Задача

Как-то у меня возникла задача (в рамках разбора высокоуровневого сетевого протокола) организовать выявление известных программе слов (составляющих словарь) среди приходящих в программу строк.
Ограничения/упрощения в рамках задачи следующие:
- строки состоят только из английских визуальных символов (скажем, в ASCII и, соответственно, с int-кодами от 33 до 122) и заканчиваются нуль-символом (`'\0'`);
- в процессе работы словарь изменяется только пополнением, причём гораздо реже чтения из него, либо вообще не изменяется;
- памяти (ОЗУ) много;
- словарь маленький (до 1000 слов, хотя жёсткого ограничения нет);
- слово представляет из себя набор визуальных символов, органиченный символами-разделителями (тоже визуальными), либо началом или концом всей строки;
- регистр символов учитывается (т.е. "word" и "WoRd" считаются разными словами);
- результатом поиска должна быть информация о том, какое именно словарное слово имеется в анализируемой строке; возможно, будет полезной информация о взаимном порядке присутствующих в анализируемой строке словарных слов относительно друг друга:
- каждое словарное слово может присутствовать в анализируемой строке больше одного раза и это должно отражаться в информации о взаимном порядке присутствующих в анализируемой строке словарных слов относительно друг друга;
- язык реализации – Си, ОС – GNU/Linux вместе с GNU библиотекой языка Си (glibc).

