Математическое моделирование, численные методы и комплексы программ

Ефанов Н.Н., *аспирант Московского физико-технического института (государственного университета)*

КЛАССИФИКАЦИЯ ПРАВИЛ ПРОВЕРКИ АТРИБУТОВ В ДЕРЕВЬЯХ ПРОЦЕССОВ LINUX

Работа посвящена построению, классификации и сведению к базовым алгоритмам поиска на деревьях правил проверок атрибутов процессов ОС Linux. Предложенные проверки используются при разборе контекста в параметризованной грамматике системных вызовов Linux для разбора деревьев процессов. Исследование приводит бесконтекстную составляющую грамматики с параметризуемыми нетерминалами, классификацию проверок по области поиска требуемых атрибутов, алгоритмическую сложность проверок и результирующий алгоритм. Результаты по классификации некоторых системных вызовов Linux также приводятся в работе. Основным выводом из предлагаемого исследования служит заключение об увеличении сложности разбора не более чем в квадрат раз от числа процессов при относительно малом числе типов системных вызовов, что говорит о целесообразности практического построения анализаторов деревьев процессов с предлагаемыми проверками атрибутов.

Ключевые слова: *Операционные системы, системные вызовы, дерево процессов, формальные грамматики, алгоритмы поиска, классификация, временные оценки.*

Efanov N.N.

RULES CLASSIFICATION FOR CHECKING ATTRIBUTES IN LINUX PROCESS TREES

The article is focused on construction, classification and reduction to the basic search algorithms for the rules of attributes checking for Linux processes. The suggested checks are used when parsing the context in the parameterized grammar of Linux syscalls for parsing process trees. The study results in a context-free component of the grammar with parameterizable nonterminals, classification of checks on the area of search for required attributes, algorithmic complexity of checks and the resulting algorithm. The results on the classification of some Linux syscalls are also given in the paper. The main conclusion of the proposed study is that the complexity of parsing is increased by not more than a quadruple from the number of processes with a relatively small number of syscalls types, which indicates the practicality of constructing process tree analyzers with proposed attribute tests.

Keywords: *Operation Systems, syscalls, process tree, parsing tree, formal grammars, search algorithms, time complexity.*

Введение

Сложность и высокие требования к качеству и высоконагруженных компьютерных систем и виртуализованных окружений мотивирует повышение эффективности и снижение накладных затрат при эксплуатации таких систем. Исполнение сложных задач на таких системах требует поддержки относительно сложной процедуры сохранения и восстановления состояния сред исполнения: процессов, виртуальных машин, контейнеров по контрольным точкам с целью воспроизводимости, поддержки процедуры живой миграции, повышения отказоустойчивости [1]. С целью снижения затрат на сохранение-восстановление предлагается [2-3] производить восстановление состояния путём воспроизведения дерева процессов, соответствующего дереву процессов при сохранении. Тем не менее, прямая генерация деревьев с заданными свойс-

твами представляется достаточно комбинаторно сложной задачей [3] для практической реализации такого метода, ввиду чего в ранних работах автора [2-3] предлагается решение данной проблемы подходом из формальных языков и грамматик: вводится формальная грамматика системных вызовов, порождающая язык деревьев процессов Unix-подобных операционных систем [3]. Разбор такого дерева в данной грамматике позволяет восстановить цепочки системных вызовов, воспроизводящие требуемое состояние. Предлагаемая грамматика является укорачивающей (Тип 0 по Хомскому) [4], однако эвристическая обработка укорачиваний позволяет осуществлять разбор данной грамматики как грамматики из мягко-контекстно-зависимых формализмов [5] за 2 прохода по промежуточному состоянию дерева разбора. Очевидно, данная грамматика должна поддерживать множество проверок контекста, ввиду зависимости свойств одних процессов от свойств других.

Целью представленной работы является введение и классификация правил непосредственных проверок свойств процессов как на анализируемом дереве процессов, так и на дереве разбора, через проверку соответствующих атрибутов вершин – идентификатора процесса, группы, сессии [2-3], списка открытых файловых дескрипторов и других идентификаторов используемых ресурсов.

