2.5. ДИНАМИЧЕСКОЕ ВЫДЕЛЕНИЕ ПАМЯТИ

В ПРЕДЫДУЩИХ РАЗДЕЛАХ было показано, как использование связей приводит к тому, что структуры данных могут располагаться в памяти непоследовательно; несколько таблиц могут независимо расти и уменьшаться в области общего пула памяти. Однако мы всегда молчаливо предполагали, что узлы имеют один и тот же размер, т. е. каждый узел занимает некоторое фиксированное количество ячеек памяти.

Для очень многих приложений можно найти компромиссное решение, при котором в действительности используется один размер узла для всех структур (например, см. упр. 2). Вместо максимального размера (и потери памяти в малых узлах) зачастую выбирается меньший размер узла и применяется метод, который можно назвать классической *философией связанной памяти*: "Если для размещения информации в одном месте не хватает памяти, разместим ее в другом месте и установим с ней связь".

Однако для очень большого количества приложений использовать единый размер узлов неразумно, ведь часто необходимы узлы различных размеров, разделяющие общую область памяти, т. е. нужны алгоритмы для резервирования и освобождения блоков переменного размера в большой области памяти, причем эти блоки должны состоять из последовательных ячеек памяти. Такие технологии, в целом, называются алгоритмами *динамического выделения памяти*.

Зачастую в моделирующих программах требуется динамическое выделение памяти для узлов весьма малого размера (скажем, от одного до десяти слов). В других случаях, чаще всего — в операционных системах, мы, в первую очередь, работаем с довольно большими блоками информации. Такие "точки зрения" приводят к несколько отличающимся подходам к динамическому выделению памяти, хотя в этих методах много общего. Чтобы унифицировать терминологию рассматриваемых подходов, в данном разделе для обозначения множества последовательных ячеек памяти вместо термина *узел* будем использовать термины *блок* и *область*.

Некоторые авторы начиная примерно с 1975 года называют пул доступной памяти кучей (heap), однако в настоящем издании этот термин используется только в более традиционном смысле, связанном с приоритетными очередями (см. раздел 5.2.3).

А. Резервирование. На рис. 42 представлена типичная *карта памяти*, или "шахматная доска", — диаграмма, отображающая текущее состояние некоторого пула памяти. В приведенном примере она разбита на 53 блока, которые "зарезервированы" (reserved), т. е. используются вперемешку с 21 "свободным" (free) или "доступным" (available) блоком, который не используется. Память компьютера спустя некоторое время работы системы динамического выделения, вероятно, будет выглядеть примерно так. Наша первая задача состоит в поиске ответов на два вопроса.

a) Как можно представить такое разбиение свободной памяти в компьютере?
b) Какой алгоритм при данном распределении свободной памяти достаточно хорош для поиска блока из n последовательных свободных ячеек и его резервирования?

Ответ на вопрос (a) кроется, конечно, в содержании *списка* свободной памяти в некотором месте; почти всегда для этой цели лучше всего использовать ту самую свободную память, сведения о которой содержатся в списке (исключением из этого

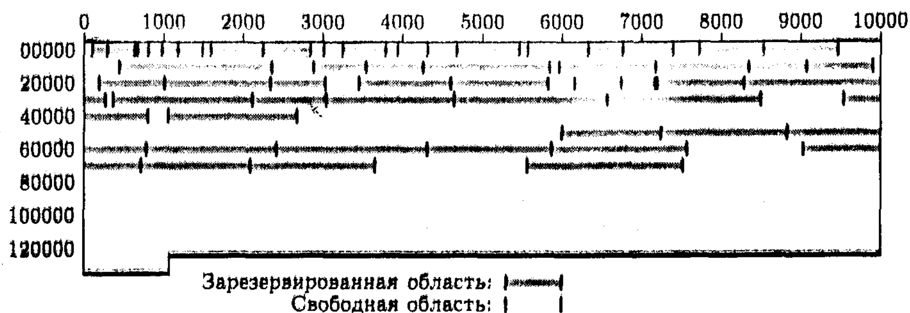


Рис. 42. Карта памяти.

правила может быть дисковая или другая память с различным временем доступа; в таком случае лучше иметь каталог доступного пространства отдельно).

Следовательно, можно *связать вместе* доступные сегменты: первое слово каждой свободной области памяти может содержать размер этого блока и адрес следующей свободной области. Свободные блоки могут быть связаны в порядке возрастания или убывания по размерам, адресам памяти либо в произвольном порядке.

Рассмотрим, например, рис. 42, на котором иллюстрируется состояние памяти объемом 131 072 слова, адресуемых от 0 до 131 071. Чтобы связать свободные блоки в порядке их адресов, потребуется переменная AVAIL, указывающая на первый свободный блок (в нашем случае AVAIL равна 0); другие же блоки будут представлены следующим образом.

Адрес	SIZE	LINK	
0	101	632	
632	42	1488	
⋮	⋮	⋮	[17 подобных записей]
73654	1909	77519	
77519	53553	Λ	[Специальный маркер для последней связи]

Следовательно, ячейки 0–100 образуют первый свободный блок; после занятых областей в ячейках 101–290 и 291–631, показанных на рис. 42, имеется свободное пространство с адресами 632–673; и т. д.

По поводу вопроса (b) понятно, что, если необходима область размером n последовательных слов, нужно выполнить поиск некоторого блока из $m \geq n$ доступных слов и уменьшить его размер до $m - n$. (Кроме того, при $m = n$ следует удалить данный блок из списка свободных.) В наличии может быть несколько блоков размером n или более ячеек, а потому один вопрос превращается в другой: "*Какая именно* область должна быть выделена?".

Два основных ответа на этот вопрос напрашиваются сами собой: можно использовать *метод наилучшего подходящего* или *метод первого подходящего*. В первом случае мы выбираем область с m ячейками памяти, где m — наименьшее значение из имеющихся, не меньшее n. Для такого выбора может потребоваться поиск по всему списку свободного пространства. Метод первого подходящего, с другой стороны, просто выбирает первую попавшуюся область размером не менее n слов.

Исторически чаще использовался метод наилучшего подходящего, так как более разумным представляется подход, сохраняющий бóльшие свободные области "на потом", когда они могут понадобиться. Однако имеется ряд претензий к этому методу: он достаточно медленный, поскольку требует длинного полного поиска, и если он не намного лучше метода первого подходящего в других отношениях, то временем выделения памяти при оценке метода пренебречь нельзя. Еще более важно то, что метод лучшего подходящего имеет тенденцию к увеличению количества блоков свободной памяти малого размера, что обычно нежелательно. Существует ряд ситуаций, в которых метод первого подходящего превосходит метод наилучшего подходящего. Так, например, предположим, что есть только две свободные области памяти размером 1300 и 1200 и три последовательных запроса на выделение блоков памяти размером 1000, 1100 и 250.

Запрос блока размером	*Доступные области, метод первого подходящего*	*Доступные области, метод наилучшего подходящего*	
—	1300, 1200	1300, 1200	(1)
1000	300, 1200	1300, 200	
1100	300, 100	200, 200	
250	50, 100	Нужного блока нет	

(Противоположный пример приведен в упр. 7.) Поскольку ни один метод явно не превосходит другой, можно порекомендовать метод первого подходящего*.

Алгоритм A (*Метод первого подходящего*). Пусть AVAIL указывает на первый доступный блок памяти, и предположим, что каждый свободный блок с адресом P имеет два поля: SIZE(P) — количество слов в блоке и LINK(P) — указатель на следующий свободный блок. Последний указатель равен Λ (что указывает на завершение списка блоков свободной памяти). Алгоритм находит и выделяет блок размером N слов (или сообщает о невозможности выделения запрошенной памяти).

A1. [Инициализация.] Установить Q $\leftarrow$ LOC(AVAIL). (Везде в алгоритме используются два указателя, Q и P, которые, вообще говоря, связаны соотношением P = LINK(Q). Мы полагаем, что LINK(LOC(AVAIL)) = AVAIL.)

A2. [Конец списка?] Установить P $\leftarrow$ LINK(Q). Если P = Λ, алгоритм завершается неудачно и блок размером N последовательных слов не может быть выделен.

A3. [Достаточен ли размер блока?] Если SIZE(P) $\geq$ N, перейти к шагу A4; в противном случае установить Q $\leftarrow$ P и вернуться к шагу A2.

A4. [Выделение блока.] Установить K $\leftarrow$ SIZE(P) − N. Если K = 0, установить LINK(Q) $\leftarrow$ LINK(P) (тем самым удаляя пустую область из списка); в противном случае установить SIZE(P) $\leftarrow$ K. Алгоритм успешно завершается, выделяя область памяти длиной N, которая начинается с адреса P + K. ▌

---

* Видимо, именно из-за отсутствия явного превосходства какого-либо метода над другим программисту во времена DOS предоставлялось право (хотя и не рекомендовалось) изменять стратегию выделения памяти операционной системой при помощи функции 58h прерывания 21h путем выбора метода первого подходящего, лучшего подходящего или последнего подходящего (включая в более поздних версиях указание на возможность использования верхней (high) памяти). *Прим. перев.*

Данный алгоритм, определенно, несколько прямолинеен. Однако всего лишь небольшое изменение стратегии может существенно повысить скорость его работы. Это весьма важное улучшение алгоритма, и читатель получит удовольствие, выполнив его поиск самостоятельно (см. упр. 6).

Алгоритм A может использоваться как для больших, так и для малых значений N. Временно предположим, однако, что нас интересуют, в первую очередь, *большие* значения N. Рассмотрим, что случится, если в алгоритме SIZE(P) равно $N+1$: перейдем к шагу A4 и уменьшим SIZE(P) до 1. Другими словами, будет создан блок доступной памяти размером 1. Он настолько мал, что практически бесполезен и только засоряет систему. Было бы лучше выделить весь блок размером $N+1$ слов вместо экономии одного слова; зачастую стóит заплатить несколькими словами памяти за избавление от несущественных деталей. Подобные примечания относятся и к блокам памяти размером $N+K$ слов при очень малом K.

