Федеральное государственное бюджетное образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский Академический университет Российской академии наук»
Центр высшего образования

Кафедра математических и информационных технологий

Горбунов Егор Алексеевич

Алгоритм генерации команд восстановления дерева процессов ОС Linux на основе модели жизненного цикла ресурсов ОС

Магистерская диссертация

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:
магистр прикладной математики и физики, Баталов Е. А.

Рецензент:
к. ф.-м. н., Емельянов П. В.

Санкт-Петербург
2017

Оглавление

Глава 1

Введение

1.1. О задаче восстановления и сохранения дерева процессов

Основная задача, которая изучается в данной работе, называется “сохранение и восстановление состояния дерева Linux процессов”. Как мы видим, задача состоит из двух частей: сохранение и восстановление.

Сохранение состояния одного процесса включает в себя сохранение состояния всех ресурсов из которых состоит процесс, а также сохранение состояния окружения, в котором исполняется процесс. Из каких ресурсов состоит типичный Linux процесс? Это регионы виртуальной памяти, открытые файлы, сокеты, устройства, идентификатор процесса, его сессия, группа, идентификатор текущего пользователя и так далее. Что входит в окружение процесса? Это пространства имен процесса (namespaces), контрольные группы (cgroups) и другое плюс их настройки. Стоит заметить, что при сохранении состояния процесса копии открытых им файлов не создаются. Это связано с тем, что файлы и так постоянно хранятся в ФС (файловой системе), а их копии могут занимать очень много места.

Что понимается под восстановлением состояния одного процесса? Это создание всех ресурсов процесса и всех объектов из его окружения, существовавших в момент сохранения, а также восстановление их состояния таким, каким оно было в момент сохранения. Важно, что после восстановления потоки процесса не должны заметить никакого изменения в поведении системных вызовов и своих машинных инструкций. То есть исполнение процесса должно продолжиться так, как если бы он не подвергался операциям сохранения и восстановления своего состояния. Внешние наблюдатели, то есть другие процессы в системе, тоже должны увидеть

этот процесс максимально похожим на исходный сохраненный.

Выше были описаны задачи восстановления и сохранения одного процесса, но на практике нередки случаи, когда приложение состоит из нескольких процессов, имеющих общего родителя (child-parent relation). Каждый ребенок родителя может породить новые процессы приложения, а те в свою очередь другие процессы. В итоге, Linux приложения обычно представляют из себя дерево процессов, связанных отношением родитель-потомок. Кроме того, такая важная в современных облачных технологиях сущность как Linux контейнер, также представляет из себя дерево процессов с корнем, являющимся init процессом всего контейнера. Задача восстановления и сохранения дерева процессов требует, помимо состояний каждого из процессов в отдельности, поддерживать ещё и взаимоотношения между этими процессами, разделяемые ресурсы и другие особенности деревьев процессов в Linux.

1.2. Актуальность задачи сохранения/восстановления

1.2.1. Живая миграция

Живая миграция это перенос работающего приложения с одного узла кластера на другой, выполняемый незаметно для их пользователя. Живую миграцию можно выполнить с помощью сохранения и восстановления процессов приложения. Для этого нужно выполнить сохранение состояния процессов в файлы ФС на исходном узле кластера, скопировать файлы на целевой узел кластера, выполнить восстановление процессов из файлов на целевом узле. Если все эти операции выполняются за незаметное для пользователя время, то миграция является “живой”.

1.2.2. Обновление ядра без остановки программ

Для ядра Linux регулярно выходят обновления безопасности. При обновлении ядра нередко требуется его перезапуск и остановка всех процессов, работающих в ОС. Если процессы достаточно большие, например, БД занимающие 100 Гб оперативной памяти, то при пропускной способности чтения с диска в 100 Мб/с понадобится 1000 с = 15 мин, чтобы приложения считали свои данные с диска и восстановили свою работу после перезапуска. 15 минут это очень длительная задержка, которой можно избежать при использовании технологии сохранения и восстановления процессов без копирования их памяти. Для этого надо сохранить состояние

приложений в оперативной памяти (без копирования страниц памяти приложений [1]), выполнить перезагрузку ядра, выполнить восстановление приложений без копирования их памяти.

1.2.3. Отложенная отладка

Благодаря технологии сохранения и восстановления процессов мы можем получить “живую” копию приложения для отладки на компьютере разработчика. Без данной технологии мы бы могли анализировать только дамп приложения - статичный снимок его состояния, полученный в определенный момент времени.

1.2.4. Снимки приложений

Технология сохранения и восстановления процессов позволяет создавать снимки состояний приложения в разные моменты времени и переключаться между ними.

1.2.5. Ускорение запуска программ

Некоторые приложения выполняют большой объем работы при своем старте. Технология сохранения и восстановления процессов позволяет сохранить состояние приложения после завершения его инициализации. Вместо запуска приложения выполняется его восстановление в уже проинициализированном состоянии.

1.3. Системная утилита `criu`

Существует несколько программных решений задачи сохранения и восстановления процессов. В данной работе мы сосредоточимся на утилите `criu` [2], обладающей следующими преимуществами перед другими проектами:

- Работает в пространстве пользователя
- Не требует подмены динамических библиотек для отслеживания поведения процессов, в отличие от проекта DMTCP [3]

- Не требует внесения тяжеловесных модификаций в ядро ОС, в отличие от проектов OpenVZ [5] и BLCR [4]
- Активно разрабатывается и поддерживается

Задача сохранения процесса решается в `criu` с помощью остановки целевого дерева процессов и вычитывания информации о нём с помощью интерфейсов, предоставляемым операционной системой: виртуальная файловая система `/proc` (информация о процессе из вне), системные вызовы и чтение памяти (изнутри самого процесса). В проекте DMTCP, к примеру, используются обёртки над некоторыми функциями `glibc`, тем самым информация о процессе в том числе собирается в течение жизни процесса [1].

На этапе сохранения процесса программа производит анализ корректного, с точки зрения ОС, дерева процессов, создавая его снимок. Но дело в том, что снимок дерева говорит нам лишь о некотором закреплённом состоянии этого дерева и он не содержит информации о том, как именно оно пришло в это состояние. С течение времени дерево процессов эволюционирует:

- Открывает файлы, сокеты, каналы и пр
- Модифицирует виртуальное пространство удаляя и добавляя области (`mmap`)
- Создаёт дочерние процессы
- Переходит из группы в группу
- Переходит из одного пространства имён в другое (namespaces)
- Переходит в новую сессию
- ...

Каждая из операций переводит процесс из одного состояния в другое и если мы представим себе граф состояний и переходов между ними, то легко увидим, что такой граф обладает нетривиальной структурой:

- Между двумя состояниями дерева может быть много путей: файлы можно открывать в разной последовательности; для того, чтобы 2 процесса, родитель и ребёнок, ссылались на один и тот же файл по одному и тому же файловому дескриптору, можно открыть файл родителем, после чего совершить `fork` ребёнка, а можно сначала совершить `fork`, а потом передать файловый дескриптор используя сокеты домена Unix.

[1] Конкретный момент времени исполнения тех или иных системных вызовов не сохраняется, поэтому по сути данный подход не добавляет никакой дополнительной полезной информации.

- Существуют состояния дерева, между которыми нет пути: процесс, находящийся в определённом пространстве идентификаторов процесса (pid-namespace) не может стать процессом из другого пространства (и наоборот); Процесс лидер сессии (session leader) не может перестать быть им (не завершаясь).

Таким образом восстановление дерева процессов должно быть реализовано так, что в каждый момент времени наше текущее состояние дерева процессов ещё имеет возможность добраться до состояния целевого, переходя по рёбрам — системным вызовам (и другим операциям, изменяющим состояние процесса). Необходимость поддерживать это свойство во время восстановления и делает задачу сложной: мы не можем просто взять и для каждого ресурса процесса написать независимую от текущего состояния этого процесса (и дерева процессов) функцию, которая бы просто выполняла восстановление этого ресурса [2].

1.4. Проблемы текущего подхода `criu` к восстановлению

`Criu` хорошо справляется с задачей восстановления большого количества процессов с различными ресурсами, но текущий подход к процедуре восстановления — это некоторая зафиксированная последовательность из большого количества шагов. Проверка файлов снимка дерева, инициализация `vdso`, частичное считывание информации из образа и инициализация различных структур, которые используются при восстановлении. В том числе тут происходит инициализация информации о процессах-помошниках, которые нужны, например, для того, чтобы верно восстановить сессии процессов, которые когда-то были «усыновлены» процессом `init` [3]. Далее начинается процедура восстановления: происходит восстановление корневого процесса и пространств имён для него (которые далее наследуются), создание дерева процессов (fork), и т.д. Такая цепочка действий чётко зафиксирована в коде `criu`.

Цепочка чётко отсортированных шагов выражает индивидуальных подход к восстановлению каждого из ресурсов. Это приводит к следующим проблемам:

- Код для восстановления каждого типа ресурсов нужно добавить в последовательность действий по восстановлению всего дерева так, чтобы его логика согласовывалась со всей остальной логикой, которой очень много, ведь последовательность действий очень длинная
- Любые нетривиальные зависимости между ресурсами требуют добавления дополнитель-

[2] Для некоторых ресурсов процесса это возможно: состояния регистров потоков процесса и др.
[3] Такое происходит, когда процесс-родитель завершается раньше ребёнка

ного кода, обрабатывающего эту нетривиальность, в цепочку действий

- Фиксированная цепочка действий может являться потенциальной причиной меньшего множества возможных к восстановлению процессов, чем могло бы быть (вспомним про метафору графа состояний дерева процессов, описанную выше)
- Отсутствие чёткого понимания того, какие конфигурации ресурсов дерева процессов `criu` гарантированно поддерживает

Сложность и отсутствие гибкости цепочки действий приводит к тому, что восстановление определённых ресурсов и взаимосвязей между ними трудно реализуемо.

Пример 1. Рассмотрим в качестве примера группы процессов в дереве процессов, изображённом на рисунке 1.1.

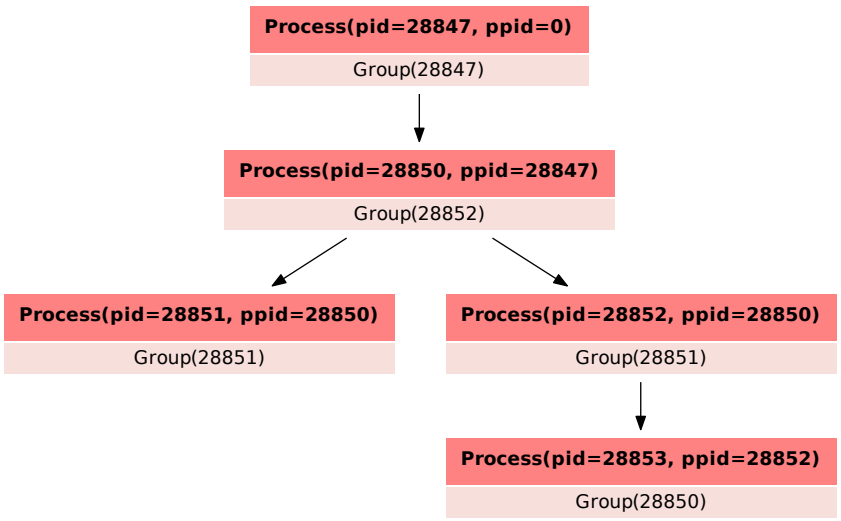


Рис. 1.1: Дерево процессов

В Linux, группа с идентификатором 28852, может быть создана только в процессе с идентификатором (pid) равным 28852. Но как мы видем из рисунка 1.1, процесс 28852 в целевом дереве находится в другой группе, а именно в 28851. На таком дереве процессов `criu` завершается с ошибкой о том, что не может присвоить процессу 28850 группу 28852. Это происходит из-за того, что на момент исполнения системного вызова установки группы, группы 28852 ещё (или уже) вовсе не существует. Данное дерево процессов является нетривиальным, так как его восстановление требует того, чтобы во время жизни процесс 28852 побывал сначала в одной группе (28852), а потом оказался в другой.

