Примеры кода

Алексей Найденов

Июль 2016

Содержание

1 Поиск строк в данных, разделенных на пакеты **1**
1.1 Пример кода. 2
1.2 Программа генерации. 9

2 Разбор разделенного на пакеты tls заголовка **16**

3 Hopscotch hash table с поддержкой ttl **30**

4 Lock-free hash table **37**

5 Lock-free object allocator **47**

6 Список литературы **50**

1 Поиск строк в данных, разделенных на пакеты

Одной из проблем при разработке системы был поиск заранее заданного набора строк в данных, которые разбиты на блоки. При этом были следующие требования к программе:

- Данные разбиты на блоки (пакеты) произвольным образом. Обработку необходимо произвести за один проход.
- Минимизировать объем сохраняемой информации.
- Строки заданы в момент компиляции, но набор может меняться.
- Необходимо добиться наибольшей производительности.
- Строки начинаются с фиксированного разделителя. Эту особенность можно использовать для увеличения скорости работы.

Для решения задачи была использована машина состояний. Т.к. набор правил мог меняться в зависимости от требований, то код генерируется из описания правил при помощи python модуля cog. Для ускорения переходов между состояниями используется jump table и переходы на метки. Т.к. строки имеют одинаковые префиксы, то используются векторные инструкции для обнаружения начала. Поиск происходит за один проход, для хранения состояния используется 2 байта.

В примере приведены правила для обнаружения “Content-Type” в http заголовке. Т.к. “octet-stream” может содержать различный тип данных, то используется вложенное сравнение. Поиск останавливается при обнаружении двойного перевода строки.

1.1 Пример кода.

```
/*[[[cog
from code_generation import *
video_rules = [("ftypmp42", "HTTP_VIDEO"),
               ("ftypisom", "HTTP_VIDEO")
               ]
content_rules = [("\r\nContent-Type: audio/", "HTTP_AUDIO"),
                 ("\r\nContent-Type: application/octet-stream",
                  ("video", video_rules)),
                 ("\r\n\r\n", "HTTP")
                ]
]]]*/
//[[[end]]]

result_t search_http_media(uint16_t state, const uint8_t *first,
                           const uint8_t *last) {
  uint8_t match_offset = state & 0xff;
  uint8_t jump_index = (state >> 8) & 0xff;
  uint32_t result = HTTP;
  /*[[[cog
  write_match_code("content", content_rules)
  ]]]*/
  uint8_t content_separator = 0xd;
  uint64_t content_separator_vector = 0xd0d0d0d0d0d0d0dull;
  uint8_t video_separator = 0x66;
  uint64_t video_separator_vector = 0x6666666666666666ull;
  const char *string_content_1 = "\x0a";
  unsigned string_content_1_length = 1;
  const char *string_content_2 = "\x0a";
  unsigned string_content_2_length = 1;
  const char *string_content_3 = "ontent-Type: a";
  unsigned string_content_3_length = 14;
  const char *string_content_4 = "plication/octet-stream";
  unsigned string_content_4_length = 22;
  const char *string_video_6 = "typ";
  unsigned string_video_6_length = 3;
  const char *string_video_7 = "som";
  unsigned string_video_7_length = 3;
  const char *string_video_8 = "p42";
  unsigned string_video_8_length = 3;
  const char *string_content_9 = "dio/";
  unsigned string_content_9_length = 4;
  static void *dispatch_table[] = {&&label_content_root, &&label_content_1,
                                   &&label_content_2,    &&label_content_3,
                                   &&label_content_4,    &&label_video_root,
                                   &&label_video_6,      &&label_video_7,
                                   &&label_video_8,      &&label_content_9};
  goto *dispatch_table[jump_index];
  while (true) {
  label_content_root:

    if (last - first < 16) {
```

```
    while (first != last) {
      if (*first == content_separator) {
        ++first;
        jump_index = 1;
        goto *dispatch_table[jump_index];
      }
      ++first;
    }
  } else {
    auto first_aligned = reinterpret_cast<const uint64_t *>(
        reinterpret_cast<uintptr_t>(first + 7) & 0xfffffffffffffff8ull);
    auto last_aligned = reinterpret_cast<const uint64_t *>(
        reinterpret_cast<uintptr_t>(last) & 0xfffffffffffffff8ull);
    while (first != reinterpret_cast<const uint8_t *>(first_aligned)) {
      if (*first == content_separator) {
        ++first;
        jump_index = 1;
        goto *dispatch_table[jump_index];
      }
      ++first;
    }
    first = simd_find(first_aligned, last_aligned, content_separator_vector);
    if (first != reinterpret_cast<const uint8_t *>(last_aligned)) {
      ++first;
      jump_index = 1;
      goto *dispatch_table[jump_index];
    }
    while (first != last) {
      if (*first == content_separator) {
        ++first;
        jump_index = 1;
        goto *dispatch_table[jump_index];
      }
      ++first;
    }
  }
  goto label_exit;
label_content_1:

  if (string_content_1_length - match_offset <= last - first) {
    while (match_offset < string_content_1_length) {
      if (*first != string_content_1[match_offset]) {
        jump_index = 0;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
  } else {
    while (first != last) {
      if (*first != string_content_1[match_offset]) {
        jump_index = 0;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
    goto label_exit;
  }

  if (first == last) {
    goto label_exit;
  }
  match_offset = 0;
  switch (*first) {
  case '\x0d':
    ++first;
    jump_index = 2;
    break;
  case 'C':
    ++first;
    jump_index = 3;
    break;
  default:
    jump_index = 0;
  }
  goto *dispatch_table[jump_index];

label_content_2:

  if (string_content_2_length - match_offset <= last - first) {
    while (match_offset < string_content_2_length) {
      if (*first != string_content_2[match_offset]) {
        jump_index = 0;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
  } else {
    while (first != last) {
      if (*first != string_content_2[match_offset]) {
        jump_index = 0;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
    goto label_exit;
  }

  jump_index = 0;
  match_offset = 0;
  result = HTTP;
  goto label_exit;

label_content_3:

  if (string_content_3_length - match_offset <= last - first) {
    while (match_offset < string_content_3_length) {
      if (*first != string_content_3[match_offset]) {
        jump_index = 0;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
  } else {
    while (first != last) {
      if (*first != string_content_3[match_offset]) {
        jump_index = 0;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
    goto label_exit;
  }

  if (first == last) {
    goto label_exit;
  }
  match_offset = 0;
  switch (*first) {
  case 'p':
    ++first;
    jump_index = 4;
    break;
  case 'u':
    ++first;
    jump_index = 9;
    break;
  default:
    jump_index = 0;
  }
  goto *dispatch_table[jump_index];

label_content_4:

  if (string_content_4_length - match_offset <= last - first) {
    while (match_offset < string_content_4_length) {
      if (*first != string_content_4[match_offset]) {
        jump_index = 0;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
  } else {
    while (first != last) {
      if (*first != string_content_4[match_offset]) {
        jump_index = 0;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
    goto label_exit;
  }

  jump_index = 5;
  match_offset = 0;
  goto *dispatch_table[jump_index];

label_video_root:

  if (last - first < 16) {
    while (first != last) {
      if (*first == video_separator) {
        ++first;
        jump_index = 6;
        goto *dispatch_table[jump_index];
      }
      ++first;
    }
  } else {
    auto first_aligned = reinterpret_cast<const uint64_t *>(
        reinterpret_cast<uintptr_t>(first + 7) & 0xfffffffffffffff8ull);
    auto last_aligned = reinterpret_cast<const uint64_t *>(
        reinterpret_cast<uintptr_t>(last) & 0xfffffffffffffff8ull);
    while (first != reinterpret_cast<const uint8_t *>(first_aligned)) {
      if (*first == video_separator) {
        ++first;
        jump_index = 6;
        goto *dispatch_table[jump_index];
      }
      ++first;
    }
    first = simd_find(first_aligned, last_aligned, video_separator_vector);
    if (first != reinterpret_cast<const uint8_t *>(last_aligned)) {
      ++first;
      jump_index = 6;
      goto *dispatch_table[jump_index];
    }
    while (first != last) {
      if (*first == video_separator) {
        ++first;
        jump_index = 6;
        goto *dispatch_table[jump_index];
      }
      ++first;
    }
  }
  goto label_exit;
label_video_6:

  if (string_video_6_length - match_offset <= last - first) {
    while (match_offset < string_video_6_length) {
      if (*first != string_video_6[match_offset]) {
        jump_index = 5;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
  } else {
    while (first != last) {
      if (*first != string_video_6[match_offset]) {
        jump_index = 5;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
    goto label_exit;
  }

  if (first == last) {
    goto label_exit;
  }
  match_offset = 0;
  switch (*first) {
  case 'i':
    ++first;
    jump_index = 7;
    break;
  case 'm':
    ++first;
    jump_index = 8;
    break;
  default:
    jump_index = 5;
  }
  goto *dispatch_table[jump_index];
```

```
label_video_7:

  if (string_video_7_length - match_offset <= last - first) {
    while (match_offset < string_video_7_length) {
      if (*first != string_video_7[match_offset]) {
        jump_index = 5;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
  } else {
    while (first != last) {
      if (*first != string_video_7[match_offset]) {
        jump_index = 5;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
    goto label_exit;
  }

  jump_index = 0;
  match_offset = 0;
  result = HTTP_VIDEO;
  goto label_exit;

label_video_8:

  if (string_video_8_length - match_offset <= last - first) {
    while (match_offset < string_video_8_length) {
      if (*first != string_video_8[match_offset]) {
        jump_index = 5;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
  } else {
    while (first != last) {
      if (*first != string_video_8[match_offset]) {
        jump_index = 5;
        match_offset = 0;
        goto *dispatch_table[jump_index];
      }
      ++first;
      ++match_offset;
    }
    goto label_exit;
    }

    jump_index = 0;
    match_offset = 0;
    result = HTTP_VIDEO;
    goto label_exit;

  label_content_9:

    if (string_content_9_length - match_offset <= last - first) {
      while (match_offset < string_content_9_length) {
        if (*first != string_content_9[match_offset]) {
          jump_index = 0;
          match_offset = 0;
          goto *dispatch_table[jump_index];
        }
        ++first;
        ++match_offset;
      }
    } else {
      while (first != last) {
        if (*first != string_content_9[match_offset]) {
          jump_index = 0;
          match_offset = 0;
          goto *dispatch_table[jump_index];
        }
        ++first;
        ++match_offset;
      }
      goto label_exit;
    }

    jump_index = 0;
    match_offset = 0;
    result = HTTP_AUDIO;
    goto label_exit;

  } // while true
label_exit:
  //[[[end]]]
  result_t combined_result;
  combined_result.stop = first;
  combined_result.result = result;
  combined_result.state = jump_index;
  combined_result.state <<= 8;
  combined_result.state += match_offset;
  return combined_result;
}
```

