УДК 517.9

Н. Н. Ефанов, П. В. Емельянов
Московский физико-технический институт
(государственный университет)
E-mail: nefanov90@gmail.com

ПОСТРОЕНИЕ ФОРМАЛЬНОЙ ГРАММАТИКИ СИСТЕМНЫХ ВЫЗОВОВ

Целью данной работы является формализация задачи восстановления последовательностей системных вызовов, приводящих к некоторой целевой конфигурации дерева процессов Linux. Исследована комбинаторная сложность задачи. Ввиду оценок количества генерируемых деревьев предложено решение задачи на основании анализа записи дерева процессов как строки, порождаемой в некоторой формальной грамматике, которая также построена и исследована. По результатам исследований рассмотрен вариант реализации анализатора.

Ключевые слова: математическая модель, операционные системы, системные вызовы, формальные грамматики, дерево разбора, промежуточное представление

Задача о восстановлении дерева процессов

Задача о восстановлении дерева процессов вместе с порождающими цепочками системных вызовов возникает в различных компьютерных проектах, ориентированных на поддержку миграции виртуальных сред и процессов [1–2]. На текущий момент задача не имеет полноценного решения, что обуславливает актуальность проведения данной работы.

Сформулируем задачу: получая на вход снимок состояния дерева процессов в некоторый момент времени (к примеру, посредством утилиты ps), получить на выходе последовательности системных вызовов, выполнив которые можно получить входящее дерево из некоторого (предварительно заданного) начального состояния. На момент подготовки публикации были использованы следующие ограничения на задачу.

1. Для воссоздания дерева процессов используются исключительно цепочки запускаемых из пространства пользователя системные вызовы.

2. На вход подаётся корректное дерево, то есть существует цепочка системных вызовов, приводящая к нему из начального состояния. Если такая цепочка не найдена, дерево считается некорректным.
3. Используются системные вызовы:
    a) fork(), создающий процесс-потомок,
    b) setsid(), создающий новую сессию с лидером-вызвавшим процессом
    c) setpgid(), назначающий процессу группу. В решении считается, что данный системный вызов замкнут на себя, то есть вызывается только от имени процесса, изменяющего группу.
    d) exit(), завершающий вызвавший процесс и переназначающий INIT-процессу потомков [1–2].

Данные системные вызовы реализуют минимальное количество операций, необходимых для всевозможных преобразований над деревом процессов.

Оценки из перечислительной комбинаторики

Оценим количество различных конфигураций дерева процессов в зависимости от иерархий, сессий и групп процессов. Для этого рассчитаем количество различных деревьев процессов, полученных из INIT – процесса [1] посредством вызовов fork(), setsid(), setpgid(). Представим процесс-родитель узлом дерева с тройкой значений (идентификатор процесса, группы, сессии). Некоторый процесс-потомок данного процесса запишем после "$\Rightarrow$". При этом не будем предъявлять жесткие требования к идентификатору процесса, кроме указанных выше. Это означает, что деревья (1 1 1) $\Rightarrow$ (2 1 1) и (1 1 1) $\Rightarrow$ (5 1 1) являются неразличимыми.

1. С одним лишь fork. Зафиксируем количество процессов и установим, сколько различных деревьев можно получить на этих процессах, полагая системный вызов fork единственным способом порождения процесса-потомка.

Утверждение 1. Количество различных деревьев из n процессов, полученных посредством системного вызова fork, равняется количеству остовных деревьев полного графа с n вершинами.

Доказательство. Пронумеруем каждую вершину соответствующим ей идентификатором процесса (PID). Для получения одного дерева достаточно соединить все вершины так, чтобы в полученном графе не было циклов. Любое дерево, полученное таким соединением, будет

остовным деревом исходного графа по определению. Что и требовалось доказать.

Для вычисления количества различных остовных деревьев можно применить формулу Кели:

$$N = n^{n-2}, \text{ где } n \text{ – число вершин.}$$

Утверждение 2. Количество различных деревьев-процессов со свободным диапазоном идентификаторов от 2 до n-1 и выделенными идентификаторами[1] 0 и 1 равняется

$$F(n) = \sum_{i=2}^{n-1} i^{i-2}.$$

Доказательство методом математической индукции.

1) $n = 2$, тогда, $F(2) = 2^0$, что соответствует единственной конфигурации с процессами 0 и 1.

2) $n = k - 1$, тогда $F(k-1) = \sum_{i=2}^{k-2} i^{i-2}$.

3) $n = k$. Для меньшего числа процессов все различимые конфигурации уже перечислены на предыдущем шаге. Осталось перечислить все возможные конфигурации из k-процессов, что можно сделать, согласно утверждению 1, по формуле

4) $$F(k) = \sum_{i=2}^{k-2} i^{i-2} + (k-1)^{(k-1)-2} = \sum_{i=2}^{k-1} i^{i-2},$$

что и требовалось доказать.

2. С различимостью сессий – setsid. Системный вызов setsid, вызываемый для создания новой сессии, устанавливает вызвавшему процессу идентификатор сессии и группы, равные идентификатору процесса, – задаёт процесс лидером созданной сессии и группы. При этом вызвавший процесс не должен быть лидером группы. Следовательно, setsid можно корректно применить только один раз для каждого из процессов.