Если допустить выделение блоков немного большего размера, чем N слов, придется помнить размер выделенного блока, чтобы при его освобождении корректно вернуть в пул свободной памяти все N+K слов. Это увеличивает накладные расходы, ведь придется использовать пространство в *каждом* блоке для того, чтобы сделать систему более эффективной в тех случаях, когда имеется почти подходящий блок. Так что подобная модифицированная стратегия не кажется очень привлекательной. Однако зачастую использование в начале каждого блока переменного размера специального *управляющего слова* представляется желательным по множеству других причин, так что предположение о том, что в первом слове каждого блока, как выделенного, так и свободного, присутствует поле SIZE, вполне разумно.

В соответствии с этими соглашениями можно изменить шаг A4 приведенного выше алгоритма следующим образом.

A4′. [Выделение $\geq$ N.] Установить K $\leftarrow$ SIZE(P) $-$ N. Если K $< c$ (где c — малая положительная константа, зависящая от того, каким количеством памяти мы готовы пожертвовать для ускорения работы), установить LINK(Q) $\leftarrow$ LINK(P) и L $\leftarrow$ P. В противном случае установить SIZE(P) $\leftarrow$ K, L $\leftarrow$ P$+$K, SIZE(L) $\leftarrow$ N. Алгоритм успешно завершается, выделив область длиной N или больше, которая начинается с адреса L.

Обычно значение константы c выбирается равным 8 или 10, хотя практически нет никаких теоретических или эмпирических оснований для сравнения этих значений с другими. При использовании метода наилучшего подходящего проверка K $< c$ *более важна*, чем в случае первого подходящего, поскольку компактное размещение (меньшие значения K) встречается при таком методе более часто, а число свободных блоков в этом алгоритме должно быть как можно меньше.

B. Освобождение. Теперь рассмотрим обратную задачу. Каким образом вернуть блоки в список свободного пространства, когда необходимость в них отпадает?

Пожалуй, заманчиво было бы избежать решения данной задачи, используя сборку мусора* (см. раздел 2.3.5). Следуя этой политике, мы просто ничего не делаем до тех пор, пока пространство не окажется заполненным, а затем ищем все используемые в текущий момент области памяти и строим новый список AVAIL.

* Именно этот метод работы с освобождаемой памятью используется, например, в языке Java. — *Прим. перев.*

Однако идея сборки мусора не рекомендуется для всех приложений. На первое место выходит вопрос строгой дисциплины использования указателей, если нужно гарантировать простое нахождение всех используемых в настоящий момент областей памяти, а именно этой дисциплины зачастую и не хватает рассматриваемым здесь приложениям. Во-вторых, как мы уже видели, метод сборки мусора имеет тенденцию к замедлению работы при почти заполненной памяти.

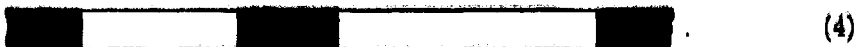
Есть и еще одна, более важная (хотя и не встречавшаяся до сих пор при обсуждении этого метода) причина, по которой сборка мусора нас не устраивает. Предположим, что имеются две соседние свободные области памяти, но из-за использования технологии сборки мусора одна из них (заштрихованная) пока не попала в список AVAIL.

(2)

На этой диаграмме черные области памяти слева и справа зарезервированы и недоступны. По запросу можно зарезервировать часть области памяти, о которой *известно*, что она свободна:

. (3)

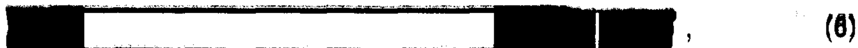
Если в этот момент произойдет сборка мусора, получатся две раздельные свободные области памяти:

 . (4)

Границы между свободной и выделенной памятью имеют тенденцию к самовоспроизводству, и со временем ситуация становится все хуже и хуже. Но если бы использовалась политика немедленного возврата освобожденных блоков памяти в список AVAIL и *слияния смежных свободных блоков*, то (2) тут же превратилась бы в

(5)

и при выделении памяти получилось бы такое распределение блоков памяти

 , (6)

которое гораздо лучше, чем (4). Как видите, рассмотренное явление вызывает большее дробление памяти при использовании метода сборки мусора, чем должно быть.

Для устранения этой проблемы можно использовать сборку мусора вместе с процессом *уплотнения памяти*, т. е. перемещения всех выделенных блоков в соседние позиции, чтобы после сборки мусора все свободные блоки были объединены. Алгоритм выделения памяти в этой ситуации становится в противоположность алгоритму A тривиальным, поскольку в любой момент есть только один свободный блок. Хотя такой подход требует времени на перемещение всех задействованных блоков и изменение в них значений связей, метод может применяться с достаточной эффективностью при условии дисциплины использования указателей и наличии запасного поля связи в каждом блоке, используемом алгоритмом сборки мусора* (см. упр. 33).

* Здесь следует отметить, что такой метод применим далеко не во всех языках программирования. При перемещении блока памяти должны быть корректно обновлены все указатели на

Поскольку многие приложения не соответствуют этим требованиям, выдвигаемым методом сборки мусора, изучим методы возврата блоков памяти в список свободного пространства. Единственная сложность в этих методах заключается в объединении соседних свободных блоков памяти. Так, когда освобождается блок, находящийся между двумя свободными блоками, все три области памяти должны быть слиты в одну. *Таким образом достигается хорошее равновесие памяти даже при длительном непрерывном процессе выделения и освобождения областей памяти.* (Для доказательства этого факта обратитесь к правилу "50%", приведенному ниже.)

В данном случае задача заключается в определении, является ли соседняя (с той или другой стороны) область памяти свободной, и если она свободна, то необходимо корректно обновить список AVAIL. Последняя операция несколько сложнее, чем кажется из ее названия.

Первое решение этой проблемы состоит в содержании списка AVAIL в порядке возрастания адресов памяти.

Алгоритм B (*Освобождение в рассортированном списке*). В предположениях алгоритма A с дополнительным предположением об упорядоченности списка AVAIL по адресам памяти (т. е. если P указывает на свободный блок и LINK(P) $\neq \Lambda$, то LINK(P) $>$ P) этот алгоритм добавляет блок из N последовательных ячеек, начинающийся с адреса P0, в список AVAIL. Естественно, предполагается, что эти N ячеек уже свободны.

B1. [Инициализация.] Установить Q $\leftarrow$ LOC(AVAIL). (См. примечание к шагу A1.)

B2. [Продвижение P.] Установить P $\leftarrow$ LINK(Q). Если P $= \Lambda$ или если P $>$ P0, перейти к шагу B3; в противном случае установить Q $\leftarrow$ P и повторить шаг B2.

B3. [Проверка верхней границы.] Если P0 $+$ N $=$ P и P $\neq \Lambda$, установить N $\leftarrow$ N $+$ SIZE(P) и установить LINK(P0) $\leftarrow$ LINK(P). В противном случае установить LINK(P0) $\leftarrow$ P.

B4. [Проверка нижней границы.] Если Q $+$ SIZE(Q) $=$ P0 (предполагается, что

$$\mathtt{SIZE(LOC(AVAIL))} = 0,$$

так что условие всегда не выполняется при Q $=$ LOC(AVAIL)), установить SIZE(Q) $\leftarrow$ SIZE(Q) $+$ N и LINK(Q) $\leftarrow$ LINK(P0). В противном случае установить LINK(Q) $\leftarrow$ P0, SIZE(P0) $\leftarrow$ N. ▮

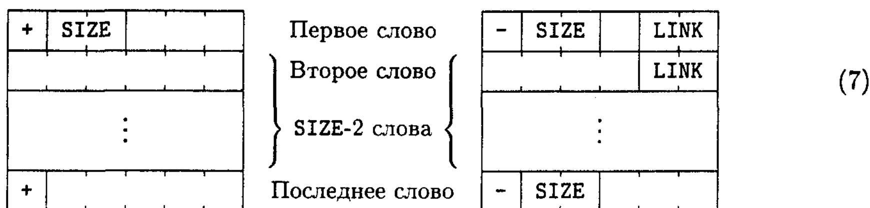
На шагах B3 и B4 выполняется требуемое слияние; учитывается тот факт, что указатели Q $<$ P0 $<$ P являются начальными адресами трех последовательных свободных блоков.

этот блок (и внутрь него). Например, в случае *Java* такой метод может применяться, поскольку в языке отсутствует понятие указателя и реальная адресация блоков памяти переменными может быть отслежена внутренними средствами языка. В то же время в языках наподобие *C* и *Pascal*, в которых используются указатели, отследить создание указателя на ту или иную область памяти невозможно, а значит, применение метода ограничено его использованием только в менеджерах памяти программ, созданных специальным образом с учетом требований такого менеджера (с дополнительными средствами регистрации всех указателей на выделяемые блоки памяти). (Вопросы косвенной адресации и защищенного режима процессоров не упоминаем, так как они не связаны с тематикой книги.) — *Прим. перев.*

Если список AVAIL не упорядочен, нетрудно понять, что попытка слияния блоков списка "в лоб" приведет к просмотру всего списка AVAIL; алгоритм B в среднем уменьшает этот поиск до примерно *половины* списка AVAIL (на шаге B2). В упр. 11 показано, каким образом можно модифицировать алгоритм B, чтобы в среднем потребовалось просмотреть около одной трети списка AVAIL. Но ясно, что, когда список AVAIL длинный, все эти методы работают существенно медленнее, чем хотелось бы. Нет ли еще какого-либо способа выделения и освобождения памяти, при котором не требовался бы столь пространный поиск в списке AVAIL?

