Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Захаров Захар Сергеевич

Реализация выделения и освобождения динамической памяти методом граничных признаков

Курсовая работа

Научный руководитель:
профессор Терехов А. Н.

Санкт-Петербург
2020

Оглавление

Введение

Работа с динамической памятью одно из главных преимуществ языка C по сравнению с другими языками программирования . На данный момент реализовано множество алгоритмов для выделения динамической памяти, но в компиляторе RuC все еще не было реализовано этих функций. В ходе курсовой работы будут рассмотрены существующие подходы для оперирования с динамической памятью. Будет приведен способ борьбы с фрагментацией памяти и будет проведен ряд тестов для проверки качества и надежности функций malloc и free.

1. Постановка задачи

Целью данной работы является реализация использования динамической памяти в компиляторе RuC. Для её достижения были выделены следующие задачи:

1. обзор существующих подходов;
2. реализация malloc();
3. реализация free();
4. тестирование с целью проверки эффективности;

2. Обзор

2.1. RuC

RuC - компактный компилятор с языка C (с некоторыми ограничениями) в коды оригинальной виртуальной машины. Первоначальным толчком создания RuC была необходимость создать простое, наглядное, но достаточно мощное средство программирования роботов, затем задача была расширена на обучение алгоритмической грамотности и информатике.

2.2. Обзор существующих вариаций алгоритмов

Помимо алгоритма рассмотренного выше, существуют следующие алгоритмы для работы с динамической памятью.

1. Метод первого подходящего.
2. Метод наиболее подходящего.
3. Система близнецов.

Систему близнецов мы не рассматриваем из-за потери одного бита в каждом блоке, так-же алгоритм требует, что бы все блоки имели длину 1, 2, 4, 8, 16, 32 и тд.
Наш алгоритм является улучшеной версией алгоритма первого подходящего так как при освобождении памяти наша ячейка не просто добавляется к списку AVAIL, а так-же умеет сливаться с рядом стоящими свободными ячейками.
Метод первого подходящего ищет первую ячейку нужного нам размера, а метод наиболее подходящего ищет ячейку размером как можно ближе к запрашиваемой.

2.3. Вид ячеек

Свободная ячейка имеет следующий вид(рис 1). На конца она хранит размер ячейки со знаком минус, это было сделано для того что

Рис. 1: Свободная ячейка

бы не хранить размер и flag. Было принято решение совместить и использовать положительные числа для занятых ячеек, а отрицательные для свободных. В ячейках NEXT хранится указатель на следующую свободную ячейку, а в PREV на предыдущую, это нужно для удобного добавления новых свободных ячеек.

Рис. 2: Занятая ячейка

Занятая ячейка имеет следующий вид(рис 2). На конца она хранит размер ячейки со знаком плюс. На хранение размера ячейки выделяется дополнительные 8 байт (по 4 с каждой стороны).

2.4. Описание алгоритма

Реализация malloc: Увеличиваем нужный нам размер на 8 байт для хранения размера и «+/-». Для удобства будем считать size > 0 ячейка занята, а size < 0 ячейка свободна. Следующий алгоритм основывается на списке AVAIL с двумя связями: указатель на следующую свободную ячейку и на предыдущую. Идем по AVAIL пока не найдем ячейку с нужным размером, после чего сохраняем информацию о нашей занятой ячейки, и изменяем размер свободной ячейки. Если размер ячейки чуть больше чем нам нужно, то забираем ее целиком для избежания коллизий.

Реализация free: Получаем указатель на занятую ячейку и смотрим есть ли слева или справа от нее свободные ячейки, если такие существу-

ют то мы их объединяем и размеры их суммируем. Если свободных ячеек нет, то добавляем нашу в список AVAIL изменяя при этом размер на отрицательный для обозначения свободной ячейки.

3. Реализация

3.1. Терминология

Введем терминологию для описания следующих двух функций на псевдокоде.

- AVAIL-список свободных ячеек.
- SIZE-размер ячейки.
- NEXT-указатель на следующую свободную ячейку.
- PREV-указатель на предыдущую свободную ячейку.
- TAG-содержащиеся в ячейке.
- END-последний указатель в списке AVAIL.

3.2. Реализация malloc()

Пусть AVAIL указывает на первый свободный блок памяти, и предположим что каждый блок памяти с адресом P имеет следующие 3 поля: SIZE(P), NEXT(P), PREV(P). Алгоритм находит и резервирует блок из N слов или сообщает об ошибке.[1]

A1:[начальная установка] Установим P <- AVAIL, увеличим size на 2 слова для хранения дополнительной информации.

