

**УДК 004.415.53**

**Метод статического поиска гонок в программах на языке Си на основе  
относительного множества блокировок**

*Фроловский А.В., студент*

*Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,  
кафедра «Программное обеспечение ЭВМ и информационные технологии»*

*Научный руководитель: Рудаков И.В., к.т.н, доцент*

*Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана*

*irudakov@bmstu.ru*

Ключевые слова:

Аннотация:

**Введение**

Интенсивное развитие информационных технологий и расширение сферы их применения привело к значительному увеличению сложности используемого программного обеспечения, а также росту количества и критичности выполняемых им функций. С увеличением сложности возрастает количество ошибок. Ущерб от них несет существенные последствия. Одними из наиболее опасных являются ошибки, связанные с гонками при работе с данными. Они носят стохастический характер, что обуславливает сложность их выявления и исправления.

Под состоянием гонки при множественном доступе к разделяемой памяти будем понимать ситуацию, когда два или более потоков одновременно совершают доступ к разделяемой области памяти, и, по крайней мере, хотя бы один из них выполняет операцию записи в неё.

В Листинг 1 показан пример программы, в которой возможно возникновение гонок при доступе к разделяемой переменной. Доступ к разделяемой переменной *count* в функции *foo* является не защищенным ни одним из средств взаимного исключения. Это может привести к возникновению гонок при одновременном доступе к ней из различных потоков.

В Листинг 2 показан пример исправленной программы из Листинг 1. Пример гонки при доступе к разделяемой переменной, в которой проблема возникновения гонок

