САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Самофалов Александр Владимирович

Способы реализации кучи и их свойства

Курсовая работа

Научный руководитель:
к. ф.-м. н. Булычев Д.Ю.

Санкт-Петербург
2013

Оглавление

Введение

Динамическое выделение памяти занимает важное положение во многих компьютерных системах. Оно применяется, если требуется выделить или освободить блок памяти с размером, неизвестным во время компиляции, неизвестное количество блоков, или обьект, время жизни которого заранее неизвестно. Существуют две основные операции: выделение непрерывного блока памяти определенного размера и освобождение ранее занятого блока. Эти операции осуществляются в определенной области памяти, которая называется кучей. Кучей также называется структура данных, которая позволяет эти операции производить.

Память является одним из критических ресурсов для любого приложения, и для эффективной его работы необходимо грамотно этим ресурсом распоряжаться. Важно, чтобы куча была эффективно реализована, иначе могут возникуть ситуации, препятствующие правильной работе программы. Например, если мы будем отдавать блоки случайным образом, не руководствуясь эффективной стратегией, при запросе на выделение блока свободной памяти, суммарно достаточного для выполнения запроса, память разбита на отделенные друг от друга блоки маленького размера, и система не сможет отдать непрерывный блок памяти. Таким образом, операция выделения памяти закончится неудачей. Такое состояние кучи называется фрагментацией.

Самый простой и распространённый способ организации кучи — хранить свободные блоки в виде списка. При запросе на выделение памяти пробегать по этому списку и, выбрав один из блоков, руководствуясь определённой стратегией, возвращать программе указатель на него. Можно организовывать кучу с помощью, например, skip lists [4], или если в программе все объекты одного размера, то можно создать “пул” блоков этого размера и выделять память из него.

Существуют разные стратегии выбора блока памяти, указатель на который возвращается программе при запросе на выделение памяти. С одной стороны, хочется как можно быстрее выделить память, с другой стороны выделить память так, чтобы минимизировать возникающую фрагментацию. Выделяют три основных стратегии:

1. “первый подходящий”: при поступлении запроса выделить блок памяти размера d просматривается список свободных блоков и возвращается указатель на первый встретившийся блок памяти размера, больше или равного d;

2. “наилучший подходящий”: при поступлении запроса выделить блок памяти размера d просматривается список свободных блоков и возвращается указатель на блок памяти, размер которого превышает d на наименьшее число байт;

3. “наихудший подходящий”: при поступлении запроса выделить блок памяти раз-

мера d просматривается список свободных блоков и возвращается указатель на самый большой по обьему блок памяти, если его размер не меньше d.

Преимущество первого подхода заключается в том, что выделение памяти (в среднем) происходит очень быстро, так как не надо каждый раз просматривать список свободных блоков до конца, однако при таком подходе фрагментация при некоторых условиях будет больше, чем у “наилучшего подходящего” или “наихудшего подходящего”.

Ещё одним распространённым подходом является “метод близнецов”. Все блоки имеют определённые размеры (например, степени двойки или числа Фибоначчи), все свободные блоки одного размера связаны в список. При запросе на выделение блока размера d, если есть свободные блоки размера d, то просто возвращается указатель на такой блок. Если же такого блока нет, то блок большего размера расщепляется на два и возвращается указатель на один из них. Расщеплённые блоки сливаются, когда они станут оба свободными. Такие блоки называются “близнецами”, что и дало название методу.

Эффективность работы программы зависит от эффективности реализации кучи, следовательно, требуется оценивать эффективность реализации кучи. Для этого необходимо, чтобы куча предоставляла информацию о своей работе, например, общий размер кучи, количество блоков, размер максимального свободного блока.