1.2 Программа генерации.

```
import itertools
import cog
```

```python
_SEPARATOR_SUFFIX = "_separator"
_SEPARATOR_VECTOR_SUFFIX = "_separator_vector"
_OFFSET_NAME = "match_offset"
_INDEX_NAME = "jump_index"
_DISPATCH_CALL = "goto *dispatch_table[jump_index];"
_STOP_PARSER = "goto label_exit;"
_STRING_NAME_TEMPLATE = 'string_{prefix}_{index}'
_STRING_LENGTH_TEMPLATE = 'string_{prefix}_{index}_length'
_STRING_TEMPLATE = ('const char *string_{prefix}_{index} = "{value}";\n'
                    + 'unsigned string_{prefix}_{index}_length = {length};')

def _create_separator_vector(prefix, rules):
    """Create uint64_t filled with separators for simd search.

    It is assumed that all rules start with the same symbol which is
    going to be used to speed up search. If rules start with different
    symbols then every byte of a packet must be checked by multiple ifs.

    """

    first_symbols = [n[0] for (n, _) in rules]
    separator = ord(first_symbols[0])
    assert(all([separator == ord(s) for s in first_symbols]))
    separator_vector = separator;
    for i in range(7):
        separator_vector <<= 8;
        separator_vector += separator
    return ('uint8_t {}{} = 0x{:x};\n'.format(prefix, _SEPARATOR_SUFFIX, separator)
            + "uint64_t {}{} = 0x{:x}ull;".format(prefix, _SEPARATOR_VECTOR_SUFFIX,
                                                   separator_vector))

def _indent(value, level):
    if isinstance(value, str):
        return 4*level*' ' + value
    return [4*level*' ' + v for v in value]

def _escape(s):
    result = ''
    for l in s:
        if l < ' ':
            l = '\\x{:02x}'.format(ord(l))
        result += l
    return result

def _chop_from_rules(rules, count):
    return [(p[count:], r) for (p, r) in rules]

class MatchTreeNode:
    """Stores information about node and generate pieces of code.

    Contains functions for tree traversal.
```

```
"""
def __init__(self, prefix, rules, jump_index, root):
    self._prefix = prefix
    self._rules = rules
    self._jump_index = next(jump_index)
    self._root = root
    self._children = {}
    self._resolution = None
    self._next_tree = None

def get_separator_definitions(self):
    definitions = []
    definition = self._get_separator_definition()
    if definition:
        definitions.append(definition)
    for child in self._children.values():
        definition = child.get_separator_definitions()
        if definition:
            definitions.append(definition)
    return '\n'.join(definitions)

def _get_separator_definition(self):
    return None

def get_constant_definitions(self):
    constants = []
    value = self._get_constant_definition()
    if value:
        constants.append(value)
    for child in sorted(self._children.values(),
                        key=lambda c: c._get_jump_index()):
        value = child.get_constant_definitions()
        if value:
            constants.append(value)
    return '\n'.join(constants)

def _get_constant_definition(self):
    return None

def _get_jump_index(self):
    return self._jump_index

def get_dispatch_table(self):
    table = ["static void* dispatch_table[] = {"]
    index_labels = sorted(self._get_index_label_pairs(),
                          key = lambda il: il[0])
    table_body = []
    for (_, label) in index_labels:
        table_body.append("&&" + label)
    table.append(", ".join(table_body))
    table.append("};")
    return '\n'.join(table)
    def _get_index_label_pairs(self):
        """Return list of pairs (jump table index, jump label).

        Before writing dispatch table it must be sorted according to
        node index. This functions create a list for a subtree, these
        lists are combined and sorted.

        """
        labels = [(self._get_jump_index(), self._get_jump_label())]
        for child in self._children.values():
            labels.extend(child._get_index_label_pairs())
        return labels

    def get_body(self):
        body = [self._get_jump_label() + ':']
        for l in self._get_body():
            body.append(_indent(l, 1))
        for child in sorted(self._children.values(),
                            key=lambda c: c._get_jump_index()):
            body.append(child.get_body())
        return '\n'.join(body)

class RootMatchTreeNode(MatchTreeNode):
    """Implementation of a tree node for a root.

    """
    def __init__(self, prefix, rules, jump_index=None):
        jump_index = jump_index if jump_index else itertools.count()
        super().__init__(prefix, rules, jump_index, None)
        self._child = InternalMatchTreeNode(prefix, rules, jump_index, self)
        self._children[' '] = self._child

    def _get_separator_definition(self):
        return _create_separator_vector(self._prefix, self._rules)

    def _get_jump_label(self):
        return "label_{}_root".format(self._prefix)

    def _get_body(self):
        dispatch_code = (
            'jump_index = {}; '.format(self._child._get_jump_index())
            + _DISPATCH_CALL)
        body = [
            """
            if (last - first < 16) {{
                while (first != last) {{
                    if (*first == {separator}) {{
                        ++first;
                        {dispatch}
                    }}
                    ++first;
                }}
            }} else {{
                auto first_aligned =
                    reinterpret_cast<const uint64_t *>(
                        reinterpret_cast<uintptr_t>(first + 7) & 0xfffffffffffffff8ull);
                auto last_aligned =
                     reinterpret_cast<const uint64_t *>(
                         reinterpret_cast<uintptr_t>(last) & 0xfffffffffffffff8ull);
                while (first != reinterpret_cast<const uint8_t *>(first_aligned)) {{
                    if (*first == {separator}) {{
                        ++first;
                        {dispatch}
                    }}
                    ++first;
                }}
                first = simd_find(first_aligned, last_aligned,
                                  {separator_vector});
                if (first != reinterpret_cast<const uint8_t *>(last_aligned)) {{
                    ++first;
                    {dispatch}
                }}
                while (first != last) {{
                    if (*first == {separator}) {{
                        ++first;
                        {dispatch}
                    }}
                    ++first;
                }}
            }}
            {stop}""".format(dispatch=dispatch_code, stop=_STOP_PARSER,
                             separator=self._prefix + _SEPARATOR_SUFFIX,
                             separator_vector=self._prefix + _SEPARATOR_VECTOR_SUFFIX)
        ]
        return body

def _get_common_prefix(strings):
    common_length = min(len(s) for s in strings)
    index = 0
    all_equal = True
    while index < common_length:
        common_letter = strings[0][index]
        for s in strings:
            if s[index] != common_letter:
                all_equal = False
        if not all_equal:
            break
        index += 1
    return strings[0][:index]

class InternalMatchTreeNode(MatchTreeNode):
    """Implementation of an internal node.

    """
    def __init__(self, prefix, rules, jump_index, root):
        super().__init__(prefix, rules, jump_index, root)
        common_prefix = _get_common_prefix([p for (p, r) in rules])
        child_rules = sorted(_chop_from_rules(rules, len(common_prefix)),
                             key=lambda r: r[0])
        self._match = common_prefix[1:]
        if not child_rules[0][0]:
            assert(len(child_rules) == 1)
            result = child_rules[0][1]
            if isinstance(result, str):
                self._resolution = result
            else:
                self._next_tree = RootMatchTreeNode(result[0], result[1], jump_index)
                self._children[''] = self._next_tree
        else:
            for letter, child_group in itertools.groupby(child_rules,
                                                         key=lambda r: r[0][0]):
                self._children[letter] = InternalMatchTreeNode(
                    prefix, list(child_group), jump_index, self._root)

    def _get_jump_label(self):
        return "label_{}_{}".format(self._prefix, self._jump_index)

    def _get_constant_definition(self):
        if not self._match:
            return None
        return _STRING_TEMPLATE.format(prefix=self._prefix, index=self._jump_index,
                                       length=len(self._match),
                                       value=_escape(self._match))

    def _get_body(self):
        body = []
        if self._match:
            body.append(self._get_match_body())
        if self._next_tree:
            body.append(self._get_next_tree_body())
        elif self._children:
            body.append(self._get_children_body())
        elif self._resolution:
            body.append(self._get_resolution_body())
        return body

    def _get_match_body(self):
        reset_string = (
            'jump_index = {}; match_offset = 0; '.format(self._root._get_jump_index())
            + _DISPATCH_CALL)
        return """
        if ({string_length} - match_offset <= last - first) {{
            while (match_offset < {string_length}) {{
                if (*first != {string}[match_offset]) {{
                    {reset}
                }}
                ++first;
                ++match_offset;
            }}
        }} else {{
            while (first != last) {{
                if (*first != {string}[match_offset]) {{
                    {reset}
                }}
                ++first;
                ++match_offset;
            }}
            {stop}
        }}""".format(stop=_STOP_PARSER, reset=reset_string,
                         string_length=_STRING_LENGTH_TEMPLATE.format(
                             prefix=self._prefix, index=self._jump_index),
                         string=_STRING_NAME_TEMPLATE.format(
                             prefix=self._prefix, index=self._jump_index))

    def _get_children_body(self):
        body = ['', 'if (first == last) {', '    ' + _STOP_PARSER, '}',
                'match_offset = 0;', 'switch (*first) {']
        body = _indent(body, 2)
        for letter, child in sorted(self._children.items()):
            body.append(_indent("case '{}':".format(_escape(letter)), 3))
            body.append(_indent('++first;', 4))
            body.append(_indent(
            'jump_index = {};'.format(child._get_jump_index()), 4))
            body.append(_indent('break;', 4))
        body.append(_indent('default:', 3))
        body.append(_indent(
            'jump_index = {};'.format(self._root._get_jump_index()), 4))
        body.append(_indent('}', 2))
        body.append(_indent(_DISPATCH_CALL, 2))
        body.append('')
        return '\n'.join(body)

    def _get_next_tree_body(self):
        body = ['', 'jump_index = {}; '.format(self._next_tree._get_jump_index())
                + 'match_offset = 0; ' + _DISPATCH_CALL, '']
        return '\n'.join(_indent(body, 2))

    def _get_resolution_body(self):
        body = ['', 'jump_index = 0; match_offset = 0;',
                'result = {};'.format(self._resolution), _STOP_PARSER, '']
        return '\n'.join(_indent(body, 2))

def write_match_code(prefix, rules):
    """Create code for searching strings in a stream of packets.

    To speed up uses computed goto. It assumes that variables
    jump_index and match_offset are preserved between calls. Pointers
    first and last are boundaries of the range to process.

    """
    root = RootMatchTreeNode(prefix, rules)
    cog.outl(root.get_separator_definitions())
    cog.outl(root.get_constant_definitions())
    cog.outl(root.get_dispatch_table())
    cog.outl(_DISPATCH_CALL);
    cog.outl("while (true) {")
    cog.outl(root.get_body())
    cog.outl("}  // while true")
    cog.outl("label_exit:")
```

