Статический поиск ошибок повторной блокировки семафора

А. Е. Бородин
alexey.borodin@ispras.ru
ИСП РАН, 109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. В статье описывается алгоритм статического поиска ошибки повторной блокировки семафоров, позволяющий выдавать предупреждения с низким уровнем ложных срабатываний. Поиск ошибок рассмотрен для абстрактной библиотеки, включающей функции блокировки семафора, разблокировки семафора и условной блокировки семафора. Определено множество регулярных языков, моделирующих блокировки и разблокировки при конкретном исполнении программы. Определён домен, аппроксимирующий множество регулярных языков. Алгоритм реализован в терминах анализа потока данных. При анализе элементы домена используются в качестве свойств потока данных. Алгоритм описан для программы с одним семафором и без алиасов. В этом случае каждое выданное предупреждение должно соответствовать ошибке при конкретном выполнении. Алгоритм реализован в системе статического анализа Svace, разрабатываемой в Институте системного программирования Российской академии наук. Svace осуществляет анализ алиасов и сопоставление формальных и фактических параметров при вызове функции. Благодаря этому можно применять алгоритм поиска повторных блокировок для программы, содержащей только один семафор, а вся остальная работа будет выполнена системой Svace. Алгоритм поиска повтороных блокировок реализованный в Svace может выдавать некоторое количество ложных срабатываний, т. к. Svace выполняет неконсервативный анализ.

Ключевые слова: статический анализ; семафор; ошибка повтороной блокировки; анализ потока данных; регулярные языки.

1. Введение

Одной из ошибок использования примитивов синхронизации семафоров является повторная блокировка семафора, которая может привести к тупиковой ситуации при выполнении приложения. Конкретное поведение зависит от операционной системы и используемых библиотек.

Семафор - это примитив синхронизации, который может находиться в двух состояниях: открытом и закрытом. С помощью функции блокировки семафор из открытого состояния переводится в закрытое состояние. Блокировка семафора, уже находящегося в закрытом состоянии, может вызвать проблемы

в зависимости от операционной системы, типа семафора и настроек. Например, в операционных системах, поддерживающих стандарт POSIX, семафоры могут иметь следующие типы [1]:

PTHREAD_MUTEX_NORMAL — нет контроля повторного захвата тем же потоком;
PTHREAD_MUTEX_RECURSIVE — повторные захваты тем же потоком допустимы, ведётся счётчик таких захватов;
PTHREAD_MUTEX_ERRORCHECK — повторные захваты тем же потоком вызывают немедленную ошибку.

Таким образом повторная блокировка семафора может привести к тупиковой ситуации для PTHREAD_MUTEX_NORMAL или к ошибке времени выполнения для PTHREAD_MUTEX_ERRORCHEC.

В данной работе рассматривается алгоритм поиска ошибок повторной блокировки с помощью статического анализа исходного текста программы.

Поиск ошибок будет рассмотрен для абстрактной библиотеки, функции которой показаны на рис.1.

```
struct MUTEX;
//переводит семафор в закрытое состояние.
void lock(MUTEX*);
//переводит семафор в открытое состояние.
void unlock(MUTEX*);
//если семафор находится в открытом состоянии,
//переводит семафор в закрытое состояние и в случае успеха возвращает 0.
int trylock(MUTEX*);
```

Рис. 1. Абстрактная библиотека работы с семафорами.

При использовании на конкретной платформе эти функции могут быть легко заменены на соответствующие аналоги.

2. Модель повторной блокировки

Рассмотрим процедуры [1] lock и unlock абстрактной библиотеки работы с семафорами. Процедура trylock будет описана ниже.

1 Функции языка Си здесь будут называться процедурами, чтобы избежать неоднозначностей с математическими функциями.