при доступе к переменной *count* устраняется посредством использования средства синхронизации — мьютекса.

```
#include <stdio.h>
#include <pthread.h>

void *foo(void *arg) {
    int *count = arg;
    unsigned int thread_id = pthread_self();
    while (*count < 10) {
        printf("thread ID = %u ,count = %d\n", thread_id, ++(*count));
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread1, thread2;
    int count = 0;

    pthread_create(&thread1, NULL, &foo, &count);
    pthread_create(&thread2, NULL, &foo, &count);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

Листинг 1. Пример гонки при доступе к разделяемой переменной

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t lock;

void *foo(void *arg) {
    int *count = arg;
    unsigned int thread_id = pthread_self();
    while (*count < 10) {
        pthread_mutex_lock(&lock);
        printf("thread ID = %u ,count = %d\n", thread_id, *count);
        (*count)++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread1, thread2;
    int count = 0;

    pthread_mutex_init(&lock, NULL);
    pthread_create(&thread1, NULL, &foo, &count);
    pthread_create(&thread2, NULL, &foo, &count);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

Листинг 2. Пример безопасного доступа к разделяемой переменной

Статические методы поиска гонок основаны на анализе исходного кода программы без его исполнения. Достоинством данных методов является теоретическая возможность анализа всех возможных путей выполнения программы. Недостатком является получение большого количества ложных предупреждений (обнаружение ситуаций гонок в тех местах программы, где их нет), что усложняет анализ и выявление тех мест программы, которые соответствуют действительным ситуациям гонок.

Нужен переход!

### Метод поиска гонок на основе относительного множества блокировок

Метод основан на методе статического поиска гонок Relay []. В основе метода лежит понятие относительного множества блокировок. Метод состоит из четырех этапов:

1. Нахождение перекрёстных ссылок.
2. Формирование относительных множеств блокировок.
3. Формирование таблиц защищенного доступа.
4. Определение мест возможного возникновения гонок.

IDEF-диаграммы метода представлены на Рисунок 1. Метод поиска гонок и Рисунок 2. Метод поиска гонок.



Рисунок 1. Метод поиска гонок

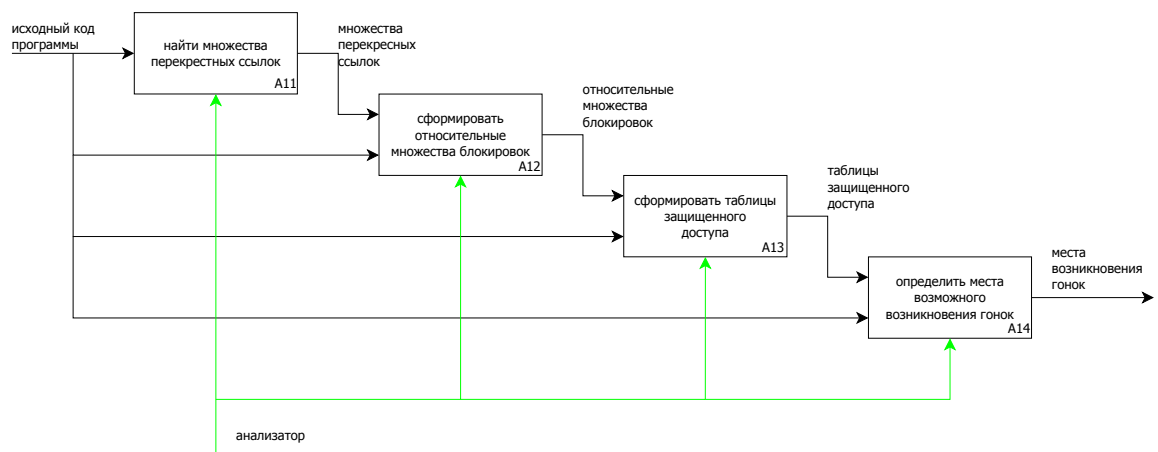


Рисунок 2. Метод поиска гонок

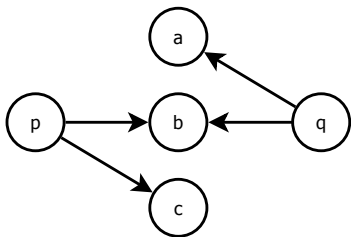
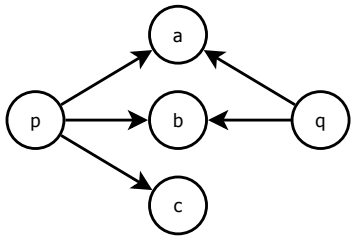
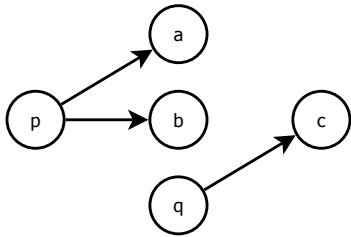
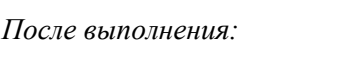
Рассмотрим далее каждый из этапов метода подробнее.

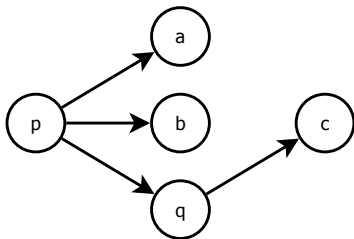
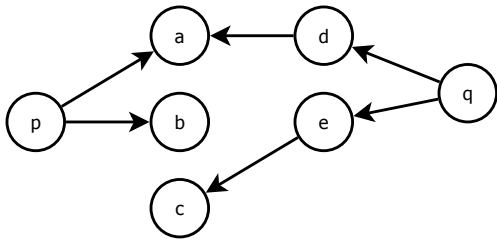