2 Разбор разделенного на пакеты tls заголовка

Функция для разбора tls заголовка и выделения необходимой информации, которая способна прерывать и продолжать работу на границе пакетов. Т.к. заголовок представляет собой вложенную структуру с полями и массивами переменной длинны, то для хранения состояния применяется стек. Для ускорения переходов между состояниями используется jump table. Макросы позволяют улучшить читаемость кода.

```
// Простая реализация stack фиксированного размера.
using parser_stack_t = fixed_stack_t<uint16_t, 12>;

enum class parse_result_t {
    kFail, kWorking, kResult, kDone
};

/**
   @brief Синхронизирует индексы и ссылки в jump таблице.

   Используется для создания массива с ссылками для jump и enum,
   который дает символьные имена смещениям внутры таблицы
   переходов. Для goto перехода необходимо знать индекс в массиве с
   labels. Индекс (число) должен быть привязан к некоторому имени, что
   бы в коде небыло "магических" чисел. Т.е. должно быть соответствие
   между символом индекса и символом метки перехода. Данный макрос
   содержит список общих частей двух типов символов. Это позволяет
   создать enum и массив labels сохранив соответствие.
*/
#define _LABELS(convert)                            \
    convert(record_header)                          \
    convert(record_type)                            \
    convert(record_version)                         \
    convert(record_size)                            \
    convert(handshake_header)                       \
    convert(handshake_type)                         \
    convert(handshake_size)                         \
    convert(client_hello)                           \
    convert(client_session_id)                      \
    convert(client_cipher_suites)                   \
    convert(client_compression_methods)             \
    convert(client_extensions)                      \
    convert(client_extension_type)                  \
    convert(client_extension_size)                  \
    convert(server_name_extension)                  \
    convert(server_name_list_length)                \
    convert(server_name_type)                       \
    convert(server_name_length)                 \
    convert(server_hello)                       \
    convert(certificate)                        \
    convert(certificate_length)                 \
    convert(certificate_identifier)             \
    convert(certificate_tbs_identifier)         \
    convert(certificate_tbs_version)            \
    convert(certificate_tbs_length)             \
    convert(certificate_entry_id)               \
    convert(certificate_entry_check)            \
    convert(certificate_skip_tail)              \
    convert(certificate_extensions)             \
    convert(certificate_extensions_size)        \
    convert(extension_sequence_id)              \
    convert(extension_sequence_size)            \
    convert(extension_id)                       \
    convert(vector_wrapper_id)                  \
    convert(vector_sequence_id)                 \
    convert(vector_sequence_size)               \
    convert(name_id)                            \
    convert(name_size)                          \
    convert(x690_size)                          \
    convert(skip_bytes)                         \
    convert(skip_vector_1)                      \
    convert(skip_vector_2)                      \
    convert(copy_data)                          \
    convert(working)                            \
    convert(result)                             \
    convert(done)                               \
    convert(fail)                               \


//! Преобразует символ в имя элемента enum.
#define _CONVERT_TO_INDICES(symbol) \
    _ ## symbol ## _index,

//! Преобразует символ в имя метки.
#define _CONVERT_TO_LABELS(symbol) \
    &&_ ## symbol,

namespace {
//! Связывает символьное имя с индексом в jump table.
enum jump_indices_t {
    _LABELS(_CONVERT_TO_INDICES) _labels_count
};
}  // unnamed namespace

#define _CHOOSE_MACRO3(_1, _2, _3, NAME, ...) NAME
#define _CHOOSE_MACRO4(_1, _2, _3, _4, NAME, ...) NAME
#define _CHOOSE_MACRO5(_1, _2, _3, _4, _5, NAME, ...) NAME

#define _ADD_CALL(...) _CHOOSE_MACRO3(                   \
    __VA_ARGS__, _ADD_CALL2, _ADD_CALL1, _ADD_CALL0) (__VA_ARGS__)
```

```
#define _ADD_CALL0(symbol)                          \
    parser_stack->push(_ ## symbol ## _index)

#define _ADD_CALL1(symbol, arg0)                           \
    do {                                                   \
        parser_stack->push(arg0);                          \
        parser_stack->push(_ ## symbol ## _index);         \
    } while (false)

#define _ADD_CALL2(symbol, arg0, arg1)                     \
    do {                                                   \
        parser_stack->push(arg1);                          \
        parser_stack->push(arg0);                          \
        parser_stack->push(_ ## symbol ## _index);         \
    } while (false)

#define _CONTINUE()                                \
    do {                                            \
        current_index = parser_stack->top();        \
        parser_stack->pop();                        \
        goto *_jump_table[current_index];           \
    } while (false)
#define _CONTINUE1(arg0)                            \
    do {                                            \
        current_index = parser_stack->top();        \
        parser_stack->pop();                        \
        parser_stack->push(arg0);                   \
        goto *_jump_table[current_index];           \
    } while (false)
#define _CONTINUE2(arg0, arg1)                      \
    do {                                            \
        current_index = parser_stack->top();        \
        parser_stack->pop();                        \
        parser_stack->push(arg1);                   \
        parser_stack->push(arg0);                   \
        goto *_jump_table[current_index];           \
    } while (false)

#define _CALL(...) _CHOOSE_MACRO4(                                \
    __VA_ARGS__, _CALL3, _CALL2, _CALL1, _CALL0) (__VA_ARGS__)

#define _CALL0(symbol)                               \
    do {                                            \
        current_index = _ ## symbol ## _index;      \
        goto *_jump_table[current_index];           \
    } while (false)

#define _CALL1(symbol, arg0)                        \
    do {                                            \
        parser_stack->push(arg0);                   \
        current_index = _ ## symbol ## _index;      \
        goto *_jump_table[current_index];           \
    } while (false)
```

```
#define _CALL2(symbol, arg0, arg1)              \
    do {                                        \
        parser_stack->push(arg1);               \
        parser_stack->push(arg0);               \
        current_index = _ ## symbol ## _index;  \
        goto *_jump_table[current_index];       \
    } while (false)

#define _CALL3(symbol, arg0, arg1, arg2)        \
    do {                                        \
        parser_stack->push(arg2);               \
        parser_stack->push(arg1);               \
        parser_stack->push(arg0);               \
        current_index = _ ## symbol ## _index;  \
        goto *_jump_table[current_index];       \
    } while (false)

#define _ADD_ARG(value)                         \
    parser_stack->push(value)

#define _ARG(n)                                 \
    (parser_stack->previous(n))

#define _BIND_ARGS(...) _CHOOSE_MACRO5(\
    __VA_ARGS__, _BIND_ARGS5, _BIND_ARGS4, _BIND_ARGS3, _BIND_ARGS2, \
    _BIND_ARGS1)(__VA_ARGS__)

#define _BIND_ARGS1(arg0)                       \
    auto &arg0 = parser_stack->previous(0)
#define _BIND_ARGS2(arg0, arg1)                 \
    auto &arg0 = parser_stack->previous(0);     \
    auto &arg1 = parser_stack->previous(1)
#define _BIND_ARGS3(arg0, arg1, arg2)            \
    auto &arg0 = parser_stack->previous(0);     \
    auto &arg1 = parser_stack->previous(1);     \
    auto &arg2 = parser_stack->previous(2)
#define _BIND_ARGS4(arg0, arg1, arg2, arg3)     \
    auto &arg0 = parser_stack->previous(0);     \
    auto &arg1 = parser_stack->previous(1);     \
    auto &arg2 = parser_stack->previous(2);     \
    auto &arg3 = parser_stack->previous(3)
#define _BIND_ARGS5(arg0, arg1, arg2, arg3, arg4)       \
    auto &arg0 = parser_stack->previous(0);             \
    auto &arg1 = parser_stack->previous(1);             \
    auto &arg2 = parser_stack->previous(2);             \
    auto &arg3 = parser_stack->previous(3);             \
    auto &arg4 = parser_stack->previous(4)

#define _EXIT()                                 \
    do {                                        \
        goto *_jump_table[_working_index];      \
    } while (false)
```

```
#define _EXIT_IF_NO_DATA()                             \
    do {                                               \
        if (last - it == 0) {                          \
            goto *_jump_table[_working_index];         \
        }                                              \
    } while (false)

#define _READ2(_value)                                         \
    do {                                                       \
        while (bytes_read != 2) {                              \
            if (last - it == 0) {                              \
                goto *_jump_table[_working_index];             \
            }                                                  \
            _value <<= 8;                                      \
            _value += *it++;                                   \
            ++bytes_read;                                      \
        }                                                      \
    } while (false)

#define _READ3(_value)                                         \
    do {                                                       \
        while (bytes_read != 3) {                              \
            if (last - it == 0) {                              \
                goto *_jump_table[_working_index];             \
            }                                                  \
            _value <<= 8;                                      \
            _value += *it++;                                   \
            ++bytes_read;                                      \
        }                                                      \
    } while (false)

parser_stack_t create_client_parser() {
    auto stack_obj = parser_stack_t();
    auto parser_stack = &stack_obj;
    _ADD_CALL(client_hello);
    _ADD_CALL(record_header, kHandshakeClientHello);
    return stack_obj;
}

parser_stack_t create_server_parser() {
    auto stack_obj = parser_stack_t();
    auto parser_stack = &stack_obj;
    _ADD_CALL(server_hello);
    _ADD_CALL(record_header, kHandshakeServerHello);
    return stack_obj;
}

std::pair<parse_result_t, const uint8_t *>
parse(const uint8_t *it, const uint8_t *last,
      parser_stack_t *parser_stack, uint8_t *result) {

    static void *_jump_table[] = {
        _LABELS(_CONVERT_TO_LABELS) &&_last_label
    };
    uint16_t current_index = 0;
    _CONTINUE();

_record_header:
    _CALL(record_type, 0);
_record_type:
    _EXIT_IF_NO_DATA();
    _ARG(0) = *it++;
    _CALL(record_version, 0, 0);
_record_version: {
        _BIND_ARGS(bytes_read, version);
        _READ2(version);
        parser_stack->pop_n(1);
        _CALL(record_size, 0, 0);
    }
_record_size: {
        _BIND_ARGS(bytes_read, size, version, type);
        _READ2(size);
        if (type != kHandshakeContentType || version < kMinTlsVersion
            || version > kMaxTlsVersion) {
            _CALL(fail);
        }
        uint16_t record_size = size;
        parser_stack->pop_n(4);
        _CALL(handshake_header, record_size);
    }
_handshake_header:
    _CALL(handshake_type, 0);
_handshake_type:
    _EXIT_IF_NO_DATA();
    _ARG(0) = *it++;
    _CALL(handshake_size, 0, 0);
_handshake_size: {
        _BIND_ARGS(bytes_read, size, type, record_size, expected_type);
        _EXIT_IF_NO_DATA();
        if (bytes_read == 0 && *it != 0) {
            _CALL(fail); // can only work with handshake size < 64k
        }
        _READ3(size);
        if (type != expected_type || size >= record_size) {
            _CALL(fail);
        }
        uint16_t handshake_size = size;
        parser_stack->pop_n(5);
        _CONTINUE1(handshake_size);
    }
    // ==================== client routines ====================
_client_hello: {
        auto &handshake_size = _ARG(0);
        parser_stack->pop_n(1); // handshake size not used
        _ADD_CALL(client_session_id);
        _CALL(skip_bytes, kHandshakeSkipSize);
    }
_client_session_id: {
        _ADD_CALL(client_cipher_suites);
        _CALL(skip_vector_1);
    }
_client_cipher_suites: {
        _ADD_CALL(client_compression_methods);
        _CALL(skip_vector_2, 0, 0);
    }
_client_compression_methods: {
        _ADD_CALL(client_extensions, 0, 0);
        _CALL(skip_vector_1);
    }
_client_extensions: {
        _BIND_ARGS(bytes_read, size);
        _READ2(size);
        auto extensions_size = size;
        parser_stack->pop_n(2);
        _ADD_ARG(extensions_size);
        _CALL(client_extension_type, 0, 0);
    }
_client_extension_type: {
        _BIND_ARGS(bytes_read, type);
        _READ2(type);
        parser_stack->pop_n(2);
        if (type == kServerNameExtensionType) {
            _ADD_CALL(server_name_extension);
            _CALL(skip_bytes, 2); // extension size
        }
