

САНКТ-ПЕТЕРБУРГСКИЙ АКАДЕМИЧЕСКИЙ УНИВЕРСИТЕТ

КАФЕДРА ?????

ВЫПУСКНАЯ РАБОТА МАГИСТРА

Тема: Алгоритм генерации команд восстановления деревапроцессов ОС Linux на
основе модели жизненного цикла ресурсов ОС

Направление: ???

Выполнил студент гр. ??? _____ Е.А. Горбунов

Руководитель, магистр ??? _____ Е.А. Баталов

Санкт-Петербург, 2017

Оглавление

1	Введение	3
1.1	О задаче восстановления и сохранения дерева процессов	3
1.2	Актуальность задачи сохранения/восстановления	4
1.2.1	Живая миграция	4
1.2.2	Обновление ядра без остановки программ	4
1.2.3	Отложенная отладка	5
1.2.4	Снимки приложений	5
1.2.5	Ускорение запуска программ	5
1.3	Системная утилита <code>criu</code>	5
1.4	Проблемы текущего подхода <code>criu</code> к восстановлению	7
1.5	Возможные подходы к решению проблемы восстановления	8
1.6	Задачи и требования	10
1.7	Содержание данной работы	10
2	Модель жизнедеятельности процесса и алгоритм построения графа действий	11
2.1	Основные понятия	11
2.1.1	Введение	11
2.1.2	Ресурс и процесс	12
2.1.3	Разделяемые и неразделяемые ресурсы	13
2.1.4	Наследуемый ресурс	14
2.1.5	Разделение ресурса между процессами	14
2.2	Действия и свойства процесса восстановления	15
2.2.1	Создание ресурса	16
2.2.2	Создание процесса	17
2.2.3	Наследование ресурса при рождении	18
2.2.4	Разделение ресурса при жизни	18

2.2.5	Зависимость между ресурсами	19
2.2.6	«Удаление» ресурса	20
2.3	Последовательность действий для восстановления	20
2.3.1	Замыкание ресурсов относительно зависимостей	21
2.3.2	Замыкание ресурсов относительно наследования	23
2.3.3	Добавление ресурса к possibleCreators	24
2.3.4	Добавление вспомогательного корневого процесса	25
2.3.5	Генерация множества действий	26
2.3.6	Отношение предшествования над действиями	28
2.3.7	Граф и сортировка	32
2.4	Примеры и гарантии	33
2.5	Классификация ресурсов	33
2.5.1	Идентификаторы процесса	34
3	Заключение	35
3.1	Итоги работы	35
3.2	Дальнейшие планы	35

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 выполняется 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 подвергся 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 $1000 \text{ с} = 15 \text{ мин}$, 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 дампы 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, предоставляемых операционной системой: вир-

туальная система /proc (информация о процессе из вне), системные вызовы и чтение памяти (изнутри самого процесса). В проекта 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 течением 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.
- 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

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 про-

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

цесса). 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 `crui` к восстановлению

`Crui` 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-помощниках, 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 `crui`.

Цепочка чётко отсортированных шагов выражает индивидуальный 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 дополнительного кода в цепочку действий, обрабатывающего эту нетривиальность
- 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 `crui` 100 процентно поддерживает

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

²Для некоторых ресурсов процесса это возможно: состояния регистров потоков процесса и др.

³Такое происходит, когда процесс-родитель завершается раньше ребёнка

процессов в дереве процессов, изображённом на рисунке 1.1.

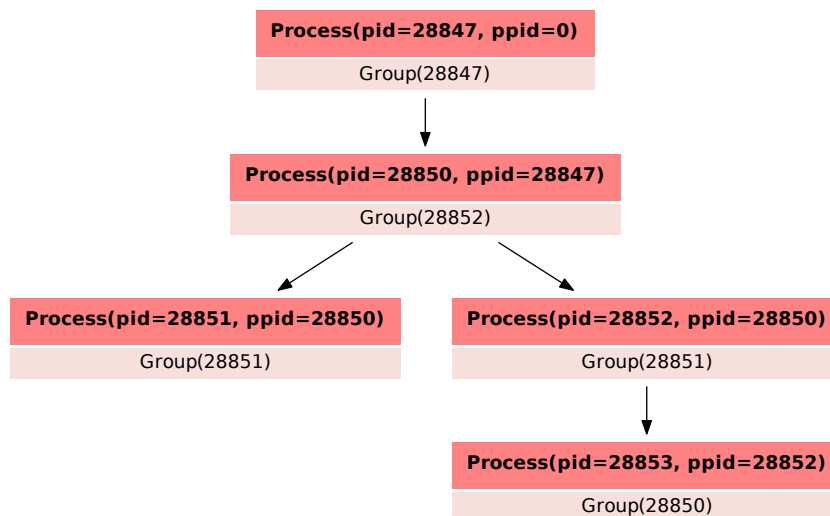


Рис. 1.1: Дерево процессов

В Linux, группа с идентификатором 28852, может быть создана только в процессе с идентификатором (pid) равным 28852. Но как мы видим из рисунка 1.1, процесс 28852 в целевом дереве находится в другой группе, а именно в 28851. На таком дереве процессов `sgui` 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 `sgui`, нет ⁴. Но в предыдущих работах, касающихся этой проблемы, был озвучен общий подход: разделение процедуры восстановления на несколько этапов (см. рис 1.2).