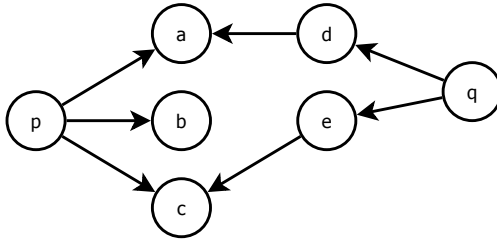
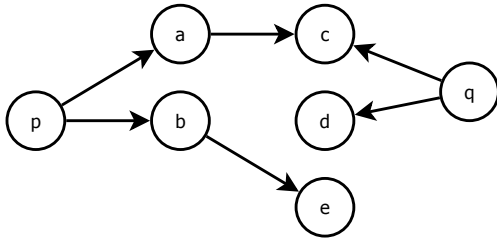
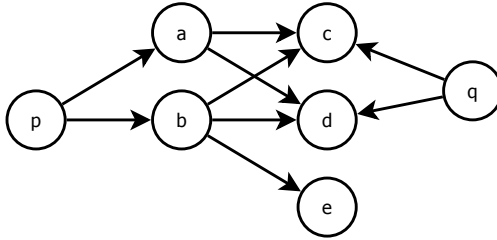
## Нахождение перекрёстных ссылок

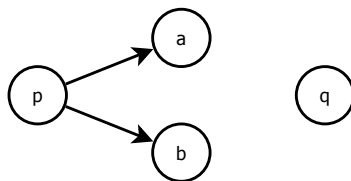
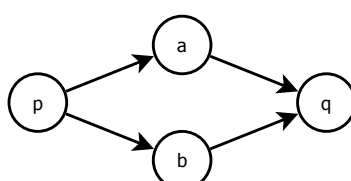
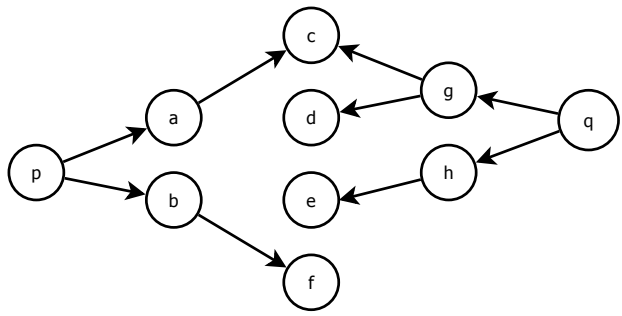
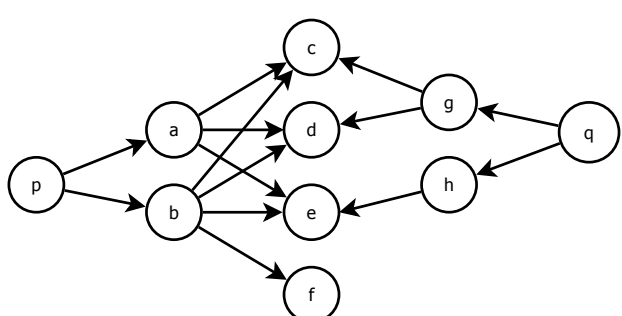
Основной целью данного этапа является определение переменных, которые могут в процессе выполнения программы ссылаться на одни и те же области памяти. В основе используемого алгоритма нахождения перекрёстных ссылок лежит нечувствительный к потоку выполнения алгоритм Андерсена [1].

Процесс нахождения перекрёстных ссылок выполняется для каждой функции независимо. Изначально множества областей, на которые может ссылаться каждая переменная, полагаются пустыми. Затем выполняется итеративный процесс формирования этих множеств областей для каждой переменной. Этот процесс продолжается до тех пор, пока множества не перестанут меняться. Анализируемые инструкции присваивания, действия, выполняемые в процессе анализа, и примеры изменения искомых множеств показаны в Таблица 1. Анализируемые инструкции присваивания.

Таблица 1. Анализируемые инструкции присваивания

Инструкция	Действия	Пример
$p = q$	Множество областей, соответствующее переменной $p$ из левой части оператора присваивания, пополняется элементами из множества, соответствующего переменной $q$ из правой части оператора присваивания.	<p><i>До выполнения:</i>  <math>PT["p"] = \{ "b", "c" \}, PT["q"] = \{ "a", "b" \}</math></p>  <p><i>После выполнения:</i>  <math>PT["p"] = \{ "a", "b", "c" \}, PT["q"] = \{ "a", "b" \}</math></p> 
$p = \&q$	Во множество областей, соответствующее переменной $p$ из левой части оператора присваивания, добавляется область, соответствующая переменной $q$ из правой части оператора	<p><i>До выполнения:</i>  <math>PT["p"] = \{ "a", "b" \}, PT["q"] = \{ "c" \}</math></p>  <p><i>После выполнения:</i></p> 

	присваивания.	$PT["p"] = \{ "a", "b", "q" \}, PT["q"] = \{ "c" \}$ 
$p = *q$	<p>Пусть <math>S</math> – множество областей, соответствующее переменной <math>q</math> из правой части оператора присваивания. Тогда Множество областей, соответствующее переменной <math>p</math> из левой части присваивания, пополняется элементами из множеств областей, соответствующих переменным из <math>S</math>.</p>	<p>До выполнения:</p> $PT["p"] = \{ "a", "b" \}, PT["q"] = \{ "d", "e" \},$ $PT["d"] = \{ "a" \}, PT["e"] = \{ "c" \}$  <p>После выполнения:</p> $PT["p"] = \{ "a", "b", "c" \}, PT["q"] = \{ "d", "e" \},$ $PT["d"] = \{ "a" \}, PT["e"] = \{ "c" \}$ 
