МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ»

Кафедра автоматизированных систем управления (АСУ)

УТВЕРЖДАЮ
Зав. кафедрой АСУ, профессор


А.М. Кориков

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Тема 4. Управление группами процессов ОС

Учебно-методическое пособие

для студентов уровня основной образовательной программы: **магистратура**
направление подготовки: **09.04.01 - Информатика и вычислительная техника**

Разработчик
доцент кафедры АСУ

В.Г. Резник

2016

Резник В.Г.

Современные операционные системы. Тема 4. Управление группами процессов ОС. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 39 с.

Учебно-методическое пособие предназначено для изучения темы №4 по дисциплине «Современные операционные системы» для студентов уровня основной образовательной программы магистратура направления подготовки: 09.04.01 «Информатика и вычислительная техника».

Оглавление

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ.....1
Введение.....4
1 Тема 4. Управление группами процессов ОС.....5
1.1 Современные механизмы управления процессами.....5
1.2 Группы процессов cgroups.....8
1.2.1 История cgroups.....8
1.2.2 OOM-Killer.....8
1.2.3 Архитектура cgroups.....10
1.2.4 Недостатки cgroups.....12
1.3 Система виртуализации на уровне ОС.....13
1.3.1 Система KVM.....15
1.3.2 Система LXC.....16
1.4 Технология systemd.....17
1.4.1 Классическая изоляция chroot.....20
1.4.2 Механизмы контейнеризации namespaces.....21
1.4.3 Контейнеры systemd-nspawn.....23
1.5 Кластерное управляющее ПО Grid Engine.....24
2 Лабораторная работа №4.....26
2.1 Практическое использование cgroups.....26
2.1.1 Управление памятью процесса.....26
2.1.2 Отключение действия на процесс OOM-Killer.....28
2.1.3 Управление устройствами.....29
2.2 Практическое использование namespaces.....32
2.2.1 Изоляция PID процессов.....32
2.2.2 Изоляция файловой системы процесса.....34
Список использованных источников.....38

Введение

Тема 4 посвящена различным аспектам управления группами процессов, многие из которых достаточно интенсивно развиваются и в настоящее время.
Учебный материал данной темы опирается на базовые теоретические знания, полученные студентами при изучении баклаврского курса «Операционные системы» [5, 6].

Общий теоретический план данной темы охватывает следующие вопросы:

- Современные механизмы управления процессами.
- Группы процессов cgroups.
- Система виртуализации на уровне ОС (KVM, LXC).
- Технология инициализации systemd.
- Кластерное управляющее ПО Grid Engine.

Указанные теоретические вопросы частично закрепляются во время проведения лабораторной работы, в которой рассматриваются практичские аспекты применения:

- механизмов технологии cgroups для ограничения ряда ресурсов процессов;
- механизмов технологии namespaces для разделения ряда пространств процессов, используемые в более общих технологических подходах, называемых - контейнерными технологиями;

По завершению лабораторной работы студент должен получить начальные практические навыки по управлению процессами и уметь самостоятельно совершенствовать эти навыки в своей последующей профессиональной деятельности.

Методика проведения лабораторных работ по данной дисциплине предполагает использование специального дистрибутива ОС УПК АСУ, которое установлено в компьютерных классах кафедры АСУ.

1 Тема 4. Управление группами процессов ОС

В учебном курсе «Операционные системы» [5, 6] были рассмотрены четыре основных концепции управления процессами:

- *монопольный* — встроенные и специализированные системы, а также промежуточный этап загрузки ОС, в котором не проверяются привилегии работы программ пользовательского режима;
- *System V init* — классический подход, выделяющий семь уровней эксплуатации ОС;
- *upstart* — модификация классического подхода, с учетом наличия групп процессов;
- *systemd* — новая концепция управления процессами ОС Linux, развиваемая ***с 2010 года*** группой программистов корпорации ***Red Hat*** под руководством Леннарта Поттеринга.

1.1 Современные механизмы управления процессами

Со времен создания первых компьютеров возникает вопрос: «Как управлять *отдельными единицами объектов*, которые называются программами?».

Программы - командные сущности ЭВМ, содержащие коды расчетных и управляющих вычислений. Они стали первыми единицами внешних по отношению к аппаратной части ЭВМ объектами, на основе которых стал формироваться *пакетный режим обработки информации*.

Пакетный режим - технология выполнения отдельных независимых друг от друга программ, цель которой — оптимальное использование дорогостоящих вычислительных ресурсов ЭВМ:

- эффективное использование *процессорного времени* — экономия простоев процессора между запусками отдельных программ;
- эффективное использование *оперативной памяти* ЭВМ (ОЗУ) для размещения отдельных частей (сегментов) многих программ.

Развитие технологии пакетного режима привело к формированию специальных управляющих программ:

- *супервизорных программ* предназначенных для автоматизации управления программами пакетного режима;
- *супервизора* — отдельной программы, органично включающей в себя функции отдельных супервизорных программ.

Первые супервизоры — прототипы ядер современных операционных систем.

Операционная система — ядро (супервизор) и набор системных программ:

- *утилит и библиотек* — программ узкоспециализированного назначения, понимаемые как неотъемлемая часть ОС;
- *модулей ядра ОС* — загружаемые в ядро специальные программы;
- *firmware* — специальные программы, загружаемые в аппаратную часть ЭВМ.

Фактически, понятие **операционная система** сформировалось из набора супервизорных программ, часть из которых было объединено в виде одной программы - ядра ОС, работающего в защищенном привелигерованном режиме, а другая - окружение ядра, работающее в непривелигерованном пользовательском пространстве.

Само понятие процесс сформировалось как отдельный программный объект пространства пользователя, который учитывается и управляется операционной системой — ядром ОС.
Цель формирования процессных объектов — разделение между программными единицами кода наиболее важных ресурсов ЭВМ:

- процессорного времени;
- памяти ОЗУ.

Таким образом, *термин программа* потерял свое первоначальное семантическое значение - единица управляемого программного обеспечения, которое сформировалось на уровне технологии пакетного режима работы ЭВМ. Его место занял *термин процесс*, который появился как следствие возникновения *термина ОС*.

Как следствие, в терминах современных ОС:

- *программа* - результат разработки приложения, представленного в виде файлов, имеющих формат обеспечивающий ее загрузку на выполнение конкретной ОС;
- *пакетный режим обработки информации* — набор программ, запускаемых на функционирующей ОС и обычно выполняемых в фоновом режиме, с низким приоритеом, не мешающим работе системного и прикладного ПО.

Классическим механизмом управления процессами является *scheduling* - *планирование*, которое осуществляется на уровне ядра ОС планировщиком, входящим в подсистему управления процессами.
Такие «механизмы» реализуются достаточно сложными алгоритмами, учитывающими:

- *состояние процессов*: запуск, готовность, выполнение, ожидание, завершение;
- *приоритетность процессов*;
- эффективное использование *выделенных квантов времени процессора*;
- *требования задач* реального времени выполнения.

Современные механизмы управления процессами ведутся в направлении:

- *нити* (потоки, threads) — реализации параллельного выполнения локальных частей кода процесса в его общем адресном пространстве;
- *управление группами* (cgroups) — дополнительная идентификация процессов и их объединение в плане совместного выполнения и ограничений по используемым ресурсам ЭВМ;
- *совершенствование общего системного взаимодействия процессов* на уровне пространства пользователя, реализуемого проектами *upstart* и *systemd*.

Указанное направление развития имеет ***две координаты***, отражающие разные его

качества:

- *системная координата*, отражающая развитие внутренних свойств управления процессами направленными на повышение быстродействия и надежности функционирования процессов, что в совокупности повышает качество виртуальной машины, формируемой ОС;
- *прикладная координата*, отражающая служебную (вспомогательную) часть ОС, которая обеспечивает функционирование как отдельных программ, так и сложных прикладных систем, что в совокупности расширяет границы и полезность применения самой ОС.

Развитие каждого из указанных качеств опирается на соответствующую функциональную поддержку, реализуемую в ядре ОС, и как показала практика — не могут быть эффективно реализованы без такой поддержки. Этот факт сам по себе обосновывает эффективность и надежность использования *монолитных ядер* по сравнению с *микроядерной архитектурой*.

Тем не менее, указанные направления имеют достаточные уровни самостоятельности, реализуемые в различных современных проектах.

Современный тренд системной координаты направлен на:

- *эффективное управление группами* (cgroups), что должно обеспечить совершенствование системной поддержки *мультизадачного режима работы ОС* и обсуждается в подразделе 1.2;
- *стандартизация и совершенствование функций* ***systemd***, обсуждаемый в подразделе 1.4, что должно унифицировать общие средства управления процессами ОС, включая функциональный потенциал работы с ***cgroups***.

Современный тренд прикладной координаты — достаточно обширен, поскольку он проецируется, прежде всего, расширяющимся списком потребностей приложений. Его можно характеризовать термином *контейнерные технологии*, что включает в себя парадигмы объединения и изоляции, относящиеся к программному обеспечению.

Контейнерные технологии — набор подходов к реализации программного обеспечения ЭВМ, порожденный все возрастающей тенденцией одновременной эксплуатации большого количества программ различной прикладной направленности и, как следствие, к лавинообразному увеличению количества управляющих связей между такими программами.

Внедрение контейнерных технологий порождает новый уровень виртуализации, который должен быть реализован на уровне ОС. Данный аспект технологий рассматривается в следующих подразделах данной темы:

- *подраздел 1.3* — общие подходы на уровне супервизоров ОС;
- *подраздел 1.5* — кластерное управляющее ПО, на примере Grid Engine.

1.2 Группы процессов cgroups

Хотя современные компьютеры обладают гораздо большими техническими возможностями, по сравнению со своими предшественниками, ресурсов ЭВМ никогда не бывает слишком много. Всегда наступает время, когда нужные ресурсы заканчиваются и необходимо их правильно распределить, или, в крайних случаях, прекратить выполнение одного или нескольких процессов, ответственных за создание критической ситуации.
Типичным примером являются серверные приложения порождающие множество отдельных процессов, каждый из которых обслуживает отдельный запрос клиента. Даже, если каждый процесс занимает небольшой объем памяти и небольшое количество времени процессора, в совокупности, если не принимать специальных мер, они могут прекратить работу системы.

В общем случае, указанная проблематика ОС связана с поддержкой *мультизадачного режима ее работы* и обязана разрешаться на уровне разработки приложений. Тем не менее, надежность работы всей вычислительной системы требует специальных решений, реализуемых на уровне ядра ОС. Одним из таких решений является развитие и реализация модели ***cgroups***.