1.5. Возможные подходы к решению проблемы восстановления

Каким образом можно избежать проблем, описанных в предыдущем разделе? На данный момент, методов, которые решают проблему восстановления дерева процессов, качественнее чем `criu`, нет[4]. Но в предыдущих работах, касающихся этой проблемы, был озвучен общий подход: разделение процедуры восстановления на несколько этапов (см. рис 1.2).

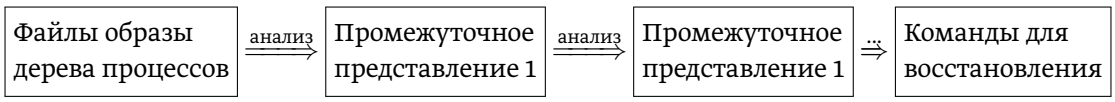


Рис. 1.2: Разделение на этапы

Каждый этап (количество этапов не специфицируется) производит анализ результатов предыдущего шага. Всё начинается с анализа файлов-образов процесса, считываемых с диска. Промежуточное представление — это представленные определённым и удобным для некоторых целей, данные. Тут можно провести аналогию с компиляторами, где промежуточными представлениями выступают структуры, такие как абстрактное синтаксическое дерево, single static assignment form и др. Они служат вспомогательными структурами, которые помогают выполнять те или иные операции эффективнее и проще.

Конечным же продуктом череды промежуточных представлений является набор команд. Этот набор команд предполагает, что он может быть исполнен процессом-интерпретатором так, что итогом исполнения является целевое дерево процессов (восстановленное). То есть начиная с одного пустого корневого процесса-интерпретатора, по мере исполнения команд, дерево процессов приходит в целевой вид.

Вопросы, который ставит данных подход:

- Каковы должны быть промежуточные представления?
- Какой вид имеет финальный список команд? Т.е. какие команды передаются на исполнение интерпретатору?

Такой подход предполагает разделение процедуры восстановления (`criu restore`) на генерацию и интерпретацию команд: генератор и интерпретатор. А значит, у нас появляется свобода выбора: насколько сложным делать генератор и интерпретатор. Ясно, что интерпретатору

[4]В рамках ограничений пользовательского приложения

придётся делать системные вызовы [5] для приведения дерева процессов в нужное состояние. Это значит, что наиболее простой с точки зрения интерпретатора набор команд — это набор системных вызовов, которые нужно последовательно исполнить. В такой ситуации генератор будет максимально сложным. Таким образом параметром данного подхода в том числе является компромисс между сложностью интерпретатора и генератора.

Одним из возможных промежуточных представлений как раз является граф ресурсов. Каждый ресурс, такой как процесс, поток, сессия, файловый дескриптор, OFD, регион памяти и так далее, является в этом графе вершиной с атрибутами, такими как PID, путь в ФС, id, размер и так далее. Ребра в таком графе направленные и представляют собой отношения между ресурсами.

Подход с разделением восстановления на многоступенчатый генератор и интерпретатор позволяет сделать процесс восстановления более гибким и простым:

- В силу того, что генератор не выполняет системных вызовов, мы можем использовать более широкий набор инструментов (в. т. ч. более высокоуровневые языки программирования)
- Генеративный подход имеет больший потенциал, чем зафиксированная последовательность действий, т.к. потенциально может сгенерировать произвольный набор команд для исполнения: теоретически мы можем запустить полный перебор всевозможных последовательностей системных вызовов.

1.6. Задачи и требования

Задача, которая была поставлена при выполнении данной работы, заключается в следующем:

- Разработать генератор команд для задачи восстановления дерева процессов в рамках подхода генератор-интерпретатор, описанного в разделе 1.5
  - Разработать промежуточные представления
  - Выбрать оптимальный баланс сложности генератора и интерпретатора

Общими требованиями к итоговому решению являются:

- Поддержка как можно большего числа ресурсов процессов в Linux

[5] Именно с помощью системных вызовов производится большая часть изменений состояния процесса. Так же запись в память процесса и регистры процессора изменяет его общее состояние

- Возможность эффективной реализации предлагаемых алгоритмов
- Генерируемые команды должны быть исполнимы из пространства пользователя

Основная цель: отойти от фиксированного порядка восстановления и найти обобщённый подход к восстановлению ресурсов, который будет проще подвергаться анализу.

1.7. Содержание данной работы

В данной части мы вкратце описали проблемы текущих подходов к восстановлению дерева процессов и возможные подходы к решению этих проблем, а так же сформулировали задачи. В части 2 подробно раскрывается конкретное промежуточное представление и подход к генерации команд, предлагаемых в данной работе. Этот подход основан на модели жизнедеятельности (обмена ресурсами) процессов в Linux и графе абстрактных действий, которые процессы совершают, чтобы этими ресурсами обмениваться, создавать их и уничтожать. В последней части, подводятся итоги и описываются дальнейшие планы на развитие подхода, предлагаемого в этой работе.

Глава 2

Модель жизненного цикла ресурсов и алгоритм построения графа действий

В этой главе мы формально вводим понятия ресурса и процесса, а так же различных свойств ресурсов (раздел 2.1). Далее мы вводим набор действий (команд), описывающих жизнедеятельность процессов и их ресурсов (раздел 2.2). В разделе 2.3 описывается алгоритм построения набора этих действий, исполнение которых решает задачу восстановления. Данный алгоритм и является основным предметом изучения этой работы.

2.1. Модель процесса и ресурса

Реальный процесс Linux, как уже говорилось, состоит из различных ресурсов, каждый из которых обладает своей спецификой. Для того, чтобы разработать обобщённый подход к восстановлению ресурсов, нужно сформулировать задачу восстановления в рамках абстрактной модели, которая бы отражала реальное поведение процессов и реальные свойства ресурсов в Linux.

2.1.1. Ресурс и процесс

Для разработки более общего подхода к алгоритму восстановления (генерации команд для восстановления), в плане покрываемых ресурсов, понятие ресурса должно быть достаточно широким.

Определение 1. *Ресурс* — r — структура в ядре ОС, которая так или иначе используются *процес-*

сом и операционной системой. У ресурса есть тип $type(r)$, с помощью которого ресурсы одного типа можно отличить от ресурсов другого типа.

Ресурсом может выступать: файл (`file struct`), группа процессов, сессия процессов, рабочая директория процесса, канал (pipe), и так далее.

Ресурсы существуют не просто так, а как описывается в определении, они используются процессами. Обычно, процесс не может взаимодействовать с ресурсом, как со структурой/объектом в ядре, напрямую. Вместо этого у процесса есть некоторый способ указать на ресурс в ядре и обратиться к нему.

Определение 2. *Указатель на ресурс — handle* — это объект, через который процесс получает доступ к ресурсу. Будем обозначать его как h и называть $handle$.

Во введении мы неформально говорили, что дерево процессов (и один процесс в частности) — это набор ресурсов. В терминах определений выше мы будем считать что процесс — это набор пар (r, h), где r — ресурс, а h — интерфейс к ресурсу.

Пример 2. При открытия файла процессом (вызов `open()`) ядро создаёт объект `file`, а процессу возвращается файловый дескриптор `fd`, посредством которого с файлом можно работать. Файловый дескриптор — это по сути просто число, с помощью которого можно идентифицировать файл. В это случае *ресурс* — это объект `file`, а *handle* — это файловый дескриптор. Из этого примера видно:

- Понятие указателя к ресурсу нельзя отождествлять с самим ресурсом, т.к. сам ресурс может иметь несколько интерфейсов: два файловых дескриптора указывают на один и тот же файл: (r, h_1), (r, h_2). Но для некоторых ресурсов понятие $handle$ однозначно. К примеру, процесс, находящий в группе g с идентификатором gid может как-то указывать на эту группу только с помощью её идентификатора gid (посылать сигнал, и пр.).

У $handle$ h тоже есть тип: $type(h)$, который позволяет отличить $handle$ разных типов. К примеру рассмотрим системный вызов `pipe`, который возвращает 2 файловых дескриптора: с помощью одного можно читать из канала, а с помощью другого можно только писать в него. Абстрактность текущего определения ресурса позволяет поступить несколькими способами, в том числе можно сказать, что `pipe` создаёт ресурс $pipe$ в ядре, к которому предоставляет два указателя разных типов: один позволяет указывать на «выход», а другой на «вход». выф

Теперь формализуем понятие процесса.

Определение 3. *Процесс* — это множество пар (r_i, h_i) из ресурсов и $handle$ к них, обладающее двумя атрибутами: pid — идентификатор процесса и $ppid$ — идентификатор родителя процесса. Будем обозначать процессы заглавными буквами и писать, например:

$$P = \{(r_1, h_1), (r_2, h_2), \ldots, (r_n, h_n)\}$$

Факт того, что процесс P владеет ресурсом r, к которому получает доступ через *handle* h, обозначаем так: $(r, h) \in P$. Также будем писать, что $r \in P$, если $\exists h : (r, h) \in P$.

Будем предполагать далее, что существует множество процессов $\mathcal{P}$, в которое входят всевозможные P. Все кванторы ($\exists, \forall$), аргументами которых является процесс, будут обозначать, что процесс этот берётся из $mathcal P$, т.е. $\exists P \iff \exists P \in \mathcal{P}$.

Определение 4. Дерево процессов — это множество из нескольких процессов: $T = \{P_1, P_2, \ldots, P_k\}$, среди которых выделен корневой процесс P_{root}, а для всех остальных процессов верно:

$$\forall P \in T, P \neq P_{root} : \exists P' \in T(P'.pid = P.ppid)$$

Т.е. родитель любого из процессов, кроме одного (P_{root}), находятся в T.

Мы будем писать: $r \in T$, если $\exists P \in T(r \in P)$.

Мы выделяем идентификатор процесса и родителя процесса в отдельные атрибуты, не являющимися ресурсами, из-за того, что их единственная роль — идентифицировать процесс среди остальных в дереве процессов.

2.1.2. Разделяемые и неразделяемые ресурсы

Ресурсы можно разбить на два множества: те, что могут быть разделены между несколькими процессами и те, что индивидуальны для каждого процесса.

Определение 5. *Разделяемый ресурс* — такой ресурс r, что

$$\exists P_1, P_2 \in \mathcal{P} : P_1 \neq P_2 \wedge (r, h_1) \in P_1 \wedge (r, h_2) \in P_2$$

То есть возможны таки два процесса, что оба процесса ссылаются на один и тот же ресурс r в ядре, причём не обязательно по одинаковым интерфейсам.

Неразделяемый ресурс, соответственно, тот, что не является разделяемым в рамках определения выше.

К примеру:

- значения регистров процесса для потока — неразделяемый ресурс
- открытый файл — разделяемый ресурс
- группа процессов — разделяемый ресурс
- сессия процессов — разделяемый ресурс

Также есть ресурсы с особенной семантикой: например, private mappings — это, казалось бы, неразделяемый ресурс, который при наследовании, всё же, становится разделяемым из-за механизма Copy On Write. Таким образом в нашей модели данный ресурс будет разделяемым, чтобы поддержать данную возможность.

2.1.3. Наследуемый ресурс

Как говорилось ранее, между процессами в дереве есть отношение родитель-ребёнок. При создании процесса-ребёнка процессом-родителем в Linux, часть ресурсов может наследоваться ребёнком. Для того, чтобы учитывать это при построении последовательности действий для восстановления, вводим:

Определение 6. $\mathrm{isInherited}(r)$ — предикат (свойство типа ресурса), который принимает значение истины, если ресурс типа $\mathrm{type}(r)$ наследуется процессом-ребёнком от процесса-родителя при создании.

