brklevel` на конец своего сегмента, а потом увеличивают `brklevel` и наращивают сегмент по мере необходимости.

Естественно, что наращивать сегмент можно только за счет следующего за ним в цепочке MCB, и MS DOS разрешит делать это только в случае, если этот сегмент не принадлежит никакой программе.

При запуске программы DOS берет последний сегмент в цепочке и загружает туда программу, если этот сегмент достаточно велик. Если он недостаточно велик, DOS говорит: "Недостаточно памяти" и отказывается загружать программу. При завершении программы DOS освобождает все блоки, принадлежавшие программе. При этом соседние блоки объединяются. Пока программы действительно завершаются в порядке, обратном тому, в котором они запускались, — все вполне нормально. Другое дело, что в реальной жизни возможны отклонения от этой схемы. Например, предполагается, что TSR-программы (Terminate, but Stay Resident — завершиться и остаться резидентно (в памяти)) никогда не пытаются по-настоящему завершиться и выгрузиться. Тем не менее любой уважающий себя хакер считает своим долгом сделать резидентную программу выгружаемой. У некоторых хакеров она в результате выбрасывает при выгрузке все резиденты, которые заняли память после нее. Другой пример — отладчики обычно загружают программу в обход обычной DOS-функции `LOAD & EXECUTE`, а при завершении отлаживаемой программы сами освобождают занимаемую ею память.

Я в свое время занимался прохождением некоторой программы под управлением отладчика. Честно говоря, речь шла о взломе некоторой игрушки... Эта программа производила какую-то инициализацию, а потом вызывала функцию DOS `LOAD & EXECUTE`. Я об этом не знал и, естественно, попал внутрь новой программы, которую и должен был взламывать.

После нескольких нажатий комбинаций клавиш <Ctrl>+<Break> я наконец-то вернулся в отладчик, но при каком-то очень странном состоянии программы. Покопавшись в программе с помощью отладчика в течение некоторого времени и убедившись, что она не хочет приходить в нормальное состояние, я вышел из отладчика и увидел следующую картину: системе доступно около 100 Кбайт в то время, как сумма длин свободных блоков памяти более 300 Кбайт, а размер наибольшего свободного блока около 200 Кбайт. Отладчик, выходя, освободил свою память и память отлаживаемой программы, но не освободил память, выделенную новому загруженному модулю. В результате посредине памяти остался никому не нужный блок изрядного размера, помеченный как используемый (рис. 4.20). Самым обидным было то, что DOS не пыталась загрузить ни одну программу в память под этим блоком, хотя там было гораздо больше места, чем над ним.

В системах с открытой памятью невозможны эффективные средства разделения доступа. Любая программная проверка прав доступа может быть легко обойдена прямым вызовом "защищаемых" модулей ядра. Даже криптографические средства не обеспечивают достаточно эффективной защиты, потому что можно "посадить" в память троянскую программу, которая будет анализировать код программы шифрования и считывать значение ключа.

В заключение можно привести основные проблемы многопроцессных систем без диспетчера памяти:

- проблема выделения дополнительной памяти задаче, которая загружалась не последней;
- проблема освобождения памяти после завершения задачи — точнее, возникающая при этом фрагментация свободной памяти;