1.2.1 История cgroups

В 2006 году, компания Google приступила к реализации проекта под названием *process containers* (контейнеры процессов).
Работа велась двумя сотрудниками компании Полом Менеджем и Рохитом Сетом, которые поставили себе целью: *усовершенствовать механизм распределения процессорного времени и памяти между задачами*.
В конце 2007 года, этот проект был переименован в *control groups* (cgroups), что более точно обозначает его направленность и освобождает расширительное понимание термина *контейнеры*.
В 2008 году, механизм ***cgroups*** был официально включен в ядро 2.6.24 ОС Linux.

1.2.2 OOM-Killer

Супервизор (ядро ОС) решает множество достаточно сложных проблем, связанных с распределением ресурсов ЭВМ. Одним из таких наиболее важных ресурсов, вызывающих различные критические ситуации работы вычислительных систем, является *оперативная память*.
Критичность нехватки этого ресурса вызвана тем, что процессор может выполнять только *код программ*, находящихся в ОЗУ (оперативном запоминающем устройстве). Когда ядро ОС выделяет память процессу и ее недостаточно, то возникает ситуация, требующая принятия решения, для обработки которой также требуется ОЗУ.

В общем случае, запускается подсистема супервизора, называемая ***OOM-Killer***, и ситуация разрещается удалением ряда процессов, определение которых является достаточно сложным и непрозрачным процессом.

OOM (Out-Of-Memory) — *нехватка памяти*.
OOM-Killer - механизм ядра Linux для освобождения памяти при исчерпании оперативной памяти за счет убийства одного из существующих процессов.

При исчерпании памяти ядро вызывает OOM-Killer, который выбирает по заданному набору правил один процесс и убивает его. Память, принадлежавшая этому процессу освобождается и передается в распоряжение ядра, после чего ядро предоставляет память тому процессу, для которого требовалось выделение памяти.
Наибольшие шансы стать жертвой OOM-Killer имеют свежезапущенные пользовательские процессы с большим объемом виртуальной памяти или имеющие большое число дочерних процессов.
Наименьшему риску подвергаются давно стартовавшие системные процессы, принадлежащие пользователю ***root***.

OOM-Killer убивает процессы сигналом *SIGKILL*, не предоставляя им возможность корректно завершить свое выполнение (сохранить данные, вызвать другие процессы). Поэтому результат работы OOM-Killer часто приводит к тяжелым последствиям для системы и для данных, и является для операционной системы средством последней надежды.
Срабатывание OOM-Killer, особенно, частое, служит признаком несбалансированной работы системы. При возникновении такой ситуации рекомендуется отнестись к ней серьезно и предпринять меры для предотвращения исчерпания памяти.

Замечание

Наличие и работа OOM-Killer является одной из проблематик операционных систем, которые недостаточно «прозрачны» в плане своего функционирования и настройки.

Поскольку дефицит ресурсов является обычной ситуацией в плане прикладного использования ОС, то желательно иметь эффективные инструменты корректного разрешения возникающих проблем.

С другой стороны, основным источником возникающих проблем является функционирование достаточно сложных приложений, которые не только могут потреблять значительное количество ресурсов, но и сами порождать проблемы, например, связанные с «утечкой памяти».

Поскольку OOM-Killer работает по некоторому (пусть даже по очень сложному) алгоритму, возникают ситуации, когда из системы будут удаляются важные с точки зрения вычислительной системы процессы.

В общем случае, принципиальное эффективное разрешение возникающих проблем возможно только при построении эффективных алгоритмов управления группами процессов.

1.2.3 Архитектура cgroups

Основная цель проекта *cgroups* — предоставить единый программный интерфейс к целому спектру средств управления процессами.
В прикладном плане, механизм *cgroups* предоставляет возможности:

- *ограничения ресурсов* (*resource limiting*), включая использование памяти;
- *приоритизации*, позволяющей разным группам выделить разное количество процессорного ресурса и пропускной способности подсистемы ввода-вывода;
- *учёт*, позволяющий выполнить расчет затрат используемых группой ресурсов;
- *изоляции*, обеспечивающей ***разделение пространств имен*** для групп таким образом, что разным группам становятся недоступны процессы, сетевые соединения и файлы других групп;
- *управления*, обеспечивающие приостановку работы групп (*freezing*), создание контрольных точек (*checkpointing*) и их перезагрузку.

Реализация механизма ***cgroups*** состоит из двух составных частей:

- *ядра* (*cgroup core*), которое образует в ядре ОС точку монтирования *cgroup*;
- *набора подсистем*, которые реализуют отдельные функциональные возможности созданного механизма.

Модель *cgroup core* опирается на современную парадигму псевдофайловых систем, для чего была разработана специальная файловая система *cgroup*, что хорошо показано на рисунке 1.1.

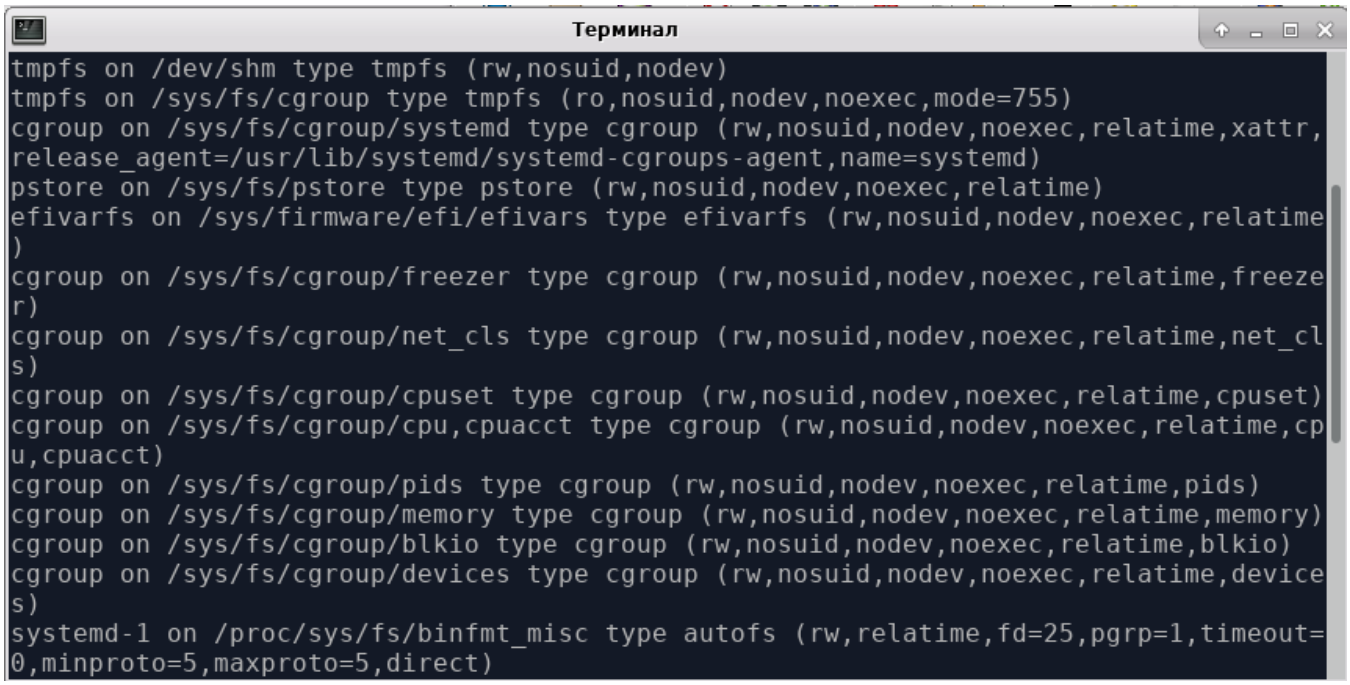

Рисунок 1.1 — Часть вывода команды mount, отражающая монтирование различных подсистем к точке ядра cgroup

Набор подсистем ***cgroups***, реализованных и функционирующих в конкретной ОС,

можно посмотреть командой вывода списка файлов, показанной на рисунке 1.2.

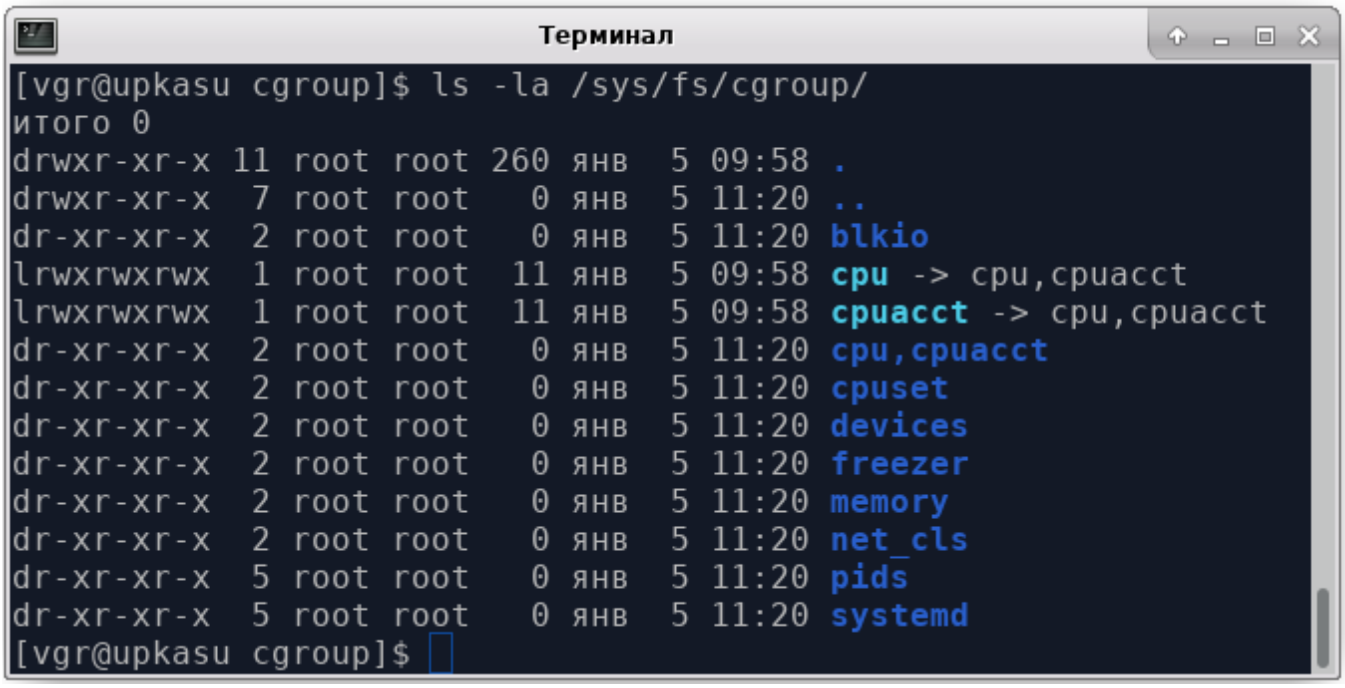

Рисунок 1.2 — Набор подсистем cgroup, реализованный в текущей версии ОС

Хорошо видно, что в текущей версии реализовано 9 подсистем:

- *blkio* — устанавливает лимиты на чтение и запись для блочных устройств;
- *cpuacct* — генерирует отчеты об использовании ресурсов процессора;
- *cpu* — обеспечивает доступ процессов в рамках контрольной группы к CPU (центральному процессору);
- *cpuset* — распределяет задачи в рамках контрольной группы между процессорными ядрами;
- *devices* — разрешает или блокирует доступ к устройствам;
- *freezer* — приостанавливает и возобновляет выполнение задач в рамках контрольной группы;
- *memory* — управляет выделением памяти для групп процессов;
- *net_cls* — помечает сетевые пакеты специальным тэгом, что позволяет идентифицировать пакеты, порождаемые определенной задачей в рамках контрольной группы;
- *pids* — используется для ограничения количества процессов в рамках контрольной группы;
- *systemd* — контрольная группа, управляемая главным родительским процессом systemd.