Рассмотрим метод, который позволяет устранить любой поиск при освобождении памяти и может быть модифицирован (см. упр. 6) для резкого сокращения времени поиска при выделении памяти. В этой технологии используется поле TAG в начале и в конце блока, а также поле SIZE в первом слове каждого блока. Такие накладные расходы не столь значительны при использовании больших блоков памяти, хотя, возможно, цена окажется слишком большой при наличии множества блоков очень малого среднего размера. Другой метод, описанный в упр. 19, требует только одного бита в первом слове каждого блока. Ценой этой экономии является несколько большее время работы и усложнение программы*.

Как бы там ни было, положим, что нет возражений против добавления некоторого количества битов управляющей информации для сохранения высокой скорости работы по сравнению с алгоритмом B при длинном списке AVAIL. В описываемом методе предполагается, что каждый блок имеет следующий вид.



(7)

Идея следующего алгоритма состоит в поддержании двусвязного списка AVAIL, так что элементы списка могут быть легко удалены из произвольной части списка. Поле TAG с обоих концов блока можно использовать для управления процессом слияния, поскольку оно позволяет легко определить, свободен ли смежный блок памяти.

Двойное связывание достигается обычным путем: LINK в первом слове указывает на следующий доступный блок в списке, а LINK во втором слове указывает на предыдущий доступный блок. Таким образом, если P — адрес блока, то

$$\mathtt{LINK}(\mathtt{LINK}(\mathtt{P})+1)=\mathtt{P}=\mathtt{LINK}(\mathtt{LINK}(\mathtt{P}+1)). \tag{8}$$

---

* Этот принцип положен, в частности, в основу управления памятью в MS DOS, где каждый блок памяти (MCB — Memory Control Block) имеет поля размера, владельца и типа блока. Менеджер памяти, основанный на списке блоков, использовался, например, в *Turbo Pascal*. — *Прим. перев.*

Для корректности "граничных условий" заголовок списка устанавливается следующим образом.

Алгоритм для выделения первого подходящего блока очень похож на алгоритм A, и потому здесь не рассматривается (см. упр. 12). Принципиально новое свойство этого метода заключается в освобождении блока за, по сути, фиксированное время.

Алгоритм C (*Освобождение с дескрипторами границ*) Предположим, что блоки памяти выглядят так, как в (7), а список AVAIL имеет две связи, как описывалось выше. Данный алгоритм помещает блок памяти с начальным адресом P0 в список AVAIL. Если пул доступной памяти располагается от адреса m_0 до m_1 включительно, в алгоритме для удобства предполагается, что

$$\mathtt{TAG}(m_0 - 1) = \mathtt{TAG}(m_1 + 1) = \text{“+”}.$$

C1. [Проверка нижней границы.] Если TAG(P0 − 1) = "+", перейти к шагу C3.

C2. [Удаление нижней области.] Установить P ← P0 − SIZE(P0 − 1), а затем установить P1 ← LINK(P), P2 ← LINK(P + 1), LINK(P1 + 1) ← P2, LINK(P2) ← P1, SIZE(P) ← SIZE(P) + SIZE(P0), P0 ← P.

C3. [Проверка верхней границы.] Установить P←P0+SIZE(P0). Если TAG(P)="+", перейти к шагу C5.

C4. [Удаление верхней границы.] Установить P1 ← LINK(P), P2 ← LINK(P+1), LINK(P1+1) ← P2, LINK(P2) ← P1, SIZE(P0) ← SIZE(P0) + SIZE(P), P ← P + SIZE(P).

C5. [Добавление в список AVAIL.] Установить SIZE(P − 1) ← SIZE(P0), LINK(P0) ← AVAIL, LINK(P0 + 1) ← LOC(AVAIL), LINK(AVAIL + 1) ← P0, AVAIL ← P0, TAG(P0) ← TAG(P − 1) ← "−". ▌

Шаги алгоритма C определяются видом блоков памяти (7); немного более длинный, но одновременно более быстрый алгоритм можно найти в упр. 15. На шаге C5 AVAIL означает аббревиатуру для LINK(LOC(AVAIL)), как показано в (9).

C. "Система двойников". Изучим теперь другой подход к динамическому выделению памяти для использования на двоичных компьютерах. В этом методе применяется по одному дополнительному биту на каждый блок; кроме того, все блоки должны иметь длину 1, 2, 4, 8, 16 и т. д. Если длина блока не равна 2^k слов для некоторого целого k, выбирается следующая более высокая степень 2 и часть выделенной памяти при этом не используется.

Суть этого метода заключается в организации отдельных списков доступных блоков каждого размера 2^k, $0 \le k \le m$. Весь пул распределяемого пространства памяти состоит из 2^m слов, адреса которых, предположим, находятся в диапазоне от 0 до $2^m - 1$. Изначально весь блок из 2^m слов свободен. Позже, при требовании блока из 2^k слов и отсутствии свободного блока такого размера больший доступный блок *разбивается* (*split*) на две равные части; в конечном итоге появится блок

необходимого размера 2^k. Когда один блок разделяется на два (каждый половинного размера по сравнению с исходным), эти блоки называются *двойниками* (*buddies**). Позже, когда оба двойника вновь становятся свободны, они объединяются в один большой блок. Таким образом, процесс выделения и освобождения может осуществляться бесконечно, если только в какой-то момент вся доступная память не окажется занятой.

Ключевой факт, лежащий в основе практической ценности этого метода, состоит в том, что если известен адрес блока (адрес первого слова в блоке) и его размер, то известен и адрес его двойника. Например, двойником блока размером 16 с двоичным адресом 101110010110000 является блок с двоичным адресом 101110010100000. Для того чтобы понять, почему это так, сначала заметим, что во время работы алгоритма *адрес блока размером* 2^k *кратен* 2^k. Другими словами, адрес в двоичной записи содержит справа как минимум k нулей. Данное наблюдение легко выводится по индукции: если это верно для всех блоков размером 2^{k+1}, то это, несомненно, справедливо и при делении блока пополам.

Значит, блок размером, скажем, 32 имеет адрес вида $xx\ldots x00000$ (где иксы представляют собой 0 или 1); при разделении блока вновь образуемые блоки-двойники имеют адреса $xx\ldots x00000$ и $xx\ldots x10000$. В общем, пусть

$$\text{двойник}_k(x)$$

обозначает адрес двойника блока размером 2^k, адрес которого равен x. Тогда находим, что

$$\text{двойник}_k(x) = \begin{cases} x + 2^k, & \text{если } x \bmod 2^{k+1} = 0; \\ x - 2^k, & \text{если } x \bmod 2^{k+1} = 2^k. \end{cases} \tag{10}$$

Эта функция легко вычисляется с помощью операции исключающее или (иногда называемой селективным дополнением или сложением без переноса), обычно имеющейся на двоичном компьютере (см. упр. 28).

Система двойников использует однобитовое поле `TAG` в каждом блоке:

$$\begin{aligned} &\texttt{TAG(P)} = 0, && \text{если блок с адресом P выделен;} \\ &\texttt{TAG(P)} = 1, && \text{если блок с адресом P свободен.} \end{aligned} \tag{11}$$

Кроме поля `TAG`, имеющегося в каждом блоке, в *свободных* блоках есть два поля связи, `LINKF` и `LINKB`, которые представляют обычные связи вперед и назад в двусвязном списке; также имеется поле `KVAL`, определяющее k для блока размером 2^k. В приведенном ниже алгоритме используются ячейки таблицы `AVAIL[0]`, `AVAIL[1]`, ..., `AVAIL[`m`]`, которые служат соответственно в качестве заголовков списков свободной памяти размером 1, 2, 4, ..., 2^m. Это списки с двойными связями, так что, как обычно, заголовок списка содержит два указателя (см. раздел 2.2.5):

$$\begin{aligned} &\texttt{AVAILF}[k] = \texttt{LINKF(LOC(AVAIL}[k]\texttt{))} = \text{связь с окончанием списка } \texttt{AVAIL}[k]; \\ &\texttt{AVAILB}[k] = \texttt{LINKB(LOC(AVAIL}[k]\texttt{))} = \text{связь с началом списка } \texttt{AVAIL}[k]. \end{aligned} \tag{12}$$

Изначально перед выделением памяти мы имеем

$$\begin{aligned} \texttt{AVAILF}[m] = \texttt{AVAILB}[m] &= 0, \\ \texttt{LINKF(0)} = \texttt{LINKB(0)} &= \texttt{LOC(AVAIL}[m]\texttt{)}, \\ \texttt{TAG(0)} = 1, \qquad \texttt{KVAL(0)} &= m \end{aligned} \tag{13}$$

---

* Дословный перевод слова *buddy* — *дружище, приятель.* — *Прим. перев.*

(что указывает на единственный свободный блок длиной 2^m, начинающийся по адресу 0) и

$$\texttt{AVAILF[}k\texttt{]} = \texttt{AVAILB[}k\texttt{]} = \texttt{LOC(AVAIL[}k\texttt{])} \qquad \text{для } 0 \le k < m \tag{14}$$

(что указывает на пустые списки свободных блоков размером 2^k для всех $k < m$).

Исходя из этого описания системы двойников, читатель может самостоятельно и не без определенного удовольствия разработать необходимые алгоритмы для выделения и освобождения областей памяти, прежде чем знакомиться с приведенными далее алгоритмами. Обратите внимание на сравнительную простоту, с которой каждый блок может быть разделен пополам в алгоритме для выделения памяти.

