Национальный исследовательский университет ИТМО

Факультет ПИиКТ

Лабораторная работа №1 по дисциплине «Низкоуровневое программирование»

Вариант: 1 (документное дерево)

Работу выполнил:

Голиков Д.И.

Группа:

P33102

Преподаватель:

Кореньков Ю. Д.

Санкт-Петербург,

2023

Цель работы

Создать модуль, реализующий хранение в одном файле данных (выборку, размещение и гранулярное обновление) информации общим объёмом от 10GB соответствующего варианту вида.

Задачи

1. Спроектировать структуры данных для представления информации в оперативной памяти
2. Спроектировать представление данных с учетом схемы для файла данных и реализовать базовые операции для работы с ним:
    a. Операции над схемой данных (создание и удаление элементов схемы)
    b. Базовые операции над элементами данных в соответствии с текущим состоянием схемы (над узлами или записями заданного вида)
3. Реализовать публичный интерфейс для приведенных выше операций
4. Реализовать тестовую программу для демонстрации работоспособности решения

Исходный код

https://github.com/Denoske/ITMO-labs/tree/main/3nd%20year/Low%20level%20programming/Lab1

Описание работы

Список модулей:

- tests/test.c – тесты
- utils/optional.c – функции для работы со вспомогательным типом optional
- zgdb/format.c – функции для работы с файлом (в т. ч. с индексами документов)
- zgdb/document.c – функции для работы с документом
- zgdb/element.c – функции для работы с элементом схемы
- zgdb/schema.c – функции для работы со схемой документа
- zgdb/list.c – функции для работы со списком свободных мест в файле
- zgdb/condition.c – функции для работы с условиями
- zgdb/query.c – функции для работы с запросами
- zgdb/iterator.c – функции для работы с набором данных через итератор

Чтобы использовать библиотеку, надо подключить все public-хедеры (на самом деле, достаточно query_public.h, потому что хедеры включают друг друга и так).

Углубляться в исходный код модулей, наверно, не стоит. Всё есть в репозитории, с большим количеством комментариев. А вот примеры использования вставлю (всё из demo.c):

```
zgdbFile* file = loadFile( filename: "demo_db");
if (!file) {
    file = createFile( filename: "demo_db");
    if (!file) {
        printf( format: "Error\n");
        exit( Code: -1);
    }
}
```

Создание или загрузка файла базы данных

```
documentSchema* rootSchema = createSchema( name: "root");
if (rootSchema) {
    addElementToSchema( schema: rootSchema,  el: intElement( key: "rootInt1",  value: 123));
    addElementToSchema( schema: rootSchema,  el: intElement( key: "rootInt2",  value: 456));
    addElementToSchema( schema: rootSchema,  el: intElement( key: "rootInt3",  value: 789));
    addElementToSchema( schema: rootSchema,  el: booleanElement( key: "isFirst",  value: true));
    addElementToSchema( schema: rootSchema,  el: doubleElement( key: "rootDouble",  value: 128.128));
    addElementToSchema( schema: rootSchema,  el: stringElement( key: "rootString",  value: "HI WORLD!"));
}
```

Создание схемы

```
condition* cond = condOr( cond1: condLess( el: intElement( key: "childInt1",  value: 1000)),  cond2: condLess( el: intElement( key: "grChildInt2",  value: 10000)));
```

Создание условия

```
query* insert = createInsertQuery( schemaName: NULL,  newValues: rootSchema,  cond: NULL);
if (insert) {
    query* insertChild = createInsertQuery( schemaName: NULL,  newValues: childSchema,  cond: NULL);
    addNestedQuery( q: insertChild,  nq: createInsertQuery( schemaName: NULL,  newValues: grandChildSchema,  cond: NULL));
    addNestedQuery( q: insert,  nq: insertChild);
}
```

Создание запроса INSERT

```
query* selectRoot = createSelectQuery( schemaName: "root",  cond: NULL);
```