Реализация набора подсистем *cgroups* также выполнена в парадигме псевдофайловых систем.
Каждая подсистема представляет собой директорию с управляющими файлами, в которых прописываются все настройки.
В каждой из этих директорий имеются следующие управляющие файлы:

- *cgroup.clone_children* — позволяет передавать дочерним контрольным груп-

пам свойства родительских групп;
- *tasks* — содержит список PID всех процессов, включенных в контрольные группы;
  cgroup.procs — содержит список *TGID* (**Talk Group ID**) групп процессов, включенных в контрольные группы (разговорные группы);
- *release_agent* — содержится команда, которая будет выполнена, если включена опция *notify_on_release*; может также использоваться, например, для автоматического удаления пустых контрольных групп;
- *notify_on_release* — содержит булеву переменную 0 или 1, включающую (или наоборот отключающую), выполнение команды, указанной в *release_agent*.

Имеются и другие управляющие файлы.
В общем случае, у каждой подсистемы имеются также собственные управляющие файлы.

Чтобы создать контрольную группу, достаточно создать вложенную директорию в любой из подсистем.
В эту директорию будут автоматически добавлены управляющие файлы.
Чтобы добавить процессы в группу, нужно просто записать их PID в управляющий файл *tasks*.
Совокупность контрольных групп, встроенных в подсистему, называется *иерархией*.

1.2.4 Недостатки cgroups

На протяжении всех лет существования механизм ***cgroups*** неоднократно подвергался критике.
Причины такой критики следующие:
- *встраивать контрольную группу* в каждую подсистему по отдельности очень неудобно;
- *имеются нелогичности* в наследовании родительских и дочерних групп: например, если мы создаём вложенную контрольную группу, то в некоторых подсистемах настройки родительской группы наследуются, а в некоторых — нет; в подсистеме cpuset любое изменение в родительской контрольной группе автоматически передаётся вложенным группам, а в других подсистемах такого нет и нужно активировать параметр *clone.children*.

Разговоры об устранении этих и других недостатков ***cgroups*** ведутся с начала ***2012 года***.
Течжен Хе, инженер Facebook, прямо указал, что ***главная проблема*** *cgroups* заключается в неправильной организации, при которой подсистемы подключаются к многочисленным иерархиям контрольных групп.
Он предложил использовать одну и только одну иерархию, а подсистемы добавлять для каждой группы отдельно.
Такой подход повлек за собой серьезные изменения в реализации этого проекта.

В частности, новый проект стал называться *cgroup*.

Замечание

В настоящее время в реализациях ОС присутствуют оба подхода и их сложно разделить. В общем случае, это направление находится в интенсивном развитии, что отражается в других вопросах изучаемой темы.

1.3 Система виртуализации на уровне ОС

Ранее, в курсе «Операционные системы», мы говорили, что любая ОС реализованная для какой-либо аппаратной платформы создает некоторую «*виртуальную машину*», которая:

- *срывает от пользователя* особенности реализации архитектуры конкретной ЭВМ;
- *создает для языков программирования* единое представление об ЭВМ, которое программист использует для реализации своих приложений.

Обобщая имеющиеся представления, мы приходим к ряду определений, которые на макроуровне присутствуют во всех вычислительных системах.
Виртуальная машина (*VM, virtual machine*) — программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой *платформы*.
Платформа — это выделенная и объявленная архитектура ЭВМ, которая в пределах модели *VM* разделяется на:

- *target-платформа* — целевая или гостевая платформа, ориентированная на исполнение приложений пользователя;
- *host-платформа* — платформа-хозяин, создающая среду функционирования target-платформ и изолирующая их друг от друга.

Технология виртуализации — общий научно-технический подход, тесно связанный с понятием моделирования, являющийся основным трендом современного развития компьютерных технологий.
В плане реализации, виртуальные технологии подразделяются на три большие группы:

- *среды языков программирования*, среди которых можно выделить: Java Virtual Machine (JVM), более известной как платформа Java, и Common Languege Runtime (CLR), являющаяся основой технологии .NET корпорации Microsoft;
- *операционные системы и гипервизоры*, среди которых широко известны: KVM и OpenVZ — Linux, Hyper-V — Microsoft, VM/CMS — IBM, VMware ESX, Xen и другие;
- *автономные эмуляторы компьютеров*: Virtual PC, QEMU, VirtualBox, VMware Fusion, VMware Workstation и так далее.

Современный спектр технологий виртуализации достаточно обширен и имеет разную прикладную направленность.

В декабре 2005 года, корпорация Red Hat приступила к реализации проекта, получившего название *libwirt*.

libvirt — свободная, кроссплатформенная библиотека управления виртуализацией, написанная на языках C и C++.

На рисунке 1.3 представлена общая идея использования этой библиотеки, где:

- *снизу* показаны поддерживаемые виртуальные машины и гипервизоры;
- *сверху* показаны приложения для управления виртуализацией с помощью библиотеки ***libvirt***.

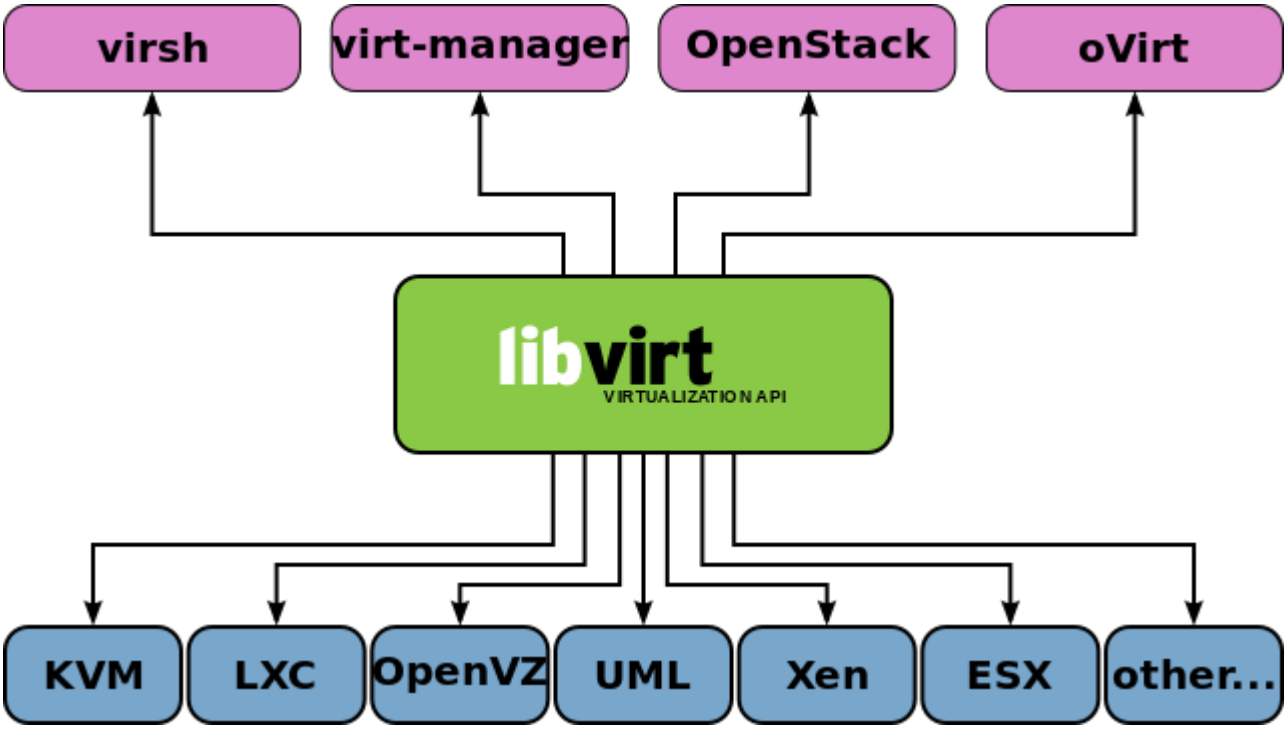

Рисунок 1.3 — Схема взаимодействия виртуальных машин и управляющих приложений через библиотеку libvirt

Преимущества использования систем виртуализации на уровне *виртуальных машин* — вполне очевидны:

- *максимально возможная изоляция* не только групп процессов или отдельных приложений, но и целых операционных систем, обеспечивающая им уникальную и защищенную среду исполнения;
- *эффективное использование ресурсов* «больших ЭВМ» при исполнении приложений, требующих ограниченное количество разнообразных по составу вычислительных ресурсов.

Недостатки систем виртуализации:

- *повышенный расход ресурсов* ЭВМ на создание среды исполнения приложений;
- *зависимость разрабатываемых приложений* от большого объема стороннего ПО достаточно сложной реализации.

Стремление устранить недостатки существующих систем виртуализации порождает альтернативные подходы, среди которых следует выделить:

- *технологии гипервизорных ядер* ОС;

- *механизмы контейнеризации* приложений.

Указанные технологии рассмотрим на двух примерах современных решений ОС Linux.

1.3.1 Система KVM

Гипервизор (*Hypervisor*) — *монитор виртуальных машин*: программа или аппаратная система, обеспечивающая параллельное выполнение нескольких ОС на одном и том же хост-компьютере.