Алгоритм R (*Выделение памяти в системе двойников*). Этот алгоритм предназначен для поиска и выделения блока памяти размером 2^k (или сообщения о невозможности такого выделения) с помощью описанной выше системы двойников.

R1. [Поиск блока.] Пусть j — наименьшее целое число в диапазоне $k \le j \le m$, для которого AVAILF[j] $\neq$ LOC(AVAIL[j]), т. е. для которого список свободных блоков размером 2^j не пуст. Если такого j не существует, алгоритм завершается неудачей, поскольку нет ни одного блока достаточного размера для выделения запрошенного количества памяти.

R2. [Удаление из списка.] Установить L $\leftarrow$ AVAILF[j], P $\leftarrow$ LINKF(L), AVAILF[j] $\leftarrow$ P, LINKB(P) $\leftarrow$ LOC(AVAIL[j]) и TAG(L) $\leftarrow$ 0.

R3. [Требуется разделение?] Если $j = k$, алгоритм завершается (найден и выделен свободный блок, начинающийся с адреса L).

R4. [Разделение.] Уменьшить j на 1. Затем установить P $\leftarrow$ L $+ 2^j$, TAG(P) $\leftarrow$ 1, KVAL(P) $\leftarrow j$, LINKF(P) $\leftarrow$ LINKB(P) $\leftarrow$ LOC(AVAIL[j]), AVAILF[j] $\leftarrow$ AVAILB[j] $\leftarrow$ P. (Тем самым разделяется большой блок памяти и неиспользуемая половина вносится в список AVAIL[j], который был пуст.) Вернуться к шагу R3. ▌

Алгоритм S (*Освобождение памяти в системе двойников*). Этот алгоритм предназначен для возврата блока размером 2^k, начинающегося с адреса L, в область свободной памяти с использованием описанной выше системы двойников.

S1. [Свободен ли двойник?] Установить P $\leftarrow$ двойник$_k$(L) (см. (10)). Если $k = m$ *или* TAG(P) $= 0$, *либо если* TAG(P) $= 1$ *и* KVAL(P) $\neq k$, перейти к шагу S3.

S2. [Объединение двойников.] Установить

$$\texttt{LINKF(LINKB(P))} \leftarrow \texttt{LINKF(P)}, \qquad \texttt{LINKB(LINKF(P))} \leftarrow \texttt{LINKB(P)}.$$

(Таким образом блок P удаляется из списка AVAIL[k].) Затем установить $k \leftarrow k + 1$ и, если P $<$ L, установить L $\leftarrow$ P. Вернуться к шагу S1.

S3. [Размещение в списке.] Установить TAG(L) $\leftarrow$ 1, P $\leftarrow$ AVAILF[k], LINKF(L) $\leftarrow$ P, LINKB(P) $\leftarrow$ L, KVAL(L) $\leftarrow k$, LINKB(L) $\leftarrow$ LOC(AVAIL[k]), AVAILF[k] $\leftarrow$ L. (Таким образом блок L помещается в список AVAIL[k].) ▌

D. Сравнение методов. Выполнение математического анализа этих алгоритмов динамического выделения памяти оказывается весьма трудной задачей, однако

имеется одно интересное явление, которое легко проанализировать, а именно — правило “50%”.

> *Если алгоритмы A и B непрерывно используются таким образом, что система стремится к равновесию, при котором в ней имеется в среднем N выделенных блоков, каждый из которых выделяется и освобождается независимо от других, а величина K в алгоритме A принимает ненулевое значение (или, более того, значения $\geq c$, как на шаге A4′) с вероятностью p, то среднее количество свободных блоков стремится приблизительно к $\frac{1}{2}pN$.*

Это правило говорит о том, какой будет приблизительная длина списка AVAIL. Когда величина p близка к 1, что случается при очень малых c и если размеры блоков редко равны, количество свободных блоков составляет примерно половину от количества занятых; отсюда и происходит название правила “50%”.

Данное правило нетрудно доказать. Рассмотрим следующую карту памяти:

На ней показаны выделенные блоки памяти трех категорий:

A: при освобождении блока количество свободных блоков уменьшается на единицу;
B: при освобождении блока количество свободных блоков не изменяется;
C: при освобождении блока количество свободных блоков увеличивается на единицу.

Теперь пусть N — число выделенных, а M — число свободных блоков. Пусть A, B и C — количество блоков описанных выше типов. Имеем

$$N = A + B + C;$$
$$M = \tfrac{1}{2}(2A + B + \epsilon), \qquad (15)$$

где $\epsilon = 0$, 1 или 2 в зависимости от условий на нижней и верхней границах.

Предположим, что N представляет, по существу, константу, а A, B, C и ϵ — случайные величины, которые достигают стационарного распределения после освобождения блока и несколько отличающегося стационарного распределения после выделения блока. Среднее изменение величины M при освобождении блока равно среднему значению $(C-A)/N$; среднее изменение величины M при выделении блока равно $1 - p$. Таким образом, с учетом достижения состояния равновесия получаем, что $C - A - N + pN = 0$. Однако тогда среднее значение величины $2M$ равно pN плюс среднее значение ϵ, поскольку $2M = N + A - C + \epsilon$ согласно (15). Отсюда следует правило “50%”.

Наши предположения о том, что каждое удаление применяется к случайному выделенному блоку, будет справедливо, если время жизни блока представляет собой экспоненциально распределенную случайную величину. С другой стороны, если все блоки имеют примерно одно и то же время жизни, сделанное предположение ложно. Джон Э. Шор (John E. Shore) указал, что блоки типа A обычно “старее” блоков типа C, если характер процесса выделения и освобождения близок к очереди (“первым вошел — первым вышел”; FIFO), поскольку последовательность смежных блоков при этом стремится выстроиться в порядке от “младших” блоков к “старшим” и последний выделенный блок почти никогда не оказывается блоком типа A. Такая

тенденция приводит к наличию меньшего числа доступных блоков, давая даже лучшую производительность по сравнению с производительностью, предсказываемой по правилу "50%" [см. *CACM* **20** (1977), 812–820].

Более детально правило "50%" рассматривается в работах D. J. M. Davies, *BIT* *20* (1980), 279–288; C. M. Reeves, *Comp. J.* **26** (1983), 25–35; G. Ch. Pflug, *Comp. J.* **27** (1984), 328–333.

Если не учитывать это интересное правило, *наши знания о производительности* алгоритмов динамического распределения памяти почти полностью базируются на экспериментах по методу Монте-Карло. При выборе алгоритма выделения памяти для конкретных машины и класса приложения (или конкретного приложения) поучительно провести собственные моделирующие динамическое распределение памяти эксперименты. Автор провел ряд таких экспериментов непосредственно перед написанием этого раздела (и правило "50%" было замечено во время этих экспериментов до того, как было найдено его доказательство). Рассмотрим здесь вкратце методы и результаты проведенных экспериментов.

Основная моделирующая программа работает следующим образом. В начальный момент работы программы, когда значение TIME равно нулю, доступна вся память.

P1. Увеличить TIME на 1.

P2. Освободить все блоки в системе, которые должны быть освобождены при текущем значении TIME.

P3. Вычислить две величины, S (случайный размер) и T (случайное время жизни), основанные на некотором распределении вероятностей, с помощью методов из главы 3.

P4. Выделить новый блок длиной S, который необходимо освободить в момент ($\mathtt{TIME} + T$). Вернуться к шагу P1. ▌

Каждый раз по достижении переменной TIME значения, кратного 200, выводились детальные статистические данные о производительности алгоритмов выделения и освобождения. Для каждой пары алгоритмов использовалась одна и та же последовательность значений S и T. После того как величина TIME превышала 2000, система обычно приходила в более или менее стабильное состояние с неизменными показателями. Однако иногда на шаге P3 алгоритм прекращал свою работу из-за невозможности выделить необходимый объем памяти.

Пусть C — общее количество доступных ячеек памяти и пусть $\bar{S}$ и $\bar{T}$ — средние величины S и T на шаге P3. Легко видеть, что ожидаемое количество занятых слов памяти в каждый момент составляет $\bar{S}\bar{T}$ при достаточно большом значении TIME. Когда в экспериментах $\bar{S}\bar{T}$ было больше, чем $\frac{2}{3}C$, обычно происходило переполнение памяти (часто прежде, чем в действительности происходил запрос на выделение C слов памяти). Память можно было заполнить на 90% при малом по сравнению с C размере блока, но когда размеры блока могли превысить $\frac{1}{3}C$ (вместе с блоками меньших размеров), наблюдалась тенденция программы к "заполнению" памяти при реальном использовании менее $\frac{1}{2}C$ ячеек памяти. Эмпирические результаты свидетельствуют о том, что *для эффективной работы систем динамического распределения памяти не должны использоваться блоки размером свыше* $\frac{1}{10}C$.

Причину такого поведения можно понять, если вспомнить правило "50%": если система достигает состояния равновесия, при котором размер среднего свободного блока f меньше размера среднего используемого блока r, то можно ожидать получения невыполнимого запроса, кроме ситуации, когда "на всякий пожарный случай" имеется достаточно большой свободный блок. Следовательно, в насыщенных системах без переполнения $f \geq r$, и мы имеем $C = fM + rN \geq rM + rN \approx \left(\frac{1}{2}p + 1\right)rN$. Общее количество использованной памяти, таким образом, составляет $rN \leq C/\left(\frac{1}{2}p+1\right)$. Когда $p \approx 1$, использовать больше примерно $\frac{2}{3}$ ячеек памяти невозможно.

Эксперименты по моделированию систем динамического выделения памяти проводились с тремя распределениями размера блоков S:

(*S1*) равновероятный выбор целого числа из диапазона от 100 до 2000;

(*S2*) выбор размера (1, 2, 4, 8, 16, 32) с вероятностями $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{32})$ соответственно;

(*S3*) равновероятный выбор размера из множества (10, 12, 14, 16, 18, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 500, 1000, 2000, 3000, 4000).

