Task

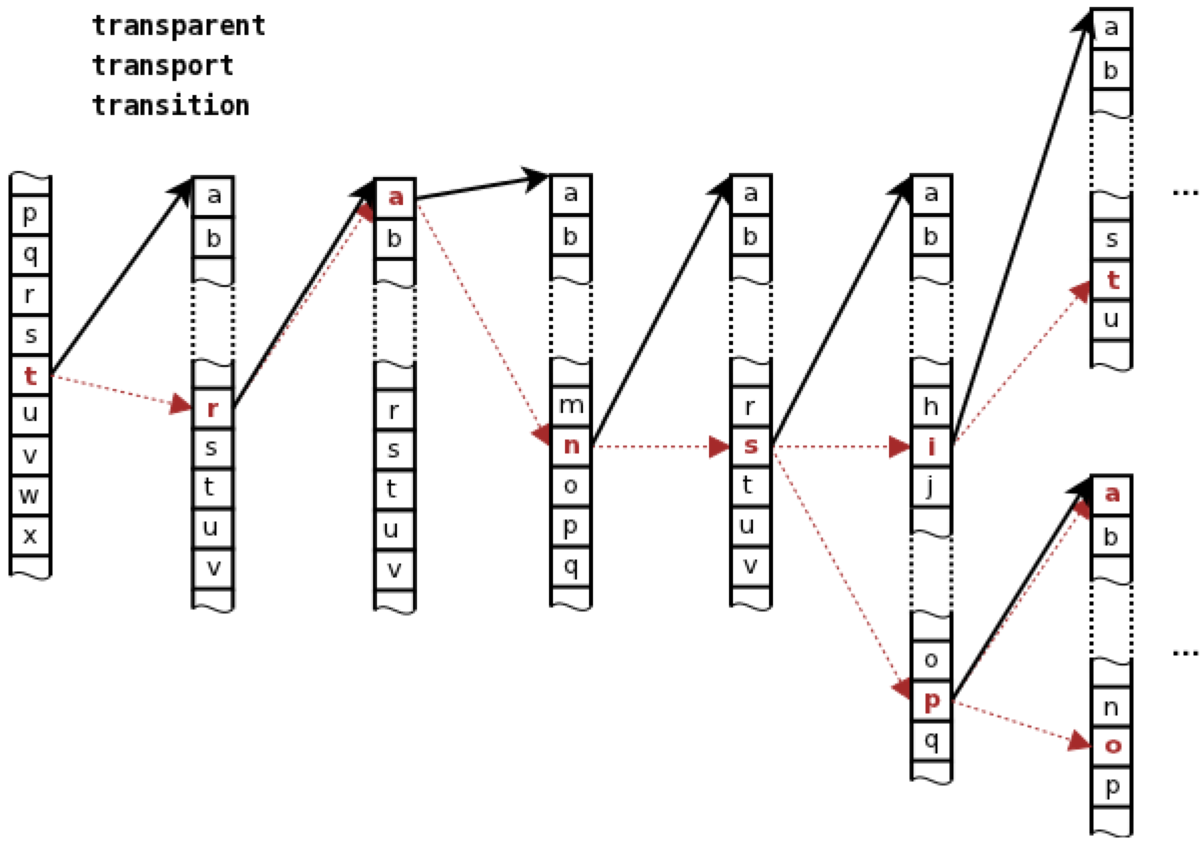
Developing high level network protocol parser I needed some tool to discover words from pre-defined dictionary in arbitrary input strings. Here are limitations/simplifications:
- strings contain only english visual symbols (say, in ASCII and with numbers from 33 to 122) and end with NULL-symbol (`'\0'`);
- dictionary is not allowed to grow or it grows too rare than reads;
- there is lot of RAM;
- dictionary size is small (up to 1000 words, but there is no hard limit);
- "word" means set of visual symbols limited by special delimiters (some of visual symbols, too) and/or the beginning/end of the string; there are several delimiters allowed between words;
- symbol case matters (i.e. "word" and "WoRd" are different words);
- search result should contain information about any dictionary word discovered in input string and could contain information about its order (relatively to other dictionary words);
- each dictionary word could be presented in input string more than once and order information in search result should contain that as new case each time;
- programming language – C (C99?), operating system – GNU/Linux with GNU C library (glibc).