Ещё раз заметим, что $\mathrm{isInherited}(r)$ — это просто свойство типа ресурса r, т.е. если r — файл, то $\mathrm{isInherited}(r) = \mathrm{true}$, но это не значит, что в процессе восстановления этот ресурс обязан наследоваться.

2.1.4. Разделение ресурса между процессами

Помимо наследования, один ресурс может быть «передан» от одного процесса другому уже после того, как эти процессы были созданы. Некоторые ресурсы могут быть разделены таким образом, а некоторые нет. В связи с этим нам понадобится ещё один предикат (свойство ресурса):

Определение 7. $\mathtt{isSharable}(r)$ — предикат, который принимает значение истины, если ресурс типа $\mathtt{type}(r)$ можно «разделить» между уже созданными процессами.

Пример 3. К примеру, процесс в Linux не может переместиться из одной сессии в другую, уже существующую (т.е. сессия не $\mathtt{isSharable}(r)$). Но при этом внутри сессии, процесс может перейти в другую группу (разделить группу с каким-то другим процессом). Так же приватные маппинги (Private virtual memory area) не могут быть переданы между живыми процессами и должны передаваться между процессами с помощью наследования, в то время как файловые дескрипторы можно разделить.

2.2. Модель жизнендеятельности дерева процессов

2.2.1. Действия процессов

Во первых разделах этой главы мы дали неформальное определение задачи восстановления: поиск (и выполнение) набора действий, которые нужно исполнить для успешного воссоздания целевого дерева. Для формализации этой задачи мы должны понять, из какого множества нам вообще выбирать эти самые «действия».

Ресурсы Linux процесса различны, а значит и их восстановление выполняется по-разному. Т.к. ресурс так или иначе находится в ядре ОС, а из пространства пользователя мы получаем доступ к ядру через системные вызовы (так же существуют специальные файловые системы в духе `/proc`), то восстановление каждого ресурса — это последовательность из каких-то системных вызовов. Таким образом, множеством возможных действий, мы можем выбрать множество системных вызовов (как говорилось в разделе 1.5, такой подход — это перевес в сторону сложного генератора). Такой подход приведёт к ряду трудностей:

- Нет абстракции от деталей восстановления каждого ресурса в каждой его возможной конфигурации, из-за чего построение алгоритмов для восстановления зависимостей между ресурсами и процессами теряется в деталях и становится сложным
- Формализация алгоритмов восстановления становится очень трудной из-за огромного количества команд (действий), в них содержащихся
- Трудность выделить общий подход (независящий от конкретного типа ресурса) к процессу восстановления

Мы попытаемся избежать этих трудностей и внести конкретные детали восстановления каждого из типов ресурсов внутрь более общих, но более простых, с логической точки зрения, действий. Таким образом, в этом разделе мы введём множество команд, достаточно абстрактных, чтобы они могли быть применимы к восстановлению как можно более широкого числа ресурсов. Так же в этом разделе мы введём несколько дополнительных сущностей (свойств), которые помогут нам при генерации (поиске) последовательности действий для восстановления.

На самом деле то, как мы ввели понятие ресурса и дерева процессов, не позволяют нам вводить низкоуровневых команд, т.к. эти понятия были введены довольно общо (и не случайно).

Замечание 1. Мы говори про некий список команд, который должен быть исполнен. Кем же он должен исполняться? Т.к. нас интересует решение, работающее в пространстве пользователя, то исполнителями команд должны быть сами процессы (+ процессы помощники, которые не находятся в целевом дереве, но как-то могут помочь его восстановлению): других выходов, как я понимаю, нет.

На этом этапе мы, перед введением множества тех самых действий, можем формализовать задачу восстановления дерева процессов. Будем считать, что действия берутся из множества допустимых действий $\mathcal{A}$.

Определение 8. Задача восстановления дерева процессов $T = \{P_1, P_2, \ldots, P_k\}$ — это задача поиска упорядоченной и конечной последовательности действий $A = [a_1, a_2, \ldots, a_n]$, $\forall i : a_i \in \mathcal{A}$, такой, что:

$$\{P_0\} \xRightarrow{A} \{P_0\} \cup T$$

Тут символ $\xRightarrow{A}$ обозначает исполнение команд из последовательности A, а $\{P_0\}$ — это стартовое дерево процессов, которое необходимо в силу того, что кто-то должен начать исполнять команды (см. замечание 1)

Действия, которые далее мы будем вводить, так или иначе моделируют действия, изменяющие состояние дерева процессов Linux и представляют из себя модель жизнедеятельности процессов ОС.

2.2.2. Создание ресурса

Пускай целевое дерево процессов состоит из одного единственного процесса:

$$P = \{(r_1, h_1), \ldots, (r_n, h_n)\}$$

Каким способом процесс P мог завладеть одним из ресурсов r_i? Он мог создать его сам: открыть файл, сокет, пайп или создать маппинг. Любой ресурс так или иначе должен быть когда-то создан внутри дерева процессов.

Таким образом, первое действие, которое мы вводим — это создание ресурса:

Определение 9. Действие создания ресурса — $CreateAction\,(P, r, [h_1, h_2, \ldots, h_k])$ — процесс P создаёт ресурс r с интерфейсами h_i к нему, причём $\forall i, j : type(h_i) \neq type(h_j)$. Данное действие исполняется самим процессом P и имеет следующий эффект по отношению к процессу:

$$P \xrightarrow{CreateAction(P,r,[h_1,h_2,\ldots,h_k])} \{(r, h_1), \ldots, (r, h_k)\} \cup P$$

Замечание 2. Заметим, что в определении мы сделали так, что ресурс создаётся сразу с несколькими интерфейсами. Это связано с тем, что может существуют такие ресурсы, создание которых сопряжено с порождением нескольких интерфейсов к этим ресурсам, но разного рода. Например, pipe или socketpair — создание этих ресурсов происходит парно. Для ресурса pipe при создании, инициализируется 2 интерфейса: один, для чтения, а другой для записи.

Действия по созданию одной пары (r, h) мы будет записывать просто как: $CreateAction\,(P, r, h)$ (для большей части ресурсов этого достаточно).

Замечание 3. Важно понимать, что ресурс — это «структура» в ядре (см. определение 1), а значит действие создания ресурса инициирует создание некоторого объекта в ядре. Ядро — это глобальное хранилище ресурсов, а значит 2 разных действия создания ресурса всегда создают разные ресурсы в ядре:

$$\forall CreateAction\,(P_1, r_1, h_1)\,, CreateAction\,(P_2, r_2, h_2) \in \mathcal{A} : r_1 \neq r_2$$

Предположим, что $(r, h) \in P$, т.е. P ссылается на ресурс r. Мы знаем по ранее сказанному, что ресурсы могут быть разделены между процессами, но при решении задачи восстановления дерева процессов, мы не можем знать, в какой последовательности ресурсы передавались

друг между другом (вспомни, что есть ресурсы, которые можно «разделить» между живыми процессами, см. определение 7). Всё, что видим мы — это просто снимок состояния дерева. Мы должны как-то понять, какой ресурс каким процессом будет создаваться. Заметим, что если процесс ссылается на ресурс r, то это не значит, что он вообще был способен создать этот ресурс самостоятельно. Из-за этого вводим следующий формализм:

Определение 10. $possibleCreators\,(T, r)$ — это множество процессов $P \in T$ такое, что P способен создать ресурс r.

Такое действительно может произойти, например, с группами процессов:

$$P_1 = \{(group2, gid = 1)\}$$
$$P_2 = \{(group2, gid = 1)\}$$
$$P_3 = \{(group1, gid = 1)\}$$

Такое дерево вполне себе может существовать в Linux, но при это при его восстановлении нам будет необходимо в какой-то момент сделать так, чтобы процесс P_1 создал группу 1.

Так же мы будем использовать следующее обозначение:

Определение 11. $resourceHolders\,(T, r)$ — множество процессов $P \in T$ такое, что $r \in P$. Аналогично будем писать $resourceHolders\,(T, r, h) = \{P \in T : (r, h) \in P\}$

Пример 4. Положим, что мы восстанавливаем дерево T.

- r — ресурс, что $type(r) = RegularFile$, тогда $possibleCreators\,(T, r) = T$ (при условии, что у процесса достаточно прав для создания файла и файл доступен для процесса в файловой системе).
- Пусть $T = \{P_1, P_2, P_3\}$. Пускай $P_1.pid = 1$, $P_2.pid = 2$, $P_3.pid = 3$. Рассмотрим ресурс $r = pgroup_2$ (группа процессов с id = 2). Тогда верно следующее:
  - $possibleCreators\,(T, r) = \{P_2\}$. Вообще говоря, создание группы 2 может быть осуществлено ещё родителем P_2 (допустим это P_1): создание ресурса процессом P_1 будет заключаться в вызове `setpgid(2, 2)`, после чего `setpgid(0, 2)`. Но в этом случае мы изменяем состояние сразу нескольких процессов, что противоречит семантике $CreateAction\,(,,)$.
  - $possibleCreators\,(\{P_1\}, r) = \emptyset$, т.к. в одиночку процесс P_1 создать этот ресурс не

способен (ибо необходим процесс с идентификатором $\mathtt{pid} = 2$)

2.2.3. Создание процесса

Мы бы могли относиться к процессу как к ресурсу, но это больше вносит неоднозначности, нежели чем помогает обобщить процесс восстановления, т.к. всё равно мы логически разделяем ресурсы и процессы. Поэтому к действию создания ресурса добавляется действие создания процесса.

Определение 12. Действие создания процесса — $\mathrm{ForkAction}\,(P_1, P_2)$ — процесс P_1 создаёт потомка (процесс) P_2 с «интерфейсом» (идентификатором процесса) $\mathtt{pid}$. Действие имеет следующий эффект:

$$\{P_1\} \xRightarrow{\mathrm{ForkAction}(P_1,P_2)} \{(\text{child task}, \mathtt{pid}) \cup P_1, P_2\}$$

При этом процесс, который создаётся (P_2) не является пустым. Он может наследовать часть ресурсов своего родителя. Также у каждого процесса, кроме корневого, всегда есть родитель: $\mathtt{parent}(P)$.

2.2.4. Наследование ресурса при рождении

Выше, в определении 6 мы вводили понятие $\mathrm{isInherited}\,(r)$ — истина, если r наследуется процессом-ребёнком от родителя при создании. Наследуемые ресурсы должны быть учтены при построении последовательности действий:

После выполнения действия $\mathrm{ForkAction}\,(P_1, P_2)$ верно, что:

$$\forall (r, h) \in P_1 \wedge \mathrm{isInherited}\,(r) : (r, h) \in P_2. \tag{2.1}$$

Замечание 4. Мы тут считаем, что интерфейс к ресурсу наследуется и сохраняет своё значение.

Выше, в разделах 2.1.3 и 2.1.4 обсуждались ресурсы, которые должны разделяться наследованием (private mappings, сессии, ...). Мы будем считать, что те ресурсы, что обязательно должны разделяться наследованием не $\mathrm{isSharable}\,(r)$.

2.2.5. Разделение ресурса при жизни

Мы уже описали два действия, позволяющие процессам получать ресурсы: создание ресурса ($CreateAction(,,)$) и получение ресурса при рождении ($ForkAction(,)$). Достаточно ли этих команд для того, чтобы описать действиями процесс восстановления любого дерева процессов T?