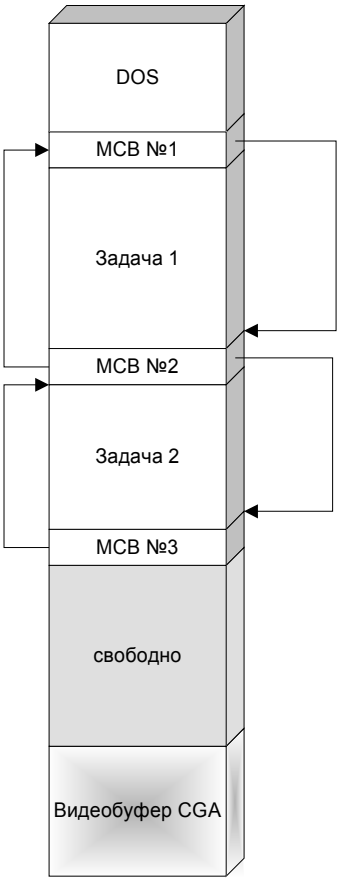

Рис. 4.19. Управление памятью в MS DOS

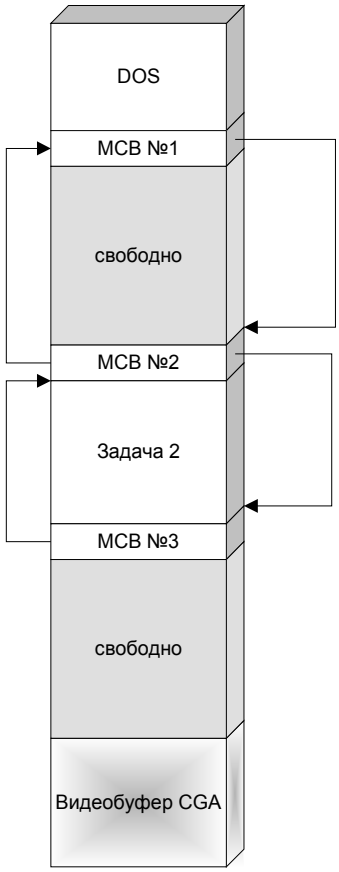

Рис. 4.20. Нарушения стекового порядка загрузки и выгрузки в MS DOS

- низкая надежность. Ошибка в одной из программ может привести к порче кода или данных других программ или самой системы;
- проблемы безопасности.

В системах с динамической сборкой первые две проблемы не так остры, потому что память выделяется и освобождается небольшими кусочками, по блоку на каждый объектный модуль, и код программы обычно не занимает непрерывного пространства. Соответственно, такие системы часто разрешают и данным программы занимать несмежные области памяти.

Такой подход используется многими системами с открытой памятью — AmigaDOS, Oberon, системами программирования для транспьютера и т. д. Проблема фрагментации при этом не снимается полностью, однако для создания катастрофической фрагментации не достаточно двух загрузок задач и од-

ной выгрузки, а требуется довольно длительная работа, сопровождающаяся интенсивным выделением и освобождением памяти.

В системе MacOS был предложен достаточно оригинальный метод борьбы с фрагментацией, заслуживающий отдельного обсуждения.

4.4.1. Управление памятью в MacOS и Win16

В этих системах предполагается, что пользовательские программы не сохраняют указателей на динамически выделенные блоки памяти. Вместо этого каждый такой блок идентифицируется целочисленным дескриптором или "*ручкой*" (handle) (рис. 4.21). Когда программа непосредственно обращается к данным в блоке, она выполняет системный вызов `GlobalLock` (запереть). Этот вызов возвращает текущий адрес блока. Пока программа не исполнит вызов `GlobalUnlock` (отпереть), система не пытается изменить адрес блока. Если же блок не заперт, система считает себя вправе передвигать его по памяти или даже сбрасывать на диск (рис. 4.22).