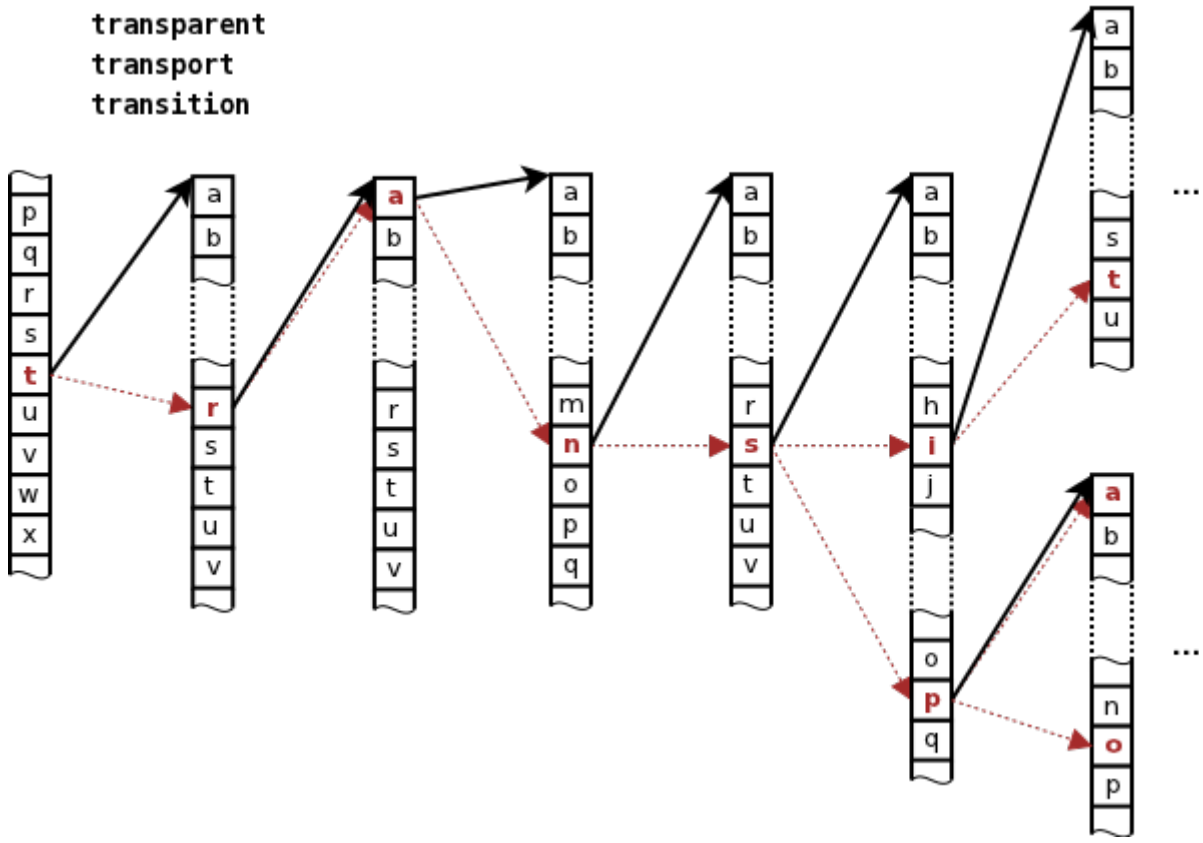
Теория

Возможно, очевидной и самой быстрой по реализации является организация словаря в виде обычного массива слов с разбором входящих строк на слова при помощи функции `strtok(3)` из GNU библиотеки языка Си и с последующим пословным сличением с содержимым словаря в цикле.

Однако, несмотря на простоту реализации, метод выглядит довольно неэффективным в отношении производительности по следующим причинам:
- функция `strtok(3)` предназначена для выделения “слов” (групп символов между символами-разделителями), из-за чего процедура поиска слова в словаре состоит из трёх шагов: 1) выявление слова в исходной строке, 2) вычленение его в отдельный объект данных (“токен”), 3) поиск токена в словаре. Основная проблема тут, на мой взгляд, в том, что наличие слова в словаре вообще-то можно выяснить уже на этапе 1, причём в каких-то случаях это возможно до достижения конца слова, что отменяет остальные действия;
- функция поиска по словарю в худшем случае (т.е. когда искомого слова в словаре нет) будет сравнивать токен с каждым словарным словом, из-за чего, очевидно, время на поиск каждого слова будет расти линейно с ростом словаря; плохо тут не только то, что зависимость от размера словаря линейная, но и то, что время от времени будет происходить заведомо неудачный перебор, в котором нет смысла.

Описанные проблемы известны и их решение тоже известно – создание индекса: например, слова в словаре упорядочиваются по алфавиту и в отдельный (индексный) массив заносятся значения индексов элементов словаря, первых для каждого символа алфавита. Теперь при поиске не надо проверять весь словарь,

достаточно перебрать только те слова, которые начинаются на тот же символ, что и проверяемое слово. При желании можно ввести индекс следующего уровня – для вторых символов слов (по тому же принципу), но уже внутри диапазона слов с зафиксированным первым символом. В пределе таким образом можно построить полностью индексированный словарь, в котором не будет нужды в каких-либо циклических переборах – наличие или отсутствие очередного символа проверяемого слова будет определяться в словаре “сразу”, скажем, по его ASCII-коду, используемому в качестве номера элемента в массиве-индексе. Схема такого словаря приведена на рисунке ниже.



На схеме в качестве примера показано размещение в таком словаре трёх слов: `transition`, `transparent` и `transport`. Черные стрелки со сплошными линиями изображают фактические связи между массивами (по указателям), красные с пунктиром – логические и олицетворяют собой слова. Во всех массивах в качестве номера (индекса) элемента используется ASCII-код символа.

В результате, по идее, получается алгоритмически предельно быстрый поиск, но хранение такого количества массивов-индексов потребует большого объёма памяти, причём, в общем случае длина каждого массива-индекса должна быть не меньше длины используемого алфавита (при использовании ASCII-кодов символов в качестве индексов). Если же рассматривать только какие-то частные случаи, то можно, с одной стороны, уменьшить алфавит до только тех символов, которые реально присутствуют в словаре, а с другой – урезать каждый индекс до тех символов, которые встречаются на данном знакоместе среди всех слов в данном диапазоне. На первый взгляд, такой подход довольно привлекателен с точки зрения расхода памяти, однако, при более внимательном рассмотрении можно понять, что привлекательность его присутствует только в тех (очень редких) случаях, в которых