Посмотрим на следующее дерево процессов:

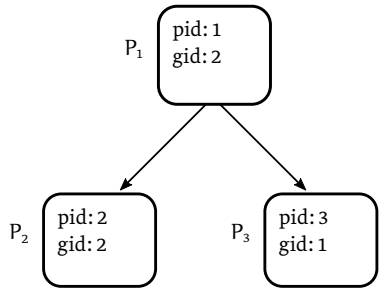


Рис. 2.1: Дерево T из 3 процессов

$$P_1 = \{(task_1,\ pid : 1), (pgroup_2,\ pgid : 2)\}$$
$$P_2 = \{(task_2,\ pid : 2), (pgroup_2,\ pgid : 2)\}$$
$$P_3 = \{(task_3,\ pid : 3), (pgroup_1,\ pgid : 1)\}$$

В силу специфики Linux, ресурс $pgroup_2$ не может быть создан процессом P_1, если процесса P_2 ещё не существует (см. пример под определением 10). Таким образом нам нужно сначала выполнить действие по созданию процесса P_2, после чего уже создавать ресурс $pgroup_2$. Но выполнение действия создания ресурса, по нашему определению, должно затрагивать лишь один процесс, а значит, что за выполнение одного $CreateAction(,,)$ ресурс $pgroup_2$ появляется только у одного из процессов (например P_2, после чего нам необходимо этот ресурс передать P_1.

Таким образом видно, что действий по созданию ресурса и процесса может быть недостаточно (глобальная причина этого заключается в том, что не каждый процесс может создать произвольный ресурс).

По этим причинам мы вводим ещё одно действие: сделка (разделение ресурса). Введение этого действия оправдывается не только проблемой, описанной выше, но так же и тем, что глупо использовать лишь часть возможностей в Linux по передаче ресурсов между процессами.

Определение 13. Действие разделения ресурса — $ShareAction(P_1, P_2, (r, h), h')$ — процесс P_1 разделяет ресурс (r, h) процессу P_2 (т.е. у P_2 до выполнения действия не было этого ресурса) так, что процесс P_2 получает доступ к r посредством *handle* h'. Перед выполнением действия

верно: $(r, h) \in P_1$. После выполнения действия верно: $(r, h') \in P_2$.

2.2.6. Зависимость между ресурсами

Одни ресурсы могут зависеть от других. В нашей модели, зависимости между ресурсами проявляются при создании этих ресурсов и этим мы ограничиваемся в данном подходе. Примеры зависимостей:

- Не приватная Virtual Memory Area зависит от того или иного файла, который необходим для создания этого маппинга (`mmap( ... )`)
- Так же, в силу того, что модель абстрактна, мы может вводить более тонкие с семантической точки зрения между ресурсами. К примеру, в Linux, при переходе в новую сессию s, процесс автоматически переходит в новую группу g и становится её лидером. Такой системный вызов мы можем смоделировать следующим образом: сказать, что ресурс g зависит от ресурса s, а значит и сессия s должна быть создана раньше, чем g (иначе мы просто не сможем выполнить создание новой сессии, т.к. процесс до этого станет лидером группы).

Зависимость между ресурсами влияет на порядок выполнения действий при восстановлении. Мы введём обозначение для множества ресурсов, от которых зависит ресурс.

Определение 14. $resourceDependencies\,(r)$ — множество пар ресурс и тип $handle$: $(r', handleType)$, от которых зависит ресурс r.

Тут возникает дополнительная сложность: вовлечения типа $handle$. В этой работе был избран подход, когда одно действие создания ресурса всегда создаёт лишь один ресурс в ядре, но с доступом к нему по нескольким различным интерфейсам. Из-за этого какой-то ресурс может зависеть именно от пары: ресурса и типа интерфейса к этому ресурсу. На данный момент этот подход обосновывается лишь причинами с точки зрения простоты реализации, но на самом деле подход с добавлением $CreateAction\,(,,)$, эффектом которого будет добавление нескольких пар разных ресурсов и $handle$, будет более гибким, но он не повлечёт никаких существенных изменений в асимптотике полученных алгоритмов.

2.2.7. «Удаление» ресурса

Может случаться так, что процесс из дерева, в целевой своей конфигурации, имеет ресурс r, т.е. $(r, h) \in P$. При этом $q \in resourceDependencies(r)$, но $(q, \_) \notin P$. Это значит, что в последовательности действий для восстановления должно фигурировать действие по удаления ресурса q из процесса, после того, как ресурс r был создан. Для того, чтобы обслуживать такую ситуацию, введём действие по удалению ресурса, а точнее пары (r, h).

Определение 15. Действие «удаления» ресурса — $RemoveAction(P, r, h)$ — процесс P удаляет «из себя» пару (r, h). Таким образом верно:

$$\{P_1 \cup \{(r, h)\}\} \xrightarrow{RemoveAction(P_1, r, h)} \{P_1\}$$

Замечание 5. Также, помимо обычных действий:

$$CreateAction(P, r, h), ShareAction(P_1, P_2, (r, h), h')$$

можно вводить их «временные» аналоги. Временное действие будет обозначать, что эффект этого действия должен быть устранён по окончании восстановления. Но мы будем использовать подход с $RemoveAction(P, r, h)$.

2.2.8. Конфликт ресурсов

В первой главе говорилось, что не каждое состояние дерева процессов может быть изменено так, чтобы получить целевое состояние. В том числе мы приводили пример (см. раздел 1.4), в котором `criu` завершается с ошибкой, гласящей о невозможности создания определённого ресурса (группы процессов) у конкретного процесса. Это значит, что состояние дерева процессов, в котором `criu` сейчас находится, конфликтует с ресурсом, который `criu` хочет создать следующим шагом. Таким образом требуемый к созданию ресурс конфликтует с каким-то ресурсом текущего состояния. Для того, чтобы обрабатывать конфликты в алгоритме построения программы действий, мы вводим понятие конфликтующих ресурсов:

Определение 16. $canExistTogether((r_1, h_1), (r_2, h_2))$ — предикат, возвращающий $True$, если (r_1, h_1) и (r_2, h_2) могут одновременно принадлежать одному и тому же процессу (в один момент времени).

Пример 5. Конфликтующие ресурсы

- Один процесс не может быть в двух сессиях или группах одновременно, а значит $\mathit{canExistTogether}(,)$ для таких ресурсов должен возвращать False
- Если r и r_1 — это ресурсы, на которые процессы указывают с помощью файлового дескриптора, то пары (r, h) и (r_1, h) — конфликтуют, т.к. процесс не может ссылаться по одному и тому же файловому дескриптору на два разных «файла»
- Процесс не может находиться в нескольких пространствах имён одного типа одновременно (видимых изнутри этого процесса)

2.3. Алгоритм построения команд восстановления

Выше мы ввели следующие команды:

- $\mathit{CreateAction}\,(P, r, h)$ (опр. 9)
- $\mathit{ForkAction}\,(P_1, P_2)\,\mathit{pid}$ (опр. 12)
- $\mathit{ShareAction}\,(P_1, P_2, (r, h), h')$ (опр. 13)
- $\mathit{RemoveAction}\,(P_1, r, h)$ (опр. 15)

Все эти команды (со всевозможными комбинациями параметров, соответственно) будут составлять множество всех команд $\mathcal{A}$.

Также мы ввели следующие вспомогательные свойства ресурсов и не только:

- $\mathit{isSharable}\,(r)$ (опр. 7)
- $\mathit{isInherited}\,(r)$ (опр. 6)
- $\mathit{resourceDependencies}\,(r, h)$ (опр. 14)
- $\mathit{possibleCreators}\,(T, r)$ (опр. 10)
- $\mathit{canExistTogether}\,((r, h), (r', h'))$ (опр. 16)

Заметим, что реализация этих предикатов и множеств зависит от внутренней специфики каждого ресурса. Алгоритм, который рассматривается этой главе, не раскрывает как должны строиться данные функции, он предполагает, что мы их имеем и можем использовать.

Теперь наша задача в том, чтобы используя всё это построить решение задачи восстановления, т.е. построить список команд из $\mathcal{A}$ для восстановления.

На входе мы имеем дерево процессов $T = \{P_1, P_2, \ldots, P_n\}$, где каждый из процессов

$$P_i = \{(r_1, h_1), (r_2, h_2), \ldots, (r_{n_i}, h_{n_i})\}$$

Основная идея алгоритма построения последовательности действий: построить промежуточное представление в виде графа, вершины которого — это действия для исполнения, а ориентированные рёбра означают отношение предшествования и задают строгий частичный порядок на вершинах-действиях: $a_1 < a_2 \implies$ действие a_1 должно быть выполнено раньше действия a_2.

Каждый раз, когда происходит исполнение очередного действия, изменяющее состояние процесса, всё дерево процессов находится в некотором состоянии (какие процессы в нём есть, какие ресурсы есть у каждого процесса). Такое состояние дерева — это *контекст исполнения* C действия. Как мы убедились (см. пример 1, раздел 2.2.6, ...) контекст определяет то, может ли действие вообще быть выполнено или нет. Например для выполнения действия удаления ресурса (r, h) процессом P контекст C должен содержать процесс P, и процесс P в контексте C должен содержать ресурс (r, h).

В предлагаемом в данной работе алгоритме мы совершаем следующие шаги:

(a) Выделяем создателей P каждого ресурса r (раздел 2.4)

(b) Имея исходное дерево T из процессов P_i, доводим его состояние до суммарного контекста, который является достаточным для инициализации любого ресурса (по сути строим объединение контекстов)

- Этот этап мы называем *замыканием*
- Суммарный контекст может быть некорректным с точки зрения ОС Linux, т.к. могут возникать конфликты ресурсов из-за того, что строится именно объединение всех контекстов. Разрешение конфликтов производится на одном из последних шагов алгоритма
- Замыкания описываются в разделе 2.5

(c) Выделяем множество действия A из множества всевозможных действий $\mathcal{A}$, которые должны быть совершены в некотором порядке (пока неизвестно в каком) для восстановления

целевого дерева (раздел 2.6)

(d) Инициализируем граф действий G вершинами из A

(e) Строим рёбра графа G вводя отношение предшествования на действиях из A (раздел 2.7)

(f) Топологически сортируем граф G (раздел 2.8)

Сразу заметим, что в листингах алгоритмов будут использоваться те же обозначения, что и описаны выше, но в другом начертании: моноширинным шрифтом

$$resourceHolders\,(T, r) \rightarrow \texttt{resourceHolders(T, r)}$$

Так же операции над множествами имеют следующие обозначение: $\cap$ = `&`, $\cup$ = `|`, $\setminus$ = `-`, $|S|$ = `len(S)`, $\in$= `in`, $\notin$= `not in`.

Так же введём следующие краткие обозначения:

- N — количество процессов в дереве T
- R — суммарное количество ресурсов r (не пар (r, h), а именно различных r) во всём дереве
- M — максимальное число $handle$ ссылающихся на один и тот же ресурс (например, число файловых дескрипторов, которые указывают на один и тот же файл) внутри одного процесса
- H — максимальное число $handle$, требуемых созданием какого-либо ресурса (см. определение 9)

2.4. Выделение создателей ресурсов

Каждый ресурс должен быть кем-то создан. В нашем алгоритме мы вводим функцию, которая для каждого ресурса возвращает процесс, который создаёт переданный ресурс. Такая функция действует исходя из естественных правил на основе введённых выше свойств ресурсов:

Функция (см. 1) принимает на вход дерево процессов и ресурс. И ищет в этом дереве процессов нужный процесс-создатель. Заметим, что данная процедура может возвращать процессы, которые в текущем состоянии переданного дерева не имеют ссылки на ресурс, который был передан аргументом. Это будет значить, что ресурс должен быть создан временно.

```python
def resourceCreator(T, r):
    holders =  resourceHolders(T, r)
    allPossibleCreators = possibleCreators(T, r)

    if isSharable(r):
        creatorCandidates = allPossibleCreators & holders
        if len(creatorCandidates) > 0:
            # ближайший к корню процесс
            return top(creatorCandidates)
        else:
            # процесс создатель не имеет ссылки на ресурс
            return top(allPossibleCreators)

    if isInherited(r):
       lca = lca(holders)  # least common ancestor всех держателей ресурса
       while lca not in allPossibleCreators:
           lca = parent(lca)
       return lca

    return next(creatorCandidates)  # приватный ресурс ⇒ один возможный создатель
```

Листинг 1: Поиск процесса-создателя ресурса

Ресурс, который распространяется по дереву процессов с помощью наследования, обязан быть создан тем процессом, поддерево с корнем в котором содержит всех держателей этого ресурса. Именно поэтому при поиске процесса-создателя такого ресурса используется алгоритм поиска наименьшего общего предка (least common ancestor) нескольких процессов в дереве.