A2:[Конец списка?] Если P = END, то мы дошли доконца выдаем ошбику

A3:[Хватает ли нам памяти?] Если SIZE(P) >= N (нужной нам), то мы забираем N или весь P при условии того, что SIZE(P) - 20<= N (это позволяет нам избежать коллизий).

A4: Устанавливаем P = NEXT(P), и возвращаемся к пункту A2.

3.3. Реализация free()

B1:[начальная установка] Установим 2 флага R, L в которых содержится информация о левой, правой ячейки. Дальше для удобства будем обозначать + если занята, - если свободна.

B2.1:[L = -, R = +] Увеличиваем размер левой ячейки на N в начале, и размер нашей ячейки на SIZE(L) и меняем это значение на отрицательное.

B2.2:[L = +, R = -] Увеличиваем размер начала нашей ячейки и меняем знак, и увеличиваем размер в конце правой ячейеи и меняем указатели предыдущей и следующей свободной ячейки на нашу.

B2.3:[L = -, R = -] Суммируем все 3 размера и удаляем правую ячейки из свободных так как она сливается с нашей.

B2.4:[L = +, R = +] Меняем SIZE(P) на отрицательный и добавляем к списку AVAIL.

4. Результаты

4.1. Тесты с различными ячейками

Первые тесты(рис 3,4) проводились для проверки поведения свободной памяти при различных начальных данных, таких как:

- количество ячеек и их размеры;
- время жизни ячейки;
- размер создаваемых ячеек;
- количество итераций;

Начальные данные	11 ячеек размером 10.000 байт	11 ячеек размером 10.000 байт	5 ячеек 1.000 байт 5 ячеек 10.000 байт 1 ячейка 20.000 байт	11 ячеек размером 20.000 байт	10 ячеек размером 500 байт 1 ячейка размером 100.000 байт
Размер выделяемых ячеек	От 1 до 2500 байт	От 1 до 1000 байт	От 1 до 3000 байт	От 1 до 3000 байт	От 1 до 1000 байт
Время жизни	От 1 до 500 итераций	От 1 до 1000 итераций	От 1 до 2000 итераций	От 1 до 2000 итераций	От 1 до 2500 итераций
Кол-во NULL на 100 итераций	2	0	40	0	0
Кол-во NULL на 200 итераций	58	0	120	40	4
Кол-во NULL на 500 итераций	200	165	400	280	210
Кол-во NULL на 1000 итераций	480	400	860	660	580

Рис. 3: Тесты над свободной памятью

Кол-во NULL на 100 итераций	2	0	40	0	0
Кол-во NULL на 200 итераций	58	0	120	40	4
Кол-во NULL на 500 итераций	200	165	400	280	210
Кол-во NULL на 1000 итераций	480	400	860	660	580

Рис. 4: количество отказов при выделении памяти

4.2. Вечная жизнь системы

Так-же был проведен тест на бесконечную жизнь системы.
Данные для теста:
Создаем ячейки размером от 1 до 1000 байт и временем жизни от 1 до 1000 итераций.
Пытаемся найти такой размер свободной ячейки, что бы система жила вечна.

Размер	Количество итераций	не нашлось места
100 КБ	10.000	4500 ячеек
200 КБ	10.000	1969 ячеек
300 КБ	10.000	56 ячеек
350 КБ	10.000	0 ячеек

Идем дальше и рассматриваем значения для 50.000 - 100.000 итераций.

Размер	Количество итераций	не нашлось места
350 КБ	50.000	89 ячеек
400 КБ	50.000	0 ячеек
400 КБ	100.000	0 ячеек

Идем дальше и рассматриваем значения для 100.000-1.000.000 итераций.

Размер	Количество итераций	не нашлось места
400 КБ	200.000	0 ячеек
400 КБ	300.000	0 ячеек
400 КБ	500.000	0 ячеек
400 КБ	1.000.000	0 ячеек

После чего был проведен тест на длительное время что бы проверить подходит ли 350.000 байт для вечной жизни системы. Тест прошел успешно и система со свободной ячейкой на 400.000 байт жила до тех пор пока тест не был прекращен.

Заключение

В рамках работы были выполнены следующие задачи:

- выбран алгоритм для работы с динамической памятью;
- проведено сравнение алгоритмов;
- реализовано выделение динамической памяти
- реализовано освобождение динамической памяти
- проведены тесты на жизнь системы;

Список литературы

[1] Д.Кнут. Исскуство программирования / Под ред. Бабенко. — К. И. : Учебник, 1976.