Процедура lock закрывает семафор, если семафор уже был закрыт, то произойдёт ошибка повторной блокировки. Процедура unlock переводит семафор в открытое состояние. Таким образом семафор может находиться в трёх состояниях: открытом, закрытом и ошибочном. Переходы между состояниями происходят только при вызове процедур lock и unlock. Вызов любой другой библиотечной процедуры не меняет состояния семафора. Начальным состоянием является открытое состояние.

Для программы, содержащей один семафор, рассмотрим конкретное выполнение (трассу) некоторой процедуры.

Обозначим I - множество инструкций процедуры. Обозначим L = {u, l}. Определим функцию IL: I → L*, сопоставляющую каждой инструкции последовательность меток из L следующим образом: IL возвращает [l] для инструкции вызова процедуры lock; [u] для инструкции вызова unlock и [] для всех остальных инструкций.

Пусть T множество конечных и бесконечных трасс (последовательностей элементов I). Для t ∈ T, |t| - обозначает длину трассы, для бесконечных трасс |t| = ω. t_k∈I (для k-1 < |t|) - k-й элемент трассы t. $t_{1..k}$ – префикс трассы, содержащий k элементов.

Введём функцию S: T → L*, отображающую трассы в последовательности меток, такую что

$S(t_{1..1}) = IL(t_1)$

$S(t_{1..k}) = S(t_{1..k-1}) \cdot IL(t_k)$, где k-1 < |t|.

Состоянием шага k трассы t будем называть $S(t_{1..k})$.

Будем говорить, что трасса содержит нелокальную повторную блокировку, если для некоторой инструкции вызова процедуры входящей в трассу, сегмент трассы, соответствующий исполнению этой инструкции, отображается функцией S на последовательность меток, содержащую подстроку ll.

Пусть T* - множество трасс, не содержащих нелокальные повторные блокировки. Рассмотрим множество префиксов трасс P ⊆ T*, заканчивающихся в некоторой точке программы. Накапливающей семантикой [2] данной точки будем называть множество {S(t) | t ∈ P}.

Для выполнения анализа будем приближать накапливающую семантику ребёр графа потока управления регулярными языками. Определим множество регулярных языков B ={ε, σ, π, e}, где e = L*llL*, ε = {ε}, σ = L*l, π = lL*.

Пусть C получено замыканием B относительно операции пересечения множеств, дополненное элементом ⊤ = L* (⊤ можно рассматривать как пересечение пустого набора множеств). Множество D определяется как замыкание C относительно операции объединения. Тогда (D,⊆) составляет полную решетку.

При анализе элементы D будут использоваться в качестве свойств потока данных. Будем говорить, что элемент d∈D корректен для точки программы, если накапливающая семантика данной точки является подмножеством d.

Интуитивно элемент e обозначает множество трасс выполнения программ, содержащих повторную блокировку; элемент σ – множество трасс, заканчивающихся блокировкой; элемент π – множество трасс, начинающихся с блокировки; ε – трассы без блокировок и разблокировок, ⊤ - все возможные трассы.

Примеры описываемых множеств последовательностей:

$ulul \subseteq \sigma$; $lul \subseteq (\pi \cap \sigma)$; $\{ulul, \varepsilon\} \subseteq (\sigma \cup \varepsilon)$.

Определение. Бинарный оператор x: DxD->D будем называть приближенной конкатенацией, если конкатенация любых элементов d1, d2 из D является подмножеством d1xd2.

Определим бинарный оператор ∘: DxD->D следующими правилами:

- Пусть c, c' ∈ C.
  - $c \circ \emptyset = \emptyset \circ c = \emptyset$.
  - $\varepsilon \circ c = c \circ \varepsilon = c$.
  - Если $c \subseteq \pi$, $c' \neq \varepsilon$, то $c \circ c' = \pi \cap c'$.
  - Если $c' \subseteq \sigma$, то $c \circ c' = \sigma \cap c$.
  - Если $c \subseteq \sigma$ и $c' \subseteq \pi$, то $c \circ c' = e$.
  - Иначе, если $c \subseteq e$ то $c \circ c' = c' \circ c = e$.
  - Иначе, $c \circ c' = \top$.
- Если d, d' ∈ D, где $d = c_1 \cup ... \cup c_m$, $d' = c_1' \cup ... \cup c_n'$, то $d \circ d' = \bigcup (c_i \cdot c'_j)$, где $0 < i \leq m$, $0 < j \leq n$.