Целью данной работы является изучение различных стратегий выделения памяти в куче, реализация некоторых из них, разработка интерфейса, позволяющего получать информацию о работе. В работе реализованы следующие виды кучи: выбор “наилучшего подходящего”, “наихудшего подходящего” и “первого подходящего” стратегий выделения памяти и их некоторые модификации, а также самосжимающаяся куча, которая при выполнении некоторых условий сжимает кучу, уменьшая фрагментацию. Реализация обеспечивает простоту переключения между разными стратегиями, возможность их замены и добавления новых, а также предоставляет информацию о работе куче, с помощью которой её можно оценить.

1. Способы реализации кучи

Для реализации самого простого аллокатора необходимо реализовать две функции:

- *malloc* резервирует для программы область памяти размера не менее, чем ей передано параметром, и возвращает указатель на начало этой области или нулевой указатель, если выделить непрерывный кусок памяти нужного размера не удалось;
- *free* освобождает ранее выделенную с помощью *malloc* область памяти, то есть делает ее доступной для нового выделения памяти.

Для того чтобы распределять память, нужно сначала ее получить у операционной системы. Основными системными вызовами для управления памятью в Unix-подобных системах являются *brk* и *sbrk*[1]. Эти вызовы изменяют размер сегмента данных для процесса, который их вызывает. Они передвигают конец сегмента (такой адрес в памяти, что, если обратиться по большему адресу, возникнет ошибка сегментации). *brk* устанавливает конец сегмета по адресу, который передается ему в качестве параметра. *sbrk* увеличивает размер сегмента данных на то количество байт, которое передано ему в качестве параметра. Также увеличить размер сегмента данных можно с помощью особого вызова *mmap*[2], передав ему одим из параметров размер запрашиваемой области в байтах.

В данной работе примущественно использовался первый метод, т.к. при использовании вызовов *brk* и *sbrk* весь сегмент данных будет расположен в одной непрерывной области памяти, что делает работу с ней более удобной.

Одним из самых распространенных способов организации кучи является хранение блоков в списке. Для этого в памяти (непосредственно перед блоком) отводится некоторая область, в которую записывается служебная информация: размер блока, отметка о том, является ли блок занятым, а также указатель на следующий за ним блок. Тогда для поиска нужного блока нужно просто бежать по списку, начиная с первого элемента, указатель на который будем хранить. Если же подходящего блока не нашлось, то запрашивается новый блок памяти у операционной системы и он добавляется в список. Для того чтобы освободить память, нужно в соответствующем блоке метаинформации отметить, что данный блок свободен.

Существует три основных способа выбора блока в процессе поиска:

- *Первый подходящий*: выбирается самый первый блок, который подходит;

[1] http://man7.org/linux/man-pages/man2/sbrk.2.html
[2] http://man7.org/linux/man-pages/man2/mmap.2.html

- *Наилучший подходящий*: из всех подходящих блоков выбирается тот, который имеет наименьший размер;
- *Наихудший подходящий*: выбирается блок, который имеет наибольший размер.

Далее приведен алгоритм работы такого выделения памяти.

Algorithm 1 Malloc(*size*)

```
1: curr ← heapBegin
2: result ← null
3: while curr ≠ null do
4:     if curr.isFree & curr.size ≥ size then
5:         result ← improve(result, curr)
6:     end if
7:     curr ← curr.next
8: end while
9: if curr = null then
10:    curr ← morecore(size + headerSize)
11:    curr.size ← size
12:    curr.next ← heapBegin
13:    heapBegin ← curr
14: end if
15: curr.isFree ← false
16: return curr.data
```

В методе первого подходящего тело цикла будет выглядеть так:

```
1: if curr.isFree & curr.size ≥ size then
2:     curr.isFree ← false
3:     return curr.data
4: end if
5: curr ← curr.next
```

Algorithm 2 Free(*ptr*)

```
1: block ← (ptr − headerSize)
2: block.isFree ← true
```

Здесь *improve* — функция, принимающая два указателя на блоки и возвращающая указатель на тот из переданных блоков, который лучше подходит по некоторому критерию. *morecore* — функция, которая запрашивает у операционной системы блок, размером не меньше, чем передано в качестве параметра. *headerSize* — размер мета-информации перед каждым блоком.