---

[1] Идентификаторы 0 и 1 используются для процессов SCHED и INIT соответственно. SCHED – процесс-планировщик, не создающий потомков, INIT -– процесс-инициализатор, являющийся общим прародителем процессов в системе: любой процесс получается цепочкой системных вызовов из INIT.

Утверждение 3. Учёт в формуле (2) различимости по сессиям даёт количество различных деревьев процессов, равное

$$F(n) = \sum_{i=2}^{n-1} i^{i-2} \cdot 2^{i-1} \cdot \sum_{j=0}^{2^l-1} 2^{(\mathbf{f}(j)\cdot\mathbf{k})},$$

где $\mathbf{f}(j)$ – вектор-функция размерности *l*-количества уровней дерева, возвращающая *j*-й бинарный вектор в порядке инкрементального перечисления, $\mathbf{k}$ – вектор, составленный из суммарного количества потомков на уровнях l … 1 соответственно.

Доказательство. Каждый процесс, не являющийся лидером группы, может вызвать setsid(), причем только один раз. Порядок вызова для процесса с потомками также имеет значение: потомок может быть порождён как до образования новой сессии, так и после, то есть существует $(1+2^{k_{children}})$ - вариантов для потомков процесса, причем каждый из них также может также сделать setsid(), добавляя $2^{k_{children}}$ вариантов для каждой из конфигураций. Проводя обход в ширину из корня дерева, считающийся нулевым уровнем глубины, получим выражение для поправки:

$(1+2^{k_1})\cdot 2^{k_1}\cdot\ldots\cdot(1+2^{k_l})\cdot 2^{k_l}$, раскрывая скобки в котором, а также учитывая, что $k_1+\ldots+k_l = i-1$ для i процессов, получаем выражение $2^{i-1}\cdot(1+2^{k_1}+\ldots+2^{k_1k_2}+\ldots+2^{k_1k_l}+\ldots+2^{k_1\cdot\ldots\cdot k_l})$, что эквивалентно поправке из утверждения.

3. С различимостью групп – setpgid(). Данный системный вызов назначает процессу, не являющемуся лидером, новую группу, если группа существует в одной сессии с процессом либо создает новую группу с указанным процессом-лидером. Предполагается, что процесс назначает группу сам себе для упрощения анализа. Учет групп вносит поправку, аналогичную виду поправки для сессий, за исключением того, что в первом сомножителе степень имеет вид $i-1-LC_{sg}$, где LC_{sg} – количество процессов, являющихся лидерами сессии либо группы, а во втором сомножителе нужно рассматривать отдельные суммы по потомкам, лежащим в одной сессии (можно считать от лидера сессии). Это очевидно из вышеописанной логики работы setpgid().

Ввиду того, что *n* в рассматриваемой задаче может достигать $2^{16}-1$, количество возможных конфигураций дерева процессов достаточно велико для применения жадного поиска на графах, а создание базовых

шаблонов для конструирования дерева приводит к необходимости тщательного профилирования системы, с учётом активности отдельных компонентов и приложений, что затрудняет восстановление для произвольной системы. Исходя из вышесказанного, авторами статьи было сконструировано решение в рамках задания правил системных вызовов в некоторой формальной грамматике над множеством строк в специальной нотации, описанной в следующем разделе.

Формальная грамматика системных вызовов

Ввиду оценок количества генерируемых деревьев предложено решение задачи на основании анализа записи дерева процессов как строки, порождаемой в некоторой формальной грамматике. Отдельный процесс в данной нотации записывается как “p g s [*]”, где p, g, s – идентификаторы процесса, группы и сессии, а символы “ [”.“]” – терминальные символы начала и конца списка потомков данного процесса. Таким образом поддерживается структура дерева уже на входной строке, что позволяет осуществлять разбор по правилу для fork() за один проход по ней. Само правило для fork() имеет вид: fork(* * * [*]) $\Rightarrow$ * * * [* \2 \3 [ ] \4].

Правило для setsid() имеет вид: setsid(* * * [*]) $\Rightarrow$ \1 \1 \1 [\4].

Правило для setpgid() имеет вид:

{p p * [*], setpgid(p,* * \3 [*])} $\Rightarrow$ {p p * [*], \1 \1 \3 [\4] | p p \3 [\4]},

где нетерминальные символы “ {”,“}” следует интерпретировать как «в системе существует или существовала такая конфигурация».

На данном этапе грамматика уже имеет тип 1, ввиду правила для setpgid. Выделяя вторую часть правила | p p * [*], соответствующую контекстно-свободному случаю создания процессом своей группы, а также кешируя встреченные сессии при первом проходе по строке, можно осуществить поддержку setsid(), setpgid() на этом проходе. Результатом последнего будет первое промежуточное представление – дерево с частично восстановленными системными вызовами, покрывающими вышеуказанные случаи для fork(), setsid(), setpgid().