Лемма 1. Оператор ∘ является приближённой конкатенацией.

Доказательство. 1) $c \cdot \emptyset = \emptyset \subseteq c \circ \emptyset$.

$\emptyset \cdot c = \emptyset \subseteq \emptyset \circ c$

2) $c \cdot \varepsilon = c \subseteq c \circ \varepsilon$.

$\varepsilon \cdot c = c \subseteq \varepsilon \circ c$.

3) Пусть $c \subseteq \pi$, тогда $c \cdot c' \subseteq \pi \cdot c'$ (т.к. $x \subseteq y \Rightarrow x \cdot z \subseteq y \cdot z$)

$\pi \cdot c' = lL^* \cdot c' \subseteq \pi$. Если также $\pi \cdot c' \subseteq c'$, то $\pi \cdot c' \subseteq \pi \cap c'$.

Докажем, что $\pi \cdot c' \subseteq c'$.

Если $c' = \top$, то $\pi \cdot c' = \pi \cdot \top \subseteq L^* \subseteq \top$.

Иначе $c' = b_1 \cap ... \cap b_n$, $b_i \neq \varepsilon$.

Если для любого i верно $\pi \cdot b_i \subseteq b_i$, то $\pi \cdot c' \subseteq \pi \cdot b_i \subseteq b_i$,

следовательно $\pi \cdot c' \subseteq b_1 \cap ... \cap b_n = c'$.

Осталось доказать, что $\pi \cdot b_i \subseteq b_i$, где $b_i \in \{\sigma, \pi, e\}$:

$\pi \cdot \pi = lL^* \cdot lL^* \subseteq lL^* = \pi$;

$\pi \cdot \sigma = lL^* \cdot L^*l \subseteq L^*l = \sigma$;

$\pi \cdot e = lL^* \cdot L^*llL^* \subseteq lL^*llL^* \subseteq L^*llL^* = e$.

4) Если $c' \subseteq \sigma$, то $c \circ c' = \sigma \cap c$ доказывается аналогично 3.

5) Если $c \subseteq \sigma$ и $c' \subseteq \pi$, тогда $c \cdot c' \subseteq \sigma \cdot \pi = L^*llL^* = e$.

6) Иначе, если $c \subseteq e$, то $c \cdot c' \subseteq c' \cdot e \subseteq T \cdot e = L^*L^*llL^* \subseteq L^*llL^* = e$

7) Иначе, $c \circ c' = \top$. По определению $\top$ верно, что для любого $d \in D$ верно $d \subseteq \top$, следовательно $c \cdot c' \subseteq T$.□

Результат применения оператора $\circ$ для элементов множества B приведён в табл. 1, элементы C могут быть представлены как пересечение элементов B, а элементы D как объединение элементов C.

Табл.1. Операция конкатенации для элементов B.

	π	**σ**	**e**	**ε**
π	**π**	**π⋂σ**	**π⋂e**	**σ**
σ	**e**	**σ**	**e**	**π**
e	**e**	**e⋂σ**	**e**	**e**
ε	**π**	**σ**	**e**	**ε**

3. Выполнение анализа

Выполним анализ потока данных, сопоставив рёбрам графа потока управления элементы домена D. Элемент, сопоставленный ребру выхода из процедуры назовём аннотацией. Введём функцию A, возвращающую аннотацию для каждой процедуры. Будем считать, что $A(lock) = \pi \cap \sigma$, $A(unlock) = T$.