$*p = q$	<p>Пусть <math>S</math> – множество областей, соответствующее переменной <math>q</math> из левой части оператора присваивания, <math>T</math> – множество областей, соответствующее переменной <math>q</math> из правой части оператора присваивания. Тогда во все множества, соответствующие элементам из <math>S</math> добавляются элементы из <math>T</math>.</p>	<p>До выполнения:</p> $PT["p"] = \{ "a", "b" \}, PT["q"] = \{ "c", "d" \},$ $PT["a"] = \{ "c" \}, PT["b"] = \{ "e" \}$  <p>После выполнения:</p> $PT["p"] = \{ "a", "b" \}, PT["q"] = \{ "c", "d" \},$ $PT["a"] = \{ "c", "d" \}, PT["b"] = \{ "c", "d", "e" \}$ 

<p><math>*p = \&amp;q</math></p>	<p>Пусть <math>S</math> – множество областей, соответствующее переменной <math>p</math> из правой части оператора присваивания. Тогда во все множества, соответствующие элементам из <math>S</math>, добавляется область, соответствующая переменной <math>q</math> из правой части присваивания.</p>	<p>До выполнения:  <math>PT["p"] = \{“a”, “b”\}</math></p>  <p>После выполнения:  <math>T["p"] = \{“a”, “b”\}, PT["a"] = \{“q”\}, PT["b"] = \{“q”\}</math></p> 
<p><math>*p = *q</math></p>	<p>Пусть <math>S</math> – множество областей, соответствующее переменной <math>p</math> из правой части присваивания, <math>T</math> – множество областей, соответствующее переменной <math>q</math> из правой части присваивания. Тогда все множества, соответствующие элементам из <math>S</math>, пополняются элементами из множеств, соответствующих элементам из <math>T</math>.</p>	<p>До выполнения:  <math>PT["p"] = \{“a”, “b”\}, PT["q"] = \{“g”, “h”\}, PT["a"] = \{“c”\}, PT["b"] = \{“f”\}, PT["g"] = \{“c”, “d”\}, PT["h"] = \{“e”\}</math></p>  <p>После выполнения:  <math>PT["p"] = \{“a”, “b”\}, PT["q"] = \{“g”, “h”\}, PT["a"] = \{“c”, “d”, “e”\}, PT["b"] = \{“c”, “d”, “e”, “f”\}, PT["g"] = \{“c”, “d”\}, PT["h"] = \{“e”\}</math></p> 

На Рисунок 3. Схема алгоритма нахождения перекрёстных ссылок представлена схема алгоритма нахождения перекрёстных ссылок для функции. На Рисунок 4. Схема алгоритма функции eval-lhsи Рисунок 5. Схема алгоритма функции eval-rhsпредставлены

схемы алгоритмов используемых в процессе нахождения перекрестных ссылок функций eval-lhs и eval-rhs соответственно.

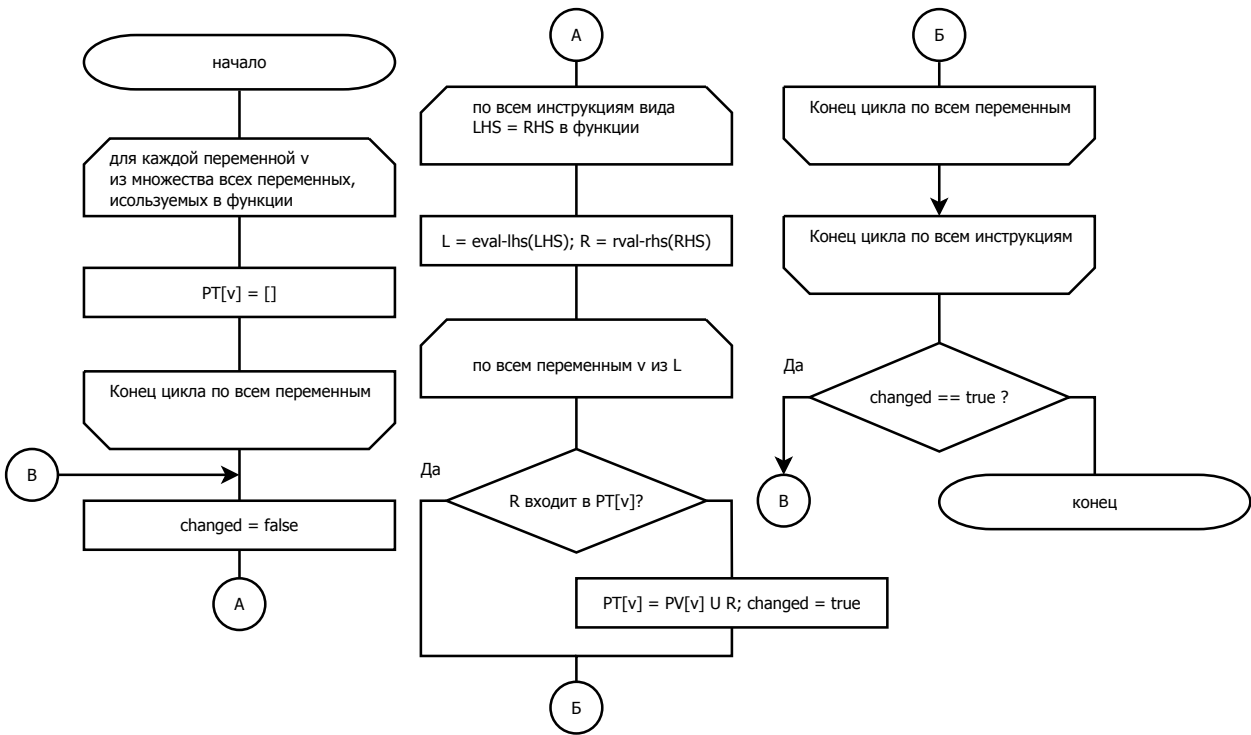


Рисунок 3. Схема алгоритма нахождения перекрёстных ссылок

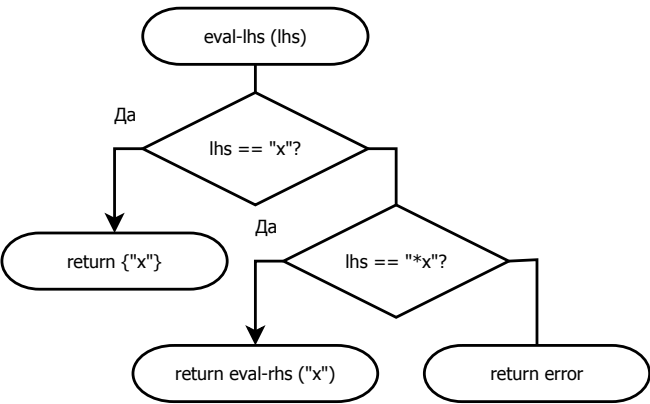


Рисунок 4. Схема алгоритма функции eval-lhs

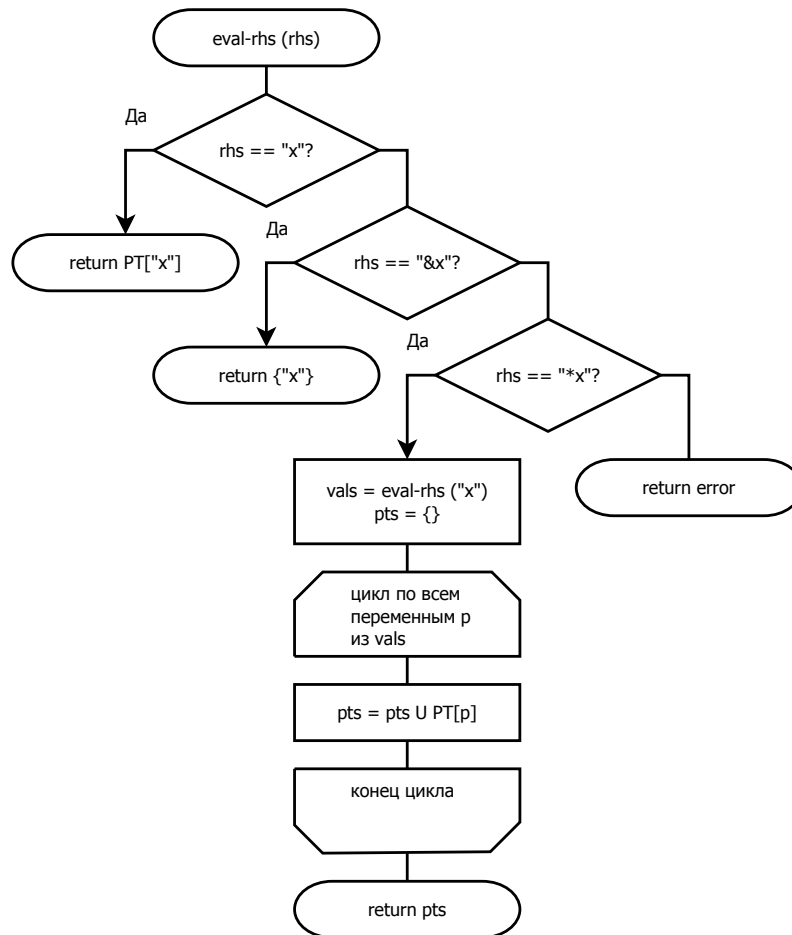


Рисунок 5. Схема алгоритма функции eval-rhs

Пример нахождения перекрёстных ссылок для последовательности инструкций, представленной в Листинг 3. Пример демонстрации алгоритма нахождения перекрёстных ссылок, с использованием описанного алгоритма, представлен в Таблица 2. Пример работы алгоритма.