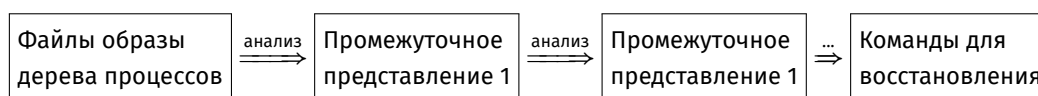


Рис. 1.2: Разделение на этапы

⁴В рамках ограничений пользовательского приложения

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).

Вопросы, который ставит данный подход:

- Каковы должны быть промежуточные представления?
- Какой вид имеет финальный список команд?

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

- Генератор можно реализовывать независимо от платформы, т.к. системных вызовов он не выполняет
- Генеративный подход имеет больший потенциал, чем зафиксированная последовательность действий, т.к. может генерировать абсолютно разные последовательности команд (это зависит от промежуточных представлений и т.д.)

⁵Именно с помощью системных вызовов производится большая часть изменений состояния процесса

1.6 Задачи и требования

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

- Предложить промежуточные представления для генератора и компромисс между сложностью генератора и интерпретатора
- Разработать генератор команд для задачи восстановления дерева процессов в рамках подхода генератор-интерпретатор, описанного в разделе 1.5

Общими требованиями к итоговому решению являются:

- Поддержка как можно большего числа ресурсов процессов в Linux
- Возможность эффективной реализации предлагаемых алгоритмов

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 жизнедеятельности процесса и алгоритм построения графа действий

2.1 Основные понятия

2.1.1 Введение

Задача данного документа — это формализация задачи восстановления дерева процессов в Linux. Перед тем как вводить основные понятия, неформально опишем задачу восстановления:

Задача восстановления дерева процессов в Linux — это задача поиска и исполнения последовательности действий, которые, будучи исполненными, приведут исходных «пустой» процесс в состояние целевого дерева процессов. Дополнительное ограничение: действия должны быть выполнимы из пространства пользователя.

Для этой задачи, целевое дерево процессов — это набор из нескольких процессов, объединённых в дерево отношением родитель-ребёнок. Каждый процесс из этого дерева, в рамках задачи восстановления, есть ни что иное, как статичный набор ресурсов — снимок состояния. Целевой процесс мы рассматриваем не как развивающийся во времени организм, а просто как состояние этого процесса в определённый момент времени.

Целевое состояние дерева процессов в Linux состоит из множества атрибутов: состояние виртуальной памяти, состояние регистров процессора, идентификатор процесса, группы, сессии и прочие идентификаторы, файловые дескрипторы, которые указывают на открытые файлы, открытые соединения, пространства имён, в которых находится процесс, и так далее. В данном документе мы будем именовать

все эти атрибуты понятием «ресурс». Таким образом целевое дерево процессов для нас — это набор ресурсов. Ниже мы попытаемся формализовать все эти понятия.

Решение задачи восстановления предполагает, что найденная последовательность действий может быть выполнена достаточно быстро, чтобы удовлетворять требованиям живой миграции.

2.1.2 Ресурс и процесс

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

Определение 1. *Ресурс* — r — некоторая структура в ядре ОС, которая так или иначе используется процессом и операционной системой. Такая структура — это абстрактное понятие само по себе, она может представлять из себя реальный экземпляр сишной структуры в ядре, но может быть чем-то менее осязаемым.

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 интерфейс к ресурсу.

Определение 2. *Интерфейс к ресурсу* — $handle$ — это объект, через который процесс получает доступ к ресурсу. Будем обозначать его как h .

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 1. При открытии 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 интерфейса 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:
 $(r, h_1), (r, h_2)$

Замечание 1. Возможно есть способ обойтись без понятия *handle*. Каким-то образом считать и файловые дескрипторы отдельными ресурсами. Но с этим понятием, пока что, всё лучше укладывается в голове.

Формализуем понятие процесса.

Определение 3. Процесс — это множество пар (r_i, h_i) из ресурсов и интерфейсов к ним. Будем обозначать процессы заглавными буквами и писать, например: $P = \{(r_1, h_1), (r_2, h_2), \dots, (r_n, h_n)\}$

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: $T = \{P_1, P_2, \dots, P_k\}$, среди которых выделен корневой процесс P_{root} , а для всех остальных процессов верно:

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

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

2.1.3 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 такие 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:

- идентификатор процесса (pid) — неразделяемый ресурс.
- значения регистров процесса для потока — неразделяемый ресурс
- открытый файл — разделяемый ресурс

- группа процессов — разделяемый ресурс
- сессия процессов — разделяемый ресурс

2.1.4 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. $isInherited(r)$ — предикат (свойство ресурса), который принимает значение истины, если ресурс r наследуется процессом-ребёнком от процесса-родителя при создании.

2.1.5 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. $isSharable(r)$ — предикат, который принимает значение истины, если ресурс r можно «разделить» между уже созданными процессами.

Пример 2. К примеру, процесс в Linux не может переместиться из одной сессии в другую, уже существующую (т.е. сессия не $isSharable(r)$). Но при этом внутри сессии, процесс может перейти в другую группу (разделить группу с каким-то другим процессом). Так же маппинги (Virtual Memory Area) не могут быть переданы между живыми процессами, в то время как файловые дескрипторы можно разделить.