Определим передаточные функции P_i: $D \rightarrow D$ для инструкции i программы:

1. Для инструкции i вызова процедуры p, где $A(p) = d_2$, $P_i(d_1) = d_1 \circ d_2$.
2. Для остальных инструкций $P_i(d) = d$.

Оператор сбора $\sqcup$: D x D → D будет объединением множеств .

Для шага k трассы t находим инструкцию $i = t_k$, конкретная семантика $c = S(t_{1..k})$, абстрактная семантика $d = OUT_i \in D$. В соответствии с определением оператора сбора и передаточной функцией, а также леммой 1, $c \in d$.

Если анализ потока данных сопоставляет некоторому ребру состояние e, то при конкретном выполнении программы в данном месте произойдёт повторная блокировка.

Доказательство: Достаточно доказать, что $S(t_{1..k}) \in d_i \Rightarrow S(t_{1..k+1}) \in d_{i+1}$, где k – шаг трассы для инструкции I, $d_i = P(d_{i-1})$. Если i – инструкция вызова процедуры, то передаточная функция определена через оператор $\circ$, для которого утверждение верно согласно лемме 1.

Для остальных инструкций передаточная функция является тождественной функцией, поэтому $d_{i+1} = d_i$ и $S(t_{1..k}) = S(t_{1..k+1})$.

Т.е. если на некотором шаге k состояние равно e, то $S(t_{1..k}) \in e = L^*llL^*$, что означает ситуацию с повторной блокировкой. □

Как видно из определения передаточной функции P и оператора сбора ⊔, если в программе возможна повторная блокировка, то алгоритм не обязательно найдёт её. Контрпример приведён на рис.2. Аннотация для процедуры foo A(foo) = $\pi \cap \sigma \cup \varepsilon$. Аннотация для lock – $\pi \cap \sigma$. Поэтому после повторного вызова lock состояние $P = \pi \cap \sigma \cup \varepsilon \cdot \pi \cap \sigma = \pi \cap \sigma \cap e \cup \pi \cap \sigma = \pi \cap \sigma$. Таким образом ошибка повторной блокировки не будет найдена. Причина подобной реализации в том, что неизвестно точно о взаимосвязи глобальной переменной glob и параметра flag. Если флаги взаимно исключают друг друга, то мы получим ложное срабатывание. Так как целью работы было создание алгоритма, выдающего предупреждения с низким уровнем ошибок, то мы предпочитаем не выдавать предупреждения для подобных ситуаций. Альтернативный подход может не использовать приближение $e \subseteq T$ и выдавать предупреждения о повторной блокировке в случае условных блокировок. Но так как в существующих библиотеках часто используются функции, которые в зависимости от параметра вызывают блокировку, либо разблокировку семафора, то такой подход приведёт к ложным срабатываниям для всех мест использования подобных функций. Также можно использовать разрешитель булевых формул для анализа взаимосвязей между переменными программы, но в этом случае возрастает сложность и увеличивается время анализа.

```
void foo(MUTEX* mutex) {
        if(glob) { lock(mutex); }
}
void func(int flag, MUTEX* mutex) {
        if(flag) { foo(mutex); }
        lock(mutex);
}
```

Рис. 2. Потенциальная ошибка.

Если программа содержит несколько семафоров, то алгоритм работает с каждым семафором отдельно.

4. Реализация для trylock

Функция trylock тестирует состояние семафора и если он находится в закрытом состоянии, производит блокировку. Таким образом двойной вызов trylock не приводит к ошибке повторной блокировки.

Если семафор находится в открытом состоянии, то после выполнения trylock, он будет в закрытом состоянии. Если же семафор находится в закрытом

состоянии, то функция trylock ничего не будет делать. Т.е. после вызова trylock семафор будет всегда находится в закрытом состоянии.