Разработанный авторами анализатор строки по данной грамматике осуществляет на первом этапе восходящий разбор без учета контекста следующим образом: вводится стековая память с механизмом стекового кадра для каждого из процессов. Каждый процесс сохраняется на стеке в виде пары строк:

1) «состояние»-> «системный вызов» $\Rightarrow$ … $\Rightarrow$ «состояние»,

2) метаданные: ссылки на родителя и процессы-потомки.

Ссылки на конкретное состояние родителя получаются в дальнейшем и хранятся в специальном упорядоченном словаре.

При встрече символа “[“ создается новый стековый кадр, а при встрече “]“ осуществляется его размотка, с восстановлением системных вызовов по бесконтекстным частям правил. По окончании ввода в стеке содержится первое промежуточное представление, по которому осуществляется нисходящий разбор с проверкой контекста. По окончании процедуры в стеке хранится дерево разбора.

Проблема укорачивания грамматики на системном вызове exit()

Авторы рассматривают проблему нетерминальных exit(), при которых информации о вышедших процессах во входной строке не содержится. Правило для exit():{1 * * [*],exit(* * * [*])} $\Rightarrow$ 1 \1 \2 [\3 \7], которое делает конструируемую грамматику укорачивающей, рассматривается в случае, если никакими другими правилами невозможно присоединить потомка к INIT-процессу. В таком случае применяются 3 эвристики:

1) сессии потомка не существует ниже по стеку. Тогда создается потомок INIT с идентификатором, равным идентификатору данной сессии, и производится setsid(). Данный процесс помечается как вышедший и кладется на стек с соответствующей пометкой. Исследуемый потомок присоединяется к созданному процессу;

2) сессия существует, но в ней нет вышедших процессов. Вышедший процесс присоединяется к лидеру сессии, после чего к нему присоединяется исследуемый через последовательность setpgid(), setpgid(). Восстановленный процесс помечается как вышедший и кладется на стек аналогично эвристике 1;

3) сессия существует, в ней есть вышедшие процессы. Находится подходящий вышедший процесс, после чего к нему присоединяется исследуемый через последовательность setpgid(), setpgid(). В случае, если подходящих процессов не найдено, реализуется эвристика 2.

Пример работы анализатора

Рассмотрим строку "1 1 1 [5 4 3 [ ] 3 3 3 [4 4 3 [ ] 9 3 3 [ ]]]". Состояние стека после первого прохода по строке:

0:[1, 1, 1]
…
4:[3, 1, 1], 'setsid', [3, 3, 3] # процесс получен в setsid(fork(INIT))
5:[1, [7, 9]] # ссылки на родителя и потомков
6:[4, 3, 3], 'setpgid', [4, 4, 3] # бесконтекстный setpgid()
7:[5,[]]
…

Также на данном этапе собираются некоторые данные для второй фазы разбора: упорядоченный словарь упорядоченных списков групп по сессиям и пр. Далее производится нисходящий разбор по КЗ-правилам для setsid(), setpgid(), а также обрабатывается проверка и восстановление для exit(). Стек имеет вид:

…
2:[5, 4, 3]
3:[11,[]] # процесс назначен потомком восстановленного
…
6:[4, 3, 3], 'setpgid', [4, 4, 3]
7:[5,[11]] # добавлена ссылка на вышедший процесс
…
10:[10, 4, 3],’exit()’ # вышедший процесс
11:[11, [3]] #

Эквивалентное дерево разбора:

[1,1,1]
|__ [3 1 1] →setsid()→[3 3 3]
| |__[9 3 3]
|__[4 3 3]→setpgid(4)→[4 4 3]
|__[10 4 3]→exit()
|__[5 4 3]

Здесь запись “|__” эквивалентна системному вызову ‘fork()’

Заключение

Построенное решение представляет собой двухпроходный анализатор [3], временная сложность которого, с учетом использования эффективных

структур данных, составляет $O(N\log(N)\log(G)\log(S))$, где N,G,S – число процессов, групп и сессий соответственно. Добавление новых системных вызовов и поддержка дополнительных атрибутов, таких как, например, открытые файлы, усложнит алгоритм анализа, как минимум, в $\log(K)$ раз, где K – количество дополнительных элементов, подлежащих проверке в нисходящем разборе. Введение новых сложных правил, подобных нетерминальному exit(), существенно усложняет работу анализатора. Грамматика, представленная такими правилами, является грамматикой типа 0. Такие усложнения могут затруднить интеграцию анализатора в практические приложения. Ввиду данных предположений, авторы планируют разрабатывать альтернативные нотации и способы решения.

Литература

1. *Лав Роберт.* Linux.Системное программирование. – 2-е изд. – М.: ИД «Вильямс», 2014.
2. *Ефанов Н.Н.* Восстановление состояния Linux-процесса оптимальными последовательностями системных вызовов на основе марковских преобразователей. //Тезисы 59-й научно-практической конференции МФТИ. – М.: МФТИ, 2016.
3. *Ахо Альфред, Лам Моника, Сети Рави, Ульман Джеффи.* Компиляторы: принципы, технологии и инструментарий. – 2-е изд. – М.: ИД «Вильямс», 2008.

Получено 20.04.2017