Время T обычно представляло собой случайное целое равномерно распределенное число в диапазоне от 1 до t для фиксированного $t = 10$, 100 или 1000.

Кроме того, были проведены эксперименты, в которых T на шаге P3 представляло собой случайное число, равномерно распределенное в диапазоне от 1 до $\min\left(\left\lfloor \frac{5}{4}U \right\rfloor, 12500\right)$, где U — количество единиц времени, оставшегося до ближайшего освобождения занятого блока из имеющихся в настоящий момент в системе. Это распределение времени использовалось для моделирования ситуации "почти последним выделен — первым освобожден" (almost-last-in-first-out): если T всегда выбирается $\leq U$, система выделения памяти вырождается в простую стековую операцию, не требующую использования сложных алгоритмов (см. упр. 1). Указанное распределение вынуждает выбранное T быть бо́льшим, чем U, около 20% раз, так что мы имеем *почти* стековую операцию. При использовании такого распределения алгоритмы, подобные алгоритмам А, В и С, ведут себя гораздо лучше, чем обычно; очень редко возникало (если возникало вообще) более двух блоков в списке `AVAIL`, в то время как выделялось около 14 блоков. С другой стороны, алгоритмы системы двойников, R и S, при использовании такого распределения оказывались медленнее, поскольку в стекообразных операциях приходится чаще, чем обычно, расщеплять и сливать блоки. Вывод теоретических свойств этих распределений слишком сложен (см. упр. 32).

На рис. 42 была приведена конфигурация памяти в момент `TIME` = 5000 с использованием распределения размеров (*S1*) и с равномерно распределенным между 1 и 100 временем при выделении по методу первого подходящего (алгоритмы А и В). В этом эксперименте вероятность p, которая входит в правило "50%", равна 1, так что можно было бы ожидать, что количество свободных блоков составит около половины количества выделенных блоков. На самом деле на рис. 42 показан 21 свободный и 53 выделенных блока. Это не опровергает правило "50%"; например, в момент `TIME` = 4600 имелось 25 свободных и 49 выделенных блоков. Конфигурация, приведенная на рис. 42, просто показывает, что правило "50%" подвержено статистическим отклонениям. Число свободных блоков, в целом, находится в диа-

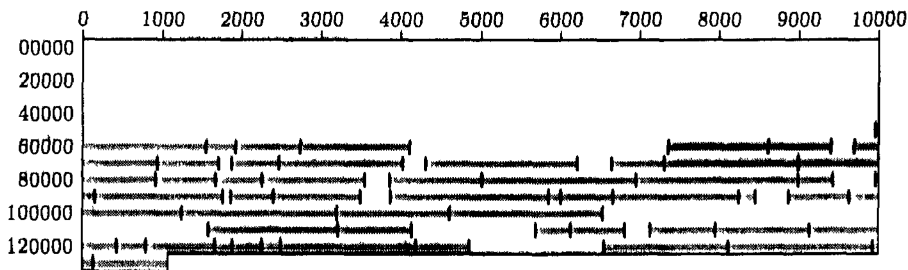


Рис. 43. Карта памяти, полученная с помощью метода наилучшего подходящего. (Сравните ее с картой, представленной на рис. 42, которая показывает результат работы метода первого подходящего, и с рис. 44, на котором в виде дерева представлена карта памяти в результате работы системы двойников с той же последовательностью запросов.)

пазоне от 20 до 30, в то время как количество выделенных блоков находится в диапазоне от 45 до 55.

На рис. 43 приведена конфигурация памяти, полученная *при тех же данных, что и конфигурация на рис. 42*, но с использованием метода наилучшего подходящего. Константа c из шага A4′ была принята равной 16 для устранения малых блоков, и как результат — вероятность p упала до 0.7 и количество свободных областей уменьшилось.

При изменении распределения времени в диапазоне от 1 до 1000 (вместо диапазона от 1 до 100) ситуация была аналогична показанной на рис. 42 и 43, но все соответствующие величины были умножены приблизительно на 10. Например, в ситуации, аналогичной показанной на рис. 42, было 515 выделенных и 240 свободных блоков; в ситуации, подобной приведенной на рис. 43, имелось 176 свободных блоков.

Во всех экспериментах по сравнению методов первого подходящего и наилучшего подходящего последний всегда превосходит первый. После исчерпания размера памяти метод первого подходящего в большинстве случаев остается работоспособным дольше метода наилучшего подходящего.

Система двойников испытывалась с теми данными, которые привели к результатам, изображенным на рис. 42 и 43. Итоговые данные представлены на рис. 44. Здесь все размеры в диапазоне от 257 до 512 рассматривались как 512, в диапазоне от 513 до 1024 — как 1024 и т. д. В среднем это означает увеличение запроса на одну треть (см. упр. 21); система двойников, конечно, лучше работает при распределении размеров (*S2*), а не при распределении (*S1*). Обратите внимание на то, что на рис. 44 имеются доступные блоки размеров 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13} и 2^{14}.

Моделирование системы двойников показало, что ее производительность выше, чем ожидалось. Ясно, что иногда эта система позволяет иметь два смежных свободных блока одинакового размера без их слияния (если они не являются двойниками). Но такая ситуация не представлена на рис. 44 и в действительности крайне редка на практике. Переполнение памяти происходило при выделении 95% памяти, что отражает удивительно хороший баланс выделения памяти. Кроме того, очень редко требуется разделение блоков в алгоритме R (и их слияние в алгоритме S); дерево остается очень похожим на изображенное на рис. 44 со свободными блоками на чаще всего используемых уровнях. Некоторые математические результаты, позво-

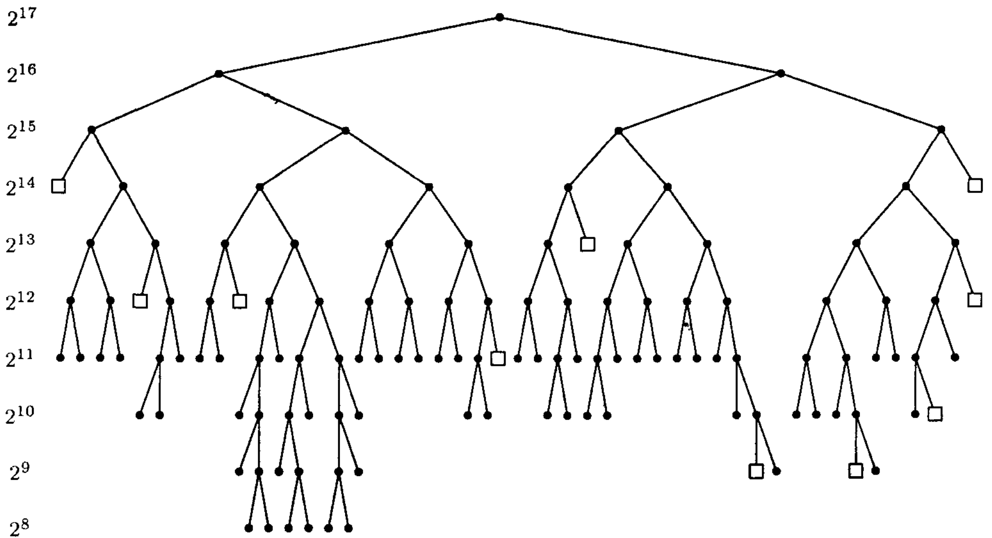


Рис. 44. Карта памяти, полученная при помощи системы двойников. (Структура дерева указывает на деление некоторых больших блоков на двойников половинного размера. Квадратами помечены свободные блоки.)

ляющие понять такое поведение на нижних уровнях дерева, были получены в работе P. W. Purdom, Jr., and S. M. Stigler, *JACM* **17** (1970), 683–697.

Еще одним сюрпризом стало превосходное поведение алгоритма A после модификации, описанной в упр. 6. Потребовалось в среднем только 2.8 проверки размера свободного блока при использовании распределения размеров (*S1*) и времени, равномерно распределенного между 1 и 1000, а более чем в половине случаев требовалось минимальное значение — одна итерация. Все это выполнялось несмотря на наличие около 250 доступных блоков. Тот же эксперимент с немодифицированным алгоритмом A показал, что в среднем необходимо около 125 итераций (т. е. каждый раз проверяется половина списка `AVAIL`); 200 и более проверок понадобилось примерно в каждом пятом случае.

Такое поведение немодифицированного алгоритма A в действительности может быть предсказано как следствие правила "50%". При равновесии в части памяти, содержащей последнюю половину выделенных блоков, находится также последняя половина свободных блоков; эта часть требует половину времени при освобождении блока и соответственно должна использоваться в половине случаев выделения памяти для поддержки состояния равновесия. Это же обоснование применимо и при замене половины любой другой частью (данные замечания были сделаны Д. М. Робсоном (J. M. Robson)).

Упражнения к этому разделу включают `MIX`-программы для двух основных методов, которые рекомендуются к использованию на основе приведенных выше замечаний: (i) модифицированный согласно упр. 12 и 16 алгоритм A (система граничных дескрипторов) и (ii) система двойников. Вот приближенные результаты

работы этих методов.