Формально, если d∈D – состояние перед выполнением trylock, а a – аннотация trylock, то если d = L*l, то d · a = L*l · a = L*l, т.е. a эквивалентно ε;

если d = L*, то d · a = L* · a = L*l, т.е. a эквивалентно σ.

Таким образом для эмуляции поведения trylock, можно принять что A(trylock) = σ. Мы говорим об эмуляции, т.к. A(trylock) = σ означает, что все трассы проходящие через функцию trylock заканчиваются блокировкой строго внутри этой функции, что некорректно.

5. Реализация

Описанный выше алгоритм был реализован в системе статического анализа Svace, разрабатываемой в ИСП РАН.

Svace осуществляет анализ алиасов и сопоставление формальных и фактических параметров при вызове функции. Благодаря этому можно разрабатывать алгоритм поиска повторных блокировок для программы, содержащей только один семафор, а вся остальная работа будет выполнена средой Svace.

Система Svace выполняет неконсервативный анализ, допуская ложные срабатывания. Поэтому алгоритм поиска повторных блокировок реализованный в Svace может выдавать ложные срабатывания. Подробнее система Svace описана в [3], [4], [5].

Для тестирования алгоритма был создан набор тестов, моделирующий различные варианты использования блокировок. Для тестов алгоритм выдал предупреждения о повторной блокировке для всех ожидаемых ситуаций. Пример теста приведён на рис. 3, ожидается что предупреждение о повторной блокировке будет выдано для строки 12.

```
 1: void mutex_unlock(MUTEX*lock) {
 2:    unlock(lock);
 3:}
 4:void lock_section(MUTEX *lock) {
 5:    mutex_unlock(lock);
 6:    lock(lock); //svace: not_emitted DOUBLE_LOCK
 7:}
 8:void test(MUTEX* mut) {
 9:    lock(mut);
10:    if(flag) {
11:       lock_section(mut);
12:       lock(mut); //svace: emitted DOUBLE_LOCK
```

```
13:  }
14:}
```

Рис. 3. Пример теста для проверки алгоритма.

Также был произведён анализ 33 проектов с открытым исходным кодом (включая ядро операционной системы линукс (linux-kernel-3.10.9), binutils-2.20.1, libxml2-2.7.6, nss-3.12.9+ckbi-1.82, pulseaudio-0.9.22). Было выдано одно сообщение о двойной блокировке для pulseaudio-0.9.22, фрагмент кода приведён на рис. 4. Процедура pa_mutex_lock блокирует семафор e->mutex на строке 1068. После чего в цикле вызывается процедура pa_memexport_process_release на строке 1070, которая на строке 1087 вызывает процедуру pa_mutex_lock и повторно блокирует семафор. В данном случае сообщение является ложным, т. к. семафор был создан с флагом PTHREAD_MUTEX_RECURSIVE, позволяющим повторную блокировку. Анализатор выдал сообщение, т. к. не учитывает способ создания семафора. Таким образом с точки зрения анализатора предупреждение является истинным.

```
1065: void pa_memexport_free(pa_memexport *e) {
1066:    pa_assert(e);
1067:
1068:   pa_mutex_lock(e->mutex);
1069:   while (e->used_slots)
1070:       pa_memexport_process_release(e, (uint32_t) (e->used_slots - e->slots));
1071:   pa_mutex_unlock(e->mutex);
1072:
1073:   pa_mutex_lock(e->pool->mutex);
1074:   PA_LLIST_REMOVE(pa_memexport, e->pool->exports, e);
1075:   pa_mutex_unlock(e->pool->mutex);
1076:
1077:   pa_mutex_free(e->mutex);
1078:   pa_xfree(e);
}

1082: int pa_memexport_process_release(pa_memexport *e, uint32_t id) {
1083:   pa_memblock *b;послать
1084:
1085:   pa_assert(e);
```

```
1086:
1087:   pa_mutex_lock(e->mutex);
1088:
1089:   if(id >= e->n_init)
1090:       goto fail;
...
```

Рис. 4. Фрагмент кода для выданного сообщения.