Theory

Probably the most obvious and quick (in context of mind work) way to organize dictionary is just array of words (char**). Input strings could be split to tokens (by delimiters) with something like `strtok(3)` and then compared with dictionary array in loop (with each token).

Nevertheless, that way is pretty much inefficient due to following resons:
- there are several steps needed to make decision about each token: 1) discover word in input string, 2) make it token, 3) search token in dictionary. Howerver, it is possible to search word in dictionary without making it token as early as stage 1;
- in worst case ("no such word in dictionary") search function will compare token with each word in dictionary so time to search will grow fast with size of dictionary and such comparison (as beforehand unsuccessful) has no sense.

That problems were solved long time ago with indexes by something like following way: 1) words in dictionary sorted alphabetically and 2) index number for each word which is first for each symbol in alphabet put to new "index" array. Now, there is no need to check all dictionary but words started with same symbol as token. It is possible to do the same at next level – for second symbols in token and dictionary words that only in area of dictionary with fixed first symbol and so on. Finally one can to build fully indexed dictionary that eliminates any words comparison loops. Any symbol presence could be checked "immediately", say, by its ASCII number used as index. The picture of such dictionary is placed below:



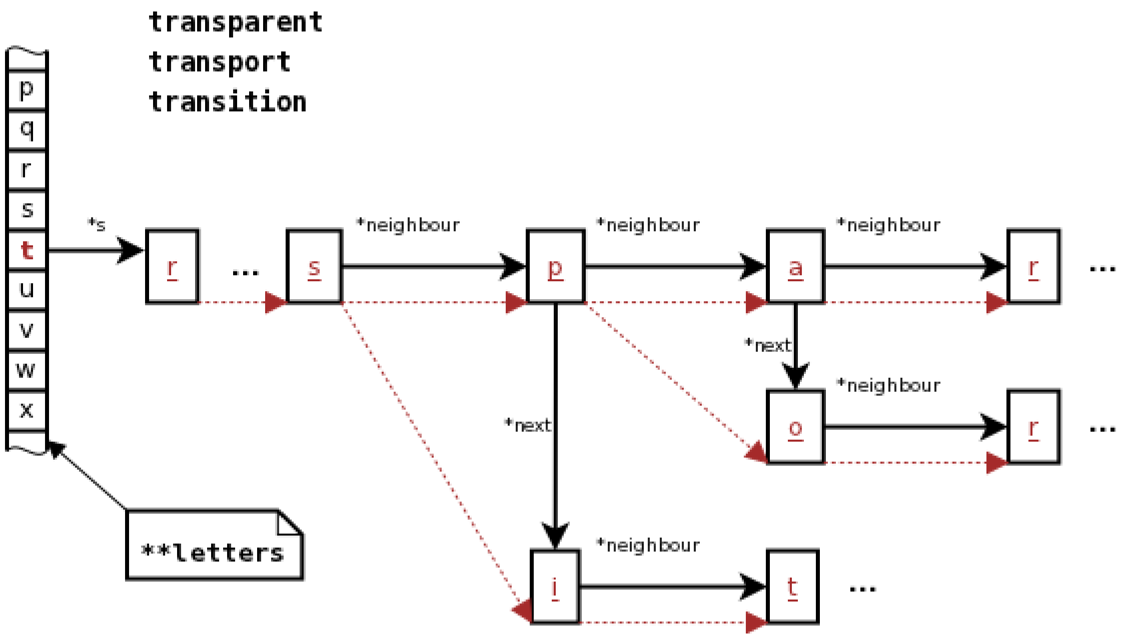
It shows fully indexed example dictionary with three words: `transition`, `transparent` and `transport`. Black arrows with solid lines express real links between arrays (with pointers), red arrows with dotted lines express logical links between symbols in words. ASCII numbers used as indexes.