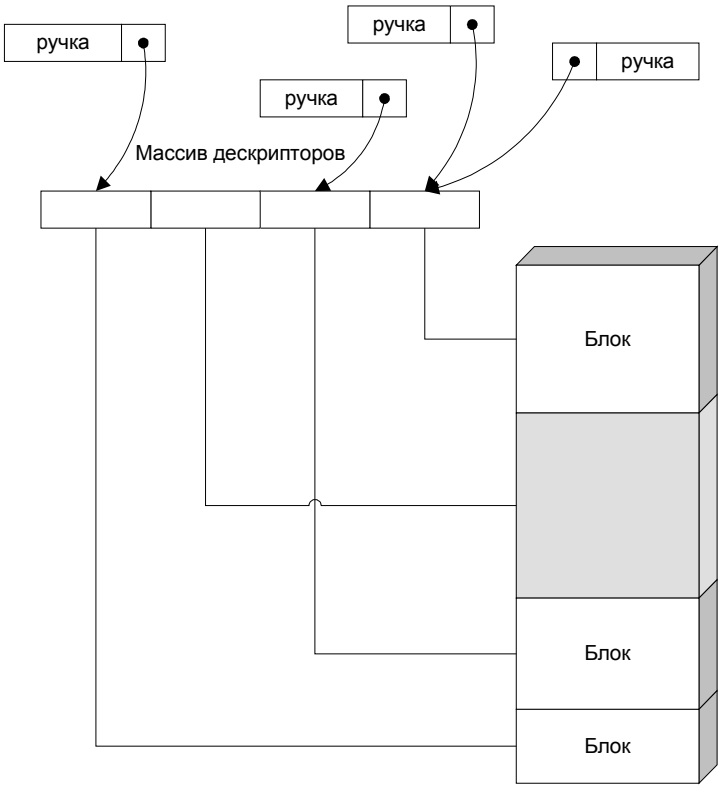

Рис. 4.21. Управление памятью с помощью "ручек"

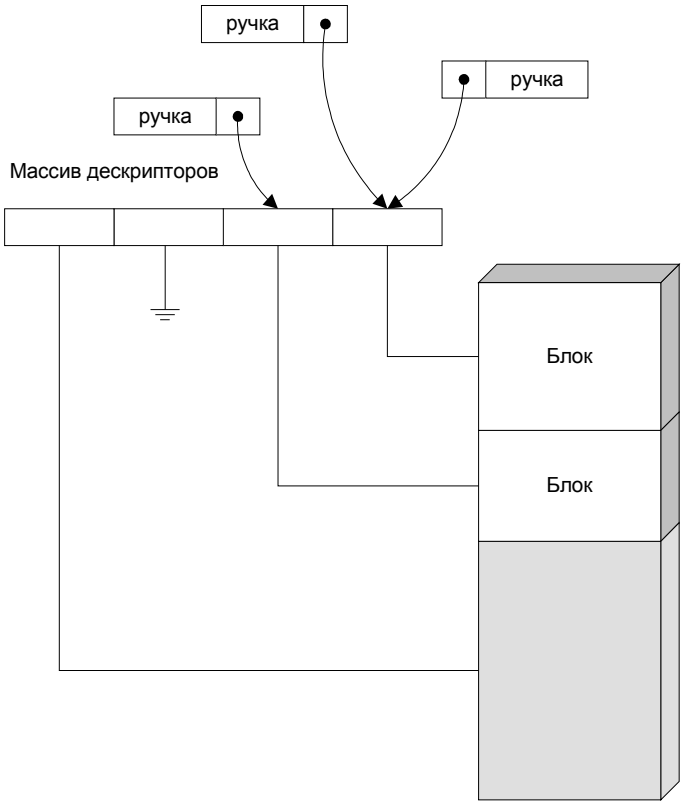

Рис. 4.22. Дефрагментация при управлении памятью с помощью "ручек"

"Ручки" представляют собой попытку создать программный аналог аппаратных диспетчеров памяти. Они позволяют решить проблему фрагментации и даже организовать некое подобие виртуальной памяти. Можно рассматривать их как средство организации оверлейных данных — поочередного отображения разных блоков данных на одни и те же адреса. Однако за это приходится платить очень дорогой ценой.

Нужно отметить, что эта цена оказывается дорогой только в системах программирования, в которых используются указатели в виде реальных адресов блоков памяти — т. е. в таких языках, как Pascal, Fortran, C/C++. В других языках, где понятие указателя или ссылки глубже спрятано от программиста — в Basic, Smalltalk, Java/C# — проблема не столь остра. В таких языках компилятор просто вставляет неявные вызовы `GlobalLock`/`GlobalUnlock` в каждое обращение к объекту. Разумеется, это снижает производительность, но преимущества, которые дает эта схема при управлении памятью и сборке мусора *(см. разд. 4.3.3)*, могут оправдать этот недостаток.