Алгоритмическая сложность операции

Самая трудоёмкая часть алгоритма — это вычисление наименьшего общего предка множества вершин `holders` в дереве T (пересечение множеств `allPossibleCreators & holders` выполнимо за $\mathcal{O}(N)$). Размер дерева — N (число процессов). Максимально возможное число «держателей» так же равно N. Операция $\mathrm{lca}(v, u)$ для двух вершин может быть реализована за $\mathcal{O}(\log N)$ [1], а значит сложность всей операции можно оценить в:

$$\mathcal{O}(N \log N) \tag{2.2}$$

Т.к. lca для множества вершин может быть выполнен просто последовательно за N вычислений попарного lca.

Имеет смысл проиндексировать создателей для всех ресурсов заранее, перед переходом к вы-

[1] С предподсчётом за линейное от размера дерева время, т.е. за $\mathcal{O}(N)$

полнению основной части алгоритма.

2.5. Замыкания исходного состояния дерева процессов

2.5.1. Замыкание ресурсов относительно зависимостей

В разделе 2.2.6 мы вводили понятие зависимых ресурсов. Зависимость в нашей модели учитывается при создании ресурса (по сути всё модель и описывает механизмы создания ресурсов). У ресурса r есть создатель $P \in T$, т.е. процесс P в какой-то момент времени должен создать ресурс r. Но ресурс r может зависеть от других ресурсов, а значит, на момент создания его процесс P должен иметь в распоряжении и зависимости ресурса r: $resourceDependencies\,(r)$.

На этапе замыкания относительно зависимостей мы добавляем процессам-создателям ресурсов ресурсы-зависимости, если эти процессы на них ещё не ссылаются. Для примера: процесс P может выступать создателем ресурса — приватного файлового маппинга, а значит до создания этого маппинга процесс должен получить доступ к файлу. Но целевой процесс может не сохранять открытый файл, а закрыть его, продолжая при этом пользоваться маппингом. Таким образом на данном шаге мы добавляем «временные» ресурсы. Процедура описана в листинге 2.

После выполнения этой операции замыкания дерева T верно следующее:

$$\forall P \in T, r \in P \wedge P = resourceCreator(T, r), q \in resourceDependencies\,(r)\,(q \in P)$$

```
def closeAgainstDependencies(T):
    for r in T: # итерация по всем ресурсам дерева
        P = resourceCreator(T, r)
        resourcesToCheck = resourceDependencies(r)

        for rDep, handleType in resourcesToCheck:
            if hasHandleOfType(P, rDep, handleType):
                continue

            addResourceWithHandleOfType(P, rDep, handleType)
```

Листинг 2: Замыкание процессов относительно зависимостей между ресурсами

Данная процедура может добавлять новые ресурсы к процессам-создателям других ресурсов. Эти ресурсы мы будем обозначать как $temporary$ (временные), т.к. эти ресурсы должны быть удалены после того, как процесс в них больше не нуждается, до завершения процедуры восстановления.

Заметим, что мы использовали в листинге следующие функции:

- `hasHandleOfType(P, r, type)` — данная функция просто проверяет, существует ли в процессе P пара (r, h), где $type(h) = type$
- `addResourceWithHandleOfType(P, r, type)` — это более хитрая функция, которая предполагает автоматический подбор $handle$ нужного типа. Это раскрывает дополнительное свойство нашей модели: $handle$ могут выбираться из некоторого множества, например, если $type(h) =$ `file descriptor`, то автоматический подбор такого $handle$ подразумевает выбор свободного файлового дескриптора. В программной реализации для каждого типа $handle$ мы заводим свою фабрику, которая умеет отвечать на вопрос: занят ли $handle$ и выдавать следующий свободный $handle$.

Замечание 6. Во время операции замыкания мы добавляем новые ресурсы к процессу. В рамках нашей модели это абсолютно допустимая операция, но всегда ли такое допустимо в настоящем Linux процессе? Думаю, что нет. Проблема в том, что когда мы имеем исходное дерево процессов $T = \{P_1, P_2, \ldots, P_k\}$, то мы можем считать, что оно верно, с точки зрения допустимости конфигураций ресурсов каждого из процессов P_i в Linux, т.к. это дерево мы берём с реального снимка (dump) реального дерева. Но как только мы начинаем искусственно добавлять новые ресурсы, мы уже не можем быть уверены в том, что получаемые конфигурации ресурсов могут реально существовать. Простой пример: допусти, процесс держит «ресурс» — `VMA(start=0, end=10)` (virtual memory area), а другой его ресурс зависит от `VMA(start=0, end=5)`. Но два таких маппинга не могут одновременно присутствовать в процессе (см 2.2.8).

Алгоритмическая сложность

Сложность процедуры `closeAgainstDependencies`: $\mathcal{O}(N \cdot R \cdot H)$ (см. обозначения в конце вступления в разделе 2.3).

2.5.2. Замыкание ресурсов относительно наследования

Рассмотрим следующую гипотетическую ситуацию: ресурс r разделяется несколькими процессами, при этом $isSharable(r) = false$, а значит ресурс r должен был разделяться между процессами наследованием и никак иначе; при этом, процессы, ссылающиеся на ресурс не образуют дерево (как на рисунке 2.2).

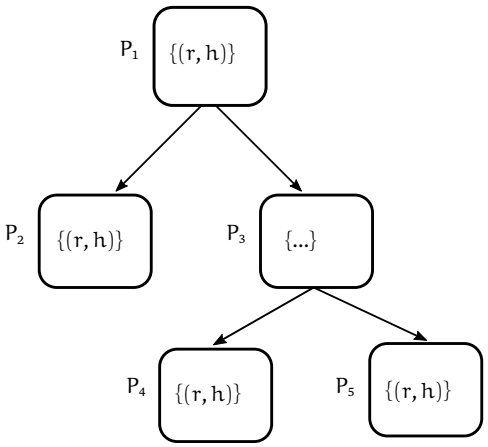


Рис. 2.2: Разделяемый наследованием ресурс, удалённый в нелистовом процессе

Такая ситуация говорит нам о том, что процесс P_3 в какой-то момент, после создания детей, избавился от ресурса (r, h), но этот ресурс когда-то присутствовал в P_3. Для простоты основной части алгоритма мы так же производим замыкание контекста в этой ситуации: добавляем ресурсы к таким процессам. Такой ресурс будет временным для процесса P_3.

Опишем процедуру последовательностью шагов:

(a) Взять очередной ресурс $r \in T$, что $\mathtt{isInherited}(r) = \mathtt{True} \wedge \mathtt{isSharable}(r) = \mathtt{False}$, т.е. ресурс должен разделяться исключительно наследованием

(b) Найти всех держателей ($\mathtt{holders}$) ресурса r ($\mathcal{O}(N)$)

(c) Найти создателя $\mathtt{creator}$ ресурса r с помощью процедуры $\mathtt{resourceCreator}$, описанной выше ($\mathcal{O}(1)$, если была произведена индексация)

(d) Найти корни деревьев из леса $\mathtt{holders}$: $\mathtt{roots}$ (поиск всех вершин в множестве, родитель которых не принадлежит этому множеству, реализуемо за $\mathcal{O}(N)$)

(e) Начать подъём от каждой вершины из $\mathtt{roots}$, попутно добавляя ресурс r к вершинам, по которым проходим, пока не встретим вершину, у которой этот ресурс уже есть (или не достигнем $\mathtt{creator}$). В каждой вершине дерева за эту процедуру мы зайдём лишь единожды, а значит $\mathcal{O}(N)$

Алгоритмическая сложность всей процедуры: $\mathcal{O}(N \cdot R)$

Заметим, что тут мы видим аналогичную проблему, описанную в замечании 6. Мы будем с этим бороться аналогично, на этапе генерации последовательности команд, используя предикат $\mathtt{canExistTogether}(,)$.

Замечание 7. Мы помним, что часть ресурсов у нас при создании процесса-ребёнка насле-

дуются. Может случиться так, что один из наследованных ресурсов r не должен находиться в целевом процессе-ребёнке, а значит он должен быть удалён, причём до создания других ресурсов (желательно), ведь они могут конфликтовать. Такие удаления ($RemoveAction(,,)$) мы добавим в самом конце нашей процедуры построения списка команд в нужное место (сразу после $ForkAction(,)$ процесса-ребёнка) В рамках модели эти ресурсы нам никак не мешают.

2.5.3. Добавление ресурса к процессу-создателю этого ресурса

Процесс, который был выбран создателем ресурса, в нашей модели, не гарантировано является держателем этого ресурса, а поэтому на этом простом шаге мы просто обходим все ресурсы и в случае, если создатель P ресурса r не ссылается на него, дополняем P этим ресурсом. Такой ресурс будет временным ($temporary$).

2.5.4. Замыкание относительно ресурсов с несколькими *handle*

В силу того, как мы ввели действие $CreateAction(P, r, [h_1, h_2, \ldots, h_k])$, процесс создатель ресурса r должен обязательно владеть всеми парами $(r, h_1), (r, h_2), \ldots$, иначе какая-то часть ресурса просто может потеряться. К примеру рассмотрим системный вызов `pipe`. Процесс, однажды вызвав его и создав дескрипторы для доступа к каналу для чтения и для записи, мог поделиться одним из таких дескрипторов с другим процессом, после чего второй дескриптор он мог закрыть. Если мы в нашей модели не дополним процесс P недостающими «частями» ресурса, то мы не сможем учесть, что P когда-то владел этой частью ресурса, а значит недостающие части мы также должны добавить.

```
def closeMultiHandleResources(T):
    for r in T: # итерация по всем ресурсам дерева
        P = resourceCreator(T, r)

        # возвращает список типов, с которыми ресурс r создаётся
        handleTypes = getResourceCreationHandleTypes(r)

        for type in handleTypes:
            if hasHandleOfType(P, r, handleType):
                continue

            addResourceWithHandleOfType(P, r, type)
```

Листинг 3: Добавление недостающих частей ресурса

Алгоритмическая сложность данной процедуры: $\mathcal{O}(N \cdot H)$ с учётом того, что $resourceCreator$

предподсчитан заранее.

2.5.5. Добавление вспомогательного корневого процесса

Для того, чтобы у всех процессов целевого дерева процессов существовал родитель, ответственный за его создание, мы добавляем вспомогательный процесс P_0, который будет ответственен за создание корня дерева T: $parent(root(T)) = P_0$.

Тут нужно подвести итог. Исходно мы имели дерево

$$T = \{P_1, \ldots, P_n\}$$

являющееся допустимым снимком дерева процессов Linux, но после описанных операций замыкания, мы имеем дерево:

$$T' = \{P'_0, P'_1, \ldots, P'_n\}$$

такое, что:

- $P'_i \cap P_i = P_i$
- $P_0 = root(T') = parent(root(T))$
- Для любого ресурса $r \in T'$ верно, что $resourceCreator(T', r) \cap resourceHolders\,(T', r) \neq \emptyset$
- $\forall P' \in T', r \in P', P' = resourceCreator(T', r)$ верно, что $resourceDependencies\,(r) \subset P'$
- Для любого ресурса $r \in T'$ такого, что r разделяется только наследованием, т.е. $isInherited\,(r) = true \wedge isSharable\,(r) = false$ верно, что $resourceHolders\,(T', r)$ образуют дерево (не лес!) относительно отношения родитель-ребёнок

Мы будем обозначать: $P_i \cap P'_i = Tmp_i$ или писать $isTemporary(P, r, h)$, $isTemporary(P, r)$, чтобы проверить, является ли ресурс r временным (есть ли хоть одна временная пара (r, h))

2.6. Генерация множества действий

Вначале мы сгенерируем множество действий A.