Этот метод является самым простым и имеет некоторые недостатки, некоторые из которых разрешают следующие модификации:

- Можно вместо одного связного списка хранить два: отдельно для свободных блоков, отдельно для занятых, тем самым улучшая время поиска, так как теперь достаточно перебирать только свободные блоки. Для поддержания этой структуры достаточно при освобождении блока перемещать его в список свободных и, наоборот, при выделении памяти перемещать блоки в список занятых.

- В данном алгоритме может возникнуть ситуация, когда система возвращает блок, размер которого превышает запрошенный размер. При этом в выделенном блоке памяти останется участок, который никак не используется и который считается занятым. Для того чтобы предотвратить такое неэффективное использование памяти, можно при выделении блока памяти разделять его на два: блок запрашиваемого размера и остаток.

- С каждым запросом блок может либо остаться неизменным, либо разбиться еще на два, меньшего размера. С данным процессом связана тенденция уменьшения размера свободных блоков. Через некоторое время эти блоки могут стать настолько маленького размера, что не будут подходить ни под один запрос. Для решения этой проблемы можно при освобождении блоков смотреть на его непосредственных соседей (блоки памяти, адреса которых граничат с заданным), и, если они тоже свободны, сливать их в один блок большего размера. Для того чтобы это можно было эффективно осуществлять, после каждого блока будет также добавляться метаинформация — адрес начала этого же блока. Тогда становится очень просто находить соседей: следущий сосед находится как сумма адреса текущего блока, его размера и размера метаинформации блока, а адрес предыдущего соседа содержится сразу перед блоком.

- При использовании алгоритма первого подходящего возникает тенденция к тому, что блоки маленького размера остаются в начале списка. Это увеличивает время поиска подходящего блока. Для решения этой проблемы можно начинать новый поиск с того места, на котором остановился предыдущий (метод следущего подходящего).

2. Самосжимающаяся куча

Одной из самых серьезных проблем динамического выделения памяти является фрагментация памяти. Один из подходов радикально решает эту пробему: в случае, когда фрагментация становится слишком высокой или аллокатор не может выделить необходимый блок памяти, происходит процесс сжатия памяти — занятые блоки перемещаются таким образом по куче, что между ними не остается свободных блоков.

Однако у этого метода есть недостатки. Для корректной работы программы адрес памяти в куче у блока должен оставаться одним и тем же вплоть до его освобождения, однако при перемещении памяти ее адрес меняется, и прежний указатель начинает указывать на некорректные данные. Для решения этой проблемы есть два основных подхода:

1. среда, в которой работает программа, должна быть спроектировна с учетом того, что данные могут быть перенесены;

2. программист должен учитывать перенос данных и разрабатывать программу с учетом этого, например, использовать некий интерфейс для работы с кучей.

В данной работе применяется второй способ. Для реализации этого подхода, для каждого обьекта в куче в отдельной области памяти создается дескриптор этого обьекта. Дескриптор — это некоторый объект, который представляет объект в куче. Как минимум, дескриптор должен содержать указатель на сам обьект, чтобы можно было иметь доступ к нему, зная его дескриптор. При компактификации будут перемещаться только сами обьекты, а их дескрипторы будут оставаться на прежнем месте. Понятно, что если *malloc* будет возвращать просто указатели на блоки данных, то после компактификации они перестанут указывать на правильные данные, поэтому надо возвращать дескриптор обьекта, из которого легко можно получить сами данные. Для удобства работы вся работа с дескрипторами инкапсулируется в отдельный класс *heap_pointer*, который в первом приближении является указателем на дексриптор, одновременно являясь “ссылкой” на обьект в куче. С помощью этого класса программист может:

1. получить значение объекта, на который *heap_pointer* сслылается, или же значение его поля или вызвать любой метод;

2. присвоить объекту или его полю какое-нибудь значение;

3. если блок памяти содержит не один объект, а массив объектов, то можно получить *heap_pointer* на какой-либо обьект в массиве по его индексу, если есть

heap_pointer, ссылающийся на начало массива (также в этом месте можно проверять массив на переполнение)

Для того чтобы можно было осуществить эти операции, можно хранить в *heap_pointer* указатель на дескриптор объекта, а также смещение от его начала. Тогда, чтобы получить указатель на сам обьект, достаточно взять адрес дескриптора, из него получить адрес данных, и относительно него с помощью смещения посчитать адрес объекта.