В тех языках, где программисту требуется настоящий указатель, использование "ручек" сильно усложняет программирование вообще и в особенности перенос ПО из систем, использующих линейное адресное пространство. Все указатели на динамические структуры данных в программе нужно заменить на "ручки", а каждое обращение к таким структурам необходимо окружить вызовами `GlobalLock/GlobalUnlock`.

Вызовы `GlobalLock/GlobalUnlock`:

- сами по себе увеличивают объем кода и время исполнения;
- мешают компиляторам выполнять оптимизацию, прежде всего не позволяют оптимально использовать регистры процессора, потому что далеко не все регистры сохраняются при вызовах;
- требуют разрыва конвейера команд и перезагрузки командного кэша; в современных суперскалярных процессорах это может приводить к падению производительности во много раз. Любопытно, что в интерпретаторах Basic, Java/C# и т. д. этот недостаток в значительной мере компенсируется за счет того, что код функций `GlobalLock/GlobalUnlock` не оформлен в виде отдельных процедур, а просто вставлен в соответствующие точки кода интерпретатора. Легко понять, что сам этот код представляет собой просто установку и снятие флага в дескрипторе "ручки", т. е. в большинстве случаев может быть реализован одной командой центрального процессора.

Попытки уменьшить число блокировок требуют определенных интеллектуальных усилий. Фактически, к обычному циклу разработки ПО: проектирование, выбор алгоритма, написание кода и его отладка — добавляются еще две фазы: микрооптимизация использования "ручек" и отладка оптимизированного кода. Последняя фаза оказывается, пожалуй, самой сложной и ответственной.

Наиболее опасной ошибкой, возникающей на фазе микрооптимизации, является вынос указателя на динамическую структуру за пределы скобок `GlobalLock/GlobalUnlock`. Эту ошибку очень сложно обнаружить при тестировании, т. к. она проявляется, только если система пыталась передвигать блоки в промежутках между обращениями. Иными словами, ошибка может проявлять или не проявлять себя в зависимости от набора приложений, исполняющихся в системе, и от характера деятельности этих приложений. В результате мы получаем то, чего больше всего боятся эксплуатационщики, — систему, которая работает *иногда*.

Не случайно фирма Microsoft полностью отказалась от управления памятью с помощью "ручек" в следующей версии MS Windows — Windows 95, в которой реализована почти полноценная виртуальная память. При переходе от

Windows 3.x к Windows 95 наработка на отказ — даже при исполнении той же самой смеси приложений — резко возросла, так что система из работающей *иногда* превратилась в работающую *как правило*. По-видимому, это означает, что большая часть фатальных ошибок в приложениях Win16 действительно относилась к ошибкам работы с "ручками".

Mac OS версии 10, построенная на ядре BSD Mach, также имеет страничную виртуальную память и никогда не перемещает блоки памяти, адресуемые "ручками".

4.5. Системы с базовой виртуальной адресацией

Как уже говорилось, в системах с открытой памятью возникают большие сложности при организации многозадачной работы. Самым простым способом разрешения этих проблем оказалось предоставление каждому процессу своего виртуального адресного пространства. Простейшим методом организации различных адресных пространств является так называемая *базовая адресация*. По-видимому, это наиболее старый из реализовавшихся на практике способов виртуальной адресации, первые его реализации относятся к началу 60-х годов XX века, т. е. к компьютерам второго поколения. Впрочем, этот способ находит применение и в современных процессорах — в устройствах, предназначенных для встраиваемых приложений или при разделении памяти между виртуальными машинами.