	Время выделения	Время освобождения
Система граничных дескрипторов:	$33 + 7A$	18, 29, 31 или 34
Система двойников:	$19 + 25R$	$27 + 26S$

Здесь $A \geq 1$ — необходимое для поиска достаточно большого блока количество итераций, $R \geq 0$ — количество разбиений блока на два (начальная разность $j - k$ в алгоритме R) и $S \geq 0$ — количество слияний двойников при работе алгоритма S. Моделирующие эксперименты указывают, что при данных предположениях относительно распределения размеров (*S1*) и времени, выбираемом в диапазоне от 1 до 1000, можно получить в среднем $A = 2.8$, $R = S = 0.04$. (Средние значения для распределения времени "почти последним выделен — первым освобожден", которое описано выше, составляют $A = 1.3$, $R = S = 0.9$.) Это показывает, что оба метода весьма быстры (система двойников для MIX несколько быстрее). Напомним, что для системы двойников необходимо примерно на 44% больше памяти, если размеры блоков не ограничены степенями 2.

Соответствующее время для выполнения алгоритма сборки мусора и уплотнения из упр. 33 составляет около 104 единиц при поиске свободного узла, если предположить, что сборка мусора происходит в момент заполненности памяти примерно наполовину, а узлы имеют среднюю длину 5 слов с двумя связями на узел. "За" и "против" этого метода обсуждаются в разделе 2.3.5. Когда память не сильно загружена и выполняются соответствующие ограничения, сборка мусора и уплотнение весьма эффективны; например, на компьютере MIX метод сборки мусора быстрее двух других, если память не заполнена более чем на треть и узлы относительно малы.

Если удовлетворяются условия, лежащие в основе метода сборки мусора, наилучшей стратегией может оказаться разделение пула памяти на две половины и выполнение всех последующих выделений памяти в одной половине. Вместо освобождения становящихся неиспользуемыми блоков мы просто ожидаем, пока текущая половина памяти не заполнится. Затем можно скопировать все активные данные в другую половину, одновременно удаляя все "дыры" между блоками при помощи метода, подобного приведенному в упр. 33. Размер каждой половины пула может изменяться при переключении с одной половины на другую.

Упомянутые выше технологии были применены и к другим алгоритмам выделения памяти. Однако эти методы оказались настолько плохи по сравнению с алгоритмами, описанными в настоящем разделе, что здесь будет дано только их краткое описание.

a) Для каждого размера используется свой список AVAIL. Единый свободный блок при необходимости разбивается на два меньших блока, но никаких попыток вновь объединить такие блоки не предпринимается. Карта памяти становится фрагментированной на все меньшие и меньшие части, пока не принимает совершенно ужасный вид. Простая схема наподобие этой практически эквивалентна раздельному распределению в несвязанных областях, по одной области для каждого размера блока.

b) Была предпринята попытка выполнения двухуровневого выделения. Память делилась на 32 больших сектора. Для выделения больших блоков размером 1, 2

или 3 (очень редко — бо́льших размеров) соседних секторов использовался метод выделения памяти "в лоб". Каждый такой большой блок разделялся для удовлетворения запросов на выделение памяти до тех пор, пока в текущем большом блоке не оставалось памяти (при этом начинал использоваться другой большой блок). Каждый большой блок возвращался в свободную память только после освобождения *всех* выделенных из него блоков памяти. Данный метод почти всегда быстро приводит к нехватке памяти.

Хотя этот частный метод двухуровневого выделения и был неработоспособен с использовавшимися автором данными при моделировании динамического распределения, возникают ситуации (на практике очень нечастые), когда многоуровневая стратегия может оказаться предпочтительной. Например, если большая программа работает в несколько стадий, возможно, что на разных стадиях используются узлы разных типов и отдельные типы узлов применяются только в отдельных подпрограммах. В некоторых программах может оказаться желательным использование нескольких различных стратегий распределения памяти для различных классов узлов. Идея выделения памяти по зонам, быть может, с помощью различных стратегий в каждой зоне и с возможностью освобождения всей зоны, рассматривается в работе Douglas T. Ross, *CACM* **10** (1967), 481–492.

Другие эмпирические результаты, полученные при изучении динамического распределения памяти, приводятся в следующих работах: B. Randell, *CACM* **12** (1969), 365–369, 372; P. W. Purdom, S. M. Stigler, and T. O. Cheam, *BIT* **11** (1971), 187–195; B. H. Margolin, R. P. Parmelee, and M. Schatzoff, *IBM Systems J.* **10** (1971), 283–304; J. A. Campbell, *Comp. J.* **14** (1971), 7–9; John E. Shore, *CACM* **18** (1975), 433–440; Norman R. Nielsen, *CACM* **20** (1977), 864–873.

***E. Метод распределенного подходящего.** Если распределение размеров блоков известно заранее и все блоки освобождаются равновероятно, независимо от момента их выделения, можно использовать технологию (которая в данных предположениях превосходит технологии общего назначения), предложенную Э. Г. Коффманом (мл.) (E. G. Coffman, Jr.) и Ф. Т. Лейтоном (F. T. Leighton) [*J. Computer and System Sci.* **38**, (1989), 2–35]. Их "метод распределенного подходящего" (distributed-fit method) работает путем разделения памяти на примерно $N + \sqrt{N}\lg N$ слотов, где N — максимальное число блоков, которые будут обрабатываться в установившемся состоянии. Каждый слот имеет фиксированный размер, хотя различные слоты могут иметь различные размеры. Главное заключается в том, что любой слот имеет фиксированные границы, и он может либо быть пустым, либо содержать единственный выделенный блок.

Первые N слотов схемы Коффмана-Лейтона расположены в соответствии с заданным распределением размеров, а оставшиеся $\sqrt{N}\lg N$ слотов имеют максимальный размер. Например, если предположить, что размеры блоков равномерно распределены между 1 и 256 и если необходимо обработать $N = 2^{14}$ таких блоков, следует разделить память на $N/256 = 2^6$ слотов каждого размера 1, 2, ..., 256, за которыми следует "область переполнения", содержащая $\sqrt{N}\lg N = 2^7 \cdot 14 = 1792$ блоков размером 256. Когда система работает при полной загрузке, ожидается работа с N блоками среднего размера $\frac{257}{2}$, занимающими $\frac{257}{2}N = 2^{21} + 2^{13} = 2\,105\,344$ ячеек памяти (это количество памяти, выделенное для первых N слотов). Кроме

того, имеется $1792 \cdot 256 = 458\,752$ ячеек памяти для обработки случайных отклонений; эти дополнительные накладные расходы составляют $O(N^{-1/2} \log N)$ от общего объема памяти, в отличие от пропорциональных N в случае системы двойников, и становятся незначимыми при $N \to \infty$. В нашем примере, однако, накладные расходы остаются на уровне 18% от общего количества памяти.

Слоты должны быть расположены в таком порядке, чтобы меньшие из них предшествовали бо́льшим. При таком расположении можно выделять блоки с помощью либо технологии первого подходящего, либо технологии наилучшего подходящего (в данном случае оба метода эквивалентны, поскольку размеры слотов упорядочены). При наших предположениях действие описанной методики заключается в начале поиска в, по сути, случайном месте среди первых N слотов при новом запросе на выделение памяти и его продолжении до тех пор, пока не будет найден пустой слот.

Если начальный слот для каждого поиска действительно выбирается случайным образом между 1 и N, частых вторжений в область переполнения не будет. В самом деле, если вставить ровно N элементов, начиная со случайных слотов, переполнение будет встречаться в среднем только $O(\sqrt{N})$ раз. Объяснение этого факта заключается в том, что можно сравнить данный алгоритм с хешированием с линейным исследованием (алгоритм 6.4L), которое имеет то же самое поведение, но поиск пустой ячейки возвращается от N к 1 вместо перехода в область переполнения. Анализ алгоритма 6.4L в теореме 6.4K показывает, что при вставке N элементов среднее смещение каждого элемента от его хеш-адреса составляет $\frac{1}{2}(Q(N) - 1) \sim \sqrt{\pi N/8}$. Из круговой симметрии следует, что это среднее значение — то же, что и среднее количество переходов поиска от слота k к слоту $k+1$ для каждого k. Переполнения в методе распределенного подходящего соответствуют поискам, переходящим от слота N к слоту 1, с той лишь разницей, что наша ситуация даже лучше, поскольку некоторого переполнения можно избежать без возвратов к началу. Таким образом, в среднем встречается менее $\sqrt{\pi N/8}$ переполнений. В этом анализе не приняты во внимание удаления, которые сохраняют предположения алгоритма 6.4L только в случае, когда мы перемещаем блоки назад при удалении другого блока, находящегося между их начальными и выделенными слотами (см. алгоритм 6.4R). Кроме того, однако, их перемещение назад только увеличивает вероятность переполнения. Наш анализ также некорректен для подсчета влияния одновременного наличия более N блоков; это может случиться, если предположить, что время между выделениями блоков составляет $1/N$ от времени их жизни. Если имеется более N блоков, необходим расширенный анализ алгоритма 6.4L, но Коффман и Лейтон доказали, что область переполнения почти никогда не потребует больше чем $\sqrt{N} \lg N$ слотов; вероятность завершения работы составляет менее $O(N^{-M})$ для всех M.

В нашем примере начальный слот для поиска во время выделения не выбирается равномерно среди слотов $1, 2, \ldots, N$. Вместо этого он равномерно выбирается среди слотов $1, 65, 129, \ldots, N-63$, поскольку имеется $N/256 = 64$ слотов каждого размера. Но такое отклонение от рассмотренной в предыдущем разделе случайной модели делает переполнение даже менее вероятным, чем предсказано. Впрочем, не стоит держать пари, если нарушаются предположения о распределении размера блоков и времени их занятости.

