# Граф действий для восстановления дерева процессов

Горбунов Егор Алексеевич

12 мая 2017 г.

## 1 Основные понятия

### 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 дерево 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 этим понятия.

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

### 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 используются ***процессом*** и операционной системой. Такая структура — это абстрактное понятие само по себе, она может представлять из себя реальный экземпляр сишной структуры, но можеть 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 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 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 в 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: $P = \{(r_1, h_1), (r_2, h_2), \ldots, (r_n, h_n)\}$

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:

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

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

## 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.0

**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

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

К примеру:

- идентификатор процесса (`pid`) — неразделяемый ресурс.
- значения регистров потока — неразделяемый ресурс
- открытый файл — разделяемый ресурс
- группа процессов — разделяемый ресурс
- сессия процессов — разделяемый ресурс

## 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.

## 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 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 Copy On Write.

**Замечание 2.** Так же отдельного внимания заслуживают `pid namespace`. Тут я о них просто упоминаю, но механизм перехода в `pid namesapce` 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 внутрь конкретной реализации исполнения действия $\mathit{ForkAction}(,,)$, о котором см. ниже)

## 1.6 Одиночный ресурс

В силу довольно общего определения термина «ресурс» в рамках нашего подхода, приходится эти ресурсы разделять на классы. Выше уже были введены классы ресурсов: $\mathit{isSharable}(r)$ и $\mathit{isInherited}(r)$. Ещё одним важным свойством некоторых ресурсов является то, что иногда ресурс может существовать в единственном экземпляре. У каждого ресурса есть тип (ведь мы как-то идентифицируем ресурсы!), будь-то файл, группа, идентификатор процесса, сессия, namespace. Таким образом для ресурсов некоторого типа верно, что одновременно два экземпляра этого типа процесс содержать не может.

**Определение 8.** Тип ресурса $r$ — $type(r)$

**Определение 9.** $\mathit{isSingle}(r)$ — предикат, истинный для ресурса $r$, если

$$\forall P \in \mathcal{P}, (r, \_) \in P : \forall (r', \_) \in P, r' \neq r : type(r) \neq type(r')$$

Т.е. если процесс $P$ содержит ресурс $r$, то для любого другого ресурса $r'$ этого процесса верно: $type(r')! = type(r)$

**Пример 3.** Примером такого «одиночного» ресурса являются:

- Группа процесса
- Сессия процесса
- Namespace любого типа
- Идентификатор процесса

В свою очередь открытый файл таким ресурсом не является.

# 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 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 10.** 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 3)

## 2.1 Создание ресурса

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

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

**Определение 11.** Действие создания ресурса — $\textit{CreateAction}(P, r, h)$ — процесс $P$ создаёт ресурс $r$ с интерфейсом $h$ к нему. Данное действие исполняется самим процессом $P$ и имеет следующий эффект по отношению к процессу:

$$P \xRightarrow{\textit{CreateAction}(P,r,h)} (r, h) \cup P$$

**Замечание 4.** 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:

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

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 12.** $\textit{possibleCreators}(T, r)$ — это множество процессов $P \in T$ такое, что $P$ способен создать ресурс $r$.

**Пример 4.** Положим, что мы восстанавливаем дерево $T$.

- $r$ — ресурс, что $type(r) = RegularFile$, тогда $\textit{possibleCreators}(T, r) = T$. (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 нуж-

ных)

- 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 $r = pgroup_2$ (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***$(\{P_1\}, r) = \varnothing$, т.к. в одиночку процесс $P_1$ создать этот ресурс не способен (ибо необходим процесс с идентификатором $pid = 2$)

## 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 процессы. Поэтому к действию создания ресурса добавляется действие создания процесса.

**Определение 13.** 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:

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

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 $pid$.