- $ForkAction(,)$: действие должно присутствовать в A для каждого процесса из дерева. См. листинг 4.

```
def get_fork_actions(T):
    acts = set()
    for P in T:
        if P == root(T):
            continue
        acts.add(ForkAction(parent(P), P))
    return acts
```

Листинг 4: Добавление действий $ForkAction(parent(P), P)$ для всех $P \in T \setminus \{P_0\}$

- $CreateAction(,,)$: действие должно быть добавлено для каждого ресурса один раз, создавать у нас ресурсы будут процессы, как мы определили в разделе 2.4. См. листинг 5.

```
def get_create_actions(T):
    acts = set()
    for r in T:
        creator = resourceCreator(T, r)
        hs = getCreatorHandles(creator, r)   # возвращает массив handle'ов
                                             # нужных для создания ресурса
                                             # Этот массив закреплён
        acts.add(CreateAction(creator, r, hs))
    return acts
```

Листинг 5: Добавление действий создания ресурсов

- $RemoveAction(,,)$: Данное действие должно быть добавлено для каждого временного ресурса (пары) (r, h) в каждом процессе ($isTemporary(P, r, h) = true$). В отличие от действия создания ресурса данное действие может выполняться несколько раз для одного и того же ресурса, но с разными handle. Генерация показана в листинге 6. Также данное действие должно быть добавлено для всех наследуемых ресурсов, но т.к. на данном этапе мы не знаем конкретной последовательности действий, то мы не можем сказать, какие конкретно ресурсы будут наследоваться, поэтому выполним это последним шагом.

```
def get_remove_tmp_acts(T):
    acts = set()
    for P in T:  # все процесса в T
        for (r, h) in P:
            if isTemporary(P, r, h):
                acts.add(RemoveAction(P, r, h))
    return acts
```

Листинг 6: Добавление действий удаления временных ресурсов

- $ShareAction(,,,)$: мы считаем, что если ресурс поддерживает возможность разделения его при жизни (как альтернатива наследованию), то нужно её использовать. Это можно

объяснить на следующим образом: любой ресурс, который разделяется наследованием, должен быть проинициализирован процессом до того, как создавать детей-наследников. Такого ограничения для разделяемых при жизни ресурсов нет, что делает налагающими меньше ограничения ресурсами. Поэтому для каждого ресурса r, который $isSharable(r) = true$, мы будем генерировать действия разделения этого ресурса нуждающимся. См. листинг 7.

```
def get_share_acts(T):
    acts = set()
    for r in T:
        if not isSharable(r):
            continue

        creator = resourceCreator(T, r)
        hs = getCreatorHandles(creator, r)
        hsMap = {type(h): h for h in hs}  # из типа handle в handle
        holders = resourceHolders(T, r)

        for P in holders:
            handles = getResourceHandles(P, r)

            if P == creator:
                # не нужно создавать ShareAction для создателя для разделения
                # ресурса по одному и тому же handle
                handles = (h for h in handles if h not in hs)

            for h in handles:
                 acts.add(ShareAction(creator, P, (r, hsMap[type(h)]), h))
```

Листинг 7: Генерация действий разделения ресурсов

Генерация действия разделения — самая трудоёмкая операция. Её сложность: $\mathcal{O}(R \cdot N \cdot M)$. Данная сложность — верхняя оценка для всей процедуры генерации множества действий (и, как следствие, на число вершин в этом графе).

2.7. Отношение предшествования над действиями

Мы имеем мешок действий, которые должны быть выполнены. Их нужно теперь как-то упорядочить. Простота и атомарность выбранных действий позволяет нам задать частичный строгий порядок, на основе которого потом построить граф и, если получится, линеаризовать его.

В этой главе описываются отношения предшествования, которые должны быть построены.

Нам понадобится несколько дополнительных понятий:

- $actsInvolvingP(P)$ — множество действий, либо исполняющихся процессом P либо использующие его:

$$CreateAction(P,,)\ RemoveAction(P,,)\ ForkAction(P,)$$
$$ShareAction(P,,,)\ ShareAction(,P,,)$$

- $obtainAction(P, r, h)$ — действие, результатом которого стало получение процессом P пары (r, h) (в том числе из-за наследования):

$$CreateAction(P, r, [\ldots, h, \ldots]), ShareAction(, P, (r, \_), h), ForkAction(, P)$$

- $actsWithRes(P, r, h)$ — действия, исполняемые процессом P, которые так или иначе используют уже созданный ресурс (r, h):

$$RemoveAction(P, r, h), ShareAction(P, \_, (r, h), \_)$$

Все отношения строятся напрямую так, как будут описаны, поэтому мы не будет приводить псевдокод их построения.

(a) Процесс P должен быть создан прежде, чем будет выполнено какое-либо действие из $actsInvolvingP(P)$, т.е.:

$$\forall forkAct = ForkAction(\_, P) \in A, a \in actsInvolvingP(P)\ (forkAct < a)$$

(b) Любой ресурс (r, h) должен быть получен процессом P прежде, чем процесс P совершит над ним какое-либо действие:

$$\forall (r, h) \in P, obtainAct = obtainAction(P, r, h), a \in actsWithRes(P, r, h)$$
$$(obtainAct < a)$$

(c) Любой временный ресурс (r, h) должен быть удалён лишь после того, как будет исполь-

зован:

$$\forall(r,h) \in P, isTemporary(P,r,h) = True, a \in actsWithRes(P,r,h)$$
$$(a < RemoveAction\,(P,r,h))$$

(d) Разделяемый наследованием ресурс (r,h) должен быть созданы до того, как будет произведён Fork процессов-детей, которые так же содержат ресурс (r,h)

$$\forall(r,h) \in P, isInherited(r) \wedge isSharable(r) = false, P = parent(P'), (r,h) \in P'$$
$$(obtainAction(P,r,h) < ForkAction\,(P,P'))$$

Выше мы написали $obtainAction(P,r,h) < ForkAction\,(P,P')$ хотя на самом деле сказать, что лишь создатель ресурса (r,h) должен создать $CreateAction\,(,r,h)$ до создания детей, которые разделяют этот ресурс.

(e) Временный разделяемый наследованием ресурс (r,h) должен быть удалён после того, как будет произведён Fork процессов-детей, которые так же содержат ресурс (r,h)

$$\forall(r,h) \in P, isInherited(r) \wedge isSharable(r) = false,$$
$$isTemporary(P,r,h), P = parent(P'), (r,h) \in P'$$
$$(ForkAction\,(P,P') < RemoveAction\,(P,r,h))$$

(f) Ресурс (r,h), от которого зависит создание ресурса (r',h') процессом P, должен быть получен этим процессом до того, как (r',h') будет создан

$$\forall(r,h) \in P, P = resourceCreator(T,r), r' \in resourceDependencies\,(r)\,,$$
$$(obtainAction(P,r',h') < CreateAction\,(P,r,[\ldots,h,\ldots]))$$

(g) Действия над ресурсами, которые конфликтуют, должны быть упорядоченны специальным образом. Этот пункт мы выносим в отдельный подраздел 2.7.1

Заметим сразу, что верхняя оценка на число вершин в графе действий (т.е. количество действий) — это $\mathcal{O}\,(N \cdot R \cdot M)$ (совпадает с оценкой сложности построения всех действий). Это значит, что верхней оценкой для числа рёбер является: $\mathcal{O}\,(N^2R^2M^2)$. На самом деле данная оценка очень грубая, т.к. наш алгоритм построения действий таков, что многие действия не будут со-

единены между собой, более того: большая часть рёбер строится из естественных наблюдений за процессами в Linux, а поэтому получаемый граф будет в большей части случаев близок к дереву, то есть число рёбер можно оценить примерно в $\mathcal{O}(N \cdot R \cdot M)$.

2.7.1. Упорядочивание действий над конфликтующими ресурсами

Также нам нужно разобраться с действиями, затрагивающими ресурсы, которые могут конфликтовать
$(canExistTogether(,) = false)$.

Пускай два ресурса (r, h) и (r', h') в рамках одного процесса P'_i конфликтуют. Не умаляя общности мы можем считать, что $(r, h) \in Tmp_i$ (т.к. только «временные» ресурсы могут вызывать конфликты, см. замечание 6). В этом случае нам нужно упорядочить работу с этими ресурсами так, что все действия над (r, h) (включая создание и удаление) происходят раньше, чем все действия над (r', h').

Очевидно, что нас интересует выполнить действия, связанные с временным ресурсом, который находится в конфликте с постоянным, до действий с постоянным ресурсом.

Также заметим следующее: пусть ресурс (r', h') — это ресурс, который процесс получает посредством наследования. Тогда этот ресурс уже существует сразу после создания процесса P'_i, а значит у нас уже не получится сделать так, что действия с (r, h) будут выполнены раньше, дабы не вступить в конфликт с (r', h'). В этом случае нам необходимо наоборот, поставить работу над (r', h') перед работой над (r, h).

Для того, чтобы обработать зависимости в соответствии с этими соображении, мы упорядочим все ресурсы в рамках каждого процесса следующим образом:

$$\Big[\overbrace{\underbrace{(r_1, h_1), (r_2, h_2), (r_3, h_3), \ldots}_{\text{полученные наследованием}}, \underbrace{(r_k, h_k), (r_{k+1}, h_{k+1}), \ldots}_{\text{полученные созданием или share}}}^{\text{временные ресурсы}}, \underbrace{(r_l, h_l), (r_{l+1}, h_{l+1}), \ldots, (r_n, h_n)}_{\text{постоянные (целевые) ресурсы}}\Big]$$

А действия над ”конфликтующими” ресурсами упорядочивать теперь можно следующим образом (в рамках одного процесса), как показано в листинге 8.

```
def handleConflictingResources(P):
    # сортируем ресурсы, как было показано выше
    sortedResources = sortResources(P)
    # tmp - "голова" из временных ресурсов с индексами (до первого не временного)
    tmp = getTmpHead(sortedResources)

    # перебираем все временные ресурсы
    for idx, (r1, h1) in enumerate(tmp):
        # перебираем все ресурсы, левее (r1, h1)
        for (r2, h2) in sortedResources[idx+1:]:
            if canExistTogether((r1, h1), (r2, h2)):
                continue

            addPrecedenceEdge(
                efrom = RemoveAction(P, r1, h1),
                eto   = obtainAction(P, r2, h2)
            )
```

Листинг 8: Разрешение конфликтов

2.8. Топологическая сортировка графа действий

В предыдущем пункте мы по сути построили набор вершин и рёбер между ними. Вершины — это действия, а рёбра между ними — это «рёбра» предшествования. Граф из этого получается естественным образом. Будем обозначать этот граф символом G и называть *графом действий*.

Заключительный этап решения задачи заключается в топологической сортировке этого графа и получения упорядоченного списка действий $[a_1, a_2, \ldots, a_N]$. Но мы помним, что данный список действий ещё не включает в себя части действий удаления тех ресурсов, которые были созданы предком до создания ребёнка, но при этом ребёнком не используются. Т.е. теперь нам нужно сэмулировать выполнение действий из списка и вставить в нужные места $RemoveAction\,(P, r, h)$ так, чтобы закрыть лишние ресурсы.