Создание запроса SELECT

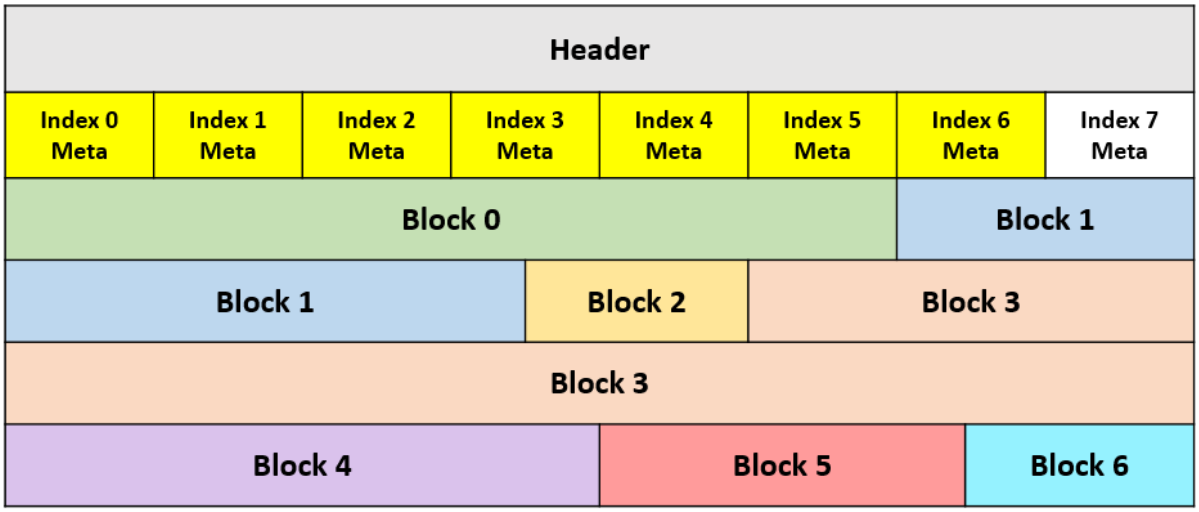
```
printf( format: executeInsert(file,  error: &error,  q: insert2) ? "true\n" : "false\n");
executeSelect(file,  error: &error,  it: &it,  q: selectRoot);
while (hasNext(it)) {
    document* doc = next(file, it);
    printDocumentAsTree(file, doc);
    destroyDocument(doc);
}
destroyIterator(it);
```

Вызов INSERT, SELECT и вывод результатов

```
true
root#641CB5B40000000000015FA7 {
    child#641CB5B40000000000001604B {
        grandChild#641CB5B40000000000001609C
    }
}
true
root#641CB5B40000000000015FA7 {
    child#641CB5B40000000000016162
    child#641CB5B40000000000001604B {
        grandChild#641CB5B400000000000160FF
        grandChild#641CB5B40000000000001609C
    }
}
true
grandChild#641CB5B400000000000160FF {
        grChildInt1 = 505
        grChildInt2 = 456
        grChildInt3 = 987
}
grandChild#641CB5B40000000000001609C {
        grChildInt1 = 505
        grChildInt2 = 456
        grChildInt3 = 987
}
child#641CB5B40000000000016162 {
        childInt1 = 121
        childInt2 = 212
}
child#641CB5B40000000000001604B {
        childInt1 = 121
        childInt2 = 212
}
root#641CB5B40000000000015FA7 {
        rootInt1 = 123
        rootInt2 = 456
        rootInt3 = 789
        isFirst = true
        rootDouble = 128.128000
        rootString = "HEY BEACH!"
}
true
```

Вывод всего demo.c

Аспекты реализации



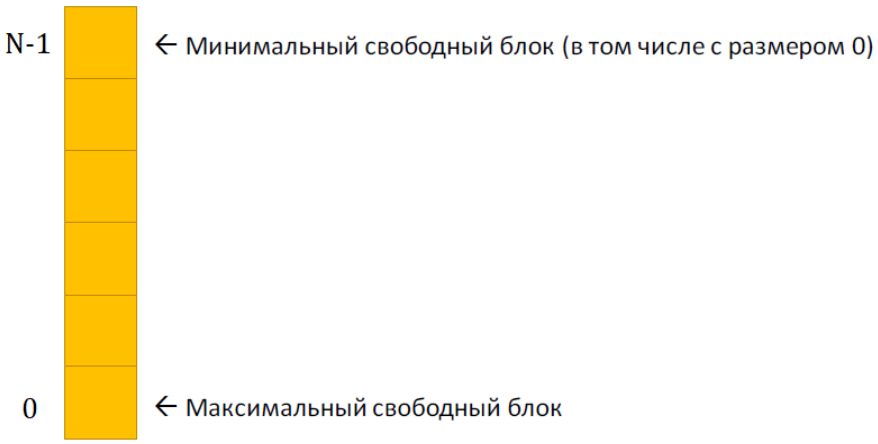
Вид файла

На картинке выше представлен общий вид файла. Файл содержит:

- Заголовок
- Массив индексов документов
- Сами документы (они же блоки)

При удалении документа его индекс обновляется и становится “мёртвым”.