As result, we probably got fastest algorithm possible but it uses large amount of memory. Moreover, each array capacity should be enough to contain all alphabet symbols in general case.
Special cases can: 1) reduce alphabet to symbols really used in dictionary words and 2) reduce each array to symbols really used in dictionary words at its symbol place. At first view it looks nice but further investigation shows that it is only possible in (very) rare cases when all (really used) symbols in one array are one continuous chain. Since “ordinary” arrays as data structres are always continuous, it means that lot of memory wasted in most cases. Besides, such “reducing” approach will lead to more complicated and slowed down push-to-dictionary function.

Refusing arrays for linked lists of functionally analogous elements (C structures, for example) will be compromise approach here I guess. They produce continuous chains at any case, reduce memory using and do no complication of source code. Some perfomance degradation possible while symbol search in each chain but it looks to me as rare case because of small number (up to 10 in small dictionaries I think) of symbols in each symbol place. This is the reason to not to use any trees here.

Practice

My suggestion of optimal dictionary organization showed below:



Black arrows with solid lines express real links between structures (by pointers), red arrows with dotted lines express logical links between symbols in words. Index array `**letters` used as start point for first words symbols and their ASCII numbers used as indexes.

The picture expresses dictionary logic and names of structures used in library (described below). All structures have one definition:

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

Each structure (named “symbol structure”) keeps either word symbol or NULL (`'\0'`) acting as word terminator (like C-string). In latter case `idx` contains word number (index). `*neighbour` points to head of list of symbols on the next symbol place, `*next` points to the symbol from the another word on current symbol place.

There are three words in dictionary showed on the picture for example: `transition`, `transparent` и `transport`. By default search goes through `*neighbour` pointers. If there is no searched symbol in the head of list on current symbol place search goes inside it through its list (by `*next` pointers) until searched symbol found or list ends. Latter means that dictionary has no word searched so search stops. If symbol found search goes by its `*neighbour` pointer to next symbol place.

If end of checked word reached (next is delimiter symbol in input string) and its last symbol is in dictionary as current then there should be NULL-symbol among other symbols on next symbol place. In this case word considered as dictionary word and its index copied from `idx` field of structure with NULL-symbol. If there is no NULL-symbol the word is not dictionary word.

Realization

I wrote GNU/Linux shared library which contains three variants of dictionary organization: “plain” (with `strtok(3)` and one index array with words – `char**`), “fully indexed” with index arrays (named “array”) and “fully indexed” with lists of structures (named “structured”). All variants are available through unified API, desired variant of organization could be set with one of parameters of dictionary initialization function and cannot be changed while dictionary is in use.

During implementation I was trying to reach following:
- maximum performance;
- maximum simplicity, even primitivity;
- minimum memory usage with achieved performance.

In process of trying that, following things were my guidelines:
- minimum duplication of anything, maximum work “at storage”;
- minimum repeats/cycles/examination of options, maximum possibilities to do something “at once”;
- minimum using of library functions, primitivity and clearness are properties of good algorithm.

All of that, of course, with limitations caused by my competence in each question and available time for thinking and investigating.

Following realization peculiarities issued from applying mentioned above:
- duplicated storage of dictionary words (except “plain” variant): there is additional “ordinary” (`char**`) array with dictionary words for quick and simple (by pointer without copying – “at once”) access to them; array indexes are the same as that ones in “main” storage;
- single-pass input string handling (except “plain” variant): the string analysed all at once and red only one time without any repeats and/or cycled reversed movements; there are no modifications made of it too;
- search functions core code (except “plain” variant) consists of about 30 lines and contains no third party libraries calls but only primitive operations – comparison, assignment and increment (to count number of entries of dictionary words to input string), so it could be implemented on assembler quite easy if needed.

I gave maximum attention to search functions (and among them – to “structured” variant). I found the others less important so they are more probable to be inefficient.

Simple and quick function of freeing memory became possible due to more complex function of getting it – there are several memory blocks requested from OS kernel at once and their calculation caused by that. Such way reduces calls to kernel and improves performance. Freeing memory function returns several blocks to kernel at once too. Numbers of requested and freed blocks are the same and set by respective parameters of dictionary initialization function. Bigger value speeds up the job but possibly increases memory usage for nothing, for example, when 100 blocks requested 5 times but only 403 blocks is in use. I guess there should be some investigation, so several “diagnostic” fields were added to dictionary descriptor (which is C structure). Here is dictionary descriptor –

```
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 numOfResults;
    int greatestLen;
    int lastAddedWordIdx;
    int memUsed; /* diagntostic */
    int actualNumOfWordPtrs; /* diagntostic */
    int actualNumOfSymbols; /* diagntostic */
    int actualNumOfLetters; /* diagntostic */
}
```

were:
`dictWords` – “ordinary” array of dictionary words for quick access to them in “structured” and “array” variants and main storage in “plain” variant.