В концептуальном плане, выделяют три типа гипервизоров:

- *тип 1, X* — *автономный гипервизор*, имеющий свои встроенные драйверы устройств и планировщик, независящие от ОС хост-платформы; например, VMware ESX, Citrics, XenServer и KVM;
- *тип 2, V* — на основе базовой ОС, работающий в одном кольце (кольцо 0) с основной ОС; например, VMware Workstation, QEMU и VirtualBox;
- *тип 1+* — *гибридный*, состоящий из двух частей: *тонкого гипервизора* (кольцо 0), контролирующего процессор и память, и *сервисной ОС*, работающей в кольце пониженного уровня; например, Microsoft Hyper-V и Xen.

Гипервизор KVM (Kernel-based Virtual Machine), общая схема которого показана на рисунке 1.4, начал разрабатываться Ави Кивити (Avi Kivity), сотрудника фирмы Qumranet, и ***в 2007 году*** был портирован в ядро Linux версии 2.6.20.

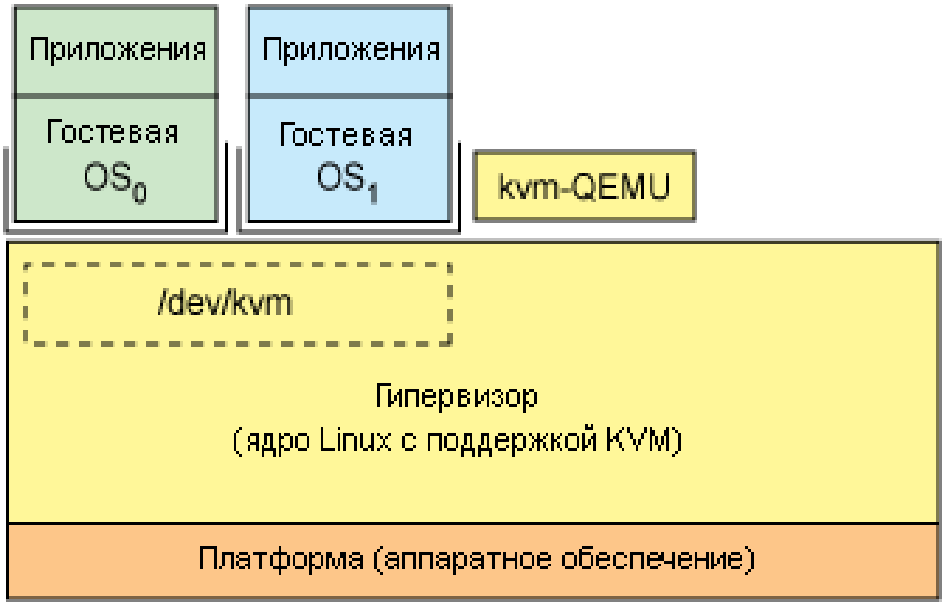

Рисунок 1.4 - Общая схема гипервизора KVM

KVM относится к гипервизорам *типа 1* и предназначен для создания виртуальной среды исполнения для аппаратной платформы ***x86***, которая поддерживает аппаратную виртуализацию Intel VT либо AMD SVM.

Для проверки наличия аппаратной виртуализации процессора, следует воспользо-

ваться командой

```
cat /proc/cpuinfo
```

и проверить наличие флагов:

- *vmx* — для Intel Virtualization Technology;
- *svm* — для AMD Secure Virtual Machine.

Реализация KVM содержит три основные компоненты:

- *универсальный модуль ядра kvm.ko*, который после загрузки требует загрузки одного из платформозависимых модулей: *kvm-intel.ko* или *kvm-amd.ko*;
- *модифицированного эмулятора QEMU* (*kvm-QEMU*), который выполняется как процесс в пользовательском пространстве и согласовывает с ядром запросы гостевой операционной системы;
- *утилита kvm*, которая загружает новые (гостевые) операционные системы.

Сам KVM не выполняет эмуляции. Для каждой гостевой операционной системы с помощью устройства `/dev/kvm` выполняется отображение из собственного виртуального адресного пространства гостевой системы в физическое адресное пространство ядра основной (хостовой) системы.
KVM в состоянии:

- *запускать* в качестве гостевых операционных систем 32-битные и 64-битные системы на базе ядра Linux, Windows и других систем;
- *использовать* немодифицированные образы дисков QEMU, VMware и других, содержащих операционные системы;
- *обеспечивает* каждой виртуальной машине своё собственное виртуальное аппаратное обеспечение: сетевые карты, диски, видеокарты и другие устройства.

1.3.2 Система LXC

LXC (*Linux Containers*) — система виртуализации на уровне ОС для запуска нескольких изолированных экземпляров операционной системы Linux на одном узле.
LXC использует технологию *cgroups* и создает изолированное виртуальное окружение с собственным пространством процессов и сетевым стеком.
Все экземпляры LXC используют один экземпляр ядра операционной системы.

Поскольку LXC не использует виртуальные машины, то значительно снижается непроизводительные расходы ресурсов ЭВМ и повышается быстродействие работы ПО приложений.
Основное назначение подобной системы виртуализации:

- *контейнерная изоляция приложений*, которая опирается на мощные средства технологии cgroups;
- *совместное использование кроссплатформенных приложений*, что обеспечивается пакетом *qemu-user-static*, содержащим множество эмуляторов популяр-

ных архитектур: ARM, CRIS, i386, M68k (ColdFire), MicroBlaze, MIPS, PowerPC, SH4, SPARC и x86-64.

В общем случае, технология LXC:

- *требует установки* дополнительного программного обеспечения в дистрибутив ОС, а также проведение совсем не тривиальных настроек, в зависимости от направленности ее применения;
- *является конкурентной альтернативой* контейнерным технологиям *Docker* и *systemd-nspawn*.

1.4 Технология systemd

В современных ОС Linux, главным родительским процессом и системой инициализации является *systemd*.
В апреле 2011 года, Леннарт Поттеринг опубликовал развернутое сравнение систем инициализации *systemd, upstart и SysVinit*, которое можно найти в Интернете по адресу: http://0pointer.de/blog/projects/why.html.
В таблице 1.1, приведены результаты этого сравнения, взятые из перевода статьи, опубликованного по адресу: http://www.opennet.ru/opennews/art.shtml?num=30412.

Таблица 1.1 — Сравнительные характеристики sysvinit, upstart и systemd

	sysvinit	**Upstart**	**systemd**
Управление через D-Bus	нет	да	да
Запуск без использования bash/shell скриптов	нет	нет	да
Включены сервисы ранней стадии загрузки, написанные на языке C	нет	нет	да
Возможность упреждающего чтения данных с диска	нет	нет[1]	да
Активация сервисов на основе сокетов	нет	нет[1]	да
Активация сервисов на основе сокетов: совместимость с inetd	нет	нет[2]	да
Активация на основе шины (Bus-based Activation)	нет	нет[3]	да
Активация на основе аппаратуры компьютера	нет	нет[4]	да
Конфигурирование зависимостей устройств, используя правила udev	нет	нет	да
Активация по событиям файловой системы (inotify)	нет	нет	да
Активация по времени	нет	нет	да
Управление точками монтирования	нет	нет[5]	да
Управление запуском fsck	нет	нет[5]	да
Управление квотами	нет	нет	да

Управление автомонтированием	нет	нет	да
Управление SWAP	нет	нет	да
Сохранение снимков состояния системы (snapshotting)	нет	нет	да
Поддержка XDG_RUNTIME_DIR	нет	нет	да
Опциональная остановка процессов пользователя после его выхода из системы	нет	нет	да
Интеграция с Linux Control Groups (cgroups)	нет	нет	да
Генерация событий аудита для запускаемых сервисов	нет	нет	да
Интеграция с SELinux	нет	нет	да
Интеграция с PAM	нет	нет	да
Управление шифрованными разделами и дисками (LUKS)	нет	нет	да
Поддержка обработки паролей к LUKS и SSL-сертификатам, с запросом пароля через такие агенты, как Plymouth, консоли, wall, tty терминалов и GNOME	нет	нет	да
Управление сетевым петлевым устройством (loopback)	нет	нет	да
Управление binfmt_misc (поддержка неродных исполняемых файлов)	нет	нет	да
Управление системной локалью	нет	нет	да
Настройка параметров консоли и клавиатуры	нет	нет	да
Инфраструктура для создания, удаления и чистки временных файлов	нет	нет	да
Управление через `/proc/sys` sysctl	нет	нет	да
Интеграция с plymouth (графическим запуском, используя KMS)	нет	нет	да
Сохранение и восстановление random seed (состояния генератора энтропии)	нет	нет	да
Поддержка статической загрузки модулей ядра	нет	нет	да
Автоматическое управление консолью COM-порта	нет	нет	да
Управление уникальным ID компьютера	нет	нет	да
Управление динамическим именем хоста и метаданными компьютера	нет	нет	да
Контролируемая остановка сервисов	нет	нет	да
Поддержка раннего логгирования через /dev/log	нет	нет	да
Включает минимальный демон логгирования на основе kmsg для встраиваемых систем	нет	нет	да
Перезаупуск сервисов в случае краха без потери соединения	нет	нет	да
Бесшовное обновление сервисов	нет	нет	да
Графический интерфейс пользователя (опциальнально)	нет	нет	да
Встроена поддержка профилирования и расширенных	нет	нет	да

инструментов			
Поддержка сервисов типа "instantiated"	нет	да	да
Интеграция с PolicyKit	нет	нет	да
Есть встроенные утилиты для удалённого доступа и управления кластером	нет	нет	да
Может показать все процессы, принадлежащие сервису	нет	нет	да
Может идентифицировать процессы сервиса	нет	нет	да
Автоматически создаёт cgroups для сервисов для равномерного распределения времени CPU	нет	нет	да
Аналогично для пользовательских процессов	нет	нет	да
Совместимость с SysV	да	да	да
Сервисы SysV контролируются как родные сервисы	да	нет	да
Управление сервисами через /dev/initctl	да	нет	да
Перезапуск сервисов с полной сериализацией (serialization) состояния	да	нет	да
Поддержка интерактивного (управляемого) запуска системы	нет[6]	нет[6]	да
Поддержка контейнеров: как расширенная замена chroot()	нет	нет	да
Загрузка, построенная на основе зависимостей	нет[7]	нет	да
Отключение сервисов без редактирования файлов	да	нет	да
Маскировка сервисов без редактирования файлов	нет	нет	да
Надёжная остановка системы, используя только один процесс	нет	нет	да
Встроенная поддержка перезапуска ядра на лету (kexec)	нет	нет	да
Динамическая генерация сервисов	нет	нет	да
Поддержка в других компонентах ОС	да	нет	да
Файлы запуска сервисов, совместимые с различными дистрибутивами	нет	нет	да
Отправка сигналов сервисам	нет	нет	да
Надёжная остановка пользовательских сессий перед остановом системы	нет	нет	да
Поддержка логгирования в utmp/wtmp	да	да	да
Легкие для написания, расширения и обработки файлы управления сервисами, подходящие для манипулирования инструментами управления предприятием	нет	нет	да

Приведенный список, сам по себе, является утверждением, что современный тренд развития ОС находится в стадии интеграции с уже имеющимися и практически оправдавшими себя технологиями.