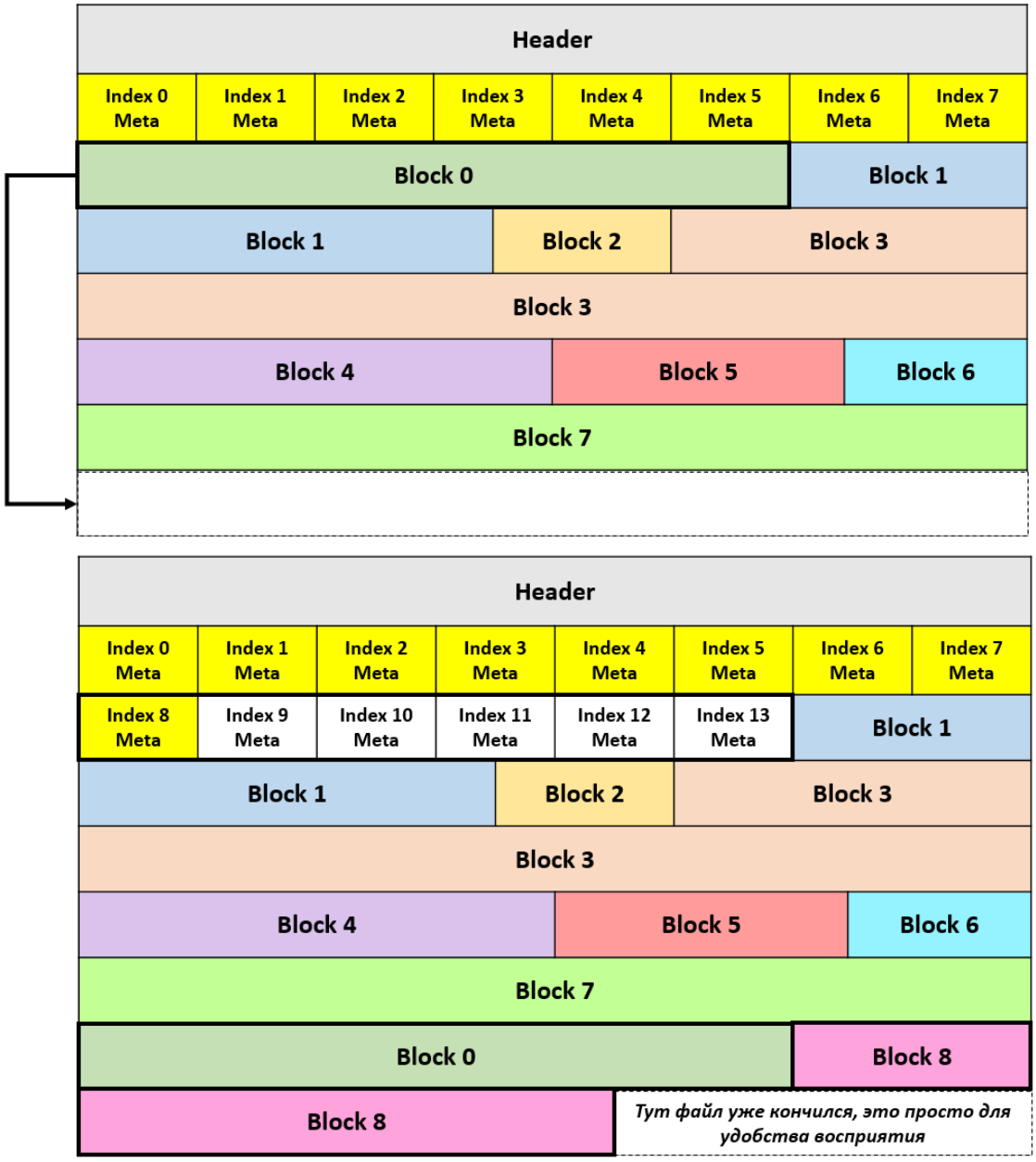
При загрузке файла массив индексов считывается. В оперативной памяти создаётся отсортированный список свободных мест в файле из “новых” (ещё не использовавшихся) и “старых” индексов:



Список свободных мест

Новые документы вставляются в блок, находящийся в начале списка (самый большой блок), если влезают в него, или в блок, находящийся в конце списка (блок с размером ноль, т.е. ещё не созданный блок).