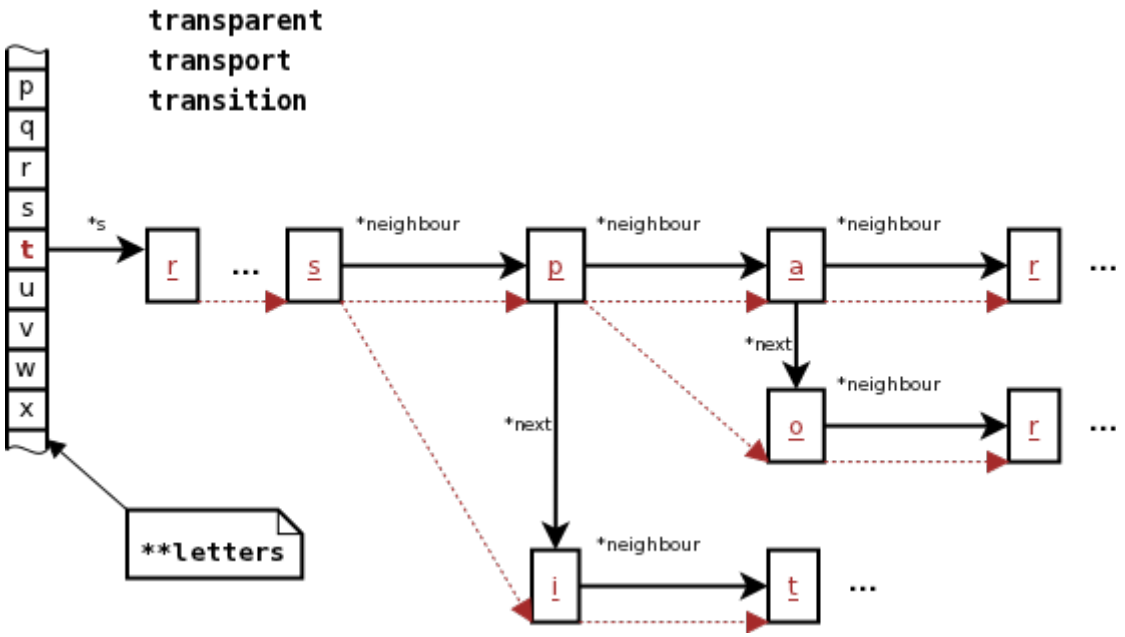
символы разных слов в рамках одного знакоместа составляют собой сплошную цепь, без разрывов. Поскольку индексные массивы как структуры данных не могут иметь разрывов и обязательно должны содержать все элементы между крайними установленными значениями вне зависимости от их использования, то разорванные цепочки символов увеличивают потребление памяти (строго говоря – впустую, просто из-за того, что массивы так устроены). Кроме того, этот подход существенно усложнит и замедлит функцию пополнения словаря.

Хорошим компромиссом тут, на мой взгляд, будет отказ от линейных массивов и переход к односвязным спискам функционально аналогичных элементов (например, в рамках языка Си - структур), что обеспечит неразрывность цепочек символов на каждом знакоместе, существенно сократит расход памяти и не приведёт к усложнению реализации. Ценой компромисса, возможно, будет небольшая потеря производительности из-за того, что для поиска нужного символа в цепочке её необходимо позвенно перебрать, “небольшая” же эта потеря из-за того, что в пределах одного знакоместа символов обычно немного (моя оценка – до 10ти, а на маленьких словарях, как в примере выше, нередко и вообще по одному).

По этой же причине (малое число элементов) я не вижу смысла реализовывать цепочки символов на конкретном знакоместе в виде деревьев – на производительности это редко когда скажется, а функция добавления слова в словарь сильно усложнится (добавится алгоритм балансировки дерева). Кроме того, может немного возрасти потребление памяти, т.к., например, в двоичных деревьях поиска узловые элементы ссылаются на двух потомков (меньшего по значению и большего либо равного), что добавит указателей в каждую структуру, представляющую символ.

Практика

Моё решение по оптимальной организации словаря выглядит следующим образом:



Черные стрелки со сплошными линиями изображают фактические связи между структурами (по указателям), красные с пунктиром – логические и олицетворяют собой слова. В качестве отправной точки словаря использован индексный массив `**letters` для первых символов слов, в котором в качестве номера (индекса) элемента выступает ASCII-код символа.