`letters` – index array by first symbols of dictionary words; the begining of dictionary; not used in “plain” variant. Because of its type (`void**`) explicit type casting takes place if needed (example: `struct structSymbol **letters = (struct structSymbol **) dictionary->letters;`).

`symbolsMem`, `lettersMem`, `lastSymbolMemBlock`, `lastLettersMemBlock` – pointers to memory areas with blocks of data of respective type, requested “several at once”. The former two pointing to areas beginnings, the latter two – to their last blocks. Not used in “plain” variant.

`arrSize`, `strSize` – sizes of internal data structres dependent on dictionary type; used in memory allocating functions. Not used in “plain” variant.

`results` – array for indexes of dictionary words found in input string. Order of indexes in array is the same as order of dictionary words in string.

`dictType` – keeps type of dictionary organization. The library includes following definitions for better use:

```
#define PLAIN_DICT 0 /* “plain” */
#define STRUCT_DICT 1 /* “structured” */
#define ARRAY_DICT 2 /* “array” */
```

`threadSafety` – controls support of multithreading. If enabled several threads can use one dictionary to handle several strings simultaneously. Each thread should take care of its own `results` array in this case. The library includes following definitions for better use:

```
#define THREADS_UNSAFE 0 /* no threads support */
#define THREADS_SAFE 1 /* threads support enabled */
```

`oneStorage` – enables or disables use of “ordinary” array of dictionay words `dictWords`. In case of only dictionary words indexes needed in “structure” and “array” variants, `dictWords` array becomes unnecessary and memory usage could be reduced. `dictWords` is the only storage in “plain” variant so in that case value of `oneStorage` ignored. The library includes following definitions for better use:

```
#define ONE_STORAGE 0 /* dictWords is not used */
#define DUPLICATED_STORAGE 1 /* dictWords is in use */
```

`numOfWords` – initial value and step of increasing of memory amount for storing dictionary words in “ordinary” array `dictWords`. Measured in number of words.

`numOfSymbols` – initial value and step of increasing of memory amount for storing symbol structures. Measured in number of symbols. Not used in “plain” variant.

`numOfResults` – expected number of search results (from one call of search function). Measured in number of words. For perfomance reasons I refused use of any memory allocation/free/move operations during input string analysis and dictionary search so `results` array declared as static (i.e. statically allocated). See description of function `searchInDict()` below.

`greatestLen` – length of longest dictionary word, used in dictionary dump function. Not used in “plain” variant.

`lastAddedWordIdx` – keeps index of dictionary word during process of inserting it to dictionary and due to that used in memory allocation functions to get actual number of words in dictionary.

`memUsed` – keeps amount of actually used memory for dictionary. Measured in bytes.

`actualNumOfWordPtrs` – actual number of memory blocks requested from OS kernel for pointers to words in “ordinary” array `dictWords`. Increased by `numOfWords` each time when next portion of blocks requested.

`actualNumOfSymbols` – actual number of symbol structures inserted into dictionary while taking into account that storage of one symbol needs one symbol structure. Not used in “plain” variant.

`actualNumOfLetters` – actual number of index arrays inserted into dictionary (only one in “structured” variant). Not used in “plain” variant.