Для того чтобы было удобно осуществлять саму компактификацию, соединим дескрипторы в связный список, отсортированыый по возрастанию адреса в памяти соответсвующего блока с данными. Алгоритм компактификации будет следующим:

Algorithm 3 Compact

```
1: threshold ← null
2: curr ← handlesBegin
3: while curr ≠ null do
4:     if curr.isFree then
5:         if threshold = null then
6:             threshold ← curr.data
7:         end if
8:         remove(curr)
9:     else if threshold ≠ null then
10:        copy(threshold, curr.data, curr.size)
11:        curr.data ← threshold
12:        threshold ← threshold + curr.size
13:    end if
14:    curr ← curr.next
15: end while
16: addBack(threshold, headEnd − threshold)
```

Здесь *handlesBegin* — начало списка дескрипторов, $copy(dest, source, size)$ копирует *size* байт из *source* в *dest*, $addBack(data, size)$ добавляет в конец списка новый дескриптор для блока с данными в *data* и размером *size*, а *remove* удаляет дескриптор из списка.

3. Результаты

В рамках данной работы была реализована система управления памятью для операционной системы Linux, написанная на C. Выбор языка обусловлен тем, что необходимо работать с памятью и с операционной системой на низком уровне, и язык C вполне удовлетворяет этим требованиям. Система реализована в виде подключаемой библиотеки, конфигурируемой перед сборкой. Можно выбрать одну из трех возможных стратегий:

- первый подходящий;
- наилучший подходящий;
- наихудший подходящий.

Также возможно настроить или модифицировать эти алгоритмы:

- раздельно хранить списки свободных и занятых блоков;
- разрешить разделение блока;
- разрешить слияние свободного блока с его соседями;
- модифицировать метод “первого подходящего” до метода “следущего подходящего”;
- определить, как именно должны быть выравнены данные;
- выбрать минимальный размер запрашиваемого у ОС блока;
- ограничить размер кучи;
- включить вывод отладочной информации.

Кроме того, создана библиотека с реализацией самосжимающейся кучи. Она также написана на C, кроме класса *heap_pointer*, который для простоты работы с ним является шаблонным, написанным на C++.

Все эти реализации предоставляют информацию о своей работе через специальный интерфейс, который позволяет узнавать:

- объем памяти, отведенный под кучу;
- объем свободной и выделенной памяти;
- количество блоков: общее количество, количество занятых и свободных;

- размер самого большого свободного блока.

Реализации позволяют трассировать свою работу, то есть выводят на печать некую служебную информацию, которая позволяет отслеживать процесс работы менеджера памяти.

Чтобы использовать данный менеджер памяти в своей программе, нужно программу скомпилировать вместе с подключаемой библиотекой. Также данный менеджер памяти возможно использовать и без переписывания исходного кода, с уже скомпилированными программами. Для этого достаточно записать в переменную окружения $LD\_PRELOAD$ адрес скомпилированной библиотеки.

Список литературы

[1] Burelle Marwan. A Malloc Tutorial. — 2009.

[2] The Memory Management Reference. — URL: `http://www.memorymanagement.org/`.

[3] Paul R. Wilson Mark S. Johnstone Michael Neely David Boles. Dynamic Storage Allocation: A Survey and Critical Review. — 1995.

[4] Pugh William. Skip Lists: A Probabilistic Alternative to Balanced Trees // Communications of the ACM. — 1990.

[5] Кнут Дональд. Искусство программирования, том 1. Основные алгоритмы. Искусство программирования. — Вильямс, 2006.