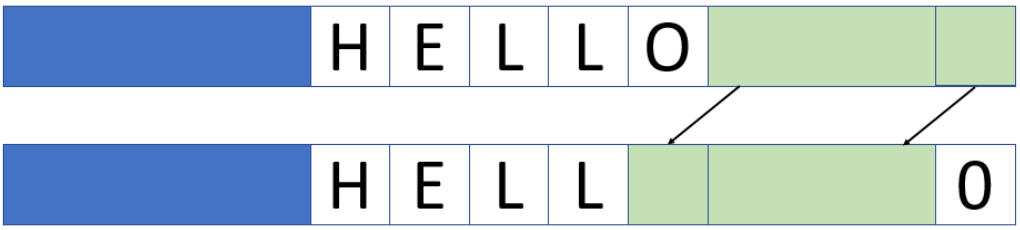
Естественно, возможны ситуации, когда свободных индексов нет. В таком случае нужно выделять место под новые индексы путём переноса первого/первых документов в другое место (в конец файла или в дырку):



Выделение новых индексов на месте первого блока

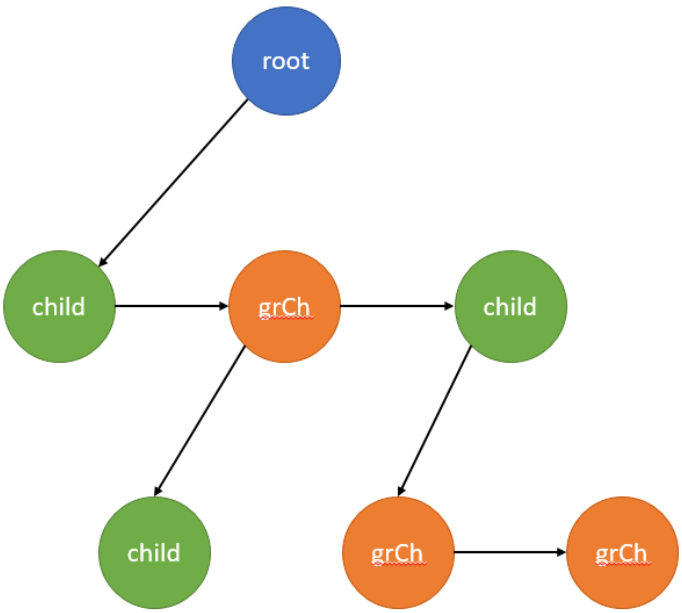
Если документ записывается в блок бОльшего размера, чем нужно, то в том месте, где полезные данные кончаются, записывается ноль, который в дальнейшем при считывании документа трактуется как конец документа.

Кстати, дырка внутри документа ещё может возникнуть при обновлении строки. Если строка стала меньше, то все элементы схемы после строки сдвигаются так, чтобы дырка была в конце документа, и в начале дырки опять-таки записывается ноль:



Уменьшение строки

Последний интересный аспект – то, как хранятся связи между документами. Тут получается так называемое «общее дерево»: в заголовке родителя хранится ссылка на первого ребёнка, а в ребёнке уже хранится ссылка на следующего, и т.д. Обратных связей нет!



Как выглядит документное дерево

Результаты

Вставка (изначально доступно 1000 индексов, время в мкс)

Вставка (изначально доступно 10000 индексов, время в мкс)

Выборка (время в мкс)

Обновление (время в мкс)

Рост размера файла при вставке (в байтах)

- Комментарий по поводу вставки: как можно видеть, на графиках присутствуют “скачки”, но, в целом, это O(1). Скачки соответствуют моментам времени, когда изначально доступные индексы заканчиваются, и надо выделять новые. Это особенно хорошо видно по первой картинке: каждый раз, когда добавляем документ, кратный 1000, выделяем ещё 1000 индексов, и так далее. Как сократить количество “скачков”? Естественно, выделить изначально больше индексов. Размер одного индекса 9 байт, так что можно позволить себе выделить, например, 1000000 индексов – это всего лишь 9 Мбайт (по сравнению с полезными данными, которых может быть 10 Гбайт+).
- Комментарий по поводу выборки и обновления: выборка происходит за O(N+N) = O(N), а обновление за O(N+M), поскольку по графику видно, что оно выполняется быстрее, чем выборка, т.е. затрагиваются не все элементы.
- Комментарий по поводу удаления: не стал вставлять график в отчёт, потому что по сути это получится то же самое, что и обновление (только выполнится ещё быстрее).

Выводы

В ходе работы была реализована “база данных” для хранения дерева документов, поддерживающая операции Select, Insert, Update, Delete, причем все указанные выше операции могут дополнительно проверять условия. Также, были разработаны тесты, которые подтверждают правильность работы программы. Основной вывод по работе можно сделать такой: лучшее – враг хорошего, потому что в общей сложности на эту лабораторную было потрачено больше 3 месяцев просто потому, что выбранная архитектура была слишком правильной, что привело к дополнительной умственной нагрузке (больше всего времени было потрачено не на сам код, а на понимание того, как он должен работать).