        _CALL(client_extension_size, 0, 0);
    }
_client_extension_size: {
        _BIND_ARGS(bytes_read, size, extensions_size);
        _READ2(size);
        uint16_t extension_size = size;
        parser_stack->pop_n(2);
        extensions_size -= kExtensionHeaderSize;
        if (extensions_size < extension_size) {
            _CALL(fail);
        }
        extensions_size -= extension_size;
        if (extensions_size == 0) {
            _CALL(done);
        }
        _ADD_CALL(client_extension_type, 0, 0);
        _CALL(skip_bytes, extension_size);
    }
_server_name_extension: {
        _CALL(server_name_list_length, 0, 0);
    }
_server_name_list_length: {
        _BIND_ARGS(bytes_read, size);
        _READ2(size);
        uint16_t list_length = size;
        parser_stack->pop_n(2);
        _ADD_ARG(list_length);
        _CALL(server_name_type);
    }
_server_name_type: {
        _EXIT_IF_NO_DATA();
        uint16_t name_type = *it++;
        _ADD_ARG(name_type);
        _CALL(server_name_length, 0, 0);
    }
_server_name_length: {
        _BIND_ARGS(bytes_read, size, name_type, list_length);
        _READ2(size);
        uint16_t name_length = size;
        parser_stack->pop_n(2);
        if (name_type != kHostNameType || list_length < name_length
            || list_length - name_length != 3
            || name_length > kMaxDomainNameLength) {
            _CALL(fail);
        }
        _ADD_CALL(done);
        _CALL(copy_data, 0, name_length);
    }
    // ==================== server routines ====================
_server_hello: {
        uint16_t handshake_size = _ARG(0);
        parser_stack->pop_n(1);
        _ADD_CALL(certificate, 0, 0);
        _ADD_CALL(record_header, kHandshakeCertificate);
        _CALL(skip_bytes, handshake_size);
    }
_certificate: {
        _BIND_ARGS(handshake_size, bytes_read, size);
        _EXIT_IF_NO_DATA();
        if (bytes_read == 0 && *it != 0) {
            _CALL(fail); // can only work with certificate size < 64k
        }
        _READ3(size);
        uint16_t certificates_array_size = size;
        parser_stack->pop_n(3);
        _CALL(certificate_length, 0, 0, certificates_array_size);
    }
_certificate_length: {
        _BIND_ARGS(bytes_read, size, certificates_array_size);
        if (certificates_array_size == 0) {
            _CALL(done);
        }
        _EXIT_IF_NO_DATA();
        if (bytes_read == 0 && *it != 0) {
            _CALL(fail); // can only work with certificate size < 64k
        }
        _READ3(size);
        uint16_t certificate_size = size;
        parser_stack->pop_n(2);
        certificates_array_size -= 3;
        if (certificate_size > certificates_array_size
            || certificate_size < kCertificateFixedDataMaxSize) {
            _CALL(fail);
        }
        certificates_array_size -= certificate_size;
        _CALL(certificate_identifier);
    }
_certificate_identifier: {
        _BIND_ARGS(certificates_array_size);
        _EXIT_IF_NO_DATA();
        if (*it != x690::kSequenceId) {
            _CALL(fail);
        }
        ++it;
        _ADD_CALL(certificate_tbs_identifier);
        _CALL(x690_size, 0, 0, 0);
    }
_certificate_tbs_identifier: {
        _BIND_ARGS(length_field_size, certificate_left,
                   certificates_array_size);
        _EXIT_IF_NO_DATA();
        if (*it != x690::kSequenceId) {
            _CALL(fail);
        }
        ++it; --certificate_left;
        parser_stack->pop_n(1); // field size
        _ADD_CALL(certificate_tbs_length);
        _CALL(x690_size, 0, 0, 0);
    }
_certificate_tbs_length: {
        _BIND_ARGS(length_field_size, tbs_size, certificate_left);
        certificate_left -= length_field_size;
        certificate_left -= tbs_size;
        parser_stack->pop_n(1);
        _CALL(certificate_tbs_version, 0);
    }
_certificate_tbs_version: {
        _BIND_ARGS(bytes_compared, tbs_left, certificate_left);
        _EXIT_IF_NO_DATA();
        switch (bytes_compared++) {
            case 0:
                // context specific, constructed tag 0
                --tbs_left;
                if (*it++ == 0xa0) {
                    _CALL(certificate_tbs_version);
                }
                break;
            case 1:
                // integer field size
                --tbs_left;
                if (*it++ == 0x03) {
                    _CALL(certificate_tbs_version);
                }
                break;
            case 2:
                // integer field id
                --tbs_left;
                if (*it++ == 0x02) {
                    _CALL(certificate_tbs_version);
                }
                break;
            case 3:
                // integer length
                --tbs_left;
                if (*it++ == 0x01) {
                    _CALL(certificate_tbs_version);
                }
                break;
            case 4:
                // reached version number
                --tbs_left;
                if (((*it++) + 1) ==x690::kCertificateAllowedVersion) {
                    parser_stack->pop_n(1);
                    _CALL(certificate_entry_id);
                }
                break;
        }
        uint16_t skip_size = tbs_left + certificate_left;
        parser_stack->pop_n(2);
        _ADD_CALL(certificate_length, 0, 0);
        _CALL(skip_bytes, skip_size);
    }
_certificate_entry_id: {
        _BIND_ARGS(tbs_left);
        if (tbs_left == 0) {
            parser_stack->pop_n(1);
            _CALL(certificate_skip_tail);
        }
        if (tbs_left < x690::kMinHeaderSize) {
            _CALL(fail);
        }
        _EXIT_IF_NO_DATA();
        --tbs_left;
        uint16_t entry_id = *it++;
        _ADD_CALL(certificate_entry_check, entry_id);
        _CALL(x690_size, 0, 0, 0);
    }
_certificate_entry_check: {
        _BIND_ARGS(length_field_size, entry_size, entry_id, tbs_left);
        tbs_left -= length_field_size;
        tbs_left -= entry_size;
        if (entry_id == x690::kExtensionsIdentifier) {
            uint16_t entry_left = entry_size;
            parser_stack->pop_n(3);
            _CALL(certificate_extensions, entry_left);
        }
        uint16_t skip_size = entry_size;
        parser_stack->pop_n(3);
        _ADD_CALL(certificate_entry_id);
        _CALL(skip_bytes, skip_size);
    }
_certificate_skip_tail: {
        _BIND_ARGS(certificate_left, certificates_array_size);
        uint16_t skip_size = certificate_left;
        parser_stack->pop_n(1);
        _ADD_CALL(certificate_length, 0, 0);
        _CALL(skip_bytes, skip_size);
    }
_certificate_extensions: {
        _BIND_ARGS(entry_left);
        _EXIT_IF_NO_DATA();
        ++it; --entry_left;
        _ADD_CALL(certificate_extensions_size);
        _CALL(x690_size, 0, 0, 0);
    }
_certificate_extensions_size: {
        _BIND_ARGS(field_size, array_size, entry_left);
        entry_left -= field_size;
        if (array_size != entry_left) {
            _CALL(fail);
        }
        parser_stack->pop_n(2);
        _CALL(extension_sequence_id);
    }
_extension_sequence_id: {
        _BIND_ARGS(entry_left);
        if (entry_left == 0) {
            parser_stack->pop_n(1);
            _CALL(certificate_entry_id);
        }
        _EXIT_IF_NO_DATA();
        --entry_left;
        if (*it++ != x690::kSequenceId) {
            _CALL(fail);
        }
        _ADD_CALL(extension_sequence_size);
        _CALL(x690_size, 0, 0, 0);
    }
_extension_sequence_size: {
        _BIND_ARGS(field_size, extension_size, entry_left);
        _EXIT_IF_NO_DATA();
        entry_left -= field_size;
        if (extension_size > entry_left) {
            _CALL(fail);
        }
        entry_left -= extension_size;
        parser_stack->pop_n(1);
        _CALL(extension_id, 0);
    }
_extension_id: {
        _BIND_ARGS(bytes_compared, extension_left);
        _EXIT_IF_NO_DATA();
        switch (bytes_compared++) {
            case 0:
                --extension_left;
                if (*it++ == 6) { // identifier
                    _CALL(extension_id);
                }
                break;
            case 1:
                --extension_left;
                if (*it++ == 3) { // id ce always 3 bytes long
                    _CALL(extension_id);
                }
                break;
            case 2:
                --extension_left;
                if (*it++ == 0x55) {
                    _CALL(extension_id);
                }
                break;
            case 3:
                --extension_left;
                if (*it++ == 0x1d) {
                    _CALL(extension_id);
                }
                break;
            case 4:
                --extension_left;
                if (*it++ == x690::kAltNameId) {
                    parser_stack->pop_n(1);
                    _CALL(vector_wrapper_id);
                }
                break;
        }
        uint16_t skip_size = extension_left;
        parser_stack->pop_n(2);
        _ADD_CALL(extension_sequence_id);
        _CALL(skip_bytes, skip_size);
    }
_vector_wrapper_id: {
        _BIND_ARGS(extension_left);
        _EXIT_IF_NO_DATA();
        --extension_left;
        if (*it++ != x690::kOctetSequenceId) {
            uint16_t skip_size = extension_left;
            parser_stack->pop_n(1);
            _ADD_CALL(extension_sequence_id);
            _CALL(skip_bytes, skip_size);
        }
        _ADD_CALL(vector_sequence_id);
        _CALL(x690_size, 0, 0, 0);
    }
_vector_sequence_id: {
        _BIND_ARGS(field_size, wrapper_size, extension_left);
        _EXIT_IF_NO_DATA();
        extension_left -= field_size;
        --extension_left;
        if (*it++ != x690::kSequenceId) {
            uint16_t skip_size = extension_left;
            parser_stack->pop_n(3);
            _ADD_CALL(extension_sequence_id);
            _CALL(skip_bytes, skip_size);
        }
        parser_stack->pop_n(2);
        _ADD_CALL(vector_sequence_size);
        _CALL(x690_size, 0, 0, 0);
    }
_vector_sequence_size: {
        _BIND_ARGS(field_size, sequence_size, extension_left);
        extension_left -= field_size;
        parser_stack->pop_n(2);
        _CALL(name_id);
    }
_name_id: {
        _BIND_ARGS(extension_left);
        if (extension_left == 0) {
            parser_stack->pop_n(1);
            _CALL(extension_sequence_id);
        }
        _EXIT_IF_NO_DATA();
        --extension_left;
        uint16_t id = *it++;
        _ADD_CALL(name_size, id);
        _CALL(x690_size, 0, 0, 0);
    }
_name_size: {
        _BIND_ARGS(field_size, name_size, name_id, extension_left);
        extension_left -= field_size;
        extension_left -= name_size;
        if (name_id == x690::kDnsNameId) {
            uint16_t copy_size = name_size;
            parser_stack->pop_n(3);
            _ADD_CALL(name_id);
            _ADD_CALL(result);
            _CALL(copy_data, 0, copy_size);
        }
        uint16_t skip_size = name_size;
        parser_stack->pop_n(3);
        _ADD_CALL(name_id);
        _CALL(skip_bytes, skip_size);
    }
    // ==================== utility functions ====================
_x690_size: {
        _BIND_ARGS(field_size, field_left, size);
        _EXIT_IF_NO_DATA();
        if (field_left == 0) {
            uint8_t first_octet = *it++;
            if (!(first_octet & 0x80)) {
                uint16_t x690_size = first_octet;
                parser_stack->pop_n(3);
                _CONTINUE2(1, x690_size);
            }
            field_left = (first_octet & 0x7f);
            field_size = field_left;
            if (field_left > x690::kLengthFieldMaxSize || field_left == 0) {
                _CALL(fail);
            }
        }
        while (field_left != 0) {
            _EXIT_IF_NO_DATA();
            size <<= 8;
            size += *it++;
            --field_left;
        }
        uint16_t x690_size = size;
        uint16_t consumed = field_size + 1 /* field size marker */;
        parser_stack->pop_n(3);
        _CONTINUE2(consumed, x690_size);
    }
_skip_bytes: {
        _EXIT_IF_NO_DATA();
        auto &size = _ARG(0);
        auto data_size = last - it;
        if (data_size < size) {
            size -= data_size;
            _EXIT();
        }
        parser_stack->pop_n(1);
        it += size;
        _CONTINUE();
    }
_skip_vector_1: {
        _EXIT_IF_NO_DATA();
        uint16_t size = *it++;
        _CALL(skip_bytes, size);
    }
_skip_vector_2: {
        _BIND_ARGS(bytes_read, size);
        while (bytes_read != 2) {
            _EXIT_IF_NO_DATA();
            size <<= 8;
            size += *it++;
            ++bytes_read;
        }
        uint16_t stored_size = size;
        parser_stack->pop_n(2);
        _CALL(skip_bytes, stored_size);
    }
_copy_data: {
        _BIND_ARGS(copied_size, remaining_size);
        auto write_pos = result + copied_size;
        auto data_size = last - it;
        if (data_size < remaining_size) {
            copied_size += data_size;
            remaining_size -= data_size;
            std::copy_n(it, data_size, write_pos);
            _EXIT();
        }
        write_pos = std::copy_n(it, remaining_size, write_pos);
        it += remaining_size;
        *write_pos = 0;
        parser_stack->pop_n(2);
        _CONTINUE();
    }
_working:
    parser_stack->push(current_index);
    return std::make_pair(parse_result_t::kWorking, last);
_result:
    return std::make_pair(parse_result_t::kResult, it);
_done:
    _ADD_CALL(done);
    return std::make_pair(parse_result_t::kDone, last);
_fail:
    _ADD_CALL(fail);
    return std::make_pair(parse_result_t::kFail, last);
_last_label:
    assert(false);
    return std::make_pair(parse_result_t::kFail, last);
}
```