На этой схеме показаны логическая организация словаря и имена использованных в программе структур. Все эти структуры – одного вида:

```
struct structSymbol {
    char c; /* значение по умолчанию: '\0' */
    int idx; /* значение по умолчанию: -1 */
    struct structSymbol *neighbour;
    struct structSymbol *next;
};
```

Одна такая структура хранит один символ слова (или нескольких, если они до этого знакоместа совпадают). Если текущий символ слова – не последний, то поле `idx` содержит значение по умолчанию. Если текущий символ слова – последний, то `idx` содержит номер (индекс) слова в словаре и тем самым обозначает конец слова. Указатель `neighbour` ссылается на начало списка символов на следующем знакоместе, указатель `next` ссылается на символ другого слова (других слов) на текущем знакоместе.

На схеме в качестве примера показано размещение в словаре трёх слов: `transition`, `transparent` и `transport`. В процессе поиска движение производится от знакоместа к знакоместу по указателям `*neighbour`, при этом, если в какой-либо структуре, доступной “сразу” по указателю `*neighbour`, хранится другой символ, то производится перебор списка символов в пределах этого знакоместа по указателям `*next` либо до обнаружения нужного символа, либо до конца списка (что будет означать отсутствие этого слова в словаре и прекращение поиска).

Реализация

Моя реализация представляет из себя разделяемую библиотеку GNU/Linux, содержащую три варианта организации словаря: “простой” (на `strtok(3)` и одном линейном массиве – “plain”), “полностью индексированный” на индексных массивах (“array”) и “полностью индексированный” на списках структур (“структурированный” – “struct”). Работа со всеми вариантами организации производится через унифицированный программный интерфейс, тип организации задаётся значением соотв. параметра функции создания словаря и в процессе использования экземпляра словаря тип его организации сменить нельзя.

В реализации своего решения (в основном, “структурированного”) я старался достичь одновременно нескольких целей:
- максимальной производительности;
- максимальной простоты, даже примитивности;
- минимального при достигнутой производительности расхода памяти.

В процессе достижения этих целей я руководствовался следующим:

- минимум копирований чего бы то ни было, максимум работы “по месту хранения”;
- минимум повторов/циклов/переборов, действие должно совершаться “сразу”;
- минимум библиотечных функций, чем примитивнее и очевиднее алгоритм – тем лучше.

Всё это, разумеется, в переделах моего понимания каждого вопроса и наличия свободного времени на размышления и исследования.

Следствием применения этих принципов стали следующие особенности реализации:
- двойное хранение словарных слов (кроме “простого” словаря): выборка слов по списку обнаруженных в строке производится из дополнительного простого массива по номеру (индексу) слова, нумерация слов в простом массиве и в словаре совпадает (в последнем для хранения номера слова используется поле `idx` структуры-символа) – т.е. за обнаруженным словом не нужно лезть словарь, можно получить указатель на него “сразу”, по возвращённому из функции поиска его индексу;
- однопроходная обработка строки (кроме “простого” словаря): входная строка анализируется вся за раз и при этом прочитывается только один раз, без каких-либо циклов или возвратов по ней; в процессе обработки она также не модифицируется;
- в функциях поиска (кроме “простого” словаря) собственно код поиска насчитывает меньше 30 строк, причём этот код не содержит библиотечных вызовов и включает только примитивные операции – сравнение, присваивание и инкремент (для посчёта числа вхождений словарных слов в строку), т.е. при сильном желании это можно без особого труда переписать на ассемблере.

Основное внимание в библиотеке уделено функциям поиска (среди которых основное внимание уделено “структурированному” варианту). Остальные функции я счёл менее критичными, из-за чего вероятность найти в них неэффектиный код выше.