F. Переполнение. Что необходимо предпринять, если больше нет свободного пространства? Предположим, что пришел запрос на n последовательных слов, в то время как все блоки слишком малы. В первый раз, когда такое происходит, обычно имеется более чем n доступных ячеек памяти, однако они расположены не последовательно. Уплотнение памяти (т. е. перемещение некоторых используемых ячеек, чтобы доступные ячейки памяти были собраны вместе) могло бы позволить продолжать обработку запросов. Однако уплотнение — медленный процесс, который требует строгой дисциплины применения указателей*. Кроме того, в подавляющем большинстве случаев при использовании метода первого подходящего независимо от того, сколько раз проводилось уплотнение, память в конечном счете оказывается полностью исчерпанной. Таким образом, вообще говоря, не имеет смысла писать программы для уплотнения, за исключением специальных случаев, связанных со сборкой мусора (см. упр. 33). Если переполнение ожидается заранее, можно "подстелить соломки", воспользовавшись некоторыми из методов переноса элементов из оперативной памяти на внешние запоминающие устройства с обеспечением возврата элемента в оперативную память при необходимости в нем**. Отсюда вытекает, что для эффективной работы все программы, работающие с областями динамической памяти, должны быть строго ограничены в плане допустимых ссылок на другие блоки, а также обеспечиваться аппаратно (например, должна иметься возможность генерирования прерывания при отсутствии данных в оперативной памяти или автоматической подкачке страниц).

Необходима некоторая процедура принятия решения о том, какие блоки являются наиболее подходящими кандидатами на удаление из оперативной памяти. Одна из идей состоит в содержании двусвязного списка выделенных блоков, при этом каждый блок двигается к началу списка при обращении к нему. Таким образом, блоки оказываются рассортированными в порядке последнего к ним обращения и блоки, расположенные в конце списка, удаляются первыми. Подобного эффекта можно достичь более просто: необходимо поместить выделенный блок в циклический список и включить в каждый блок бит "недавно использован". Этот бит устанавливается равным 1 при обращении к блоку. Когда наступает время удаления блока, указатель перемещается по циклическому списку, сбрасывая все биты "недавно использован" в нуль, пока не будет найден блок, к которому не было обращений со времени последнего прохода указателя по этой части списка.

Д. М. Робсон (J. M. Robson) показал [*JACM* **18** (1971), 416–423], что стратегии динамического выделения памяти, которые никогда не переносят выделенные блоки, не могут гарантировать эффективное использование памяти. Всегда найдутся патологические ситуации, при которых метод перестанет работать. Например, даже когда блоки ограничены размерами 1 или 2, переполнение может произойти при заполнении памяти примерно на 2/3, какой бы алгоритм не использовался! Интересные результаты Робсона рассматриваются в упр. 36–40, а также в упр. 42

---

* В этом случае справедливы все замечания, сделанные выше в связи с методом сборки мусора. — *Прим. перев.*

** Этот процесс известен в современной литературе как *свопинг* (*swapping*). Еще одно замечание заключается в том, что внешним запоминающим устройством, в принципе, может быть и сама оперативная память — стоит только вспомнить, каким образом в DOS использовалась недоступная для прямой адресации память EMS/XMS. — *Прим. перев.*

и 43, в которых показано, что метод наилучшего подходящего имеет очень плохой наихудший случай по сравнению с методом первого подходящего.

G. Для дальнейшего чтения. Всестороннее исследование и критический обзор технологий динамического выделения памяти, основанный на более многолетнем опыте, чем опыт автора этой книги, можно найти в работе Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles, *Lecture Notes in Computer Science* **986** (1995), 1–116.

УПРАЖНЕНИЯ

1. [*20*] Какие упрощения алгоритмов выделения и освобождения памяти, описанных в этом разделе, можно сделать, если предположить, что запросы на выделение и освобождение памяти всегда приходят в стековом порядке ("последним выделен — первым освобожден"), т. е. ни один блок не освобождается, пока не освобождены все выделенные после него блоки?

2. [*HM23*] (Э. Вольман (E. Wolman).) Предположим, что необходимо выбрать фиксированный размер узла для элементов переменной длины, и предположим также, что, когда каждый узел имеет длину k, а элемент — длину l, для хранения такого элемента используется $\lceil l/(k-b) \rceil$ узлов (здесь b — константа, обозначающая, что b слов каждого узла содержат управляющую информацию, например связь со следующим узлом). Если средняя длина l элемента равна L, то какой выбор k минимизирует среднее количество необходимой памяти? (Положим, что среднее значение $(l/(k-b)) \bmod 1$ равно $1/2$ для любого фиксированного k и переменного l.)

3. [*40*] При помощи компьютерного моделирования сравните следующие методы выделения памяти: наилучшего подходящего, первого подходящего и *наихудшего подходящего*. В последнем случае всегда выбирается наибольший доступный блок. Есть ли при этом существенная разница в использовании памяти?

4. [*22*] Напишите MIX-программу для алгоритма A, обращая особое внимание на ускорение работы внутреннего цикла. Положите, что поле SIZE — это (4:5), поле LINK — (0:2), а $\Lambda < 0$.

▶ **5.** [*18*] Предположим, известно, что в алгоритме A N всегда не меньше 100. Сто́ит ли устанавливать $c = 100$ на модифицированном шаге A4′?

▶ **6.** [*23*] (*Следующий подходящий.*) При постоянном использовании алгоритма A возникает тенденция к тому, что блоки малого размера остаются в начале списка AVAIL, поэтому приходится часто проводить длительный поиск блока нужного размера. Например, обратите внимание на рис. 42, на котором четко видно, как увеличиваются размеры блоков (как занятых, так и свободных) от начала памяти к ее концу. (Список AVAIL рассортирован по увеличению адресов памяти, как того требует алгоритм B.) Можете ли вы предложить такой вариант модификации алгоритма A, что (a) короткие блоки при его работе не скапливаются в некоторой области и (b) список AVAIL остается упорядоченным по адресам памяти для работы алгоритма наподобие B?

7. [*10*] Пример (1) показывает, что иногда метод первого подходящего заведомо превосходит метод наилучшего подходящего. Приведите аналогичный пример, когда метод наилучшего подходящего заведомо превосходит метод первого подходящего

8. [*21*] Покажите, каким образом можно просто модифицировать алгоритм A для работы по методу наилучшего подходящего (вместо изначального метода первого подходящего).

▶ **9.** [*26*] Каким образом следует разработать алгоритм выделения памяти, работающий по методу наилучшего подходящего, чтобы он не проходил в поисках необходимого блока

памяти весь список AVAIL? (Попытайтесь придумать метод, снижающий необходимый поиск настолько, насколько это возможно.)

10. [*22*] Покажите, каким образом можно модифицировать алгоритм B, чтобы блок из N последовательных ячеек, начинающийся с адреса P0, становился свободным без предположения о занятости всех N ячеек. Считается, что освобождаемая область может перекрывать несколько уже освобожденных блоков.

11. [*M25*] Покажите, что предложенное в упр. 6 усовершенствование алгоритма A можно также использовать для некоторого улучшения алгоритма B, что приведет к снижению средней длины поиска от половины длины списка AVAIL до одной трети. (Предполагается, что освобождающийся блок будет вставлен в упорядоченный список AVAIL в случайном месте.)

▶ **12.** [*20*] Модифицируйте алгоритм A таким образом, чтобы он следовал соглашениям "помеченных границ" (7)–(9), использовал измененный шаг A4′, описанный в тексте раздела, и включал усовершенствования из упр. 6.

13. [*21*] Напишите MIX-программу для алгоритма из упр. 12.

14. [*21*] Какие отличия появились бы в алгоритме C и алгоритме из упр. 12, если бы (a) в последнем слове свободного блока отсутствовало поле SIZE и (b) поле SIZE отсутствовало в первом слове выделенного блока?

▶ **15.** [*24*] Покажите, как ускорить работу алгоритма C за счет небольшого удлинения программы, не изменяя связей больше, чем это абсолютно необходимо в каждом из четырех случаев, в зависимости от того, чем является каждый из дескрипторов TAG(P0 − 1) и TAG(P0 + SIZE(P0)) — плюсом или минусом.

16. [*24*] Напишите MIX-программу для алгоритма C, включающую идеи из упр. 15.

17. [*10*] Каким должно быть содержимое LOC(AVAIL) и LOC(AVAIL) + 1 в (9) при отсутствии доступных блоков?

▶ **18.** [*20*] Рис. 42 и 43 получены с помощью одних и тех же данных и по сути одинаковых алгоритмов (алгоритмов A и B), но рис. 43 подготовлен модифицированным алгоритмом A с выбором наилучшего подходящего вместо первого подходящего. Почему при этом на рис. 42 большая свободная область находится в *старших* адресах памяти, в то время как на рис. 43 она же находится в *младших* адресах?

▶ **19.** [*24*] Предположим, что блоки памяти имеют вид (7), но без полей TAG или SIZE в последнем слове блока. Предположим далее, что для освобождения блока используется следующий алгоритм: Q ← AVAIL, LINK(P0) ← Q, LINK(P0 + 1) ← LOC(AVAIL), LINK(Q + 1) ← P0, AVAIL ← P0, TAG(P0) ← "−". (Он не выполняет объединение соседних свободных областей.)

Разработайте алгоритм для выделения памяти, аналогичный алгоритму A, который выполняет объединение смежных свободных блоков во время поиска в списке AVAIL и при этом исключает любую излишнюю фрагментацию памяти, как в (2)–(4).

20. [*00*] Почему в системе двойников вместо линейных списков желательно иметь двусвязные списки AVAIL[k]?

21. [*HM25*] Исследуйте отношение a_n/b_n при $n \to \infty$, где a_n — сумма первых n членов ряда $1 + 2 + 4 + 4 + 8 + 8 + 8 + 8 + 16 + 16 + \cdots$, а b_n — сумма первых n членов ряда $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + \cdots$.

▶ **22.** [*21*] В тексте раздела неоднократно упоминалось, что система двойников позволяет иметь только блоки размеров 2^k, и упр. 21 показывает, что это может привести к существенному увеличению требуемой памяти Но если приходит запрос на блок размером 11 слов, то почему нельзя найти блок размером 16 слов и разделить его на выделяемый блок размером 11 слов и два свободных блока с размерами 4 и 1 слово?