3 Hopscotch hash table с поддержкой ttl

Реализация hopscotch hash table [1]. Добавлена поддержка удаления элементов, к которым не было обращения в течении заданного промежутка времени.

```
template <class Key, class T, class Hash = std::hash<Key>,
          class KeyEqual = std::equal_to<Key>>
class hopscotch_ttl_hash_t {
 public:
  using key_type = Key;
  using mapped_type = T;
  using value_type = std::pair<Key, T>;
  using reference = value_type &;
  using pointer = value_type *;
  using hasher = Hash;
  using key_equal = KeyEqual;
  using time_type = uint16_t;
  using bucket_mask_type = uint32_t;

  struct entry_type {
    value_type key_value{};
    bucket_mask_type bucket_mask{0};
    time_type timestamp{0};
    bool is_empty{true};
  };

  class iterator {
  public:
    explicit iterator(entry_type *position = nullptr,
                      const entry_type *last = nullptr)
        : m_position{position}, m_last{last} {
      if (last == nullptr) {
        m_last = m_position;
      }
      assert(m_position <= m_last);
    }
    iterator(const iterator &) = default;
    iterator &operator=(const iterator &) = default;

    iterator &operator++() {
      while (m_position != m_last) {
        ++m_position;
        if (!m_position->empty()) {
          break;
        }
      }
      return *this;
    }

    iterator operator++(int) {
      auto tmp = *this;
      ++*this;
      return tmp;
    }

    reference operator*() {
      assert(m_position != nullptr);
      return m_position->key_value;
    }

    pointer operator->() {
      assert(m_position != nullptr);
      return &(m_position->key_value);
    }

    bool operator==(const iterator &other) const {
      return m_position == other.m_position;
    }
```

```
    bool operator!=(const iterator &other) const { return !(*this == other); }

  private:
    entry_type *m_position;
    const entry_type *m_last;
  }; // class iterator

  static constexpr uint32_t kBucketLength = 8 * sizeof(bucket_mask_type);
  static constexpr uint32_t kBucketMask = kBucketLength - 1;
  static constexpr uint32_t kInvalidIndex =
      std::numeric_limits<uint32_t>::max();

  hopscotch_ttl_hash_t() {}

  void set_storage(uint8_t *buffer, std::size_t size);
  void set_ttl(time_type ttl) { m_ttl = ttl; }

  bool empty() const { return size() == 0; }
  uint32_t size() const { return m_size; }
  uint32_t max_size() const;

  iterator begin() {
    auto result = iterator(m_entries, m_entries + max_size());
    if (m_entries->is_empty) {
      ++result;
    }
    return result;
  }
  iterator end() { return iterator(m_entries + max_size()); }

  // Вставляет элемент. Возвращает true, если удалось вставить (был
  // создан новый элемент). Итератор указывает на положение нового
  // элемента, элемента с тем же ключом или на конец (если не удалось
  // вставить из за переполнения).
  std::pair<iterator, bool> insert(time_type current, const key_type &key,
                                   const mapped_type &value);
  iterator find(time_type current, const key_type &key);
  std::size_t erase(const key_type &key);
  void clear();

private:
  uint32_t increment(uint32_t index) const {
    assert(index < max_size());
    return (index + 1u) & m_index_mask;
  }
  uint32_t decrement(uint32_t index) const {
    assert(index < max_size());
    return (index - 1u) & m_index_mask;
  }
  uint32_t distance(uint32_t from, uint32_t to) const {
    assert(from < max_size());
    assert(to < max_size());
    return (max_size() + to - from) & m_index_mask;
  }
  uint32_t get_index(const key_type &key) const;
  bool is_outdated(time_type timestamp, time_type current) const;
  // Найти пустую ячейку виртуальном bucket элемента или попытаться
  // переместить пустую ячейку. Возвращает kInvalidIndex если нет
  // пустого места. Может вернуть индекс не пустой ячейки, если ключи
  // совпадают.
  uint32_t find_insert_location(time_type current, const key_type &key);
  // Попытаться переместить пустую ячейку в виртуальный bucket
  // элемента.
  uint32_t move_empty_space_towards(uint32_t empty_index,
                                    uint32_t desired_index);
  // Перемещает элемент в пустую ячейку, если она находиться в
  // виртуальном bucket элемента. Упрощает код перемещения пустой
  // ячейки.
  bool try_move(uint32_t index, uint32_t empty_index);
  // Вспомогательная функция для поиска элементов, используется при
  // поиске и удалении. Возвращает kInvalidIndex, если найти не удалось.
  // Обновляет timestamp элемента если он был найден.
  uint32_t find_key(time_type current, const key_type &key);
  // Используется для удаления устаревших элементов.
  void remove(uint32_t index);

  entry_type *m_entries{nullptr};
  uint32_t m_log2_table_length{0};
  uint32_t m_index_mask{0};
  uint32_t m_size{0};
  time_type m_ttl{std::numeric_limits<time_type>::max()};
  hasher m_hash;
  key_equal m_key_equal;
};  // class hopscotch_ttl_hash_t

template <class Key, class T, class Hash, class KeyEqual>
void hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::set_storage(
    uint8_t *buffer, std::size_t size) {
  assert(buffer != 0);
  std::tie(buffer, size) =
      align_buffer(buffer, size, std::alignment_of<entry_type>::value);
  m_log2_table_length = floor_log2(size/sizeof(entry_type));
  assert((1u << m_log2_table_length) > kBucketLength &&
         "Buffer is too small for hash table");
  m_index_mask = (1u << m_log2_table_length) - 1;
  m_entries = reinterpret_cast<entry_type *>(buffer);
  clear();
}

template <class Key, class T, class Hash, class KeyEqual>
std::pair<typename hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::iterator, bool>
hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::insert(time_type current,
                                                     const key_type &key,
                                                     const mapped_type &value) {
  assert(m_entries != nullptr);
  auto insert_index = find_insert_location(current, key);
  if (insert_index == kInvalidIndex) {
    return std::make_pair(end(), false);
  }
  auto it = iterator(m_entries + insert_index, m_entries + max_size());
  if (m_entries[insert_index].is_empty) {
    auto bucket_index = get_index(key);
    assert(!is_bit_set(m_entries[bucket_index].bucket_mask,
                       distance(bucket_index, insert_index)));
    m_entries[bucket_index].bucket_mask =
        set_bit(m_entries[bucket_index].bucket_mask,
                distance(bucket_index, insert_index));
    m_entries[insert_index].is_empty = false;
    m_entries[insert_index].key_value.first = key;
    m_entries[insert_index].key_value.second = value;
    m_entries[insert_index].timestamp = current;
    ++m_size;
    return std::make_pair(it, true);
  }
  return std::make_pair(it, false);
}

template <class Key, class T, class Hash, class KeyEqual>
typename hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::iterator
hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::find(time_type current,
                                                    const key_type &key) {
  auto index = find_key(current, key);
  if (index != kInvalidIndex) {
    return iterator(m_entries + index, m_entries + max_size());
  }
  return end();
}

template <class Key, class T, class Hash, class KeyEqual>
std::size_t
hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::erase(const key_type &key) {
  auto index = find_key(0, key);
  if (index != kInvalidIndex) {
    remove(index);
    return 1;
  }
  return 0;
}

template <class Key, class T, class Hash, class KeyEqual>
void hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::clear() {
  assert(m_entries != nullptr);
  m_size = 0;
  std::fill_n(m_entries, max_size(), entry_type{});
}

template <class Key, class T, class Hash, class KeyEqual>
uint32_t hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::max_size() const {
  if (m_log2_table_length == 0) {
    return 0;
  }
  return 1 << m_log2_table_length;
}

template <class Key, class T, class Hash, class KeyEqual>
uint32_t hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::find_insert_location(
    time_type current, const key_type &key) {
  auto proper_index = get_index(key);
  auto bucket_last_index = (proper_index + kBucketLength) & m_index_mask;
  auto index = proper_index;
  // optimistically try to find space in virtual bucket
  for (; index != bucket_last_index; index = increment(index)) {
    if (m_entries[index].is_empty) {
      return index;
    }
    if (m_key_equal(key, m_entries[index].key_value.first)) {
      return index;
    }
    if (is_outdated(m_entries[index].timestamp, current)) {
      remove(index);
      return index;
    }
  }
  // check if virtual bucket is full
  if (m_entries[proper_index].bucket_mask ==
      std::numeric_limits<bucket_mask_type>::max()) {
    return kInvalidIndex;
  }
  // try to move items around to free space in virtual bucket
  for (; index != proper_index; index = increment(index)) {
    if (m_entries[index].is_empty) {
      break;
    }
    if (is_outdated(m_entries[index].timestamp, current)) {
      remove(index);
      break;
    }
  }
  if (index == proper_index) {
    return kInvalidIndex;
  }
  return move_empty_space_towards(index, proper_index);
}

template <class Key, class T, class Hash, class KeyEqual>
uint32_t hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::move_empty_space_towards(
    uint32_t empty_index, uint32_t desired_index) {
  assert(m_entries[empty_index].is_empty);
  assert(!m_entries[desired_index].is_empty);
  auto candidate_index = decrement(empty_index);
  while (distance(candidate_index, empty_index) < kBucketLength) {
    assert(distance(desired_index, empty_index) >= kBucketLength);
    if (try_move(candidate_index, empty_index)) {
      empty_index = candidate_index;
      if (distance(desired_index, empty_index) < kBucketLength) {
        return empty_index;
      }
    }
    candidate_index = decrement(candidate_index);
  }
  return kInvalidIndex;
}

template <class Key, class T, class Hash, class KeyEqual>
bool hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::try_move(
    uint32_t index, uint32_t empty_index) {
  assert(!m_entries[index].is_empty);
  assert(m_entries[empty_index].is_empty);
  auto proper_index = get_index(m_entries[index].key_value.first);
  if (distance(proper_index, empty_index) >= kBucketLength) {
    return false;
  }

  auto old_bucket_offset = distance(proper_index, index);
  auto new_bucket_offset = distance(proper_index, empty_index);
  assert(is_bit_set(m_entries[proper_index].bucket_mask, old_bucket_offset));
  m_entries[proper_index].bucket_mask =
      clear_bit(m_entries[proper_index].bucket_mask, old_bucket_offset);
  assert(!is_bit_set(m_entries[proper_index].bucket_mask, new_bucket_offset));
  m_entries[proper_index].bucket_mask =
      set_bit(m_entries[proper_index].bucket_mask, new_bucket_offset);

  m_entries[empty_index].key_value = m_entries[index].key_value;
  m_entries[empty_index].timestamp = m_entries[index].timestamp;
  m_entries[index].is_empty = true;
  m_entries[empty_index].is_empty = false;
  return true;
}

template <class Key, class T, class Hash, class KeyEqual>
uint32_t
hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::find_key(time_type current,
                                                        const key_type &key) {
  auto index = get_index(key);
  auto bucket_mask = m_entries[index].bucket_mask;
  while (bucket_mask != 0) {
    if ((bucket_mask & 1u) != 0 &&
        m_key_equal(key, m_entries[index].key_value.first)) {
      assert(!m_entries[index].is_empty);
      m_entries[index].timestamp = current;
      return index;
    }
    bucket_mask >>= 1;
    index = increment(index);
  }
  return kInvalidIndex;
}

template <class Key, class T, class Hash, class KeyEqual>
uint32_t hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::get_index(
    const key_type &key) const {
  uint32_t hash = m_hash(key);
  return (hash >> (32 - m_log2_table_length)) & m_index_mask;
}

template <class Key, class T, class Hash, class KeyEqual>
bool hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::is_outdated(
    time_type timestamp, time_type current) const {
  return static_cast<uint16_t>(current - timestamp) >= m_ttl;
}

template <class Key, class T, class Hash, class KeyEqual>
void hopscotch_ttl_hash_t<Key, T, Hash, KeyEqual>::remove(uint32_t index) {
  assert(index < max_size());
  assert(!m_entries[index].is_empty);
  m_entries[index].is_empty = true;
  auto proper_index = get_index(m_entries[index].key_value.first);
  auto in_bucket_index = distance(proper_index, index);
  assert(is_bit_set(m_entries[proper_index].bucket_mask, in_bucket_index));
  m_entries[proper_index].bucket_mask =
      clear_bit(m_entries[proper_index].bucket_mask, in_bucket_index);
  --m_size;
}
```