Для ускорения и упрощения функции очистки памяти из-под словаря несколько усложнено её выделение (кроме “простого” варианта) – за раз выделяется сразу несколько блоков запрашиваемого размера, в связи с чем ведётся их учёт. Вследствие этого, при последующих запросах памяти отдаются неиспользованные блоки из числа уже выделенных (до их исчерпания), без обращения к ядру ОС за новыми блоками. Благодаря такому подходу функция освобождения памяти отдаёт её ядру тоже по несколько блоков за раз. Количество выделяемых и освобождаемых за раз блоков одинаково и регулируется соотв. параметрами функции создания объекта словаря – бОльшее значение ускоряет работу, но может вызвать бОльшее потребление памяти впустую (например, когда выделено 5 раз по 100 блоков, а используется только 403). Полагаю, что это значение нужно подбирать экспериментально, для чего в структуру-описатель словаря введено несколько “диагностических” полей. Сама структура имеет следующий вид –

```
struct dict {
    char **dictWords;
    void **letters;
    struct memArea *symbolsMem;
    struct memArea *lastSymbolMemBlock;
    struct memArea *lettersMem;
    struct memArea *lastLettersMemBlock;
    int arrSize;
```

```
    int strSize;
    intptr_t *results;
    int dictType;
    int threadSafety;
    int oneStorage;
    int numOfWords;
    int numOfSymbols;
    int greatestLen;
    int lastAddedWordIdx;
    int memUsed; /* диагностическое */
    int actualNumOfWordPtrs; /* диагностическое */
    int actualNumOfSymbols; /* диагностическое */
    int actualNumOfLetters; /* диагностическое */
}
```

где:
`dictWords` – “простой” массив словарных слов, для ускорения доступа к ним и как основное место хранения слов в “простом” варианте.

`letters` – индекс по первым символам словарных слов, с него начинается словарь, в “простом” варианте не используется. При использовании в словаре конкретного типа (“структурированного” или на массивах), явно приводится к нему (например, так: `struct structSymbol **letters = (struct structSymbol **) dictionary->letters;`).

`symbolsMem`, `lettersMem`, `lastSymbolMemBlock`, `lastLettersMemBlock` – указатели на области памяти с несколькими блоками данных соотв. типов, выделяемыми за раз. Первые два указателя указывают на начала этих областей, вторые – на их последние блоки. В “простом” варианте не используются.

`arrSize`, `strSize` – рассчитанные после указания типа словаря размеры внутренних структур данных, используются в функциях выделения памяти. В “простом” варианте не используются.

`results` – массив для хранения номеров (индексов) словарных слов, обнаруженных в строке. Порядок следования номеров в этом массиве совпадает с порядком появления словарных слов в анализируемой строке. Фактический размер массива превышает запрошенный на 1 элемент – нулевой элемент используется для хранения запрошенного размера этого массива: это необходимо для разработки модулей расширения (например, для Perl) при помощи SWIG.

`dictType` – тип словаря, задаёт способ его организации. В рамках библиотеки для указания типа словаря введены следующие макроопределения:

```
#define PLAIN_DICT 0 /* простой */
#define STRUCT_DICT 1 /* структурированный */
#define ARRAY_DICT 2 /* на массивах */
```

`threadSafety` – включает поддержку многопоточной работы. При включённой поддержке несколько потоков могут использовать один словарь для одновременной обработки разных строк. В этом случае в каждом потоке должен быть определён свой массив `results`, который нужно будет передавать в функцию поиска. В рамках библиотеки для указания режима работы введены следующие макроопределения:

```
#define THREADS_UNSAFE 0 /* поддержки многопоточной работы нет */
#define THREADS_SAFE 1 /* поддержка многопоточной работы есть */
```

`oneStorage` – включает или выключает использование массива `dictWords`. Если вне поиска в самих обнаруженных словах нужды нет и достаточно только их индексов, то можно несколько уменьшить расход памяти, отключив хранение копий словарных слов в отдельном массиве. Игнорируется в “простом” варианте, так как в нём массив `dictWords` является единственным местом хранения слов. В рамках библиотеки для указания режима использования массива `dictWords` введены следующие макроопределения:

```
#define ONE_STORAGE 0 /* массив dictWords не используется */
#define DUPLICATED_STORAGE 1 /* массив dictWords используется */
```

`numOfWords` – начальное значение и шаг наращивания выделенной памяти для хранения словарных слов в “простом” массиве (который используется для быстрой выборки), в “штуках” (кол-ве слов).