Топологическая сортировка графа выполняется за количество действий, пропорциональное размеру (числу вершин) этого графа (один обход в глубину), а поэтому время работы данного шага: $\mathcal{O}\,(N \cdot R \cdot M)$ (см. раздел 2.6). А это значит, что среднее время работы всего алгоритма так же равно $\mathcal{O}\,(N \cdot R \cdot M)$, а если же брать верхнюю очень грубую оценку, то получим $\mathcal{O}\,(N^2R^2M^2)$ (см. конец раздела 2.7).

Вопрос, на который нам остаётся ответить: действительно ли полученный таким способом граф будет ациклическим для того, чтобы мы всегда могли построить программу для восстановления?

2.9. Примеры и гарантии

2.9.1. Простой пример

В качестве простого примера рассмотрим дерево из одного процесса, изображённое слева на рисунке 2.3.

Process(pid=2324, ppid=0)	
Group(2324)	
Session(2324)	
RegFile(1, 'simple') *at* 6	tmp
RegFile(2, 'libc-2.23.so') *at* 4	tmp
RegFile(3, 'ld-2.23.so') *at* 5	tmp
RegFile(4, 'null') *at* 0	
RegFile(4, 'null') *at* 1	
RegFile(4, 'null') *at* 2	
RegFile(5, 'log.txt') *at* 3	
Internals(VmInfo)	
Internals(FSProps)	
Internals(ProcessCore)	
Internals(ThreadCore)	
Internals(PageMap)	
VMA(140285946966016-140285948796928, 2)	
VMA(140285948796928-140285950894080, 2)	
VMA(140285950894080-140285950910464, 2)	
VMA(140285950910464-140285950918656, 2)	
VMA(140285950918656-140285950935040, 0)	
VMA(140285950935040-140285951090688, 3)	
VMA(140285953040384-140285953052672, 0)	
VMA(140285953175552-140285953183744, 0)	
VMA(140285953183744-140285953187840, 3)	
VMA(140285953187840-140285953191936, 3)	
VMA(140285953191936-140285953196032, 0)	
VMA(140725443121152-140725443260416, 0)	
VMA(140725443649536-140725443657728, 0)	
VMA(140725443657728-140725443665920, 0)	
VMA(18446744073699065856-18446744073699069952, 0)	
VMA(4194304-4202496, 1)	
VMA(6295552-6299648, 1)	
VMA(6299648-6303744, 1)	

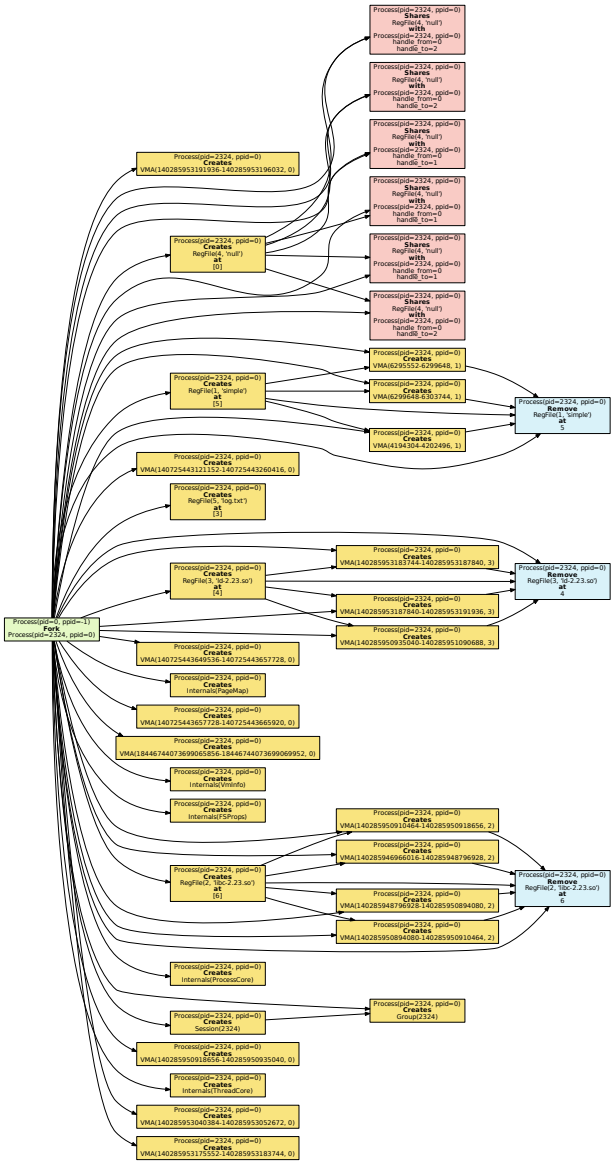


Рис. 2.3: Дерево из одного процесса и граф действий восстановления для него

Граф действий с выстроенными рёбрами будет выглядеть для него так, как изображено справа на рисунке 2.3. Граф получается довольно большим, но структура этого графа проста и древесна. В полученном дереве лишь три уровня.

2.9.2. Пример с группами

Рассмотрим следующее дерево процессов, изображённую на рисунке 2.4. Это дерево идентично тому, которое использовалось в примере 1. И `criu` возвращает ошибку при попытке восстановить дерево из таких процессов.

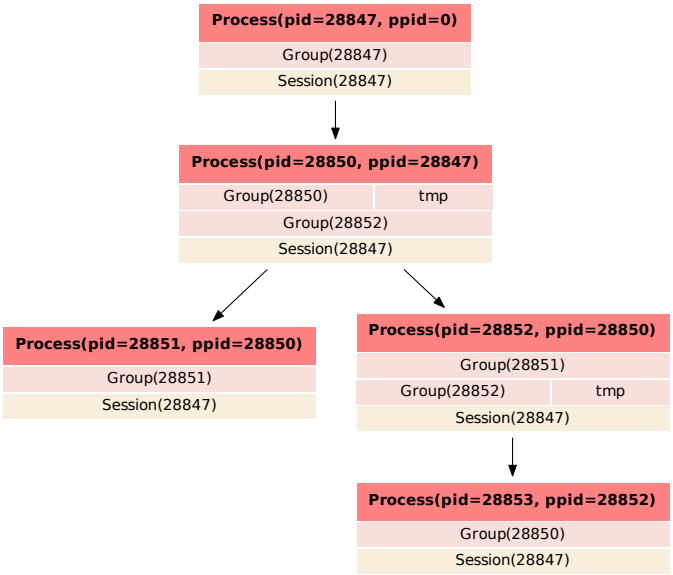


Рис. 2.4: Дерево для примера с группами

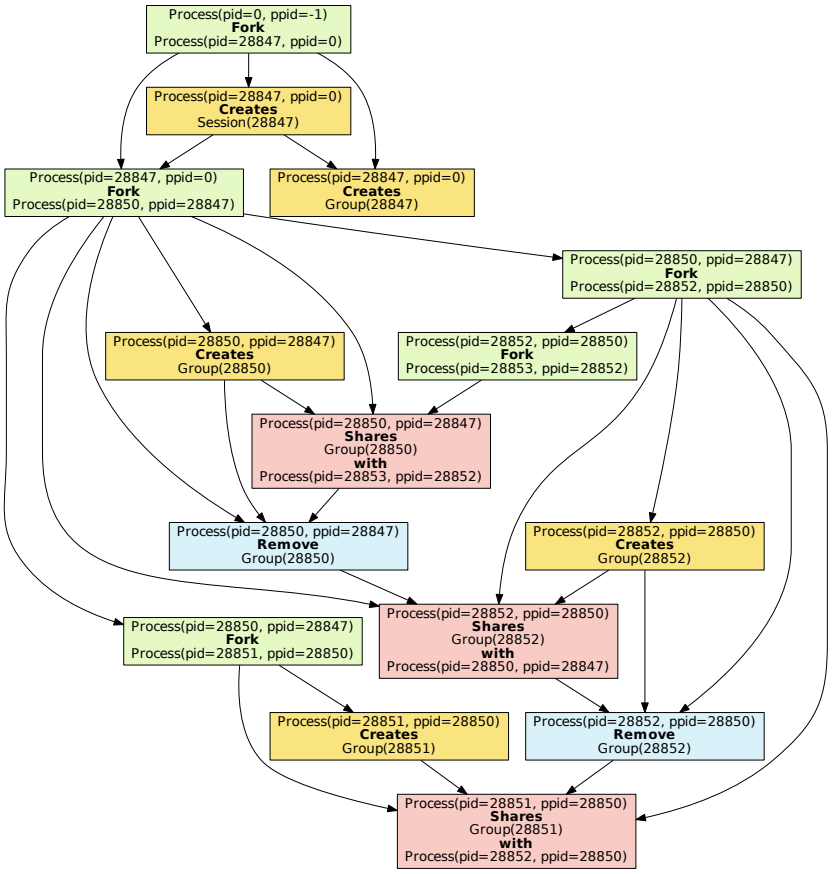
После построения графа действий мы получаем граф, изображённый на рисунке 2.5a. Предложенный алгоритм строит ациклический граф и возвращает последовательность действий, изображённую на рисунке 2.5b.

Полученная последовательность действий и граф хорошо показывают, что полученный алгоритм может разрешать довольно нетривиальные зависимости между ресурсами.

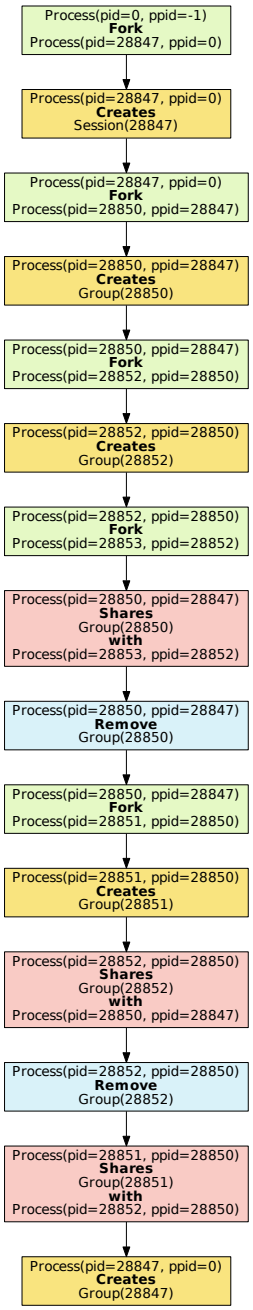
2.9.3. Цикличность графа действий

Предложенный в этой работе подход является достаточно гибким для описания огромного числа зависимостей между ресурсами и конфигураций процессов, но пока что он не способен восстановить абсолютно любое дерево процессов. Это связано с тем, что граф действий не всегда будет ациклическим.