```

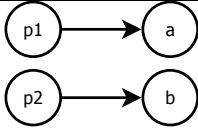
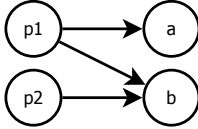
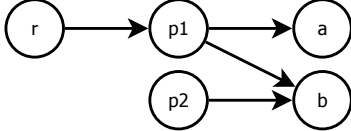
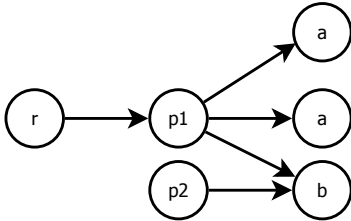
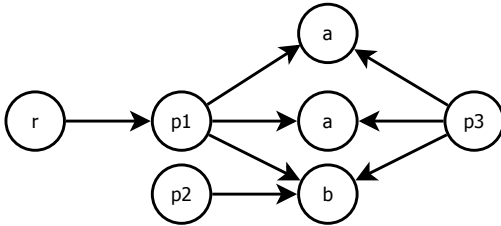
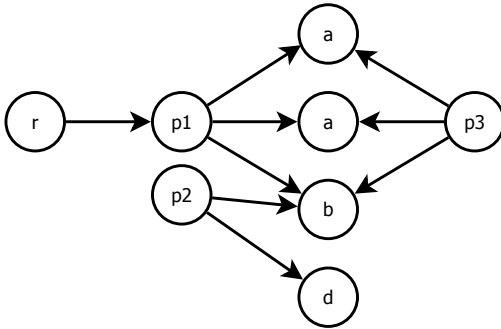
p1 = &a;
p2 = &b;
p1 = p2;
r = &p1;
*r = &c;
p3 = *r;
p2 = &d;
  
```

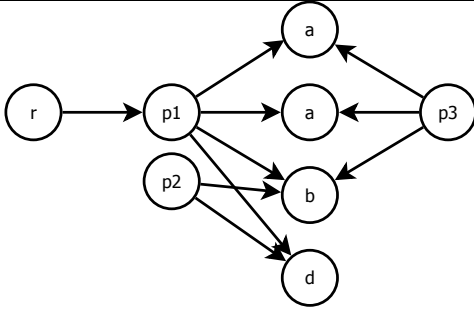
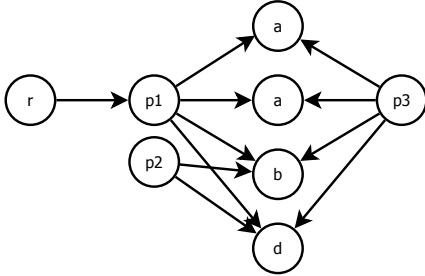
Листинг 3. Пример демонстрации алгоритма нахождения перекрёстных ссылок

Таблица 2. Пример работы алгоритма нахождения перекрёстных ссылок

Номер итерации	Выполняемая инструкция	Состояние множеств областей
1	p1 = &a	PT["p1"] = {"a"} 
1	p2 = &b	PT["p1"] = {"a"}, PT["p2"] = {"b"}



		 <pre> graph LR     p1((p1)) --&gt; a((a))     p2((p2)) --&gt; b((b)) </pre>
1	p1 = p2	<p>PT["p1"] = {"a", "b"}, PT["p2"] = {"b"}</p>  <pre> graph LR     p1((p1)) --&gt; a((a))     p1 --&gt; b((b))     p2((p2)) --&gt; b </pre>
1	r = &p1	<p>PT["r"] = {"p1"}, PT["p1"] = {"a", "b"}, PT["p2"] = {"b"}</p>  <pre> graph LR     r((r)) --&gt; p1((p1))     p1 --&gt; a((a))     p1 --&gt; b((b))     p2((p2)) --&gt; b </pre>
1	*r = &c	<p>PT["r"] = {"p1"}, PT["p1"] = {"a", "b", "c"}, PT["p2"] = {"b"}</p>  <pre> graph LR     r((r)) --&gt; p1((p1))     p1 --&gt; a1((a))     p1 --&gt; a2((a))     p1 --&gt; a3((a))     p2((p2)) --&gt; b((b)) </pre>
1	p3 = *r	<p>PT["r"] = {"p1"}, PT["p1"] = {"a", "b", "c"}, PT["p2"] = {"b"}, PT["p3"] = {"a", "b", "c"}</p>  <pre> graph LR     r((r)) --&gt; p1((p1))     p1 --&gt; a1((a))     p1 --&gt; a2((a))     p1 --&gt; a3((a))     p1 --&gt; b((b))     p2((p2)) --&gt; b     p3((p3)) --&gt; a1     p3 --&gt; a2     p3 --&gt; a3     p3 --&gt; b </pre>
1	p2 = &d	<p>PT["r"] = {"p1"}, PT["p1"] = {"a", "b", "c"}, PT["p2"] = {"b", "d"}, PT["p3"] = {"a", "b", "c"}</p>  <pre> graph LR     r((r)) --&gt; p1((p1))     p1 --&gt; a1((a))     p1 --&gt; a2((a))     p1 --&gt; a3((a))     p1 --&gt; b((b))     p2((p2)) --&gt; b     p2 --&gt; d((d))     p3((p3)) --&gt; a1     p3 --&gt; a2     p3 --&gt; a3 </pre>
2	p1 = p2	<p>PT["r"] = {"p1"}, PT["p1"] = {"a", "b", "c", "d"}, PT["p2"] = {"b", "d"}, PT["p3"] = {"a", "b", "c"}</p>

		
2	$p3 = *r$	<p> <math>PT["r"] = \{ "p1" \}</math>, <math>PT["p1"] = \{ "a", "b", "c", "d" \}</math>, <math>PT["p2"] = \{ "b", "d" \}</math>, <math>PT["p3"] = \{ "a", "b", "c", "d" \}</math> </p> 

### Формирование относительных множеств блокировок

На данном этапе выполняется формирование и сопоставление каждой инструкции относительного множества блокировок, которое уже сформировалось к началу её выполнения. Под относительным множеством блокировок  $L$  понимается пара  $(L_+, L_-)$ , где  $L_+$  - множество обязательно захваченных блокировок,  $L_-$  - множество освобождённых блокировок. Оно отражает изменение множества блокировок, производимое во время выполнения.

Для формирования множеств обязательно захваченных блокировок необходимо знать какие базовые блоки графа потока управления функции входят в ядро функции, т.е. встречаются на всех путях выполнения функции из графа потока управления. Введём ограничение на количество раз, которое базовый блок может встретиться в пути, равное  $K$ . Тогда алгоритм определения ядра функции, основанный на поиске в глубину может быть описан следующим псевдокодом, показанным на Листинг 4. Алгоритм определения ядра функции4.

```

def walk(v, p):
    # v - текущий базовый блок
    # p - базовый блок

    p = p + v # добавляем блок v в путь p

    # v.next() - получение списка базовых блоков, на которые
    # может быть совершён переход из блока v
    if v.next() is empty:
        return set(p)

    # получаем множества блоков,
    # встречающихся в путях, проходящих через v
    C = []
    N = 0
    for w in v.next():
        # count(w in p) - получение количества раз,
        # которое блок w встречается в пути p
        if count(w in p) <= K:
            C[N] = walk(w, p)
            N = N + 1

    # строим пересечение множеств блоков, которые
    # встречаются в путях, проходящих через v
    core = C[0]
    for I = 0 to N:
        if core is empty:
            break
        core = intersect(core, S[I])