`numOfSymbols` – начальное значение и шаг наращивания выделенной памяти для хранения структур-символов при древовидной и полностью индексированной организациях словаря, в “штуках” (кол-ве символов). В “простом” варианте не используется.

`greatestLen` – длина наибольшего словарного слова, используется в функциях распечатки словарей. В “простом” варианте не используется.

`lastAddedWordIdx` – используется для определения фактического количества слов в словаре в функциях выделения памяти и для указания номера слова при его добавлении в словарь.

`memUsed` – объём фактически использованной словарём памяти в байтах.

`actualNumOfWordPtrs` – фактическое количество запрошенных у ядра блоков памяти под указатели на словарные слова в простом массиве. Наращивается каждый раз на значение `numOfWords`.

`actualNumOfSymbols` – фактическое количество внесённых в словарь структур-символов с учётом того, что один символ вносимого в словарь слова требует для своего хранения одну такую структуру. В “простом” варианте не используется.

`actualNumOfLetters` – фактическое количество внесённых в словарь индексных массивов (при древовидной организации – один, с которого и начинается словарь). В “простом” варианте не используется.

Использование

Работа со словарём производится через следующие интерфейсные функции:

`struct dict *``initDict(`**`int`**` dictType, `**`int`**` threadSafety, `**`int`**` oneStorage, `**`int`**` numOfWords, `**`int`**` numOfSymbols, `**`int`**` numOfResults)` – выделяет память под основные структуры словаря, инициализирует их и возвращает указатель на объект словаря, который далее используется в других функциях. Параметры соответствуют одноимённым полям структуры-описателя словаря (кроме

numOfResults, который используется только при инициализации массива results и далее хранится уже в нём). При включённой поддержке многопоточной работы (threadSafety == THREADS_SAFE) параметр numOfResults игнорируется и память под массив results не выделяется (это нужно делать в каждом потоке независимо).

int putToDict(**struct dict ***dictionary, **char ***str) – помещает слово по *str в словарь dictionary. Возвращает 0 в случае успеха и индекс (номер) слова в словаре при попытке добавить в словарь уже имеющееся в нём слово. Проверка на дубликат не производится для “простого” варианта, в котором всегда возвращается 0.

int searchInDict(**struct dict ***dictionary, **char ***str, **intptr_t ***results) – ищет в строке str все слова из словаря dictionary и возвращает общее число вхождений (и единственных, и множественных). Если возвращённое число вхождений превышает значение numOfResults, заданное при инициализации словаря или через функцию resizeResults(), то это означает, что ёмкости массива results нехватило для размещения всех вхождений и информация о части из них была отброшена. Параметр results используется при поточнобезопасном поиске (в этом случае у каждого потока должен быть свой массив для результатов) по общему словарю. Перед каждым поиском массив results очищается.

void resizeResults(**struct dict ***dictionary, **int** newSize) – изменяет размер массива dictionary->results на указанное в newSize (в “штуках”). Не работает в режиме поддержки многопоточности.

void printDict(**struct dict ***dictionary) – выводит содержимое словаря dictionary на экран (точнее, в stdout);

void freeDict(**struct dict ***dictionary) – освобождает всю занятую словарём dictionary память и уничтожает сам объект словаря, после вызова этой функции использовать указатель *dictionary (и всё, что “под” ним) в прежнем контексте нельзя.

Кроме интерфейсных функций, имеется ряд “внутренних” вспомогательных функций, не предназначенных для использования вне словаря:

char ****getWordsMem(struct dict ***dictionary) – увеличивает размер линейного массива dictWords на величину dictionary->numOfWords для размещения очередной группы слов и возвращает указатель на него.

void ***getSymbolMem(struct dict ***dictionary) – возвращает указатель на очередную проинициализированную (все поля содержат 0 или NULL, соответственно своему типу) структуру (например, structSymbol) для хранения символов в контексте знакомест;

void ****getLettersMem(struct dict ***dictionary) – возвращает указатель на очередной проинициализированный (все элементы содержат NULL) массив-индекс (которым, в частности, является упомянутый выше **letters).