4 Lock-free hash table

Реализация расширяемой lock-free hash table [2].

```
/**
   @brief Простой расширяемый shared массив.

   Может только расти до определенного размера. Разбивает диапазон
   индексов на сегменты. Резервирует в памяти сегменты, к которым было
   обращение. Операции будут lock-free, если хранимый тип является
   одним из целочисленных типов. Используется в hash_table для
   хранения указателей. Создание и уничтожение объекта не
   concurrent. Не вызываемся destructor объектов при освобождении
   памяти.
*/
template <typename T, std::size_t N, class Allocator>
class atomic_segmented_array_t {
public:
  using contained_type = T;
  using value_type = std::atomic<T>;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type &;
  using const_reference = const value_type &;
  using pointer = value_type *;
  using const_pointer = const value_type *;
  using allocator_type = Allocator;

  static_assert(sizeof(typename allocator_type::value_type) /
                        sizeof(value_type) >
                    0,
                "Pool object size is too small");

  static constexpr size_type kSize = N;
  static constexpr size_type kLog2SegmentSize =
      floor_log2(sizeof(typename Allocator::value_type) / sizeof(value_type));
  static constexpr size_type kSegmentSize = (1 << kLog2SegmentSize);
  static constexpr size_type kSegmentMask = kSegmentSize - 1;
  static constexpr size_type kSegmentCount =
      (kSize + kSegmentSize - 1) / kSegmentSize;

  using segment_ptr = std::atomic<pointer>;

  atomic_segmented_array_t(atomic_segmented_array_t const&) = delete;
  atomic_segmented_array_t& operator=(atomic_segmented_array_t const&) = delete;

  explicit atomic_segmented_array_t(allocator_type allocator)
      : m_allocator{allocator} {
    std::fill_n(m_segments, kSegmentCount, nullptr);
  }

  reference operator[](size_type index) {
    assert(index <= kSize && "Out of bound access");
    auto segment_index = index >> kLog2SegmentSize;
    auto values = m_segments[segment_index].load(std::memory_order_consume);
    if (values == nullptr) {
      auto new_values = reinterpret_cast<pointer>(m_allocator.allocate(1));
      if (new_values == nullptr) {
        return m_invalid_value;
      }
      assert(((reinterpret_cast<uintptr_t>(new_values) & 7) == 0) &&
             "Pointer is not properly aligned");
      std::fill_n(new_values, kSegmentSize, contained_type{});
      if (!m_segments[segment_index].compare_exchange_strong(values,
                                                             new_values)) {
        m_allocator.deallocate(
            reinterpret_cast<typename allocator_type::pointer>(new_values), 1);
        values = m_segments[segment_index].load(std::memory_order_consume);
      } else {
        values = new_values;
      }
    }
    assert(values != nullptr && "Something is wrong with concurrent access");
    return values[index & kSegmentMask];
  }

  // check if returned reference is valid, index access might fail
```

```
  // because it might reserve memory, this object api should be redesigned
  bool is_reference_valid(reference ref) {
    return &m_invalid_value != &ref;
  }

  ~atomic_segmented_array_t() {
    for (auto &segment : m_segments) {
      auto segment_pointer = segment.load();
      if (segment_pointer != nullptr) {
        m_allocator.deallocate(
            reinterpret_cast<typename allocator_type::pointer>(segment_pointer),
            1);
      }
    }
  }

private:
  allocator_type m_allocator;
  value_type m_invalid_value; // used in case allocator fails
  segment_ptr m_segments[kSegmentCount];
}; // class atomic_segmented_array_t

/**
   функции для работы с помеченными указателями.

   Используются для реализации упорядоченной очереди. Помеченные
   указатели позволяют копировать дополнительную информацию за одну
   CAS операцию. Для маркировки используется самый младший бит,
   который в реальном указателе должен быть нулем для большинства
   типов. Используются функции а не тип для упрощения использования с
   std::atomic.
*/
template <typename T> T *mark_pointer(T *pointer) {
  static_assert(std::alignment_of<T>::value > 1,
                "Can not mark pointer because type is too short");
  pointer = reinterpret_cast<T *>(reinterpret_cast<uintptr_t>(pointer) | 1u);
  return pointer;
}
template <typename T> T *clear_pointer(T *pointer) {
  static_assert(std::alignment_of<T>::value > 1,
                "Can not mark pointer because type is too short");
  uintptr_t mask = 1u;
  mask = ~mask;
  pointer = reinterpret_cast<T *>(reinterpret_cast<uintptr_t>(pointer) & mask);
  return pointer;
}
template <typename T> bool is_pointer_marked(const T *pointer) {
  static_assert(std::alignment_of<T>::value > 1,
                "Can not mark pointer because type is too short");
  return (reinterpret_cast<uintptr_t>(pointer) & 1u) != 0;
}

template <typename T> struct node_t {
  node_t() : value{} {}
  explicit node_t(const T &v) : value{v} {}

  T value;
  std::atomic<node_t<T> *> next{nullptr};
};

/**
   @brief Lock-free упорядоченный односвязанный список

   Используется для хранения элементов hash table. Элементы хранятся в
   упорядоченном списке. Операции insert поддерживает порядок. Все
   функции имеют форму с дополнительным параметром hint, который
   используется для ускорения поиска (hash table дополнительно
   поддерживает jump table на основании hash).

   Предполагается, что ключ содержит некоторый payload, который не
   участвует в сравнении, но несет некоторую полезную
   информацию. Функция insert не заменяет найденный элемент (что бы
   сильно не отклоняться от описанного алгоритма), поэтому если
   необходимо заменить payload, то необходимо сначала удалить старую
   запись а затем вставить новую.
*/
template <typename Key, class Allocator = std::allocator<node_t<Key>>,
          class Compare = std::less<Key>>
class ordered_list_t {
 public:
  using key_type = Key;
  using value_type = Key;
  using allocator_type = Allocator;
  using key_compare = Compare;

  using pointer = node_t<value_type> *;

  explicit ordered_list_t(const allocator_type &allocator,
                          const key_compare compare = key_compare{})
      : m_allocator{allocator}, m_compare{compare} {}

  std::pair<bool, pointer> insert(const value_type &value) {
    return insert(&m_head, value);
  }
  std::pair<bool, pointer> insert(pointer hint, const value_type &value) {
    auto new_node = m_allocator.allocate(1);
    if (new_node == nullptr) {
      return std::make_pair(false, nullptr);
    }
    new_node->value = value;
    auto is_found = false;
    auto found_value = value_type{};
    auto snapshot = snapshot_t{};
    while (true) {
      std::tie(is_found, found_value, snapshot) =
          find_insert_position(hint, value);
      if (is_found) {
        // already present, abort insertion
        m_allocator.deallocate(new_node, 1);
        return std::make_pair(false, snapshot.current);
      }
      new_node->next.store(snapshot.current);
      // attempt to insert into the list
      if (snapshot.previous->next.compare_exchange_strong(snapshot.current,
                                                           new_node)) {
        return std::make_pair(true, new_node);
      }
    }
  }

  std::pair<bool, value_type> find(const value_type &value) {
    return find(&m_head, value);
  }
  std::pair<bool, value_type> find(pointer hint, const value_type &value) {
    auto is_found = false;
    auto found_value = value_type{};
    auto position = snapshot_t{};
    std::tie(is_found, found_value, position) =
        find_insert_position(hint, value);
    return std::make_pair(is_found, found_value);
  }

  bool erase(const value_type &value) { return erase(&m_head, value); }
  bool erase(pointer hint, const value_type &value) {
    auto is_found = false;
    auto found_value = value_type{};
    auto snapshot = snapshot_t{};
    while (true) {
      std::tie(is_found, found_value, snapshot) =
          find_insert_position(hint, value);
      if (!is_found) {
        return false;
      }
      auto marked_next = mark_pointer(snapshot.next);
      // try to mark current node
      if (!snapshot.current->next.compare_exchange_strong(snapshot.next,
                                                           marked_next)) {
        continue;
      }
      // try to delete current node
      if (snapshot.previous->next.compare_exchange_strong(snapshot.current,
                                                           snapshot.next)) {
        m_allocator.deallocate(snapshot.current, 1);
      } else {
        // something was inserted before current node (previous node
        // contains unexpected value), current is marked but is in the
        // list, find_insert_position will find the value again and
        // delete marked node
        find_insert_position(hint, value);
      }
      return true;
    }
  }

 private:
   // То что в описании алгоритма называют локальными переменными. Три
   // указателя возращаемые find_insert_position: current - указатель
   // на элемент с наименьшим индексом, который больше или равен
   // искомому значению; previous - элемент, который ссылается на
   // current; next - элемент, на который ссылается current. Функция
   // find_insert_position гарантирует, что в какой то момент времени
   // значения указателей соответствовали значениям в списке,