23. [*05*] Каков двоичный адрес двойника блока размером 4, двоичный адрес которого равен 011011110000? Каким бы он был, если бы блок имел размер 16?

24. [*20*] Согласно приведенному в тексте раздела алгоритму наибольший блок (размером 2^m) не имеет двойника, так как представляет собой всю память. Будет ли правильным определение

$$\text{двойник}_m(0) = 0$$

(по сути, блок станет собственным двойником), и можно ли таким образом избежать проверки $k = m$ на шаге S1?

▶ **25.** [*22*] Раскритикуйте следующую идею: "Динамическое выделение памяти с помощью системы двойников на практике никогда не приводит к выделению блока размером 2^m (поскольку этим исчерпывается вся память), и, вообще говоря, имеется некоторый максимальный размер 2^n, такой, что блоки большего размера никогда не выделяются. Значит, начинать работу с такого размера блоков — пустая трата времени, как и объединение в алгоритме S блоков, образующих в результате свободный блок размером, превосходящим 2^n".

▶ **26.** [*21*] Поясните, каким образом можно использовать систему двойников для динамического выделения памяти с адресами от 0 до `M` − 1 даже в случае, когда `M` не имеет вид 2^m, как требуется в тексте раздела.

27. [*24*] Напишите `MIX`-программу для алгоритма R и определите время ее работы.

28. [*25*] Напишите `MIX`-программу для алгоритма S и определите время ее работы, считая `MIX` двоичным компьютером с новой операцией `XOR`, с помощью обозначений из раздела 1.3.1 определяемой так: "C = 5, F = 5. Для каждого бита в ячейке M, равного 1, соответствующий бит в регистре A дополняется (изменяется с 0 на 1 или с 1 на 0); знак rA остается неизменным, время выполнения равно $2u$"

29. [*20*] Может ли система двойников работать без бита дескриптора в каждом выделенном блоке?

30. [*M48*] Проанализируйте среднее поведение алгоритмов R и S, задавая обоснованные распределения для последовательности запросов на выделение памяти.

31. [*M40*] Можно ли построить аналогичную системе двойников систему динамического распределения памяти, основанную на последовательности чисел Фибоначчи, а не на степенях двойки? (Таким образом, можно начать с F_m свободных слов и разделить доступные блоки из F_k слов на два двойника длиной F_{k-1} и F_{k-2} соответственно.)

32. [*HM47*] Определите $\lim_{n\to\infty} \alpha_n$, если он существует, где α_n — среднее значение t_n в случайной последовательности, определенной таким образом. Даны значения t_k для $1 \le k < n$; t_n равновероятно выбирается из множества значений $\{1, 2, \ldots, g_n\}$, где

$$g_n = \lfloor \tfrac{5}{4} \min(10000,\ f(t_{n-1} - 1),\ f(t_{n-2} - 2),\ \ldots,\ f(t_1 - (n-1))) \rfloor$$

и $f(x) = x$ при $x > 0$ и $f(x) = \infty$ при $x \le 0$. [*Примечание.* Некоторые ограниченные эмпирические тесты показывают, что α_n может быть примерно равно 14, но, вероятно, это не слишком точное значение.]

▶ **33.** [*28*] (*Сборка мусора и уплотнение.*) Положим, что ячейки памяти 1, 2, ..., `AVAIL` − 1 используются в качестве пула памяти для узлов переменного размера, имеющих следующий вид: первое слово `NODE(P)` содержит поля

`SIZE(P)` = количество слов в `NODE(P)`;
`T(P)` = количество полей связей в `NODE(P)`; `T(P)` < `SIZE(P)`;
`LINK(P)` = специальное поле связи для использованиятолько при сборке мусора.

В памяти за `NODE(P)` следует `NODE(P + SIZE(P))`. Предположим, что в `NODE(P)` в качестве связей с другими узлами используются `LINK(P + 1)`, `LINK(P + 2)`, ..., `LINK(P + T(P))` и

каждое из этих полей связей представляет собой либо Λ, либо адрес первого слова другого узла. И наконец, предположим, что в программе имеется еще одна переменная связи с именем USE, которая указывает на один из узлов.

Разработайте алгоритм, который (i) определяет все узлы, прямо или косвенно доступные из переменной USE, (ii) перемещает эти узлы в ячейки памяти от 1 до K − 1 для некоторого K, изменяя все связи так, чтобы сохранились структурные отношения, и (iii) устанавливает AVAIL ← K.

Например, рассмотрим следующее содержимое памяти, где INFO(L) обозначает содержимое ячейки L, исключая LINK(L):

1: SIZE = 2, T = 1	6: SIZE = 2, T = 0	AVAIL = 11,
2: LINK = 6, INFO = A	7: CONTENTS = D	USE = 3.
3: SIZE = 3, T = 1	8: SIZE = 3, T = 2	
4: LINK = 8, INFO = B	9: LINK = 8, INFO = E	
5: CONTENTS = C	10: LINK = 3, INFO = F	

Ваш алгоритм должен преобразовать эти данные в

1: SIZE = 3, T = 1	4: SIZE = 3, T = 2	AVAIL = 7,
2: LINK = 4, INFO = B	5: LINK = 4, INFO = E	USE = 1.
3: CONTENTS = C	6: LINK = 1, INFO = F	

34. [*29*] Напишите MIX-программу для алгоритма из упр. 33 и определите время ее работы.

35. [*22*] Сопоставьте методы динамического распределения памяти из этого раздела с технологиями последовательных списков переменного размера, рассмотренными в конце раздела 2.2.2.

▶ **36.** [*20*] В некоторой закусочной в Голливуде (Калифорния) имеется 23 места для посетителей, которые приходят поодиночке или вдвоем. Хозяйка закусочной показывает посетителям их места Докажите, что она всегда сможет рассадить посетителей, не разбивая пары, если одновременно в закусочной будет находиться не более 16 посетителей, а одиночки не сидят на местах 2, 5, 8, …, 20 (предполагается, что каждая пришедшая пара уходит вместе).

▶ **37.** [*26*] Продолжая упр. 36, докажите, что хозяйка не всегда может так удачно рассадить посетителей, если в закусочной только 22 места. Независимо от используемой ею стратегии может возникнуть ситуация, когда в закусочной будет находиться всего 14 посетителей, но для вошедшей пары не найдется двух соседних пустых мест.

38. [*M21*] (Д. М. Робсон (J. M. Robson).) Задача о закусочной, изложенная в упр. 36 и 37, может быть обобщена для определения производительности в наихудшем случае для любого алгоритма динамического выделения памяти, который никогда не перемещает выделенные блоки. Пусть $N(n,m)$ — наименьшее количество памяти, такое, что любая серия запросов на выделение и освобождение может быть выполнена без переполнения в случае, когда все размеры блоков не превышают m, а общее количество затребованной памяти не превосходит n. В упр. 36 и 37 доказано, что $N(16,2) = 23$. Определите точное значение $N(n,2)$ для всех n.

39. [*HM23*] (Д. М. Робсон.) Используя обозначения из упр. 38, покажите, что

$$N(n_1 + n_2,\, m) \le N(n_1,\, m) + N(n_2,\, m) + N(2m-2,\, m).$$

Следовательно, для фиксированного m существует

$$\lim_{n\to\infty} N(n,m)/n = N(m).$$

40. [*HM50*] Продолжая упр 39, определите $N(3)$, $N(4)$ и $\lim_{m\to\infty} N(m)/\lg m$, если таковой существует.

41. [*M27*] Назначение этого упражнения состоит в рассмотрении наихудшего случая использования памяти в системе двойников В частности, плохой случай возникает, например, если начать с пустой памяти и действовать следующим образом сначала выделить $n = 2^{r+1}$ блоков длиной 1, которые будут занимать адреса с 0 по $n-1$, затем для $k = 1, 2, \ldots, r$ освободить все блоки начинающиеся с адресов, которые не делятся на 2^k, и выделить $2^{-k-1}n$ блоков длиной 2^k, которые будут занимать адреса с $\frac{1}{2}(1+k)n$ по $\frac{1}{2}(2+k)n-1$ Для этой процедуры необходимо в $1+\frac{1}{2}r$ раз больше памяти, чем выделено

Докажите, что наихудший случай не может быть существенно хуже этого когда все запросы приходят на блоки размером 1, 2, …, 2^r и общий размер запрошенной памяти в любой момент не превышает n, где n кратно 2^r, система двойников никогда не переполнит область памяти размером $(r+1)n$

42. [*M40*] (Д М Робсон, 1975) Пусть $N_{\rm BF}(n,m)$ — количество памяти, необходимой, чтобы гарантировать отсутствие переполнения при использовании для выделения метода наилучшего подходящего, как в упр 38 Найдите "атакующую" стратегию, показывающую, что $N_{\rm BF}(n,m) \ge mn - O(n+m^2)$

43. [*HM35*] Продолжая упр 42, положим, что $N_{\rm FF}(n,m)$ — необходимая при методе первого подходящего память Найдите "оборонительную" стратегию, показывающую, что $N_{\rm FF}(n,m) \le H_m n/\ln 2$ (Следовательно, наихудший случай стратегии первого подходящего не так далек от наилучшего из наихудших случаев)

44. [*M21*] Предположим, что функция распределения $F(x)$ = (вероятность того, что размер блока $\le x$) непрерывна Например, $F(x)$ равна $(x-a)/(b-a)$ при $a \le x \le b$, если размеры равномерно распределены между a и b Приведите формулу, выражающую размеры первых N слотов при использовании метода распределенного подходящего