6. Заключение

Был разработан и реализован алгоритм поиска двойных блокировок семафоров. Алгоритм разработан таким образом, чтобы выдавать как можно меньше ложных срабатываний. Для реализации использовалась система статического анализа Svace. Результаты тестирования показали, что алгоритм может быть полезен, но выдаётся слишком мало срабатываний. На наш взгляд причина заключается в большом количестве взаимосвязанных условных выражений, которые используются вместе с семафорами. Текущая реализация не выдаёт предупреждения для условных блокировок семафоров. Поэтому в планах по улучшению анализа – добавление в систему Svace разрешителя булевых формул, после чего алгоритм сможет выдавать предупреждения для более сложных ситуаций. При этом не потребуется менять сам алгоритм поиска двойных блокировок, а только инфраструктуру Svace.

Список литературы

[1]. POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0464 [121]

[2]. Cousot, P., & Cousot, R. (1992). Abstract interpretation and application to logic programs. The Journal of Logic Programming, 13(2), 103-179.

[3]. В.С. Несов. Автоматическое обнаружение дефектов при помощи межпроцедурного статического анализа исходного кода. Материалы XI Международной конференции «РусКрипто'2009».

[4]. А. Аветисян, А. Белеванцев, А. Бородин, В. Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН, 2011, том 21.

[5]. Иванников, В. П., Белеванцев, А. А., Бородин, А. Е., Игнатьев, В. Н., Журихин, Д. М., Аветисян А.И., Леонов, М. И. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды Института системного программирования РАН, 2014, 26(1), с. 231-250, doi: 10.15514/ISPRAS-2014-26(1)-7.

Static detection of error of double locking of mutex

Alexey Borodin
<alexey.borodin@ispras.ru>
ISP RAS, 25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. This paper describes algorithm for static search for error of double locking of mutex. The algorithm allows emitting warnings with low level of false positives. We considered finding errors for abstract library containing functions of mutex lock, unlock and conditional locking. We defined a set of regular languages, which models locks and unlocks during concrete execution of a program. We have defined a domain approximated set of regular languages. The algorithm is implemented in terms of data flow analysis. In the analysis elements of the domain are used as the data flow properties. The algorithm is described for a program that has only one mutex and does not contain any aliases. In that case every emitted warning will correspond to a real error which may occur during program execution. The algorithm is implemented in system of static analysis Svace developed in Institute for System Programming of the Russian Academy of Sciences. Svace performs alias analysis and matching of formal and actual function parameters. This makes it possible to apply the algorithm to search for double locking of a program containing only one mutex, and the rest of the work will be executed by Svace. The search algorithm of the double lock implemented in Svace can emit some number of false positives because Svace performs nonconservative analysis.

Keywords: static analysis; mutex; error of double locking; data-flow analysis; regular languages.

References

[1]. POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0464 [121]
[2]. Cousot, P., & Cousot, R. (1992). Abstract interpretation and application to logic programs. The Journal of Logic Programming, 13(2), 103-179.
[3]. Nesov V. S. Automatically Finding Bugs in Open Source Programs. Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2009).
[4]. Avetisyan A.I., Belevantsev A.A., Borodin A.E., Nesov V. S. Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode programm [Using static analysis to find vulnerabilities and critical errors in the source code of programs]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol 21 (in Russian).
[5]. Ivannikov V.P., Belevantsev A.A., Borodin A.E., Ignatyev V.N., Zhurikhin D.M., Avetisyan A.I., Leonov M.I. Staticheskij analizator Svace dlya poiska defektov v iskhodnom kode programm [Svace: static analyzer for detecting of defects in program source code]. Trudy ISP RAN [The Proceedings of ISP RAS], 2014, vol 26(1), pp. 231-250 (in Russian), doi: 10.15514/ISPRAS-2014-26(1)-7.