Рассмотрим следующий пример дерева процессов, изображённый на рисунке 2.6. В этом дереве мы видим два процесса, которые обменялись группами (третий процесс, с ppid=0 является



(a) Граф действий для примера



(b) Упорядоченная последовательность действий для примера с группами

лидером всех сессии): 21650 и 21651. Поскольку текущий подход пытается создавать команды создания и разделения ресурсов, которые исполняются только теми процессами, которые присутствуют в дереве для восстановления, то данное дерево получается невозможным для восстановления, т.к. с одной стороны процесс 21650 должен создать группу 21650 и поделиться ею с 21651, причём так, что этот обмен будет последним действием, вовлекающим процесс 21651 в изменение группы (т.к. это конечная целевая группа для этого процесса). Но при этом то же самое должно произойти и со стороны процесса 21651. Эта ситуация наглядно проявляется в

графе действий, который, вместе с возникающим циклом, изображён на рисунке

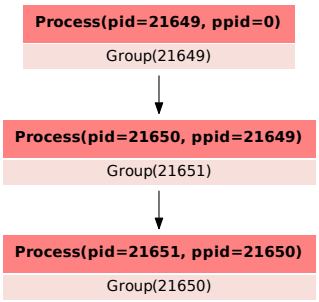
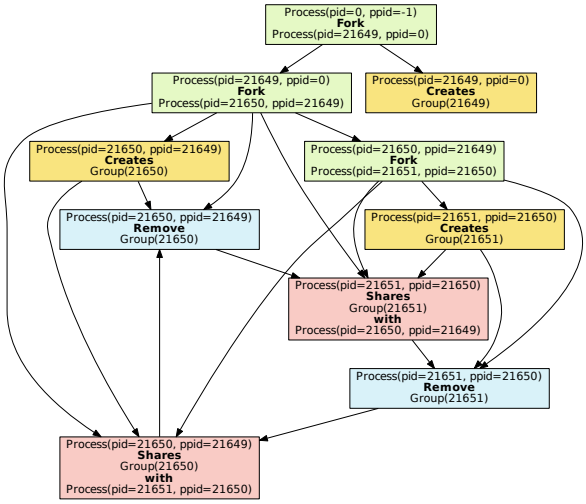
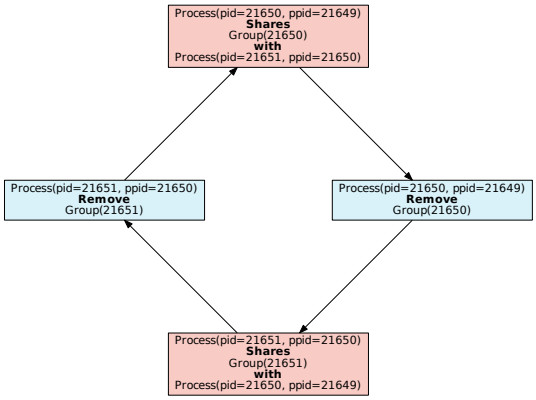


Рис. 2.6: Дерево, которое не восстановить в рамках текущей модели



(a) Граф действий для примера с обменом конфликтующими



(b) Возникший цикл на действиях

Такие циклы будут возникать каждый раз, когда в графе действий возникает ситуация циклического обмена конфликтующими между собой ресурсами. Одним из возможных способов решения этой проблемы является добавление вспомогательных процессов, которые бы могли выступать «третьим» лицом, сохраняющим ресурс при обмене (можно провести аналогии с обменом значений двух переменных).

2.10. Программная реализация генератора команд

- Генератор команд, реализованный в рамках данной работы, написан на языке `python`
- Переходы между модулями приложения изображены на рисунке 2.8

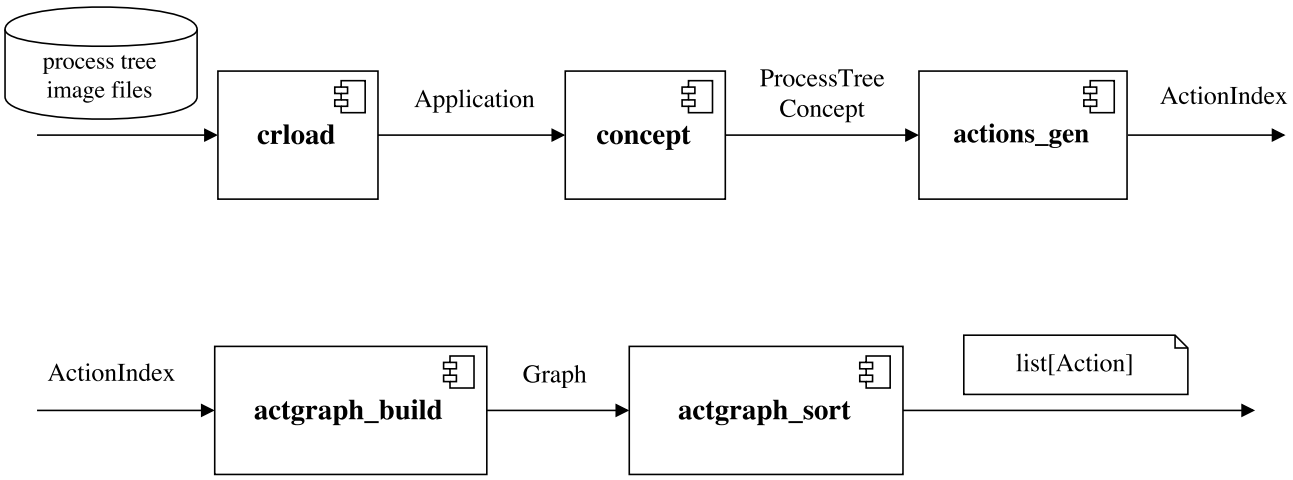


Рис. 2.8: Поток данных между модулями приложения

- Модели каждого конкретного ресурса объединены в простую иерархию, изображённую на рисунке 2.9

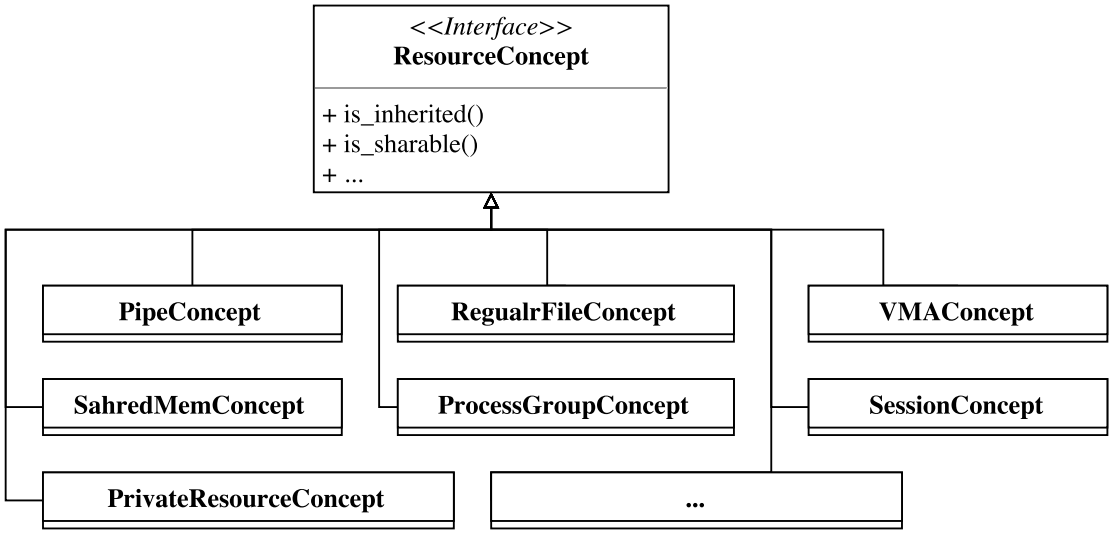


Рис. 2.9: Концепция ресурса

- Действия также объединены в иерархию и управляются выделенным классом, который, по мере генерации действий модулем `actions_gen`, индексирует их, предоставляя удобные (для реализации построения графа) интерфейсы для поиска различных действий. Структуры схематично изображены на рисунке 2.10

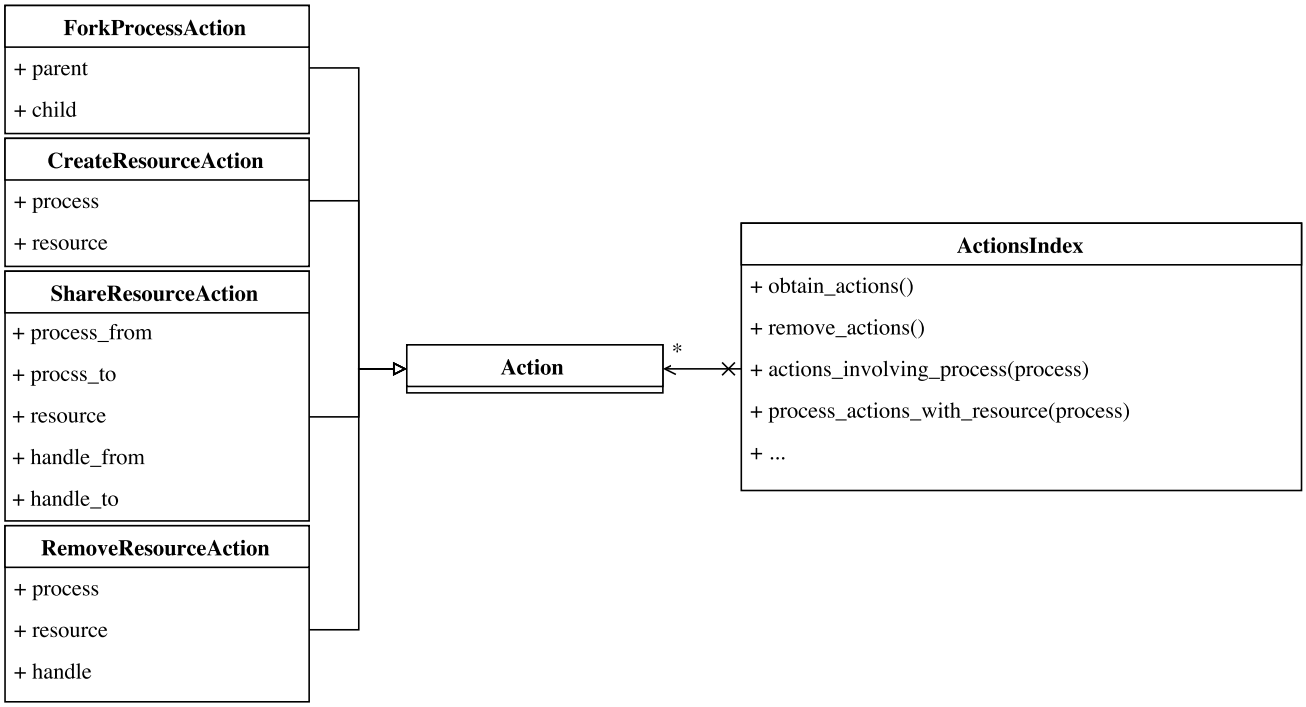


Рис. 2.10: Структуры для работы с действиями

Глава 3

Заключение

3.1. Итоги работы

В данной работе были получены следующие результаты:

- Введена формальная обобщённая модель жизнедеятельности процессов в ОС Linux для решения задачи восстановления
- Придуман и реализован алгоритм генерации промежуточного представления в виде графа действий, а так же генерации последовательности действий для восстановления дерева процессов
- На примере показана гибкость предложенного подхода, в сравнении с текущим подходом к восстановлению в системной утилите `criu`
- Написан набор программных утилит для визуализации промежуточных представлений, используемых генератором действий

Исходный код проекта можно найти по ссылке: `https://github.com/egorbunov/v2criu`.

3.2. Дальнейшие планы

- Описанная в данной работе модель, как было показано в разделе 2.9.3 , обладает некоторыми недостатками: существует класс процессов, граф действий для которых не будет ациклическим. Как уже говорилось, одним из подходов к решению этой проблемы является добавление вспомогательных временных процессов к дереву. Поэтому следующим

шагом на пути к решению задачи восстановления будет улучшение и усложенине приведённой модели.

- Следующим логичным продолжением данной работы, также, будет реализация раскрытия абстрактных команд, генерируемых реализованным решением, в наборы более специфичных, для конкретных ресурсов, команд-сестемных вызовов и их интерпретации.
- Также, в силу того, что программу для восстановления в нашей модели мы представляем в виде графа, возможным развитием данной работы может стать параллельное исполнение такой программы из действий.

Использованные источники и ссылки

[1] https://lwn.net/Articles/557046/ — *PRAM: Persistent over-kexec memory storage*

[2] https://criu.org/Main_Page — *Официальная страница проекта CRIU*

[3] http://dmtcp.sourceforge.net — *Страница проекта DMTCP*

[4] http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/ — *Страница проекта BLCR*

[5] https://openvz.org/Main_Pagea — *Страница проекта OpenVZ*