Также есть ресурсы с особенной семантикой: например, `private mappings` — это, казалось бы, неразделяемый ресурс, который при наследовании, всё же, становится разделяемым из-за механизма `Cow On Write`.

Замечание 2. Так же отдельного внимания заслуживают `pid namespace`. Тут я о них просто упоминаю, но механизм перехода в `pid namespace` должен быть как-то встроен в модель, описываемую в этом документе: пока я об это не думал. Это частный случай! С ним всё не так красиво будет, скорее всего (но возможно, его можно будет как-то спрятать внутри конкретной реализации исполнения действия `ForkAction(, ,)`, о котором см. ниже)

2.2 Действия и свойства процесса восстановления

Во введении 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. 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, команд. 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.

Замечание 3. Мы говорим 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 вос-

становления дерева процессов. Будем считать, что действия берутся из множества допустимых действий \mathcal{A} .

Определение 8. Задача восстановления дерева процессов $T = \{P_1, P_2, \dots, P_k\}$ — это задача поиска упорядоченной и конечной последовательности действий $A = [a_1, a_2, \dots, a_n]$, $\forall i : a_i \in \mathcal{A}$, 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 3)

2.2.1 Создание ресурса

Действия будем стараться вводить исходя из возможностей Linux. Пускай целевое дерево процессов состоит из одного единственного процесса: $P = \{(r_1, h_1), \dots, (r_n, h_n)\}$. Каким способом процесс P мог завладеть одним из ресурсов r_i ? Он мог создать его сам: открыть файл, сокет, пайп или создать маппинг.

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

Определение 9. Действие создания ресурса — $CreateAction(P, r, [h_1, h_2, \dots, h_k])$ — процесс P создаёт ресурс r с интерфейсами h_i к нему. Данное действие выполняется самим процессом P и имеет следующий эффект по отношению к процессу:

$$P \xRightarrow{CreateAction(P, r, h)} \{(r, h_1), \dots, (r, h_k)\} \cup P$$

Замечание 4. Заметим, что в определении мы сделали так, что ресурс создаётся сразу с несколькими интерфейсами. Это связано с тем, что может существовать 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 будем записывать просто как: $CreateAction(P, r, h)$

Замечание 5. Важно понимать, что ресурс — это «структура» в ядре (см. определение 1), а значит действие создания ресурса иницирует создание некоторого объекта в ядре. Ядро — это глобальное хра-

нилище 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 .

Пример 3. Положим, что мы восстанавливаем дерево T .

- r — ресурс, что $type(r) = RegularFile$, тогда $possibleCreators(T, r) = T$. (тут вопрос: на деле, файл может открыть только тот процесс, у которого достаточно для этого прав, но в нашей модели мы можем запускать процесс восстановления с наивысшими правами, а потом всё понижать до нужных)
- Пусть $T = \{P_1, P_2, P_3\}$. Пускай $(task_1, pid : 1) \in P_1, (task_2, pid : 2) \in P_2, (task_3, pid : 3) \in P_3$. Рассмотрим ресурс $r = pggroup_2$ (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 $pid = 2$)

2.2.2 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 процес-

сы. Поэтому к действию создания ресурса добавляется действие создания процесса.

Определение 11. Действие создания процесса — $ForkAction(P_1, P_2, pid)$ — процесс P_1 создаёт потомка (процесс) P_2 с «интерфейсом» (идентификатором процесса) pid . Действие имеет следующий эффект:

$$\{P_1\} \xrightarrow{ForkAction(P_1, P_2, pid)} \{(child\ task, pid) \cup P_1, P_2\}$$

При этом процесс, который создаётся (P_2) не является пустым. Он может наследовать часть ресурсов своего родителя и может просто получать некоторые ресурсы при рождении, например собственный pid . Также у процесса всегда есть родитель: $parent(P)$.

2.2.3 Наследование ресурса при рождении

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

После выполнения действия $ForkAction(P_1, P_2, pid)$ верно, что:

$$\forall (r, h) \in P_1 \wedge isInherited(r) : (r, h) \in P_2. \quad (2.1)$$

Замечание 6. Мы тут считаем, что интерфейс к ресурсу наследуется и сохраняет своё значение.

Выше, в разделах 2.1.4 и 2.1.5 обсуждались ресурсы, которые должны разделяться наследованием (*private mappings*, ...). Мы будем считать, что те ресурсы, что обязательно должны разделяться наследованием не $isSharable(r)$.

2.2.4 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:

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

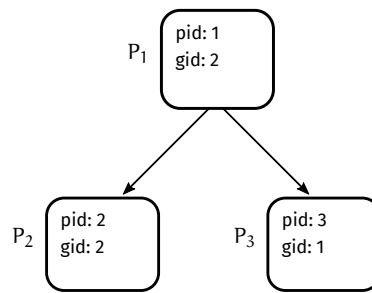


Рис. 2.1: Дерево T из 3 процессов

$P_1 = \{(task_1, pid : 1), (pgroup_2, pgid : 2)\}$

$P_2 = \{(task_2, pid : 2), (pgroup_2, pgid : 2)\}$

$P_3 = \{(task_3, pid : 3), (pgroup_1, pgid : 1)\}$

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, как я понимаю, 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 12. 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: $(r, h') \in P_2$.