Usage

Following functions are intended for work with dictionary:

struct dict ***`initDict(`**int** `dictType,` **int** `threadSafety,` **int** `oneStorage,` **int** `numOfWords,` **int** `numOfSymbols,` **int** `numOfResults)` – allocates memory for dictionary main structures, initiates them and returns pointer to new dictionary descriptor which will be used in other functions. Function parameters corresponding to fields of dictionary descriptor with the same names. Memory not allocated for `results` array and parameter `numOfResults` ignored in case of multithreading support enabled (`threadSafety == THREADS_SAFE`).

void `putToDict(`**struct dict** ***`dictionary,` **char** ***`str)` – inserts word from `*str` to `*dictionary`.

int `searchInDict(`**struct dict** ***`dictionary,` **char** ***`str,` **intptr_t** ***`results,` **int** `numOfResults)` – searches all words from `*dictionary` in string `str` and returns total number of discovered dictionary words. If it greater than `numOfResults`, it means that capacity of `results` array was not enough to store all results and information about some words (which are closer to end of string) was refused. Parameters `results` and `numOfResults` used if multithreading support enabled – each thread should use its own values here in that case. Every call of this function cleans `results` array before search starts.

void `resizeResults(`**struct dict** ***`dictionary,` **int** `newSize)` – changes size of `dictionary->results` array and value of `dictionary->numOfResults` to `newSize`. Does nothing in case of multithreading support enabled.

void `printDict(`**struct dict** ***`dictionary)` – dumps `dictionary` on screen (more precisely, to `stdout`);

void `freeDict(`**struct dict** ***`dictionary)` – frees all memory allocated for `dictionary` and destroys dictionary descriptor. `*dictionary` should not be used in its previous context after call to this function – new dictionary descriptor should be created instead.

Also, there are number of “service” (internal) functions not intended for direct usage:

char ****`getWordsMem(`**struct dict** ***`dictionary)` – increases size of “ordinary” array `dictWords` by `dictionary->numOfWords` to contain next portion of dictionary words and returns pointer to it.

void ***`getSymbolMem(`**struct dict** ***`dictionary)` – allocates next symbol structure, initializes it (all fields set to `0` or `NULL`, accordingly to their types) and returns pointer to it.

void ****`getLettersMem(`**struct dict** ***`dictionary)` – allocates next index array, initializes it (all elements set to `NULL`) and returns pointer to it.

I wrote two simple programs to show how to work with library – `dictDemo.c` with just example usage and `dictsComparison.c` for performance tests and comparison between dictionary variants. Example program allows to set variant of organization: '0' for “plain”, '1' is default and for “structured“, '2' for “array”. Comparison program allows to set number of calls to search function with same dictionary and input string to dictionary of each type of organization. Default value is 100 000.

To build library and programs type 'make' and press Enter. Hope you have C development tools installed in your system. To run programs you need to add current directory (with library file) to your LD_LIBRARY_PATH list.

Result

My test system with Intel Core i7-2600K processor and 8 GBytes DDR3 PC-10666 1333 MHz RAM in dual channel mode shows following result (times were summed):

```
> ./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) = [ 7.227557], mem used: 788 bytes
(1, STRUCT): [ 33] dict words found in string, time (secs) = [ 0.709926], mem used: 7092 bytes
(2, ARRAY): [ 33] dict words found in string, time (secs) = [ 0.745603], mem used: 1511356 bytes
```

It is interesting to note that “structured” variant works by 10 times faster than “plain” variant at cost of 9 times growth of memory usage.

Known problems

1. Since `results` array is statically allocated its capacity should be chosen with reserve. I decided to do so to avoid perfomance degradation on memory allocations if its capacity is depleted during search.

2. Search algorithm is pretty simple and capitulates before any conflict situations. For example, it does nothing in case of symbol which is neither part of dictionary word nor list of delimiters and placed between dictionary word and one of delimiters in input string, as it happened in my programs with ')' symbol. The last sentence of second paragraph in text I choose for my programs looks like following:

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

“`getrlimit(2)`” is dictionary word in my programs, but symbol ')' between it and delimiter symbol '.' forces algorithm to discover word “`getrlimit(2))`” which is not dictionary word.

There is no such problem in my initial task so its solution currently not actual for me.

3. The list of delimiter symbols hardcoded. It could be nice to initialize it with dictionary at least but for the present I have no idea how to do that without performance degradation.

4. The programs and the library are not foolproof'ed.

5. The library was being written simply and quickly so it contains very few checks of returned values. For example, it has no checks of memory allocations results.

6. Probably, the library is not ready for immediate usage. For example, it isn't ready for full parsing pairs that look like “parameter=value”.