    return core

entry = <начальный базовый блок функции>
core = walk(entry, [])

```

#### Листинг 4. Алгоритм определения ядра функции

Перед началом анализа функции текущее относительное множество блокировок полагается пустым. Далее производится последовательный анализ инструкций из каждого базового блока. Относительное множество блокировок может измениться в процессе анализа только после вызова какой-либо функции. Относительное множество блокировок, отражающее изменение состояния множества блокировок в процессе выполнения некоторой функции  $f$ , называется обобщением относительного множества блокировок функции  $lockSummary(f)$ . Для функции захвата блокировки  $lock(l_a)$  обобщение равно  $(\{l_a\}, \{\})$ , где  $l_a$  – захватываемая блокировка, а для функции освобождения –  $(\{\}, \{l_r\})$ , где  $l_r$  – освобождаемая блокировка. Таким образом, относительное множество блокировок во время анализа после вызова функции может быть получено с использованием функции  $lockUpdate((L_+, L_-), (L'_+, L'_-)) = ((L_+ \cup L'_+) - L'_-, (L_- \cup L'_-) - L'_+)$ , где  $(L_+, L_-)$  – текущее множество блокировок,  $(L'_+, L'_-)$  – обобщение относительно множества блокировок вызываемой функции, в котором формальные параметры функции заменяются передаваемыми в функцию аргументами. Схема алгоритма формирования относительных

множеств блокировок для функции представлена на Рисунок 6. Алгоритм формирования относительных множеств блокировок, Рисунок 7. Алгоритм формирования относительных множеств блокировок и Рисунок 8. Алгоритм формирования относительных множеств блокировок.

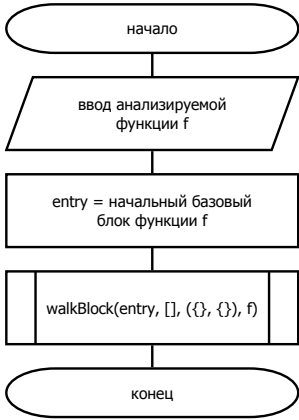


Рисунок 6. Алгоритм формирования относительных множеств блокировок

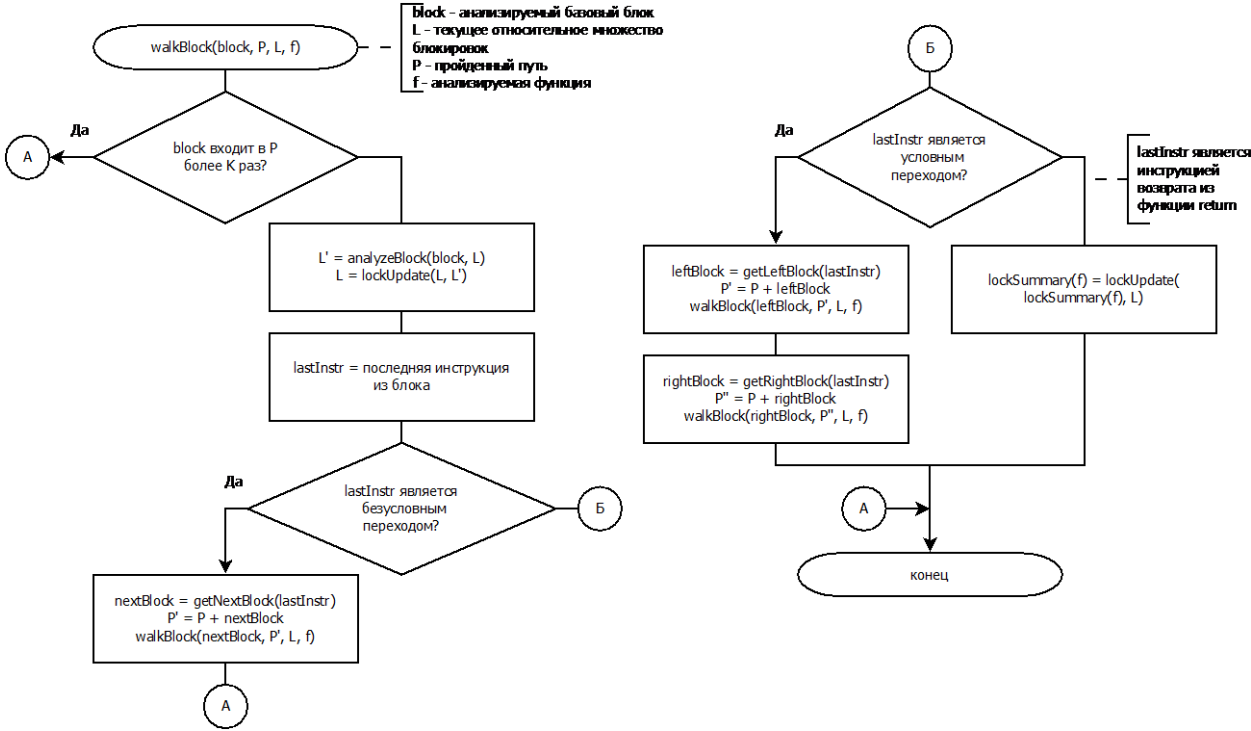


Рисунок 7. Алгоритм формирования относительных множеств блокировок

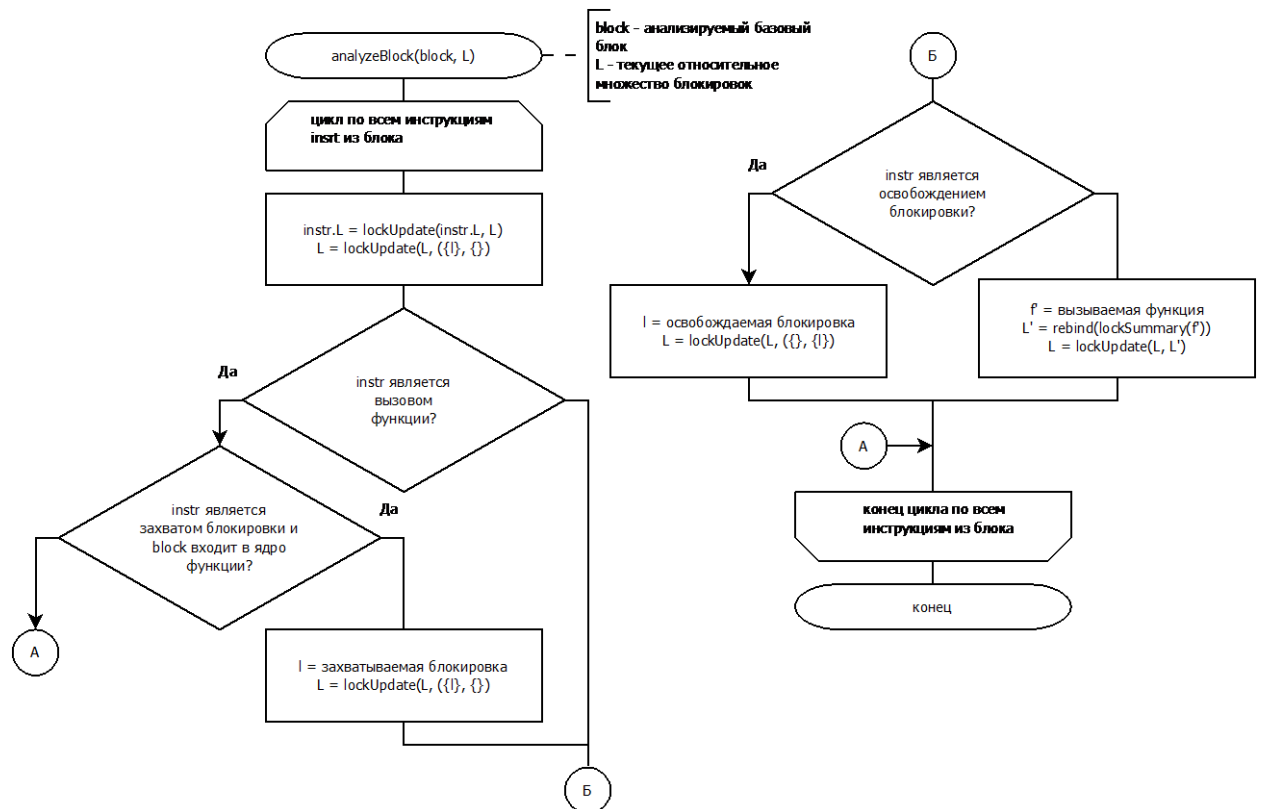


Рисунок 8. Алгоритм формирования относительных множеств блокировок

Рассмотрим пример из Листинг 5. Пример программы. Функции *thread1* и *thread2* являются точками входа в потоки. В строке 4 производится операция захвата блокировки *mutex*, поэтому состояние относительного множества блокировок после её исполнения становится равным  $(\{mutex\}, \{\}) = lockUpdate(\{\}, \{\}, (\{mutex\}, \{\}))$ . В строке 6 производится её освобождение, поэтому состояние относительного множества блокировок становится равным  $(\{\}, \{\}) = lockUpdate((\{mutex\}, \{\}), (\{\}, \{mutex\}))$  и, следовательно, обобщение относительного множества блокировок для функции *incr* становится равным  $(\{\}, \{\})$ . В силу этого вызов функции *incr* не изменит состояний относительных множеств блокировок ни в функции *thread1*, ни в функции *thread2*.