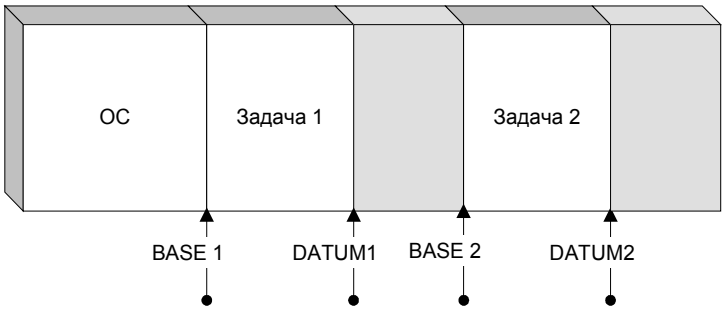

Рис. 4.23. Виртуальная память на основе базовой адресации

Вы можете заметить, что термин "базовая адресация" уже занят — мы называли таким образом адресацию по схеме `reg[offset]`. Метод, о котором сейчас идет речь, состоит в формировании адреса по той же схеме. Отличие состоит в том, что регистр, относительно которого происходит адресация, не доступен прикладной программе. Кроме того, его значение прибавляется ко

всем адресам, в том числе к "абсолютным" адресным ссылкам или переменным типа указатель.

Как правило, машины, использующие базовую адресацию, имеют два регистра (рис. 4.23). Один из регистров задает базу для адресов, второй устанавливает верхний предел. В системе ICL1900/Одренок эти регистры называются соответственно `BASE` и `DATUM`. Если адрес выходит за границу, установленную значением `DATUM`, возникает *исключительная ситуация (exception) ошибочной адресации*. Как правило, это приводит к тому, что система принудительно завершает работу программы.

Базовая адресация в процессорах PowerPC

В процессорах PowerPC предусмотрено два режима работы диспетчера памяти:

1. Virtual mode (виртуальный режим), при котором используется двухуровневая сегментно-страничная трансляция адресов.
2. Real mode (реальный режим), в котором обеспечивается прямой доступ к физической памяти.

Принципы работы диспетчера памяти в первом режиме будут рассматриваться в следующей главе.

При прямом доступе к памяти адрес все-таки может подвергаться модификации с помощью регистров RLMR (Real Mode Limit Register — регистр-ограничитель реального режима) и RMOR (Real Mode Offset Register — регистр смещения реального режима). Если эффективный адрес при обращении к памяти превышает значение RLMR, возникает исключение доступа к памяти, по которому вызывается процедура-обработчик.

Реальный режим главным образом используется во встраиваемых приложениях; процессоры линии Power находят применение во многих таких приложениях, начиная от управляющих процессоров лазерных принтеров и заканчивая бортовыми компьютерами марсианских зондов. Некоторые модели процессоров, например PPC 401CF, вообще не имеют сегментно-страничного диспетчера памяти и поддерживают только реальный режим адресации.

В процессорах с диспетчером памяти реальный режим и базовая адресация может использоваться для реализации гипервизора (hypervisor) или, что то же самое, диспетчера виртуальных машин. Каждая "задача", исполняющаяся в такой среде, представляет полноценную операционную систему, которая управляет своей памятью с помощью страничного диспетчера. Базовая же адресация позволяет защитить и скрыть эти "задачи" — образы виртуальных машин — друг от друга.

С помощью этих двух регистров мы сразу решаем две важные проблемы.

Во-первых, мы можем изолировать процессы друг от друга — ошибки в программе одного процесса не приводят к разрушению или повреждению образов других процессов или самой системы. Благодаря этому мы можем обеспечить защиту системы не только от ошибочных программ, но и от злонамеренных действий пользователей, направленных на разрушение системы или доступ к чужим данным.