Естественным образом, программные средства *systemd* исползуют различные механизмы контейнерных технологий, многие из которых находятся в интенсивной разработке. Далее, кратко будут рассмотрены только три таких средства: *chroot*, *namespaces* и *systemd-nspawn*.

1.4.1 Классическая изоляция chroot

Функционирование любой ОС опирается наличие доступных ей файловых систем, которые, в общем случае, образуют дерево каталогов, регулярных и специальных файлов.
Невозможно запустить команду (программу), если она «невидима» в дереве файловой системы.

В 1979 году, в ОС UNIX были добавлены утилита и системный вызов, с одноименным названием *chroot*, которые позволяли указывать каталог любой файловой системы, делая его корнем для последующих дочерних команд дочерних процессов.
В частности, при загрузке ОС, переключение с временной файловой системы на корневую ФС выполняется с помощью системного вызова *chroot(...)*:

```
    #include <unistd.h>

    int chroot(const char *path);
```

где *path* — указатель на имя полного пути в дереве исходной файловой системы.

Соответственно, утилита *chroot* способна запускать программы, одновременно изменяя видимую корневую директорию и ряд опций запуска:

```
    chroot [OPTION] NEWROOT [COMMAND [ARG]…]

    chroot OPTION

--groups=G_LIST — указывает список групп, как: g1,g2,..,gN ;

--userspec=USER:GROUP — указывает пользователя и группу при запуске;

--skip-chdir — не изменяет корневую директорию  '/' ;

--help - выводит help и заканчивает работу;

--version - выводит информацию о версии и заканчивает работу.
```

Замечание

При использовании этой технологии, необходимо копировать все исполняемые файлы и динамические библиотеки, что приводит к существенному дублированию.
Имеются также существенные недостатки с точки зрения безопасности.

1.4.2 Механизмы контейнеризации namespaces

Пространство имен (*Namespace*) — это механизм ядра Linux, обеспечивающий изоляцию процессов друг от друга.
Работа над этим проектом началась ***в 2002 году***, при реализации ядра Linux версии 2.4.19.
На текущий момент, в ядре Linux поддерживается порядка шести типов пространств имён, перечень и назначение которых представлен в таблице 1.2.

Таблица 1.2 — Список пространств имен ядра Linux

Пространство имен	**Что изолирует**
PID	PID процессов, обеспечивая несколько иерархий учета PID процессов.
NETWORK	Сетевые устройства, стеки, порты..
USER	ID пользователей и групп.
MOUNT	Точки монтирования.
IPC	SystemV IPC, очереди сообщений POSIX.
UTS	Имя хоста и доменное имя компьютера (NIS).
CGROUP	Главная директория Cgroup.

Основное назначение *namespases* — формирование групп процессов, которые имеют раздельный учет и управление по перечисленным в таблице 1.1 пространствам имен.
Большинство из указанных свойств обеспечиваются системным вызовом *clone()*, являющимся своеобразным раширением известного системного вызова *fork()*:

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, void *newtls, pid_t *ctid */ );
```

где *fn* — указатель на произвольную функцию, которая станет основой дочернего процесса, функционирующего в отдельном пространстве имен;
child_stack — указатель на массив данных, который будет использоваться как стек дочернего процесса;

flags — набор флагов (см. ниже), с помощью которых указывается набор пространств имен создаваемого дочернего процесса;
arg — указатель на аргумент, передаваемый функции *fn*.

В случае успешного запуска, *clone()* возвращает номер дочернего процесса, в пространстве номеров родительского процесса.
В случае ошибки, *clone()* возвращает *-1*, а переменная *errno* — содержит код описания этой ошибки.

Основные особенности работы системнного вызова *clone()* определяются аргументом флагов *flags*, используемых в различной комбинации:
SIGCHLD — флаг означающий, что родительский процесс должен получить код завершения дочернего процесса, возвращаемый функцией *fn*;
CLONE_NEWPID — означает, что будет создано новое дерево учета PID процессов, в котором функция *fn* сформирует главный родительский процесс с PID=1; если этот флаг будет отсутствовать, то нумерация дочернего процесса будет аналогична системному вызову *fork()*;
CLONE_NEWNET — будет создано новое пространство имен *NETWORK*;
CLONE_NEWUSER — будет создано новое пространство имен *USER*;
CLONE_NEWNS — будет создано новое пространство имен *MOUNT*;
CLONE_NEWIPC — будет создано новое пространство имен *IPC*;
CLONE_NEWUTS — будет создано новое пространство имен *UTS*;
CLONE_NEWCGROUP — будет создано новое пространство имен *CGROUP*.

В псевдофайловой системе */proc*, для каждого используемого пространства имен, будут формироваться файлы заголовков пространства имен:

```
/proc/[pid]/ns/pid
/proc/[pid]/ns/net
/proc/[pid]/ns/user
/proc/[pid]/ns/mnt
/proc/[pid]/ns/ipc
/proc/[pid]/ns/uts
/proc/[pid]/ns/cgroup
```

где *[pid]* — идентификатор PID, возвращаемый системным вызовом *clone()* родительскому процессу.

Замечание

Для ограничения ресурсов *namespaces* используются механизмы *cgroups*.
Более подробное изучение свойств *namespaces* следует изучать по источникам:

- **man 2 clone**;
- **man 7 namespaces**;
- статье: https://blog.selectel.ru/mexanizmy-kontejnerizacii-namespaces/.

1.4.3 Контейнеры systemd-nspawn

systemd-nspawn представляет собой один из интенсивно развиваемых модулей проекта *systemd*. Оно представляет собой сокращение от *namespaces spawn*.

Основное назначение модуля *systemd-nspawn*, как и проекта *namespaces*, управление изоляцией процессов, но в тесном взаимодействии с systemd, который способен изолировать и ресурсы ЭВМ.
В результате, с помощью *systemd-nspawn* можно создать полностью изолированое окружение, в котором автоматически будут:

- смонтированы псевдофайловые системы */proc* и */sys*;
- созданы изолированный *loopback-интерфейс* и отдельное изолированное *пространство имен* для идентификаторов процессов (PID), внутри которого можно запускать ОС, основанную на ядре Linux.

Далее, можно воспользоваться всей интегрированной мощью системы *systemd* по организации управления созданными контейнерами и хост-системой, в целом.
В частности, для более эффективного управления контейнерами, в рамках проекта *systemd*, развивается утилита *machinectl*.

Замечание

К основным недостаткам проекта *systemd-nspawn* следует отнести:

- специфическая привязка к общему проекту *systemd*;
- специфическая технология формирования образов запускаемых гостевых ОС, к тому же ориентированную на файловую систему *btrfs*;
- неустойчивые по технологиям исполнения и перспективе программные решения, вызванные интенсивным развитием самого проекта, еще далекого от завершения.

Дальнейшее изучение этого подхода можно продолжить по источникам:

- **man systemd-nspawn**;
- **man machinectl**;
- статье: https://blog.selectel.ru/systemd-i-kontejnery-znakomstvo-s-systemd-nspawn/.

1.5 Кластерное управляющее ПО Grid Engine

Возможно, столь сложное управление, основанное на использовании *cgroups*, кажется излишним для ОС рабочих станций, но современные задачи часто бывают настолько большими и сложными, что требуют для своего исполнения:

- *множества компьютеров*, объединенных сетью или специальными каналами связи;
- *участия коллектива исполнителей*, одновременно работающих на разных компьютерах, но решающих общую задачу в виде отдельных заданий.

В таких случаях, говорят о распределенных вычислениях, реализуемых на распределенной вычислительной инфраструктуре.
С учетом того, что любая программа (процесс) выполняются в некоторой вычислительной среде, которая всегда контролирует такие параметры как дерево доступной *файловой системы* и *пользователя* (группы), использование механизмов *cgroups* становится особенно актуальным.

Одним из способов организации распределенных вычислений являются *кластерные решения*.
Кластер — группа компьютеров, объединенных высокоскоростными каналами связи и представляющих, с точки зрения пользователя, единый аппаратный ресурс.
Кластер - слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений, и представляющихся пользователю единой системой.
Грид-вычисления (*grid* — решётка, сеть) — это форма распределенных вычислений на базе «*виртуального суперкомпьютера*» представленого в виде кластеров слабосвязанных гетерогенных компьютеров, работающих вместе для выполнения большого количества заданий (операций, работ).

Замечание

Организация распределенных вычислений, прежде всего, порождает две проблемы:

- *адресация* вычислительного ресурса (узла, cell), выполняющего вычисления;
- *обеспечение надежности* вычислений, порожденных отказом или недоступностью нужного вычислительного ресурса (вычислительного узла).

Первая проблема решается выделением главного хоста, который занимается распределением отдельных заданий между компьютерами кластера.
Вторая проблема решается перераспределением заданий отказавших узлов на другие (рабочие) узлы кластера.

Классическим примером реализации кластерных технологий является ПО *Sun Grid Engine*.
В 2000 году, корпорация *Sun Microsistems* приобрела фирму *Gridware*, на основе продуктов которой и была реализована технология *Grid*.
В 2010 году, корпорация *Sun Microsistems* была куплена корпораций *Oracle* и продукт был переименован в *Oracle Grid Engine*.

В 2013 году, корпорация *Oracle* продала все права на этот продукт фирме *Univa* и система переименована в *Univa Grid Engine*.

На рисунке 1.5 показан общий алгоритм обработки заданий на главном хосте системы Grid Engine.

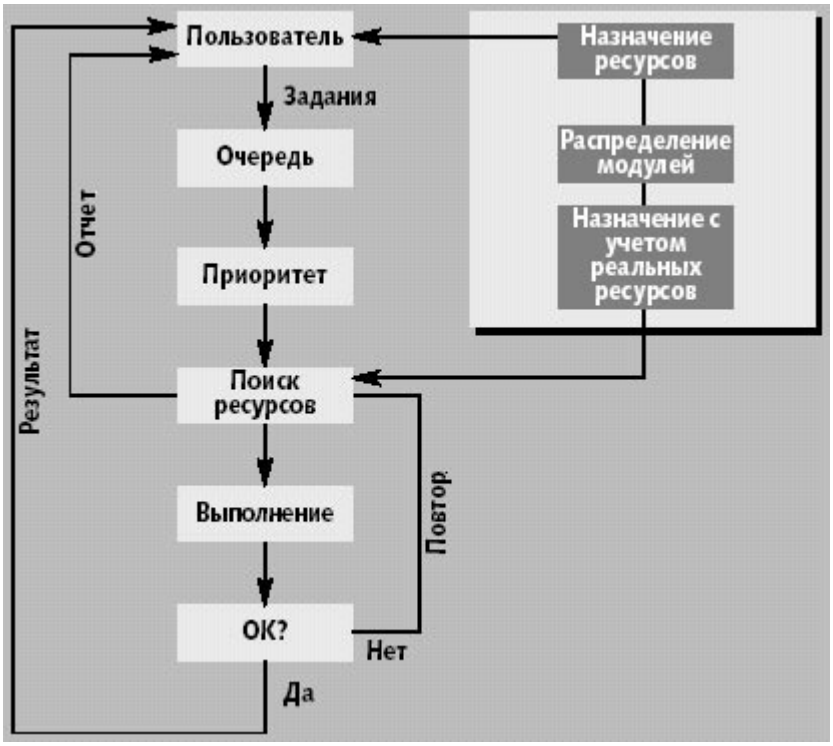

Рисунок 1.5 — Схема алгоритма распределения заданий системы Grid Engine

К главным недостатку реализации ПО *Grid Engine* следует отнести необходимость регистрации всех пользователей системы на всех компьютерах кластера, что существенно увеличивает объем административной работы и снижает качество работы всей системы.
Таким образом, разработки технологий *cgroups* и *namespaces* являются весьма актуальными в системах распределенных вычислений.

2 Лабораторная работа №4

Лабораторная работа №4 посвящена практическому освоению технологий, предоставляемых механизмами *cgroups* и *namespaces*.

2.1 Практическое использование cgroups

В этой части лабораторной работы, мы рассмотрим ряд простых примеров, обеспечивающих ограничение ресурсов процесса для некоторой учебной группы с именем *group0*.
Все работы будут проводиться в командной строке терминала, поэтому следует запустить виртуальный терминал от имени пользователя *root*.

Учебная цель данного подраздела — управление ресурсами процессов.
В частности, запущенный в терминале интерпретатор *bash* является процессом, узнать номер PID которого можно командой: **`echo $$`**, например, как показано на рисунке 2.1.

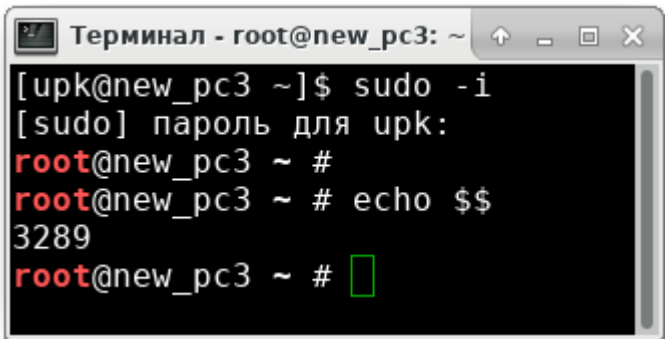

Рисунок 2.1 — Получение номера текущего интерпретатора bash

Замечание
Перед выполнением конкретных заданий данного подраздела, следует внимательно изучить теоретическую часть, изложенную в подразделе 1.2 данного руководства.
Все примеры данного подраздела заимствованы из статьи, размещенной по электронному адресу: https://habrahabr.ru/company/selectel/blog/303190/, с содержанием которой также желательно ознакомиться.

2.1.1 Управление памятью процесса

Часто, процесс или группу процессов следует ограничить по объему используемой памяти ЭВМ.
Попробуем ввести такие ограничения для текущего процесса *bash*.

Для управления ресурсами памяти, в *cgroups* имеется подсистема *memory*.
Чтобы создать в подсистеме ***memory*** контрольную группу *group0*, необходимо просто создать директорию `/sys/fs/cgroup/memory/group0`.
Автоматически, в ней будет создан набор управляющих файлов, как показано на рисунке 2.2.

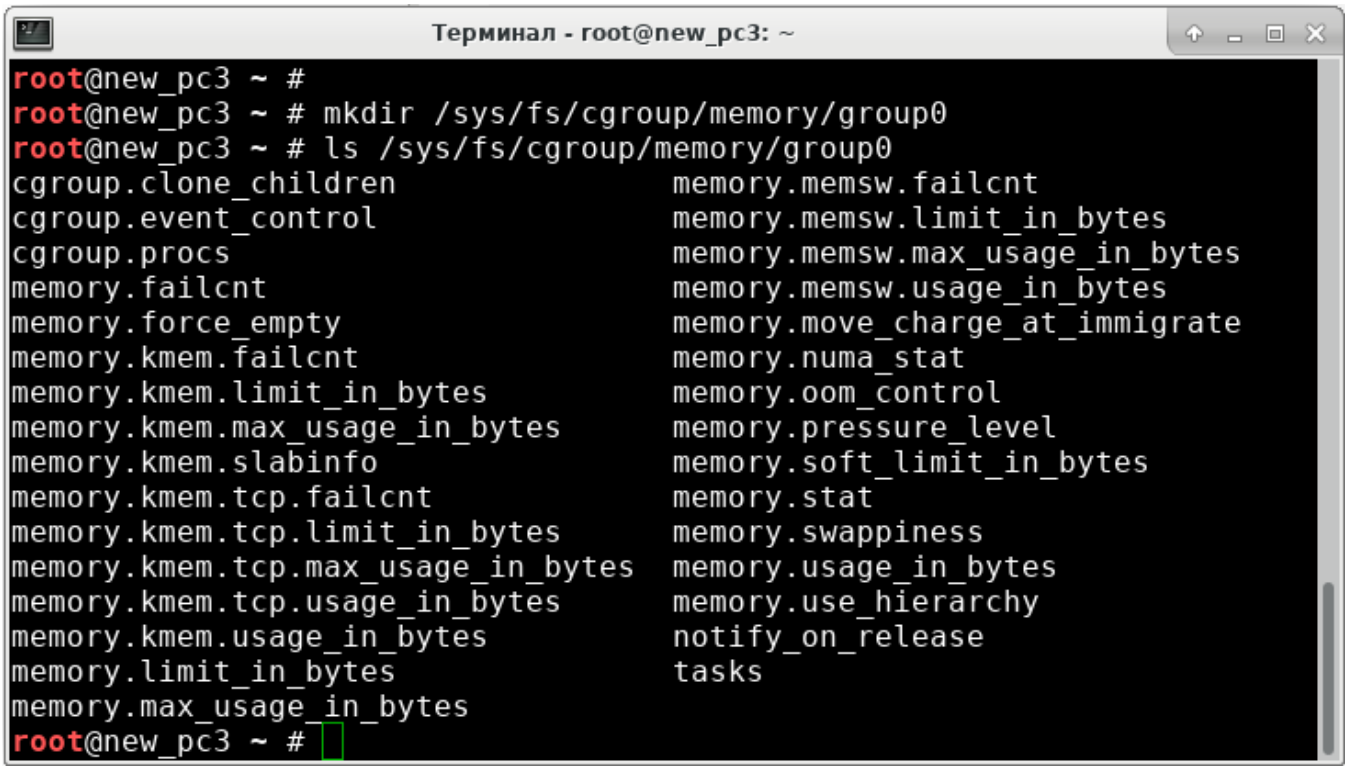

Рисунок 2.2 — Управляющие файлы контрольной группы group0 для подсистемы memory

Чтобы добавить текущий процесс в созданную контрольную группу, нужно записать его PID в файл *tasks*, например:

```
echo $$ > /sys/fs/cgroup/memory/group0/tasks
```

Соответственно, прочитать содежимое этого файла сожно командой:

```
cat /sys/fs/cgroup/memory/group0/tasks
```

Убедитесь, что номер процесса записался в файл *tasks*.

Замечание

Каждый номер контролируемого процесса должен быть записан в отдельную строку, но поскольку мы имеем дело с псеводофайлом, то добавление выполнятся командой:

```
echo [pid] > /sys/fs/cgroup/memory/group0/tasks
```

где [pid] — номер добавляемого процесса.

Для ограничения контрольной группы *group0* по потреблению памяти, необходимо записать соответствующий лимит в файле *memory.limit_in_bytes*, например, как показано на рисунке 2.3.

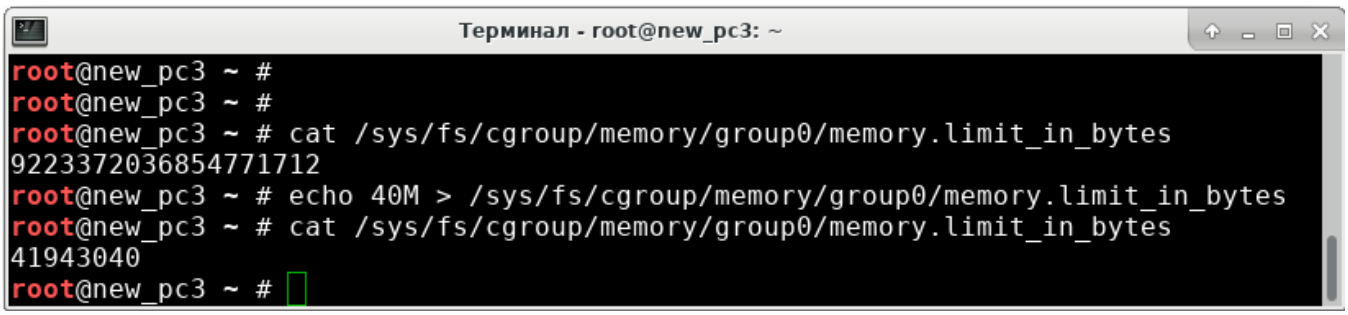

Рисунок 2.3 — Изменение лимита по памяти для группы group0

Указанные изменения будут распространяться на все процессы, включенные в группу *group0*.

2.1.2 Отключение действия на процесс OOM-Killer

Как было отмечено в пунке 1.2.2 данного руководства, когда оперативная память заканчивается в ОЗУ ЭВМ и в пространстве свопинга, специальный модуль ядра ОС, называемый ***OOM-Killer***, может удалить нужный нам процесс.
Чтобы этого не случилось, необходимо отключить ***OOM-контроль***, например, как показано на рисунке 2.4.

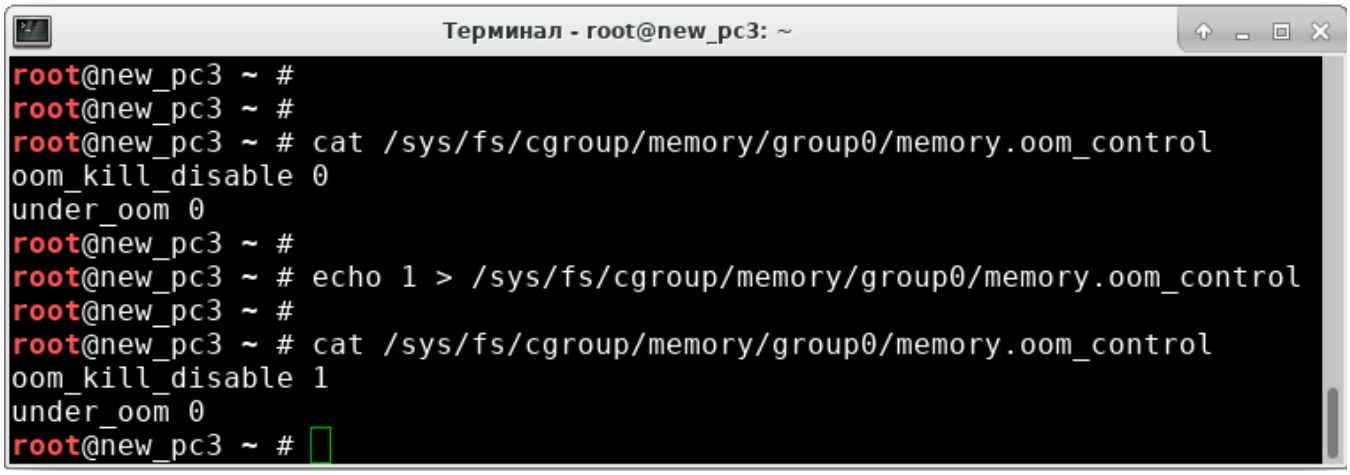

Рисунок 2.4 — Отключение OOM-контроля над группой group0

В случае, когда контроль подсистемы над группой уже ненужен, следует:

- *перенести* все контролируемые процессы в другую группу, например, в родительскую;
- *удалить* директорию контрольной группы из соответствующей подсистемы.

Например, на рисунке 2.5 показана процедура удаления контрольной группы *group0* из подсистемы *memory*.

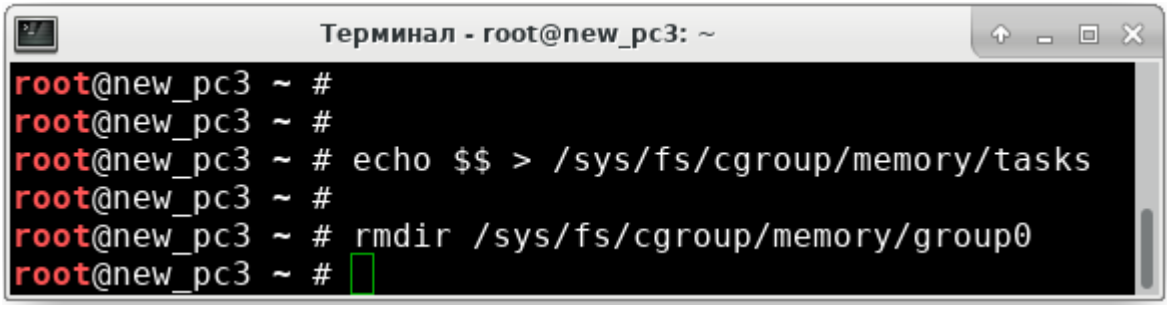

Рисунок 2.5 — Удаление контрольной группы group0 из подсистемы memory

2.1.3 Управление устройствами

В общем случае, ОС обеспечивает всем процессам равные условия для доступа к устройствам ввода-вывода, ограниченное уже известными правами:

- владелец, группа, другие;
- возможности: чтения, записи и запуска.

В ряде случаев, необходимо группе процессов *дополнительно* запретить или разрешить доступ к устройствам.
Для этих целей используется подсистема *devices*.

Создадим в подсистеме *devices* контрольную группу *group0*, выведем список управляющих файлов и поместим в эту группу процесс текущего командного интерпретатора и, как показано на рисунке 2.6.

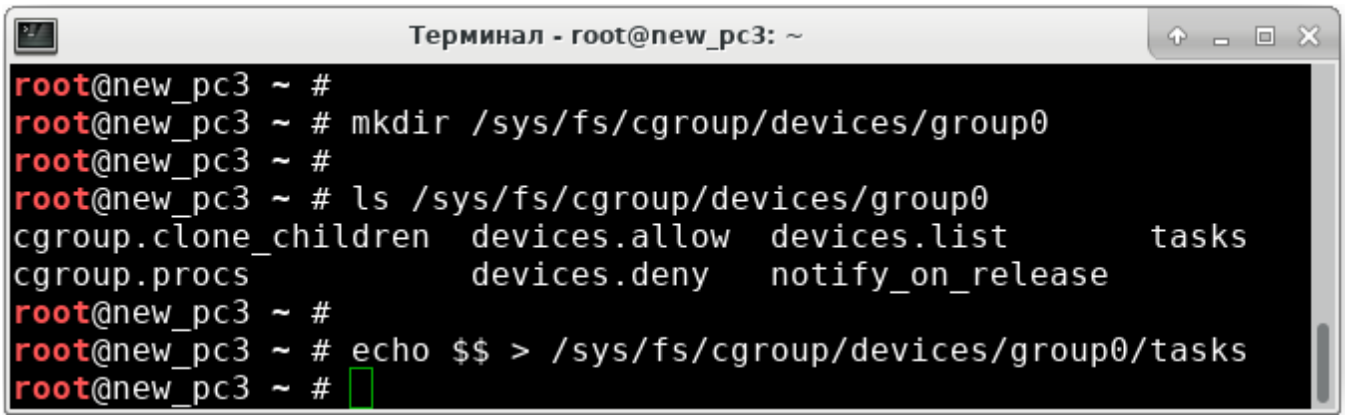

Рисунок 2.6 — Создание контрольной группы group0 для текущего процесса bash в подсистеме devices

Для управления ограничениями созданной группы достаточно использование файлов:

- *devices.list* — список текущего состояния устройств (только для чтения);
- *devices.allow* — запись команды разрешений;

- *devices.deny* — запись команды запретов.

Общий формат записей в этих файлах:

```
    тип major:minor rwm
```

где ***тип*** — принимает одно из значений:

- *a* — все устройства;
- *c* — символьные устройства;
- *p* — каналы;

major:miner — старший и младший номера устройств;
rwm — набор разрешений (ограничений):

- *r* — чтение из устройства;
- *w* — запись в устройство;
- *m* — создание устройства.

Непосредственные эксперименты проведем для устройства */dev/null*, которое в общем случае обеспечивает запись в него для всех процессов.

Шаг 1. Определим характеристики устройства */dev/null*, возможность записи в него и состояние доступа к устройствам для группы *group0*, как показано на рисунке 2.7.

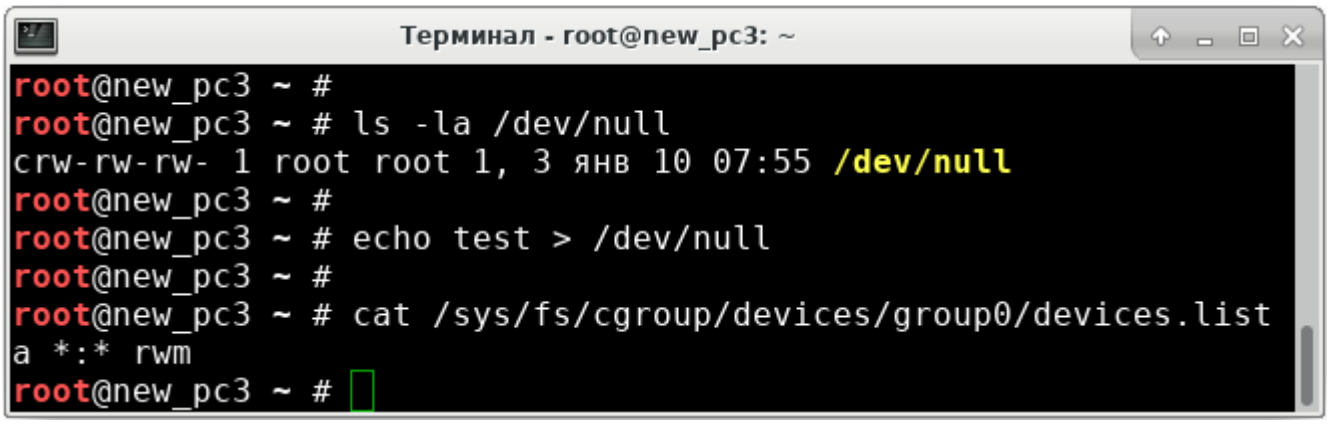

Рисунок 2.7 — Первый шаг эксперимента

Хорошо видно, что:

- /dev/null — символьное устройство, major=1, minor=3, запись разрешена для всех пользователей;
- запись в устройство проходит без ошибочных сообщений;
- текущему процессу доступны все типы устройств на чтение, запись и создание устройств.

Шаг 2. Введем для группы *group0* запрет на все возможные действия с устройством */dev/null* и повторим эксперимент, как показано на рисунке 2.8.

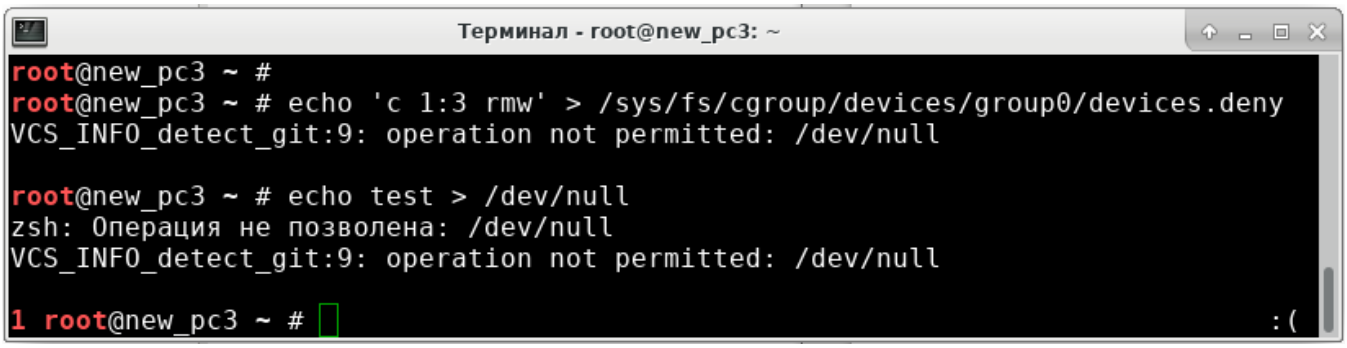

Рисунок 2.8 — Второй шаг эксперимента

Хорошо видно, что мы получаем сообщения о запрете записи в устройсво */dev/null*.

Шаг 3. Вернем первоначальные установки для группы *group0* на работу с устройством */dev/null* и повторим эксперимент, как показано на рисунке 2.9.

Рисунок 2.9 — Третий шаг эксперимента

Результат эксперимента говорит сам за себя.

Замечание

После проведения экспериментов, следует удалить group0 из подсистемы devices. Делается это также, как и для группы memory:

- сначала номера процессов переносятся в родительскую подсистему;
- затем, удаляется директория: /sys/fs/cgroup/devices/group0.

2.2 Практическое использование namespaces

Технология *namespaces* позволяет изолировать процессы в разных именных пространствах.
В данной части лабораторной работы проводятся эксперименты с изоляцией PID процессов и изоляция файловой системы.

2.2.1 Изоляция PID процессов

Для демонстрации примера изоляции ***PID*** процессов, обратимся к исходному тексту программы, представленной на листинге 2.1, в которой:

- из головной функции *main()*, с помощью системных вызовов *clone()*, создаются два процесса, порожденные функциями *child1()* и *child2()*;
- первый вызов *clone()* осуществляется без флага *CLONE_NEWPID*, а второй — с использованием этого флага;
- обе функции *child1()* и *child2()* печатают свой ***PID*** и ***PID*** родительского процесса.

Листинг 2.1 — Исходный текст проекта lab4_1