   // т.е. структура содержит snapshot куска списка. Необходимы все 3
   // значения, для операций удаления и добавления. В алгоритме на
   // месте prevoius используется указатель на указатель, однако это
   // неудобно из-за необходимости введения дополнительного внешнего
   // типа и неоднородности типа ссылок. Сейчас для указания внутрь
   // использзуется только node_t *. Если не изменить тип previous, то
   // надо использовать указатель на atomic. Недостаток в том, что в
   // некоторых случаях будет копироваться лишний ключ.
   struct snapshot_t {
     pointer previous{nullptr};
     pointer current{nullptr};
     pointer next{nullptr};
   };

   // Указатель hint должен быть действительным узлом, не удаленным и
   // не помеченным для удаления. Т.е. либо действительным началом
   // списка, либо алгоритм, использующий список, должен гарантировать
   // его присутствие. Первое значение индикатор, было ли точное
   // совпадение. Если snapshot_t::current != nullptr, то второе
   // значение больше или равно искомому.
   std::tuple<bool, value_type, snapshot_t>
   find_insert_position(pointer hint, const value_type &value) {
     assert(hint != nullptr);
     auto snapshot = snapshot_t{};
     while (true) { // traverse retry, start from fixed valid point
       assert(is_pointer_marked(hint->next.load()) == false &&
              "Starting node was marked for deletion");
       snapshot.previous = hint;
       snapshot.current = clear_pointer(snapshot.previous->next.load());
       while (true) { // traverse the list
         if (snapshot.current == nullptr) { // end of the list
           return std::make_tuple(false, value_type{}, snapshot);
         }
         snapshot.next = snapshot.current->next.load();
         auto is_current_marked = is_pointer_marked(snapshot.next);
         snapshot.next = clear_pointer(snapshot.next);
         auto current_key = snapshot.current->value;
         if (snapshot.previous->next.load() != snapshot.current) {
           // something happened to current or previous node,
           // everything is invalid, restart traverse
           break;
         }
         if (is_current_marked) {
           // marked for deleteion, delete it in case the thread that
           // marked it is hung
           if (snapshot.previous->next.compare_exchange_strong(snapshot.current,
                                                               snapshot.next)) {
             m_allocator.deallocate(snapshot.current, 1);
           } else {
             // start from fixed position, current was deleted
             break;
           }
         } else {
           if (!m_compare(current_key, value)) {
             return std::make_tuple(!m_compare(value, current_key), current_key,
                                    snapshot);
           }
           snapshot.previous = snapshot.current;
         }
         snapshot.current = snapshot.next;
       } // traverse the list
     } // traverse failed, start over
   }

   // Фиктивный узел для однородного представления указателей на
   // необходимые элементы.
   node_t<value_type> m_head;
   allocator_type m_allocator;
   key_compare m_compare;
};  // class ordered_list_t

namespace internal {
/**
   @brief Степень двойки меньше или равная аргументу.

   Может иметь другое определение на других платформах. Возвращает 0
   для нуля, соответствующая ассемблерная инструция возвращает
   неопределенное значение.
*/
inline unsigned floor_log2(unsigned x) {
  unsigned zero_arg_workaround_mask = static_cast<unsigned>(x == 0) - 1;
  return zero_arg_workaround_mask & (8*sizeof(x) - 1 - __builtin_clz(x));
}

inline unsigned get_parent(unsigned value) {
  unsigned mask = (1u << floor_log2(value)) - 1;
  return value & mask;
}

inline uint32_t reverse_bits(uint32_t x) {
  x = (((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1));
  x = (((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2));
  x = (((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4));
  x = (((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8));
  return ((x >> 16) | (x << 16));
}

}  // namespace internal

/**
   @brief Shared hash table основанная на split ordered lists.
*/
template <class Key, class T, class Hash = std::hash<Key>,
          class Compare = std::less<Key>,
          template<typename> class Allocator = std::allocator>
class hash_table_t {
public:
  static constexpr unsigned kLoadFactor = 2u;
  static constexpr unsigned kLog2MaxBucketCount = 20u; // 1M entries
  static constexpr unsigned kMaxBucketCount = (1u << kLog2MaxBucketCount);

  using key_type = Key;
  using mapped_type = T;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using hasher = Hash;
  using key_compare = Compare;

  struct value_type {
    key_type key{};
    mapped_type payload{};
    uint32_t reversed_hash{0};
  };

  class value_compare_t {
   public:
    value_compare_t() {}
    explicit value_compare_t(const key_compare &compare) : m_compare{compare} {}

    bool operator()(const value_type &lhs, const value_type &rhs) const {
      if (lhs.reversed_hash == rhs.reversed_hash) {
        return m_compare(lhs.key, rhs.key);
      }
      return lhs.reversed_hash < rhs.reversed_hash;
    }

   private:
    key_compare m_compare;
  };

  using reference = value_type &;
  using const_reference = const value_type &;
  using pointer = value_type *;
  using const_pointer = const value_type *;
  using allocator_type = Allocator<node_t<value_type>>;
  using list_type = ordered_list_t<value_type, allocator_type, value_compare_t>;
  using array_type =
      atomic_segmented_array_t<typename list_type::pointer, kMaxBucketCount,
                               allocator_type>;

  explicit hash_table_t(const allocator_type &allocator = allocator_type{})
      : m_compare{}, m_allocator{allocator},
        m_buckets{m_allocator}, m_nodes{m_allocator, m_compare} {
    auto first_dummy_value = value_type{};
    auto is_inserted_dummy_node = m_nodes.insert(first_dummy_value);
    assert(is_inserted_dummy_node.first);
    assert(is_inserted_dummy_node.second != nullptr);
    m_buckets[0] = is_inserted_dummy_node.second;
  }

  uint32_t size() {
    return m_size.load();
  }

  bool insert(const key_type &key, const mapped_type &payload) {
    auto value = value_type{};
    value.key = key;
    value.payload = payload;
    auto hash_value = m_hasher(key);
    value.reversed_hash = (internal::reverse_bits(hash_value) | 1u);
    auto bucket_index = hash_value & (m_reserved.load() - 1);
    if (m_buckets[bucket_index].load() == nullptr) {
      if (!initialize_bucket(bucket_index)) {
        return false;
      }
    }
    auto is_inserted_position =
        m_nodes.insert(m_buckets[bucket_index].load(), value);
    if (!is_inserted_position.first) {
      return false;
    }
    auto reserved = m_reserved.load();
    if (m_size.fetch_add(1) > kLoadFactor * reserved &&
        reserved < kMaxBucketCount) {
      m_reserved.compare_exchange_weak(reserved, 2 * reserved);
    }
    return true;
  }

  std::pair<bool, mapped_type> get(const key_type &key) {
    auto value = value_type{};
    value.key = key;
    auto hash_value = m_hasher(key);
    value.reversed_hash = (internal::reverse_bits(hash_value) | 1u);
    auto bucket_index = hash_value & (m_reserved.load() - 1);
    if (m_buckets[bucket_index].load() == nullptr) {
      if (!initialize_bucket(bucket_index)) {
        return std::make_pair(false, mapped_type{});
      }
    }
    auto is_found_value = m_nodes.find(m_buckets[bucket_index].load(), value);
    return std::make_pair(is_found_value.first, is_found_value.second.payload);
  }

  bool erase(const key_type &key) {
    auto value = value_type{};
    value.key = key;
    auto hash_value = m_hasher(key);
    value.reversed_hash = (internal::reverse_bits(hash_value) | 1u);
    auto bucket_index = hash_value & (m_reserved.load() - 1);
    if (m_buckets[bucket_index].load() == nullptr) {
      if (!initialize_bucket(bucket_index)) {
        return false;
      }
    }
    bool is_erased = m_nodes.erase(m_buckets[bucket_index].load(), value);
    if (is_erased) {
      m_size.fetch_sub(1);
    }
    return is_erased;
  }

private:
  bool initialize_bucket(uint32_t index) {
    uint32_t parent_index = internal::get_parent(index);
    if (!m_buckets.is_reference_valid(m_buckets[parent_index])) {
      return false;
    }
    if (m_buckets[parent_index].load() == nullptr) {
      if (!initialize_bucket(parent_index)) {
        return false;
      }
    }
    auto dummy_value = value_type{};
    dummy_value.reversed_hash = internal::reverse_bits(index);
    auto parent_bucket = m_buckets[parent_index].load();
    auto is_inserted_position = m_nodes.insert(parent_bucket, dummy_value);
    if (is_inserted_position.second == nullptr) {
      return false;
    }
    // need to assign independed of if node was inserted or not
    // because some thread might have inserted but did not update buckets
    if (!m_buckets.is_reference_valid(m_buckets[index])) {
      return false;
    }
    m_buckets[index].store(is_inserted_position.second);
    return true;
  }

  std::atomic<uint32_t> m_size{0}; // element count
  std::atomic<uint32_t> m_reserved{2};
  hasher m_hasher{};
  value_compare_t m_compare;
  allocator_type m_allocator;
  array_type m_buckets;
  list_type m_nodes;
}; // class hash_table_t
```