Бесконтекстная составляющая грамматики деревьев процессов.

Исходя из работы [2], простейшая схема разбора дерева процессов – 2-х проходная схема с «наивным» разбором дерева КС-грамматике на 1-м проходе, и анализом контекста на 2-м. Следовательно, с точки зрения повышения эффективности разбора, результирующая грамматика может быть построена как грамматика с проверкой атрибутов, где некоторые терминалы поддерживают проверку условий на вывод. Ввиду особенности прикладной задачи – локализации системного вызова [2-3], проверка контекста производится только на уровне «процесс-процесс», что позволяет произвести бесконтекстный разбор некоторого дерева на 1-м шаге, после чего осуществить бесконтекстный разбор дерева, с последующим получением итогового дерева из промежуточного состояния после 1-го шага проверкой свойств процессов.

Приведём простейшее описание бесконтекстной грамматики, описывающей дерево процессов как дерево с фиксированными атрибутами в корне. Для упрощения повествования, списки атрибутов процессов выводятся как терминалы, и представляются как символьные строки. Такой подход укрупнения «атомарных» единиц грамматики соответствует практическому смыслу и позволяет уменьшить число грамматических правил.

Определение. Бесконтекстной составляющей грамматики системных вызовов Linux является грамматика $\{T, N, S, P\}$, где

T – множество терминалов – подстрок, перечисляющих атрибуты соответствующих процессов, получаемых при проверках контекста, а также "(1..1)" – строчка для init-процесса с фиксированными атрибутами[1].

N – {<*S*>, <*branch*>, <*node*>, <*init*>, <*mul*>, ε и <*children*>} – множество нетерминалов, используемых в бесконтекстных правилах.

S – стартовый нетерминал. Здесь им можно положить дерево, подаваемое на вход, формально считая его типовой конструкцией, разбираемой в правилах ниже.

P – набор правил, приведённых в форме Бэкуса-Наура [5]:

<*S*> = <*init*> | <*init*>, <*children*> (1)

<*children*> = <*mul*>

<*mul*> = ε| <*mul*>, <*branch*>

<*branch*> = <*node*> | <*node*>, <*children*>

<*init*> = "(1..1)"

[1] Init-процесс считается корневым, с единичными идентификаторами.

<node> = "check_rules(...)" , где

"check_rules(...)" – символическое обозначение вызова процедуры проверки атрибутов процессов, в которой происходит дальнейшая обработка промежуточного состояния, входящая в грамматику (1) как терминал.

Данная форма предложенной грамматики разбирается в за $O(n^3)$, а восходящий разбор естественным образом отражает топологию изначальных входных данных – дерева процессов.

Дерево разбора в предложенной грамматике можно использовать как промежуточное представление для дальнейшей проверки атрибутов и применимости правил системных вызовов.

Классификация правил проверки атрибутов.

Согласно типовым операциям , производимым при передаче управления из пространства пользователя в ядро ОС, автором были выделены 4 класса правил проверки атрибутов, на основании требований области «видимости» атрибутов одних процессов из других:

Глобальные (“global”)– для проверки требуется обход всего дерева.

Локальные (“local”) – проверки не требуется, либо проверяется локальный атрибут. Локальным атрибутом считается либо атрибут процесса, либо некоторый атрибут родителя процесса. Данный класс включает только безконтекстные проверки в смысле грамматики (1), ввиду того, что все они могут проводиться «внутри» нетерминала *<node>*.

Проверка поддерева (“subtree”) – требуется обход поддерева, в котором лежит текущая вершина, начиная с некоторого корня, выбираемого условно. Применяется для проверки группировки процессов во вложенные конструкции: группы процессов в сессиях, пространства имён и др.

Проверка ветви (“branch”) – атрибута, который наследуется сверху.

Ввиду классификации выше, для поддержания проверок атрибутов следует применить 3 операции:

обход дерева: работа использует обходы поиском в глубину [6] на дереве, требующие $O(n + m)$ при – вершин и $m = n - 1$ рёбер.

поиск вершины с нужными свойствами по ветви от текущей вершины до корня за $O(n)$.