## 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:

$$\forall (r, h) \in P_1 \wedge \textit{isInherited}(r) : (r, h') \in P_2. \tag{1}$$

**Замечание 5.** Часто $h == h'$ (не всегда ли?) в Linux. Например, открытые файловые дескрипторы процесса-родителя сохраняют свои значения в процессе-ребёнке. Аналогично происходит с адресами начала и конца маппингов и др. Равенство тут понимается не в смысле того, что это одинаковые объекты, а в смысле того, что это одинаковые значения.

Выше, в разделах 1.4 и 1.5 обсуждались ресурсы, которые должны разделяться наследованием (private

mappings, ...). Мы будем считать, что те ресурсы, что обязательно должны разделяться наследованием не $\mathit{isSharable}(r)$.

## 2.4 Разделение ресурса при жизни

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

Посмотрим на следующее дерево процессов:

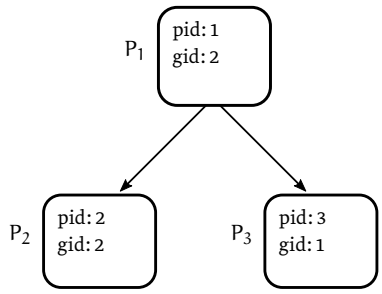


Рис. 1: Дерево T из 3 процессов

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

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 12). 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 $\mathit{CreateAction}(,,)$ 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 14.** Действие разделения ресурса — $\mathit{ShareAction}(P_1, P_2, r, h)$ — процесс $P_1$ разделяет ресурс $r$ 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* $r$. После выполнения действия верно: $(r, h) \in P_2$

## 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. Введём обозначение для краткого описания того, что один ресурс зависит от другого:

**Определение 15.** *DependsProperty*$(r_1, r_2)$ — свойство целевого дерева процессов, гласящее, что ресурс $r_1$ зависит от ресурса $r_2$

## 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 *DependsProperty*$(r, q)$, 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 16.** Действие «удаления» ресурса — *RemoveAction*$(P, r, h)$ — процесс $P$ удаляет «из себя» пару $(r, h)$. Таким образом верно:

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

Выше, в определении 9, 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 6.** **PS**: также, помимо обычных действий (*CreateAction*$(P, r, h)$, *ShareAction*$(P_1, P_2, r, h)$) 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*$(P, r, h)$

# 3 Последовательность действий для восстановления

Выше мы ввели следующие команды:

- ***CreateAction***$(\mathrm{P}, \mathrm{r}, \mathrm{h})$ (опр. 11)
- ***ForkAction***$(\mathrm{P}_1, \mathrm{P}_2, \mathtt{pid})$ (опр. 13)
- ***ShareAction***$(\mathrm{P}_1, \mathrm{P}_2, \mathrm{r}, \mathrm{h})$ (опр. 14)
- ***RemoveAction***$(\mathrm{P}_1, \mathrm{r}, \mathrm{h})$ (опр. 16)

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***$(\mathrm{r})$ (опр. 7)
- ***isInherited***$(\mathrm{r})$ (опр. 6)
- ***DependsProperty***$(\mathrm{r}_1, \mathrm{r}_2)$ (опр. 15)
- ***possibleCreators***$(\mathrm{T}, \mathrm{r})$ (опр. 12)

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

На входе мы имеем дерево процессов $\mathrm{T} = \{\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_n\}$, где каждый из процессов

$$\mathrm{P}_i = \{(\mathrm{r}_1, \mathrm{h}_1), (\mathrm{r}_2, \mathrm{h}_2), \ldots, (\mathrm{r}_{n_i}, \mathrm{h}_{n_i})\}$$

# 4 Классификация ресурсов

## 4.1 Идентификаторы процесса

| **Ресурс** | ***handle*** | **Наследование** | **Разделение (share)** |
|---|---|---|---|
| Сам процесс | pid | нет | нет |
| Группа процесса | pgid | да | да, `setpgid()` |
| Сессия процесса | ssid | да | нет |
| Идентификатор пользователя | uid | да | да, `setuid()` |