Для иллюстрации работы с библиотекой и оценки возможностей вариантов организации словарей я написал две простые программы – `dictDemo.c` для иллюстрации работы и `dictsComparison.c` для сравнительных тестов. Каждая программа допускает указание одного параметра: для программы-демонстрации это выбор типа словаря (0 – простой, 1 – “структурированный”, по умолчанию, 2 – на индексных массивах), для программы-сравнения – количество повторов поиска (по умолчанию 100 000).

Для сборки библиотеки и программ наберите в командной строке `'make'` (или `'make help'` для просмотра подсказки) и нажмите Enter. Для сборки необходимо наличие в системе средств разработки на языке Си. Для запуска программ необходимо добавить текущий каталог (содержащий библиотеку) к списку `LD_LIBRARY_PATH`.

Результат

В системе с Intel Core i7-2600K и 8 ГБ DDR3 PC-10666 1333 МГц ОЗУ, работающего в 2-канальном режиме, и установленной Slackware64 13.37, запуск программы-сравнения со 100 000 повторов поиска даёт следующие результаты (время выполнения просуммировано):

```
> ./dictsComparison
Parameter 'number of cycles' (positive integer) allowable.
Doing 100000 cycles of searching with each type of dictionary...
(0, PLAIN): [ 33] dict words found in string, time (secs) = [ 7.074024], mem used: 876 bytes
(1, STRUCT): [ 33] dict words found in string, time (secs) = [ 0.692989], mem used: 6160 bytes
(2, ARRAY): [ 33] dict words found in string, time (secs) = [ 0.688388], mem used: 1510952 bytes
```

Примечательно, что примерно 10-кратное ускорение поиска на “структурированной” организации словаря по сравнению с “простой” организацией достигается за счёт примерно 7-кратного роста потребления ОЗУ.

Строго говоря, “простой” вариант в процессе поиска потребляет памяти больше, чем показано выше. Из-за того, что `strtok(3)` изменяет разбираемую строку, внутри функции поиска для “простого” варианта организации словаря создаётся копия исходной строки, с которой и производится работа. Иными словами, в процессе работы “простой” вариант потребляет приведённое выше количество памяти + длина исходной строки в байтах, что ещё больше сокращает разницу между “простым” и “структурированным” вариантами по потреблению памяти (последний в процессе поиска дополнительной памяти не потребляет).

Известные проблемы

1. Поскольку массив `results` – статический, то при инициализации словаря необходимо выбирать его ёмкость с запасом. Это сделано для исключения потерь производительности во время поиска на выделение дополнительной памяти под результаты при исчерпании ёмкости этого массива.

2. Поиск по словарю не сильно интеллектуальный и не разбирает какие-либо конфликтные ситуации. Например, не разрешается ситуация с символом, который одновременно и не является частью словарного слова, и не входит в список разделителей, но расположен между словарным словом и одним из разделителей, как это произошло с символом правой скобки в том тексте, который я выбрал для

демонстрации. Последнее предложение второго абзаца анализируемого текста выглядит так:

“`Allocations performed using mmap(2) are unaffected by the RLIMIT_DATA resource limit (see getrlimit(2)).`”

“`getrlimit(2)`” является словарным словом, но из-за того, что между ним и разделителем ('.') присутствует ещё один символ не-разделитель, алгоритм поиска считает, что обнаружил слово “`getrlimit(2))`”, которое словарным уже не является.

В моей задаче такая проблема не возникает, поэтому её решение для меня пока неактуально.

3. Список символов-разделителей захардкоден, хотя хорошо бы задавать его при инициализации словаря. Я пока не придумал как сделать его изменяемым без существенных потерь производительности.

4. Библиотека и программы не рассчитаны на использование “дураком”, поэтому не содержат “защиты от дурака”.

5. Библиотека писалась быстро и просто, потому в ней минимум проверок возвращаемых значений. В частности, в ней не проверяется успех операций выделения памяти.

6. Библиотека, возможно, не подходит для немедленного применения – например, для разбора множества пар вида “параметр=значение”.