Во-вторых, мы получаем возможность передвигать образы процессов по физической памяти так, что программа каждого из них не замечает перемещения. За счет этого мы решаем проблему фрагментации памяти и даем процессам возможность наращивать свое адресное пространство. Действительно, в системе с открытой памятью процесс может добавлять себе память только до тех пор, пока не доберется до начала образа следующего процесса. После этого мы должны либо говорить о том, что памяти нет, либо мириться с тем, что процесс может занимать несмежные области физического адресного пространства. Второе решение резко усложняет управление памятью как со стороны системы, так и со стороны процесса, и часто оказывается неприемлемым (подробнее связанные с этим проблемы обсуждаются в *разд. 4.4*). В случае же базовой адресации мы можем просто сдвинуть мешающий нам образ вверх по физическим адресам (рис. 4.24).

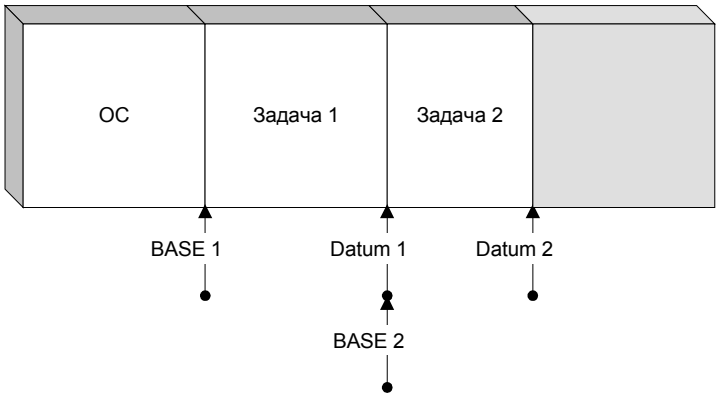

Рис. 4.24. Дефрагментация при использовании базовой адресации

Часто ОС, работающие на таких архитектурах, умеют сбрасывать на диск образы тех процессов, которые долго не получают управления. Это самая простая из форм *своппинга* (swapping — обмен) (русскоязычный термин "*страничный обмен*" довольно широко распространен, но в данном случае его использование было бы неверным, потому что обмену подвергаются не страницы, а целиком задачи).

Решив перечисленные ранее проблемы, мы создаем другие, довольно неожиданные. Мы оговорили, что базовый регистр недоступен прикладным задачам. Но какой-то задаче он должен быть доступен! Каким же образом процессор узнает, исполняет ли он системную или прикладную задачу, и не сможет ли злонамеренная прикладная программа его убедить в том, что является системной?

Другая проблема состоит в том, что, если мы хотим предоставить прикладным программам возможность вызывать систему и передавать ей параметры,

мы должны обеспечить процессы (как системные, так и прикладные) теми или иными механизмами доступа к адресным пространствам друг друга.

На самом деле эти две проблемы тесно взаимосвязаны — например, если мы предоставим прикладной программе свободный доступ к системному адресному пространству, нам придется распрощаться с любыми надеждами на защиту от злонамеренных действий пользователей. Раз проблемы взаимосвязаны, то и решать их следует в комплексе.

Стандартное решение этого комплекса проблем состоит в следующем. Мы снабжаем процессор флагом, который указывает, исполняется системный или пользовательский процесс. Код пользовательского процесса не может манипулировать этим флагом, однако ему доступна специальная команда. В различных архитектурах данные специальные команды имеют разные мнемонические обозначения, далее мы будем называть эту команду `SYSCALL`. `SYSCALL` одновременно переключает флаг в положение "системный" и передает управление на определенный адрес в системном адресном пространстве. Процедура, находящаяся по этому адресу, называется *диспетчером системных вызовов* (рис. 4.25).