5 Lock-free object allocator

Простой allocator объектов фиксированной длины на основе lock-free очереди [3]. Используется в lock-free hash table.

```
template <typename T> class object_allocator_t {
public:
  static_assert(sizeof(T) >= sizeof(uint64_t),
                "Object size must be at least 8 byte long");

  using value_type = T;
  using pointer = value_type *;
  using const_pointer = const value_type *;

  object_allocator_t() = default;
  object_allocator_t(object_allocator_t const &) = default;
  object_allocator_t &operator=(object_allocator_t const &) = default;

  object_allocator_t(uint8_t *buffer, std::size_t size) {
    initialize_implementation(buffer, size);
  }

  pointer allocate(std::size_t n) {
    assert(n == 1 && "Array allocation is not supported");
    return reinterpret_cast<pointer>(pull());
  }

  void deallocate(pointer p, std::size_t n) {
    assert(n == 1 && "Array allocation is not supported");
    push(reinterpret_cast<node_t *>(p));
  }

  pointer first() const {
    return m_pimpl->elements;
  }
  pointer last() const {
    return m_pimpl->elements_last;
  }

  // Используется для проверки работы object allocator, не должна
  // вызываться одновременно с allocate/deallocate.
  unsigned object_count() {
    auto index = get_next_index(m_pimpl->head.load());
    auto tail_index = get_next_index(m_pimpl->tail.load());
    auto count = 0u;
    while (index != tail_index) {
      auto node = m_pimpl->node_at(index);
      index = get_next_index(node->load());
      ++count;
    }
    return count;
  }

private:
  static constexpr uint32_t kInvalidIndex = 0xffffffffu;
  using node_t = std::atomic<uint64_t>;

  /**
     @brief Структура для хранения информации об очереди.

     Используется pimpl для возможности копирования и отложенного
     создания allocator. Данные хранятся в начале области памяти,
     используемой в очереди.
  */
  struct implementation_t {
    explicit implementation_t(uint8_t *buffer, unsigned size)
        : elements{reinterpret_cast<pointer>(buffer)},
          elements_last{reinterpret_cast<pointer>(buffer + size)} {}

    // ensure that pointer never changes after creation
    pointer const elements{nullptr};
    pointer const elements_last{nullptr}; // used in debugging
    node_t head;
    node_t tail;

    node_t *node_at(uint32_t index) {
      assert(elements != nullptr);
      assert(index != kInvalidIndex);
      return reinterpret_cast<node_t *>(elements + index);
    }

    uint32_t index_of(node_t *node) {
      auto element_pointer = reinterpret_cast<pointer>(node);
      assert(element_pointer >= elements);
      return static_cast<uint32_t>(element_pointer - elements);
    }
  };

  void initialize_implementation(uint8_t *buffer, std::size_t size) {
    assert(((reinterpret_cast<uintptr_t>(buffer) & 0x7) == 0) &&
           "Storage is not aligned");
    assert(buffer != nullptr);
    assert(size > sizeof(value_type) + sizeof(*m_pimpl));

    auto elements_storage = buffer + sizeof(*m_pimpl);
    m_pimpl = new (buffer)
        implementation_t(elements_storage, size - sizeof(*m_pimpl));
    m_pimpl->node_at(0)->store(compose(kInvalidIndex, 0));
    m_pimpl->head.store(compose(0, 0));
    m_pimpl->tail.store(compose(0, 0));

    auto it = elements_storage + sizeof(value_type); // one dummy node used
    auto next = it + sizeof(value_type);
    auto last = buffer + size;
    while (next < last) {
      push(reinterpret_cast<node_t *>(it));
      it = next;
      next += sizeof(value_type);
    }
  }

  bool try_update_tail(uint64_t old_content, uint64_t update_content) {
    auto new_tail_content = compose(get_next_index(update_content),
                                    get_tag(old_content) + 1);
    return m_pimpl->tail.compare_exchange_strong(old_content, new_tail_content);
  }

  void push(node_t *new_node) {
    new_node->store(compose(kInvalidIndex, 0));
    auto tail_content = uint64_t{0};
    auto last_node_content = uint64_t{0};
    while (true) {
      tail_content = m_pimpl->tail.load();
      last_node_content =
          m_pimpl->node_at(get_next_index(tail_content))->load();
      if (tail_content != m_pimpl->tail.load()) {
        // local tail is not consistent with global tail, that is
        // read last_node_content is not consistent
        continue;
      }
      if (get_next_index(last_node_content) == kInvalidIndex) {
        // tail_node is last, insert new node
        auto new_node_index = m_pimpl->index_of(new_node);
        auto new_content =
            compose(new_node_index, get_tag(last_node_content) + 1);
        if (m_pimpl->node_at(get_next_index(tail_content))
                ->compare_exchange_weak(last_node_content, new_content)) {
          last_node_content = new_content;
          break;
        }
      } else {
        // tail was not pointing to the last node, try to change tail
        try_update_tail(tail_content, last_node_content);
      }
    } // while true
    // node was inserted
    try_update_tail(tail_content, last_node_content);
  }

  node_t *pull() {
    auto head_content = uint64_t{0};
    while (true) {
      head_content = m_pimpl->head.load();
      auto tail_content = m_pimpl->tail.load();
      auto first_node_content =
          m_pimpl->node_at(get_next_index(head_content))->load();
      if (head_content != m_pimpl->head.load()) {
        // the value of first_node_content that was just read is not
        // consistent because head node pointer changed, retry
        continue;
      }
      if (get_next_index(head_content) == get_next_index(tail_content)) {
        // the queue is empty or tail was not updated yet
        if (get_next_index(first_node_content) == kInvalidIndex) {
          return nullptr;
        }
        try_update_tail(tail_content, first_node_content);
      } else {
        // update head
        auto new_content =
            compose(get_next_index(first_node_content), get_tag(head_content));
        if (m_pimpl->head.compare_exchange_weak(head_content, new_content)) {
          break;
        }
      }
    } // while true
    return m_pimpl->node_at(get_next_index(head_content));
  }

  static uint64_t compose(uint32_t index, uint32_t tag) {
    return (static_cast<uint64_t>(index) << 32) + static_cast<uint64_t>(tag);
  }
  static std::pair<uint32_t, uint32_t> decompose(uint64_t value) {
    auto tag = static_cast<uint32_t>(value & 0xffffffffu);
    auto index = static_cast<uint32_t>((value >> 32) & 0xffffffffu);
    return std::make_pair(index, tag);
  }
  static uint32_t get_next_index(uint64_t value) {
    return static_cast<uint32_t>((value >> 32) & 0xffffffffu);
  }
  static uint32_t get_tag(uint64_t value) {
    return static_cast<uint32_t>(value & 0xffffffffu);
  }

  implementation_t *m_pimpl{nullptr};
};
```

6 Список литературы

[1] Maurice Herlihy and Nir Shavit and Moran Tzafrir, *Hopscotch Hashing*, Proceedings of the 22nd international symposium on Distributed Computing, Springer-Verlag, 2008

[2] Ori Shalev and Nir Shavit, *Split-Ordered Lists: Lock-Free Extensible Hash Tables*, Journal of the ACM, Vol. 53, No. 3, 2006.

[3] Maged M. Michael and Michael L. Scott, *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms*, Symposium on Principles of Distributed Computing, 1996.