2.2.5 Зависимость между ресурсами

Одни ресурсы могут зависеть от других. В моём текущем понимании, зависимости между ресурсами проявляются при создании этих ресурсов и этим можно ограничиться. Примеры зависимостей:

- Не приватная Virtual Memory Area зависит от того или иного файла, который необходим для создания этого маппинга (`mmap(. . .)`)
- ...

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

Определение 13. $resourceDependencies(r, h)$ — множество ресурсов (r', h') , от которых зависит ресурс r (с handle h).

2.2.6 «Удаление» ресурса

Может случаться так, что процесс из дерева, в целевой своей конфигурации, имеет ресурс r , т.е. $(r, h) \in P$. При этом $(q, _) \in resourceDependencies(r, h)$, но $(q, _) \notin P$. Это значит, что в последовательности действий для восстановления должно фигурировать действие по удалению 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 14. Действие «удаления» ресурса — $RemoveAction(P, r, h)$ — процесс P удаляет «из себя» пару (r, h) . Таким образом верно:

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

Выше, в определении ??, мы вводили понятие ресурса, который может присутствовать у процесса лишь в одном экземпляре. По сути это значит, что при восстановлении дерева, действие $CreateAction(P, ,)$ для ресурса этого типа может быть выполнено только один раз. Тут нам приходит на помощь действие $RemoveAction(, ,)$: с помощью него можно несколько раз создать такой ресурс, только так, чтобы каждому созданию сопутствовало удаление (возможно, кроме последнего создания). Т.е. если некоторый ресурс r таков, что $isSharable(r) = false$ и при этом в целевом дереве T есть несколько процессов ссылающихся на r , то разделяться в процессе восстановления этот ресурс должен методом наследования.

Замечание 7. PS: 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 $RemoveAction(P, r, h)$.

2.3 Последовательность действий для восстановления

Выше мы ввели следующие команды:

- *CreateAction*(P, r, h) (опр. 9)
- *ForkAction*(P_1, P_2, pid) (опр. 11)
- *ShareAction*($P_1, P_2, (r, h), h'$) (опр. 12)
- *RemoveAction*(P_1, r, h) (опр. 14)

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:

- *isSharable*(r) (опр. 7)
- *isInherited*(r) (опр. 6)
- *resourceDependencies*(r, h) (опр. 13)
- *possibleCreators*(T, r) (опр. 10)

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 $T = \{P_1, P_2, \dots, P_n\}$, где каждый из процессов

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

2.3.1 Замыкание ресурсов относительно зависимостей

Выше говорилось, что одни ресурсы могут зависеть от других (при создании), но при этом если для какого-то процесса $P \in T$ верно, что $(r, h) \in P$ и $(q, _) \in \text{resourceDependencies}((r, h))$, то может случиться так, что $(q, _) \notin P$. Поэтому первый шаг, который мы выполним перед генерацией последовательности действий — это замкнём ресурсы процессов относительно зависимости между ресурсами. В листинге 1 описана процедура замыкания.

После выполнения замыкания мы имеем новое дерево процессов T' такое, что: для любого процесса $P \in T'$ верно, что если $(r, h) \in P$, то и для любой зависимости $(q, _) \in \text{resourceDependencies}((r, h)) \Rightarrow (q, _) \in P$.

Сложность процедуры `close_against_dependencies`: $\mathcal{O}\left(n \cdot \left| \bigcup_{i=1}^n P_i \right| \right)$, в смысле числа итераций (выполнения операции объединения множеств $|\Rightarrow|$), где n — число процессов.

Listing 1 Замыкание процессов относительно зависимостей между ресурсами

```
def close_against_dependencies(T):  
    """  
    @param T - исходное дерево процессов  
    @return новое дерево процессов, той же мощности, что и T, но в котором все процессы  
            замкнуты относительно зависимостей  
    """  
    new_T = set()  
  
    for P in T:  
        new_P = P  
        deps = new_P  
  
        while len(deps) > 0:  
            next_deps = set()  
  
            for (r, h) in deps:  
                next_deps |= resourceDependencies(r, h)  
  
            new_P |= next_deps  
            deps = next_deps  
  
        new_T.add(new_P)  
  
    return new_T
```

Далее будем решать задачу для нового дерева T' , но в конце обязательно перейдём к решению исходной задачи путём добавления нужных действий $RemoveAction(, ,)$.

Замечание 8. 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 $T = \{P_1, P_2, \dots, P_k\}$, 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: допустим, 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!

В силу замечания выше введём ещё один вспомогательный предикат:

Определение 15. $canExistAtOnce(r_1, h_1, r_2, h_2,)$ — 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).

Замечание 9. Заметим, что данный предикат в том числе позволяет в течение процесса восстановления

перемещать один процесс из одной группы в другую, из одной сессии в другую и прочее.

2.3.2 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).