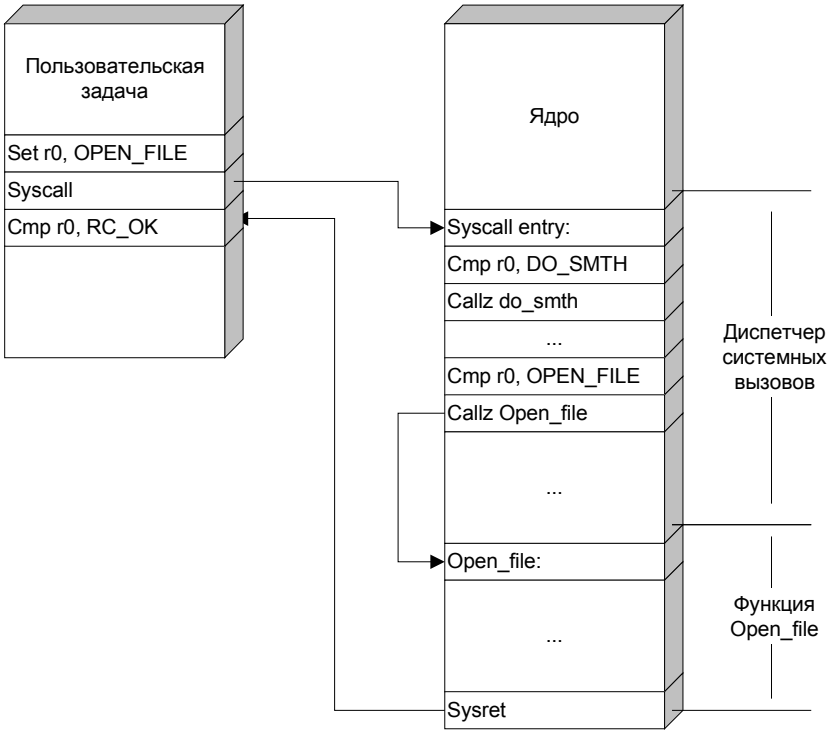

Рис. 4.25. Диспетчер системных вызовов

Возврат из системного вызова осуществляется другой специальной командой, назовем ее SYSRET. Эта команда передает управление на указанный адрес в указанном адресном пространстве и одновременно переводит флаг в состояние "пользователь". Необходимость выполнять эти две операции одной командой очевидна: если мы сначала сбросим флаг, мы потеряем возможность переключать адресные пространства, а если мы сначала передадим управление, никто не может нам гарантировать, что пользовательский код добровольно выйдет из системного режима.

Протокол общения прикладной программы с системой состоит в следующем: программа помещает параметры вызова в оговоренное место — обычно в регистры общего назначения или в стек — и исполняет SYSCALL. Одним из параметров передается и код системного вызова. Диспетчер вызовов анализирует допустимость параметров и передает управление соответствующей процедуре ядра, которая и выполняет требуемую операцию (или не выполняет, если у пользователя не хватает полномочий). Затем процедура помещает в оговоренное место (чаще всего опять-таки в регистры или в пользовательский стек) возвращаемые значения и передает управление диспетчеру, или вызывает SYSRET самостоятельно.

Сложность возникает, когда ядру при исполнении вызова требуется доступ к пользовательскому адресному пространству. В простейшем случае, когда все параметры (как входные, так и выходные) размещаются в регистрах и представляют собой скалярные значения, проблемы нет, но большинство системных вызовов, особенно запросы обмена данными с внешними устройствами, в эту схему не укладываются.

В системах с базовой адресацией эту проблему обычно решают просто: в "системном" режиме базовый и ограничительный регистры не используются вообще, и ядро имеет полный доступ ко всей физической памяти, в том числе и к адресным пространствам всех пользовательских задач (рис. 4.26). Это решение приводит к тому, что хотя система и защищена от ошибок в пользовательских программах, пользовательские процессы оказываются совершенно не защищены от системы, а ядро — не защищено от самого себя. Ошибка в любом из системных модулей приводит к полной остановке работы.

В архитектурах с более сложной адресацией нередко предоставляются специальные инструкции для передачи данных между пользовательским и системным адресными пространствами. Однако и в этом случае ядро ОС обычно имеет полный доступ к адресным пространствам пользовательских задач.

В современных системах базовая виртуальная адресация используется редко. Дело не в том, что она плоха, а в том, что более сложные методы, такие как сегментная и страничная трансляция адресов, оказались намного лучше. Часто под словами "виртуальная память" подразумевают именно сегментную или страничную адресацию.