проверка текущей вершины: $O(1)$.

В условиях совершения проверки для каждой из вершин, временная сложность возрастает в $O(n)$ – раз.

Результирующая временная сложность проверок и подходящие системные вызовы приводятся в сводной таблице 1:

Таблица 1.

Применение различных классов проверок атрибутов в правилах для некоторых системных вызовов

Тип правила	Применение, системные вызовы	Временная сложность
global	Проверка открытых файлов, сетевых соединений, IPC: pipe, др. Обработка укорачиваний: exit.	$O(n)\ (O(n) + O(n + m)) = O(n^2)$

subtree	Проверка вложенных групповых свойств: группы процессов – setpgid и др.	$O(n)\ (O(n) + O(n + m)) = O(n^2)$
local	Создание процесса, работа с ресурсами: файлами, виртуальной памятью – fork, clone, exec, open, read, write, close, getpid, getsid, getpgrp и др.	$O(n)$
branch	Проверка сессии и других атрибутов, наследуемых сверху – setsid и др.	$O(n)\ O(n) = O(n^2)$

Алгоритм проверки атрибутов

Приведём алгоритм проверки атрибутов процессов, который может использоваться для разбора контекста в параметризованной грамматике, построенной на базе вводимой ранее. При описании алгоритма используется Python-подобный псевдокод.

Алгоритм 1.

Вход: attr # словарь атрибутов вида {ключ:значение}

Выход: res # список корректных (условно выполняемых) правил

```
begin:
 res = [] # пустой список
 rules=get_rules() # получить правила, применимость которых проверяется
 for rule in rules:
 if check(attr, rule.type):
 res.append(rule)
 return rules
subprogram check(attr, type):
 if type == ‘ branch’:
 return check_branch(attr)
 if type == ‘ subtree ‘:
 root=find_root(attr) # поиск вверх по ветви дерева
 return dfs(root, attr) # поиск в глубину от корня root
 if type==’global’:
 root=find_root({‘p’:1}) # поиск вверх по ветви дерева
 return dfs(root, attr) # поиск в глубину от корня root
 else: # атрибуты проверяются локально:
 return check_local_attr(...)
```

Использование приведённого алгоритма позволяет осуществить поддержку различных правил системных вызовов, вообще говоря, не более чем за $O(n^2)$ по времени для m правил и n вершин в промежуточном состоянии дерева разбора при $m << n$, что является практически важным результатом и открывает широкие возможности по реализации и интеграции средств анализа деревьев процессов, основанных на вводимых методах, в информационно-технические системы, поддерживающие сохранение и восстановление состояний по контрольным точкам [1-3, 7].

ЛИТЕРАТУРА

1. *Andrey Mirkin, Alexey Kuznetsov, Kir Kolyshkin.* Containers checkpointing and live migration. In Proceedings of the In Ottawa Linux Symposium. Volume Two. Ontario, Canada, 2008.
2. *Ефанов Н. Н., Емельянов П.В.* Построение формальной грамматики системных вызовов // М. Информационное обеспечение математических моделей, 2017 – 83-90 С.
3. *Nikolay Efanov and Pavel Emelyanov.* 2017. Constructing the formal grammar of system calls. In Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17). ACM, New York, NY, USA, Article 12, 5 pages. DOI: https://doi.org/10.1145/3166094.3166106
4. *Partee B.H., Ter Meulen A., Wall R.E.* Turing Machines, Recursively Enumerable Languages and Type 0 Grammars // Mathematical Methods in Linguistics (Studies in Linguistics and Philosophy). 1993. V. 30. P. 505–525.
5. *David J.* Weir. "Characterizing Mildly Context-Sensitive Grammar Formalisms". Ph.D. thesis, University of Pennsylvania, Philadelphia, USA, 1988.
6. *Sedgewick R.* Algorithms in C++ Part 5: Graph Algorithms, 3rd Edition // Pearson Education. 2002.
7. Checkpoint-Restore In Userspace (CRIU). Comparison to other CR projects. 2015. https://criu.org/Comparison_to_other_CR_projects