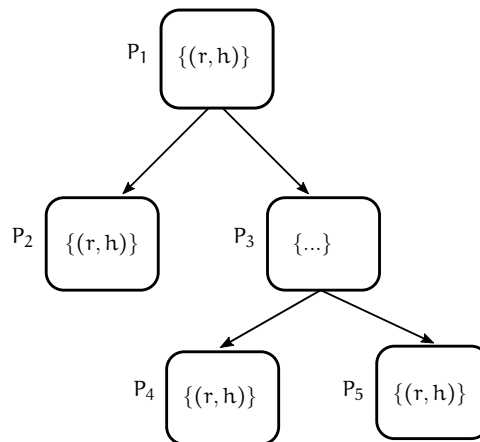


Рис. 2.2: Разделяемый наследованием ресурс, удалённый в нелистовом процессе

Случай, который похож на случай с зависимостью, на которую не ссылается процесс. И разрешить проблему можно похожим образом: добавить временно ресурсы в нужный процессы так, чтобы процессы, разделяющие ресурс передаваемый исключительно наследованием, формировали дерево. Конечно, после использования временные ресурсы должны быть удалены. Процедура замыкания описана в листинге 2.

Асимптотическая сложность данной процедуры: $\mathcal{O}\left(n \cdot \left| \bigcup_{i=1}^n P_i \right| \right)$ (опять же, я тут не учитываю сложности отдельных операций, т.к. их считаю достаточно простыми: все они вкладываются в полиномиальные рамки с лёгкостью). Тут важно увидеть, что внутренний цикл `while` выполняется суммарно не более $\text{len}(\text{new_T}) = n$. А внешний цикл совершает итераций столько, сколько у нас всего ресурсов в объединении всех процессов.

Заметим, что тут мы видим аналогичную проблему, описанную в замечании 8. Мы будем с этим бороться аналогично, на этапе генерации последовательности команд, используя предикат $\text{canExistAtOnce}(\cdot)$.

Будем считать, что у нас теперь есть деревья newT и T , что T — целевое, а newT — целевое, но замкнутое относительно зависимостей и наследования. Будем работать с первым, после чего подкрутим последовательность действий, чтобы удовлетворить исходной задаче. Заметим, что $\text{canExistAtOnce}(r_1, h_1, r_2, h_2)$

Listing 2 Замыкание относительно наследования

```
def close_against_inheritance(T):
    new_T = copy(T)
    all_resources = get_all_resources_with_handles(new_T)

    for (r, h) in all_resources:
        processes = get_holders(new_T, r, h) # возвращает множество процессов,
                                              # ссылающихся на ресурс (r, h)
        top = get_top_process(processes) # возвращает самый верхний (ближе к корню) процесс

        for P in processes:
            cur_P = parent(P)

            while P != top and cur_P not in processes:
                # значит cur_P не содержит (r, h), т.е. нашли "дырку"
                cur_P.add(r, h)
                cur_P = parent(cur_P)

    return new_T
```

может быть false только если $(r_1, h_1) \in \text{newT} \wedge (r_1, h_1) \notin T$ или $(r_2, h_2) \in \text{newT} \wedge (r_2, h_2) \notin T$ (т.е. хотя бы один из ресурсов r_1 или r_2 должен быть в newT)

Замечание 10. Мы помним, что часть ресурсов у нас при создании процесса-ребёнка наследуются. 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 ($\text{RemoveAction}(\cdot, \cdot)$) мы добавим в самом конце нашей процедуры построения списка команд в нужное место (сразу после $\text{ForkAction}(\cdot, \cdot)$ процесса-ребёнка) В рамках модели эти ресурсы нам никак не мешают.

2.3.3 Добавление ресурса к possibleCreators

Когда мы вводили определение 10, мы заметили, что не каждый процесс может создать определённый ресурс. Например, группа с идентификатором ID может быть создана только процессом с $\text{pid} = \text{ID}$. Можно представить ситуацию, что процесс P — единственный, кто может создать ресурс r , но при этом мы, на снимке дерева, наблюдаем то, что на r ссылается только $Q \neq P$, а у P этого ресурса вообще нет. Что делать? 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 его восстановлении нам будет необходимо в какой-то момент сделать так, чтобы процесс P_1 создал группу 1.

Для обслуживания таких ситуаций мы сделаем следующее. Ещё немного измени исходное дерево про-

цессов так, чтобы для любого ресурса r : $possibleCreators(T, r) \cap resourceHolders(T, r) \neq \emptyset$.

Listing 3 Расширение ресурсов процесса, который ответственен за создание ресурса, но не ссылается на него

```
def ensure_creators_have_holder(T):
    new_T = copy(T)
    for r in get_all_resources(new_T):
        creators = possibleCreators(new_T, r)
        if len(creators) == 0:
            # Наш алгоритм тут возвращает ошибку, пока мы не поддерживаем таких
            # ресурсов, которые не могут быть созданы ни одним из процессов из
            # снимка (такое возможно, опять же, с группами)
            raise ProcessTreeIsNotComplete()
        if len(creators & get_holders(r)) == 0:
            P = top(creators) # наиболее высокий в дереве процесс
            P.add(r, construct_handle(P, r))
    return new_T
```

2.3.4 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

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

являющееся допустимым снимком дерева процессов Linux, но после описанных операций добавления временных ресурсов и прочего, мы имеем дерево:

$$T' = \{P_0, P'_1, \dots, P'_n\}$$

такое, что:

- $P'_i \cap P_i = P_i$
- $P_0 = root(T') = parent(root(T))$
- Для любого ресурса $r \in T'$ верно, что $possibleCreators(T', r) \cap resourceHolders(T', r) \neq \emptyset$
- $\forall P' \in T'$: для любой пары $(r, h) \in P'$, верно, что $resourceDependencies(r, h) \subset P'$
- Для любого ресурса $r \in T'$ такого, что r разделяется только наследованием, т.е. $isInherited(r) =$

$\text{true} \wedge \text{isSharable}(r) = \text{false}$ верно, что $\text{resourceHolders}(T', r)$ образуют дерево (не лес!) относительно отношения родитель-ребёнок

- $\forall i \forall (r, h), (r', h') \in P_i : \text{canExistAtOnce}(r, h, r', h') = \text{true}$ но при этом, возможно, что для какой-то пары ресурсов $(r, h) \in P'_i \cap P_i, (r', h') \in P'_i : \text{canExistAtOnce}(r, h, r', h') = \text{false}$

Мы будем обозначать: $P_i \cap P'_i = \text{Tmp}_i$

2.3.5 Генерация множества действий

На первом шаге мы сгенерируем множество действий A , которые далее соберём в граф, линейаризация (топологическая сортировка) которого и будет ответом на задачу.

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

Listing 4 Добавление действий $\text{ForkAction}(\text{parent}(P), P, P.\text{pid})$ для всех $P \in T \setminus \{P_0\}$

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

- $\text{CreateAction}(, ,)$: действие должно быть добавлено для каждого ресурса, причём создавать этот ресурс обязан тот процесс, который может это сделать (P из $\text{possibleCreators}(T, r)$). См. листинг 5.

Listing 5 Добавление действий создания ресурсов

```
def get_resource_creator(T, r):
    return top(possibleCreators(T, r) \cap get_holders(T, r))

def get_create_actions(T):
    acts = set()
    resources = get_all_resources(T)
    for r in resources:
        creator = get_resource_creator(T, r)
        h = get_creator_handle(creator, r) # возвращает любой из хэндлов, по
                                           # которому creator ссылается на r,
                                           # но всегда один!
        acts.add(CreateAction(creator, r, h))
    return acts
```

- $\text{RemoveAction}(, ,)$: Данное действие должно быть добавлено для каждого временного ресурса ($r \in \text{Tmp}_i$) в каждом процессе. 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.

Listing 6 Добавление действий удаления временных ресурсов

```
def get_remove_tmp_acts(T):
    acts = set()
    for P in T:
        tmps = get_tmp_resources(P) # возвращает множество временных ресурсов
                                   # см. выше Tmpi

        for (r, h) in tmps:
            acts.add(RemoveAction(P, r, h))
    return acts
```

- *ShareAction*(,,): мы считаем, что если ресурс поддерживает возможность разделения его при жизни (как альтернатива наследованию), то нужно её использовать. Поэтому для каждого ресурса *r*, который *isSharable*(*r*) = *true*, мы будем генерировать действия разделения этого ресурса нуждающимся. См. листинг 7.

Listing 7 Генерация действий разделения ресурсов

```
def get_share_acts(T):
    acts = set()
    rs = get_all_resources(T)
    for r in rs:
        creator = get_resource_creator(T, r)
        cr_handle = get_creator_handle(creator, r)
        holders = get_holders(T, r)
        for P in holders:
            handles = get_resource_handles(P, r) # возвращает список handle'ов на ресурс
                                                  # напр: [h1, h2] значит, что в P есть пары
                                                  # (r, h1), (r, h2)

            if not isSharable(r) and (len(handles) > 1 or handles[0] != cr_handle):
                raise NonSharableResourceWithMoreThanOneHandle()

            for h in handles:
                if h == cr_handle:
                    continue
                acts.add(ShareAction(creator, P, (r, cr_handle), h))
```

Замечание 11. Как видим из листинга 7, если у нас есть какой-то *false = isSharable*(*r*) ресурс, с разными хэндлами в разных процессах, то мы падаем с ошибкой. Это, например, может иметь место быть тогда, когда некий *private mapping* был отнаследован, а потом перемарлен (*mapped*) в другое место. Пока что такое мы не поддерживаем. Но в будущем можно добавить, например, с помощью дополнительного действия *MoveResource* или чего-то подобного.

2.3.6 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 порядок (sic!), на основе которого потом построить граф и, если получится, линейаризовать его (если не получится, то значит, что мы столкнулись с деревом, которое наш алгоритм восстановить не в силах, об этом будем говорить подробнее позже).

Предшествование должно строиться по следующим правилам:

- (1) Все действия, которые исполняются от лица процесса P:

$$CreateAction(P, _, _), RemoveAction(P, _, _), ForkAction(P, _, _)$$

или так или иначе задействую процесс P:

$$ShareAction(P, _, _, _), ShareAction(_, P, _, _)$$

должны выполняться после того, как было выполнено создание этого процесса: $ForkAction(_, P, _)$

- (2) Все действия от лица процесса P с ресурсом (r, h):

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

должны выполняться после действия, эффектом которого было появление ресурса (r, h) у процесса P:

$$CreateAction(P, r, h) \vee ShareAction(_, P, (r, _), h) \vee ForkAction(_, P, _)$$

- (3) Все действия, в рамках процесса P, которые так или иначе используют ресурс (r, h), кроме удаления:

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

должны быть выполнены до $RemoveAction(P, r, h)$, если такое действие в наличии (его может не быть для ресурсов, которые должны присутствовать в финальном дереве) (действие создания будет предшествовать удалению в любом случае из-за предыдущего пункта).