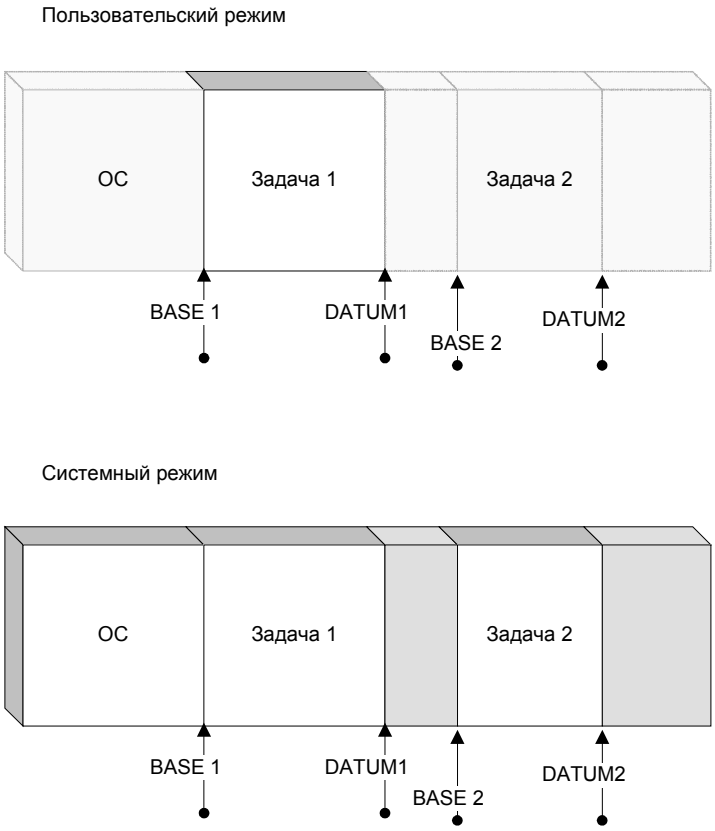

Рис. 4.26. Системный и пользовательский режимы

Вопросы для самопроверки

1. Что такое внешняя фрагментация?
2. Почему в большинстве случаев невозможно перемещать занятые блоки памяти?
3. Что такое внутренняя фрагментация? При каких условиях она возникает?
4. Может ли внутренняя фрагментация возникать при выделении памяти блоками переменного размера?
5. Почему страгия first fit считается лучшей из известных стратегий поиска свободных блоков?

6. Почему алгоритм работы `malloc/free` из библиотеки GNU LIbC назван аналогом алгоритма парных меток, несмотря на то, что никаких парных меток GNU LibC не использует?
7. Каково основное преимущество алгоритма близнецов?
8. Каким образом использование слабов позволяет перераспределять память между очередями блоков разного размера?
9. Назовите основные ограничения алгоритма подсчета ссылок.
10. Перечислите недостатки сборки мусора просмотром ссылок.
11. Как вы думаете, почему не применяются гибридные алгоритмы сборки мусора, сочетающие подсчет и просмотр ссылок?
12. Почему генерационные алгоритмы сборки мусора иногда называют "работающими без пауз"? Корректно ли такое название?
13. Что такое remembered set и почему без него невозможно реализовать "малую" сборку мусора?
14. Почему инкрементальную сборку мусора называют также репликационной?
15. Верно ли, что инкрементальная сборка мусора может выполняться фоновой нитью без блокировки рабочих нитей программы?
16. Почему невозможно осуществить сборку мусора с просмотром ссылок и какие бы то ни было ее варианты, в том числе генерационную и инкрементальную, без полной или хотя бы частичной остановки системы?
17. Какие проблемы может решить виртуальная память?
18. Почему даже простейшая реализация виртуальной памяти, такая как базовая адресация, требует специальных механизмов для исполнения системных вызовов?