```

1. static int x = 0, y = 0;
2. pthread_mutex_t m1, m2;
3. void incr(int * value, pthread_mutex_t* mutex) {
4.     pthread_mutex_lock(mutex);
5.     (*value)++;
6.     pthread_mutex_unlock(mutex);
7. }
8. void* thread1(int *arg) {
9.     int* z = &x;
10.    incr(z, &m1);
11.    incr(&y, &m2);
12. }
13. void* thread2(int *arg) {
14.    incr(&x, &m1);
15.    int* z = &y;
16.    incr(z, &m1);
17. }
  
```

## Листинг 5. Пример программы

### Формирование таблиц защищенного доступа

Целью данного этапа является формирование таблиц защищенного доступа для каждой функции. В каждой строке таблицы содержится структура, называемая защищенным доступом. Под защищенным доступом понимается тройка  $(o, L, k)$ , где  $o$  – lvalue [], к которому производится доступ,  $L$  – относительное множество блокировок в момент доступа к  $o$ ,  $k$  - тип доступа (“чтение” или “запись”).

Вначале анализа таблица защищенного доступа для функции полагается пустой. Затем анализируется каждая инструкция на предмет доступа к разделяемой переменной. В контексте анализа функции под разделяемой переменной понимаются глобальные переменные и формальные параметры функции. Если текущая анализируемая инструкция содержит доступ к переменной, то в таблицу добавляется соответствующая ему запись защищенного доступа. В случае, когда анализируемая инструкция является вызовом функции, нужно все записи из её таблицы защищенного доступа изменить в соответствии с текущим состоянием множеств блокировок и добавить в таблицу анализируемой функции. Схема алгоритма формирования таблицы защищенного доступа для функции показана на Рисунок 9. Схема алгоритма формирования таблиц защищенного доступа и на Рисунок 10. Схема алгоритма формирования таблицы защищенного доступа.

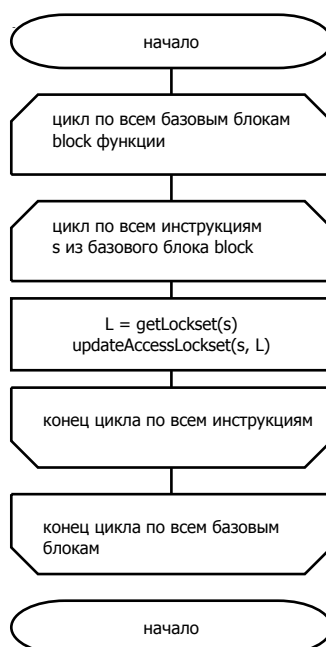


Рисунок 9. Схема алгоритма формирования таблиц защищенного доступа

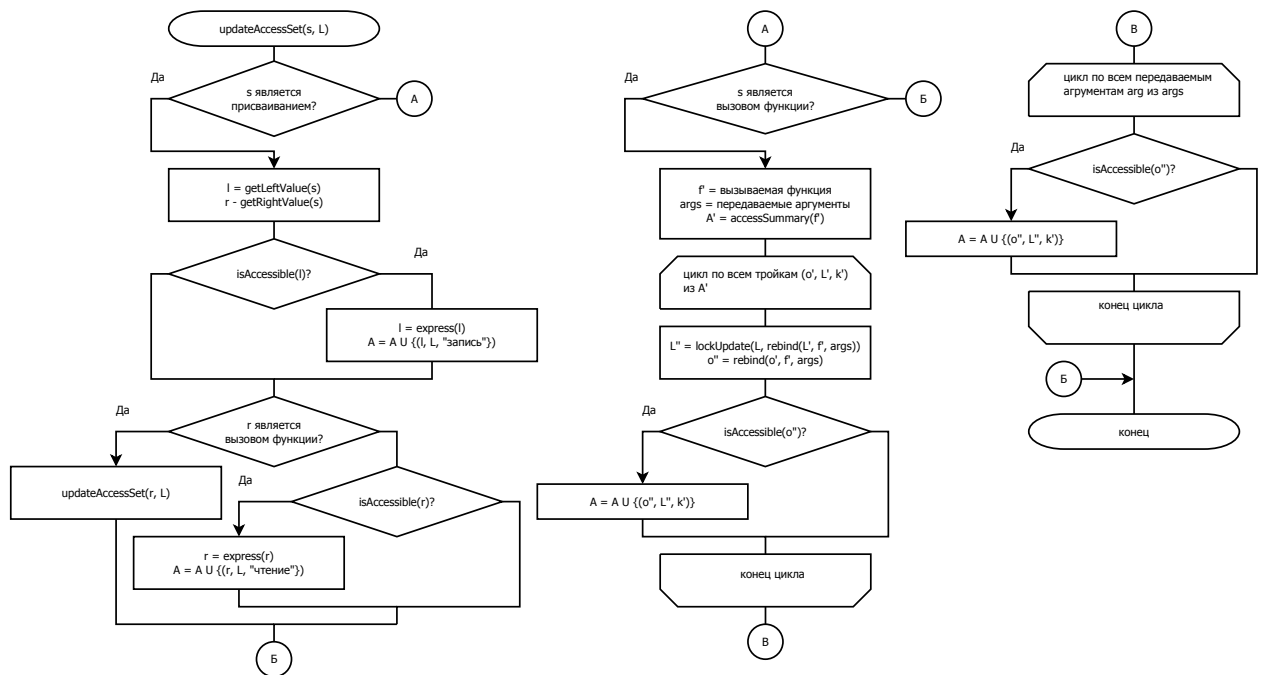


Рисунок 10. Схема алгоритма формирования таблицы защищенного доступа

Рассмотрим пример из листинга 5. В строке 5 производится доступ и на чтение, и на запись к разделяемой области, адрес которой передается в функцию через формальный параметр *value*. Относительное множество на момент доступа равно  $(\{mutex\}, \{\})$ , поэтому добавляем в таблицу доступов для функции *incr* 2 строки:  $(*value, (\{mutex\}, \{\}), \text{чтение})$  и  $(*value, (\{mutex\}, \{\}), \text{запись})$ . Защищенный доступ для функции *incr* представлен в Таблица 3. Таблица защищенного доступа функции *incr*. Поскольку в строке 10 в функции *thread1* осуществляется вызов функции *incr*, необходимо выполнить конкретизацию Таблица 3. Таблица защищенного доступа функции *incr* и добавить её строки в таблицу для функции *thread1*. Аналогичные действия выполняются в строках 11, 14 и 16. Результирующие защищенные доступы для функций *thread1* и *thread2* показаны в Таблица 4 и Таблица 5 соответственно.

Таблица 3. Таблица защищенного доступа функции *incr*