- (4) Для всех ресурсов (r, h), которые разделяются больше чем одним процессом, т.е. $|get_holders(T, r, h)| > 1$ и при этом разделяться должны наследованием, т.е. $isInherited(r) = true \wedge isSharable(r) =$

false, должно выполняться следующее: получение ресурса (это либо создание, либо непосредственно *fork*) ресурса должно происходить до выполнения *ForkAction*(, ,) процессов, которые тоже разделяют этот ресурс.

То есть, в рамках процесса *P*, если

$$(r, h) \in P \wedge |\text{get_holders}(T, r, h)| > 1 \wedge \text{isSharable}(r) = \text{false} \wedge \text{isInherited}(r) = \text{true}$$

и если $\text{resource_creator}(r, h) = P$, то *CreateAction*(*P*, *r*, *h*) должно быть выполнено до *ForkAction*(*P*, *P'*, *pid*) для любого *P'*, что $(r, h) \in P'$

- (5) Также нужно обслужить зависимости между ресурсами. Пускай для процесса *P* верно, что $(r, h) \in P$ и $(r', h') \in P \wedge (r', h') \in \text{resourceDependencies}(r, h)$ и при этом *P* является создателем ресурса *r'*. Тогда действие получения ресурса (r, h) :

$$\text{CreateAction}(P, r, h) \vee \text{ShareAction}(_, P, (r, _), h) \vee \text{ForkAction}(_, P, _)$$

должно быть выполнено до:

$$\text{CreateAction}(P, r', h')$$

- (6) Также нужно разобраться с действиями, затрагивающими ресурсы, которые могут конфликтовать ($\text{canExistAtOnce}(_, _) = \text{false}$). Тут дела обстоят немного более хитро. Пускай два ресурса (r, h) и (r', h') в рамках одного процесса P'_i конфликтуют. Не умаляя общности мы можем считать, что $(r, h) \in \text{Tmp}_i$ (т.к. только «временные» ресурсы могут вызывать конфликты, см. замечание 8). 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: пускай 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) .

Так или иначе, исходя из соображений в этом пункте, нам для каждого процесса нужно упорядочить его ресурсы следующим образом:

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

А действия над "конфликтующими" ресурсами упорядочивать теперь можно следующим образом:

- $\forall i < j, \text{canExistAtOnce}(r_i, h_i, r_j, h_j) = \text{false}$: нужно сказать, что действие $\text{RemoveAction}(P, r_i, h_i)$ предшествует действию, эффектом которого является получение процессом P ресурса (r_j, h_j) (аналогично пункту (2)). i будет пробегать только индексы временных ресурсов, а значит действие $\text{RemoveAction}(P, r_i, h_i)$ существует для всех таких i .
- Как видим, если два наследуемых временных ресурса конфликтуют, то восстановление вообще невозможно (т.к. граф будет не ациклическим), ибо любое действия исполняемое процессом должно выполняться после форка этого процесса, поэтому один из ресурсов не может быть удалён до создания исполняющего это действия процесса.

Каждый из пунктов выше описывает правило, по которому одни действия должны исполняться раньше других. Ниже мы опишем все эти правила в псевдокоде, чтобы понять, как реализуется генерация «рёбер» предшествования. В листингах с 8 по 12 описаны вспомогательные процедуры.

Listing 8 Вспомогательная функция, возвращающая только те действия, в которых участвует процесс P

```
def get_actions_for_process(all_acts, P):
    """
    @param all_acts - множество всех действий
    @param P - интересующий процесс
    """
    acts = set()
    for act in all_acts:
        if act == CreateAction(P, _, _) or
           act == RemoveAction(P, _, _) or
           act == ShareAction(P, _, _) or
           act == ShareAction(_, P, _, _) or
           act == ForkAction(P, _, _):
            acts.add(act)
    return acts
```

Listing 9 Вспомогательная функция, возвращающее действие, эффект которого – создание процесса P

```
def get_process_fork_act(all_acts, P):
    for act in all_acts:
        if act == ForkAction(_, P, _):
            return act
    raise Error("No creator!")
```

Listing 10 Вспомогательная функция, возвращающая действие, благодаря которому процесс P получил ресурс (r, h)

```
def get_proc_obtain_resource_act(all_acts, P, (r, h)):
    if (r, h) not in P:
        raise Error()
    for act in all_acts:
        if act == CreateAction(P, r, h) or
           act == ShareAction(_, P, (r, _), h):
            return act
    # если с P никто не делится ресурсом и он его не создаёт сам, то
    # остаётся только наследование
    return get_process_fork_act(all_acts, P)
```

Listing 11 Вспомогательная функция, возвращающее действие, которое удаляет ресурс (r, h) из процесса P

```
def get_proc_remove_resource_act(all_acts, P, (r, h)):
    if (r, h) not in P:
        raise Error()
    for act in all_acts:
        if act == RemoveAction(P, r, h):
            return act
    # нет действия удаления, значит ресурс и не нужно удалять
    return None
```

Listing 12 Вспомогательная функция, возвращающее список действий использующих ресурс, но не связанных с его созданием или удалением

```
def get_acts_with_resource(all_acts, P, (r, h)):
    acts = set()
    for act in all_acts:
        # только действие разделения ресурса никак не связано с
        # удалением или созданием ресурса для процесса P
        if act == ShareAction(P, _, (r, h), _):
            acts.add(act)
    return acts
```

Мы будем генерировать «рёбра» предшествования: $\text{Preceed}(a_1, a_2)$, где $a_1, a_2 \in A$, что обозначает: действие a_1 предшествует действию a_2 .

В листинге 13 описана вспомогательная процедура генерации таких рёбер из всех действий одного множества по все действия второго.

В листинге 14 описана процедура генерации всех «рёбер» предшествования.

Listing 13 Вспомогательная процедура, генерирующая рёбра предшествования из всех действия из as ко всем действиям из bs

```
def gen_as_preceed_bs(as, bs):
    edges = set()
    for a in as:
        for b in bs:
            edges.add(Preceed(a, b))
    return edges
```

Listing 14 Общая процедура генерации «рёбер» предшествования

```
def generate_precedence_relations(all_acts, T):  
    """  
    @param all_acts - список всех действий  
    @param T - дерево процессов  
    """  
  
    edges = set()  
  
    for P in T:  
        proc_fork_act = get_process_fork_act(all_acts, P)  
        proc_acts = get_actions_for_process(all_acts, P)  
  
        # пункт (1)  
        edges.add(gen_as_preceed_bs([proc_fork_act], proc_acts))  
  
        for (r, h) in P:  
            obtain_act = get_proc_obtain_resource_act(proc_acts, P, (r, h))  
            remove_act = get_proc_remove_resource_act(proc_acts, P, (r, h))  
            resource_acts = get_acts_with_resource(proc_acts, P, (r, h))  
  
            # пункт (2) (если remove_act = None, то считаем, что он не добавляется)  
            edges.add(gen_as_preceed_bs([obtain_act],  
                                       resource_acts + [remove_act] if remove_act else []))  
  
            # пункт (3)  
            edges.add(gen_as_preceed_bs(resource_acts, remove_act))  
  
            # пункт (4)  
            edges.add(handle_resource_inheritance(T, P, all_acts, (r, h)))  
  
            # пункт (5)  
            edges.add(handle_resource_dependencies(T, P, all_acts, (r, h)))  
  
        # пункт (6)  
        edges.add(handle_can_exist_at_once(T, P, all_acts))  
  
    return edges
```

2.3.7 Граф и сортировка

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 $[a_1, a_2, \dots, a_N]$. 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 смулировать выполнение дей-

Listing 15 Обработка случая (4): добавления рёбер предшествования между действия создания ресурса и создания ребёнка, его наследующего

```
def handle_resource_inheritance(T, P, all_acts, (r, h)):
    edges = set()
    resource_holders = get_holders(T, (r, h))
    if not isInherited(r) or isSharable(r) or len(resource_holders) == 1:
        return set()

    # может быть только create или fork
    obtain_act = get_proc_obtain_resource_act(all_acts, P, (r, h))
    if obtain_act != CreateAction(P, r, h):
        return set()

    # имеем дело с ресурсом, передаваемым наследованием, который создаётся P
    for H in resource_holders:
        if H == P:
            continue
        fork_act = get_process_fork_act(all_acts, H)
        if fork_act == ForkAction(P, H, _):
            edges.add(Precede(obtain_act, fork_act))

    return edges
```

Listing 16 Обработка случая (5)

```
def handle_resource_dependencies(T, P, all_acts, (r, h)):
    # прямая реализация того, что описано в самом пункте
```

Listing 17 Обработка случая (6)

```
def handle_can_exist_at_once(T, P, all_acts):
    # прямая реализация того, что описано в самом пункте
```

ствий из списка и вставить в нужные места *RemoveAction*(P, r, h) так, чтобы закрыть лишние ресурсы.

2.4 Примеры и гарантии

TODO: тут будет несколько примеров работы алгоритма, пример дерева, которое алгоритм восстановить не сможет и, возможно, какие-то теоремы про мощность данного подхода

2.5 Классификация ресурсов

TODO: тут будет информация про реальные ресурсы Linux, про то, какими свойствами они обладают в рамках приведённой модели.

2.5.1 Идентификаторы процесса

Ресурс	<i>handle</i>	<i>isInherited(r)</i>	<i>isSharable(r)</i>
Сам процесс	pid	нет	нет
Группа процесса	pgid	да	да, setpgid()
Сессия процесса	ssid	да	нет
Идентификатор пользователя	uid	да	да, setuid()

Глава 3

Заключение

3.1 Итоги работы

3.2 Дальнейшие планы

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_Page — *Страница проекта OpenVZ*
- [6] Chris Manning and Hinrich Schütze, *Foundations of Statistical Natural Language Processing*, MIT Press. Cambridge, MA: May 1999.
- [7] Dan Klein, Christopher D. Manning, *Accurate Unlexicalized Parsing*, 2003, доступно по ссылке: <http://dl.acm.org/citation.cfm?id=1075150>
- [8] Mitchell Marcus, Grace Kim, Mary Ann Marcinkievich, *The Penn Treebank: annotating predicate argument structure*, University of Pennsylvania