Доступ	Относительное множество блокировок	Тип доступа
$*value$	$(\{mutex\}, \{\})$	чтение
$*value$	$(\{mutex\}, \{\})$	запись

Таблица 4. Таблица защищенного доступа функции *thread1*

Доступ	Относительное множество блокировок	Тип доступа
$x$	$(\{m1\}, \{\})$	чтение
$x$	$(\{m1\}, \{\})$	запись
$y$	$(\{m2\}, \{\})$	чтение

у	{m2}, { }	запись
---	-----------	--------

Таблица 5. Таблица защищённого доступа функции thread2

Доступ	Относительное множество блокировок	Тип доступа
х	{m1}, { }	чтение
х	{m1}, { }	запись
у	{m1}, { }	чтение
у	{m1}, { }	запись

### Определение мест возможного возникновения гонок

На данном этапе на основе полученных на предыдущем этапе таблиц защищённого доступа производится определение мет в программе, в которых возможно возникновение гонок при доступе к разделяемым между несколькими потоками областям. Вначале производится определение точек входа в потоки. Затем осуществляется конкретизация таблиц защищённого доступа для каждой точки входа: замена формальных параметров, присутствующих в таблице, на передаваемые в поток аргументы. После этого производится перебор всех пар точек входа в потоки и сравнение соответствующих им таблиц защищённого доступа. В случае, когда в таблицах, соответствующих разным точкам входа, присутствуют доступы к одной и той же области, и при этом хотя бы один из них является доступом на запись, и пересечение множеств захваченных блокировок пусто, то данная область помечается как потенциально опасное место возникновения гонок. Схема описанного алгоритма показана на Рисунок 11. Схема алгоритма поиска мест возможного возникновения гонок.

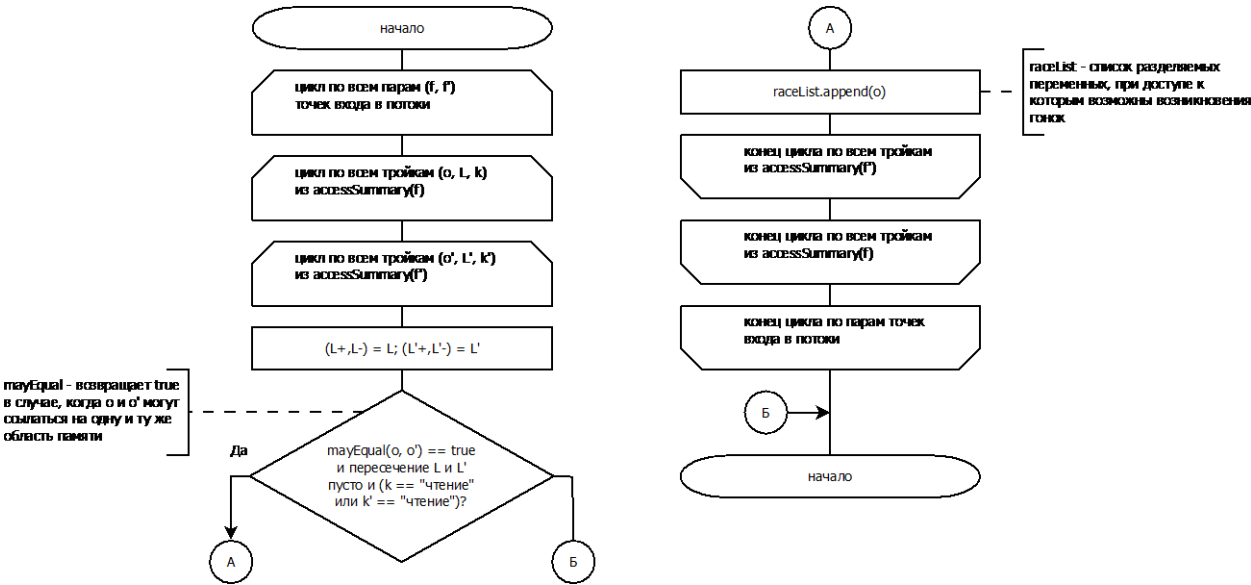


Рисунок 11. Схема алгоритма поиска мест возможного возникновения гонок



Рассмотрим пример из листинга 5. Видно, что в защищенных доступах функций *thread1* и *thread2*, представленных в Таблица 4 и Таблица 5, при доступе к глобальной переменной *y* захватываемые множества блокировок не пересекаются, и производится доступ на запись, следовательно, возможно возникновение гонки.

### **Заключение**

В данной работе представлен метод статического поиска гонок в программах на языке Си, основанный на концепции использования относительного множество блокировок. Рассмотрены детали каждого из этапов и приведены примеры, поясняющие выполняемые при анализе действия.

Основным достоинством метода является частичная независимость по данным анализа одной функции от анализа другой. Это даёт возможность эффективно выполнить распараллеливание анализа программы, что может существенно снизить время его выполнения.

Одним из недостатков метода является предположение о параллельном выполнении всех потоков, хотя некоторые исполняются заведомо последовательно. Также в представленном методе поиск перекрёстных ссылок внутри функции выполняется вне зависимости от пути исполнения программы, в результате чего могут появиться дополнительные ошибки 2 рода (ложные предупреждения, англ. false alarms).