```
/*
 ============================================================================
 Name        : lab4_1.c
 Author      : Reznik V.G., 10.01.2017
 Version     :
 Copyright   : Your copyright notice
 Description : Hello World in C, Ansi-style
 ============================================================================
 */

#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

static char stack1[1048576];
static char stack2[1048576];

static int child1() {
  printf("\tchild1: PID = %ld\n", (long)getpid());
  printf("\tparent: PID = %ld\n", (long)getppid());
  return 0;
}

static int child2() {
  printf("\tchild2: PID = %ld\n", (long)getpid());
  printf("\tparent: PID = %ld\n", (long)getppid());
```

```
  return 0;
}

int main() {
  puts("\nЗапуск без флага CLONE_NEWPID ...");
  pid_t child_pid1 = clone(child1, stack1+1048576,
            SIGCHLD, NULL, 0);
  printf("main: clone(): PID = %ld\n", (long)child_pid1);

  waitpid(child_pid1, NULL, 0);

  puts("\nЗапуск с флагом CLONE_NEWPID ...");
  pid_t child_pid2 = clone(child2, stack2+1048576,
            CLONE_NEWPID | SIGCHLD, NULL, 0);
  printf("main: clone(): PID = %ld\n", (long)child_pid2);

  waitpid(child_pid2, NULL, 0);

  return 0;
}
```

На первом этапе исследования:

- запустим на рабочем столе систему разработки Eclipse для языка C;
- создадим проект с именем *lab4_1*;
- перенесем в него исходный текст листинга 2.1;
- создадим исполняемый файл *lab4_1*, проведя компиляцию исходного текста программы.

Замечание

На данном этапе следует учесть два момента:

- на этапе компиляции и создании проекта, среда разработки будет негативно реагировать на флаг *CLONE_NEWPID* системного вызова *clone()*;
- не следует отлаживать и запускать программу *lab4_1* из среды разработки Eclipse, поскольку это следует делать от имени пользователя ***root***.

На втором этапе исследования, следует:

- запустить виртуальный терминал, в котором запустить файловый менеджер ***mc*** от имени пользователя ***root***;
- перейти в файловом менеджере в директорию: *~/workspaceC/lab4_1/Debug*;
- запустить исполняемый файл проекта командой: *./lab4_1*

В результате указанных действий, будет получен результат подобный тому, что показан на рисунке 2.10.

Студенту следует разобраться с работой программы и дать адекватное описание полученного результата, описав его в личном отчете.

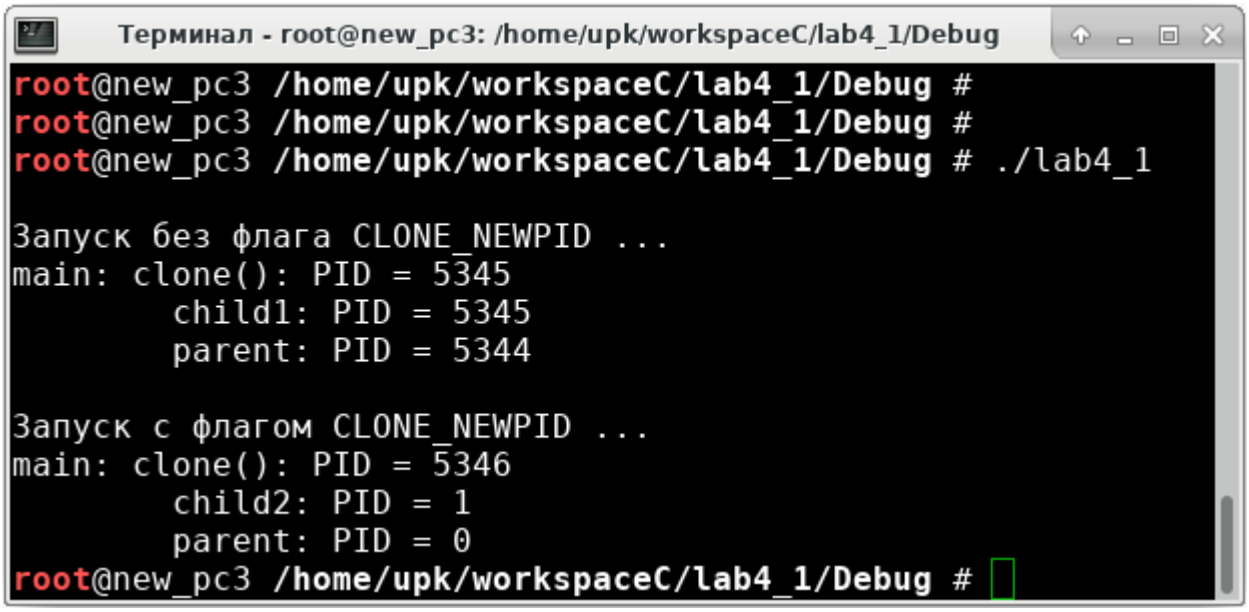

Рисунок 2.10 — Результат вывода программы lab4_1

2.2.2 Изоляция файловой системы процесса

В случае, когда необходимо, чтобы процессы работали с некоторой директорией, к которой монтированы разные файловые системы, можно воспользоваться *механизмом изоляции файловых систем*.
Один из способов такой изоляции — использование системного вызова *clone()* с флагом *CLONE_NEWNS*.

На листинге 2.2 приведен исходный текст программы, которая:

- создает слон процесса с флагами: `CLONE_NEWNS | CLONE_NEWPID`;
- функция *child2()*, реализующая алгоритм работы клонированного процесса, сначала выводит ***PID*** — свой и родительского процесса, а затем заменяет свое тело на интерпретатор ***bash***, с помощью системного вызова *execve()*.

Таким образом, мы получаем программный инструмент, позволяющий нам провести нужные эксперименты.

Листинг 2.2 — Исходный текст проекта lab4_2

```
/*
 ============================================================================
 Name        : lab4_2.c
 Author      : Reznik V.G., 10.01.2017
 Version     :
 Copyright   : Your copyright notice
 Description : Hello World in C, Ansi-style
 ============================================================================
 */

#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/mount.h>
#include <unistd.h>

static char stack2[1048576]; // Массив стека
char * args[] = {"bash", NULL};     // Аргументы программы bash
extern char **environ;              // Внешняя переменная с окружением

static int child2() {
  printf("\tchild2: PID = %ld\n", (long)getpid());
  printf("\tparent: PID = %ld\n", (long)getppid());

  execve("/bin/bash", args, environ);

  return 0;
}

int main() {
  puts("\n3апуск clone() с флагом CLONE_NEWPID | CLONE_NEWPID | SIGCHILD...");
  pid_t child_pid2 = clone(child2, stack2+1048576,
              CLONE_NEWNS | CLONE_NEWPID | SIGCHLD, NULL, 0);
  printf("main:  clone(): PID = %ld\n", (long)child_pid2);

  waitpid(child_pid2, NULL, 0);

  return 0;
}
```

На первом этапе исследования, следует:

- запустить на рабочем столе систему разработки Eclipse для языка C;
- создать проект с именем *lab4_2*;
- перенести в него исходный текст листинга 2.2;
- создать исполняемый файл *lab4_2*, проведя компиляцию исходного текста программы.

Замечание

На данном этапе следует игнорировать предупреждающие сообщения по семантической неопределенности использования флагов системного вызова *clone()*.

На втором этапе — определимся с методикой экспериментов:

- обычно, директория */mnt* используется администраторами ОС для служебных целей, поэтому она является пустой, в чем можно убедиться простым ее просмотром в файловом менеджере;
- клонированный, с помощью программы *lab4_2*, процесс будет видет видеть копию той же файловой системы, но из своего именованного пространства;
- клонированный процесс является интерпретатором ***bash***, поэтому с помощью утилиты ***mount*** можно монтировать в директорию */mnt* любую файловую систему;
- сравнивая «видимости» обычного и клонированного процессов, можно экспериментально убедиться в правильности теоретических утверждений.

Замечание

Главный управляющий процесс *systemd* делает файловые системы обычного и клонированного процесса — разделяемыми, поэтому необходимо дополнительно воспользоваться командой:

```
mount --make-rprivate /
```

которая сделает файловую систему клонированного процесса *неразделяемой* с остальными процессами.

В качестве монтируемой файловой системы будет использоваться flashUSB студента, поэтому следует:

- подключить к ЭВМ flashUSB;
- отмонтировать это устройство (не извлекая его);
- определить имя устройства с помощью команды: ***sudo fdisk -l***

Далее, приводится пошаговое описание эксперимента, предполагая, что flashUSB имеет имя: */dev/sdc1*

Шаг 1. Запустим виртуальный терминал и от имени пользователя ***root*** выполним:

- запуск программы lab4_2;
- укажем, что файловые системы являются не разделяемыми (no shared);
- монтируем файловую систему flashUSB в директорию /mnt.

Результат указанных действий показан на рисунке 2.11.

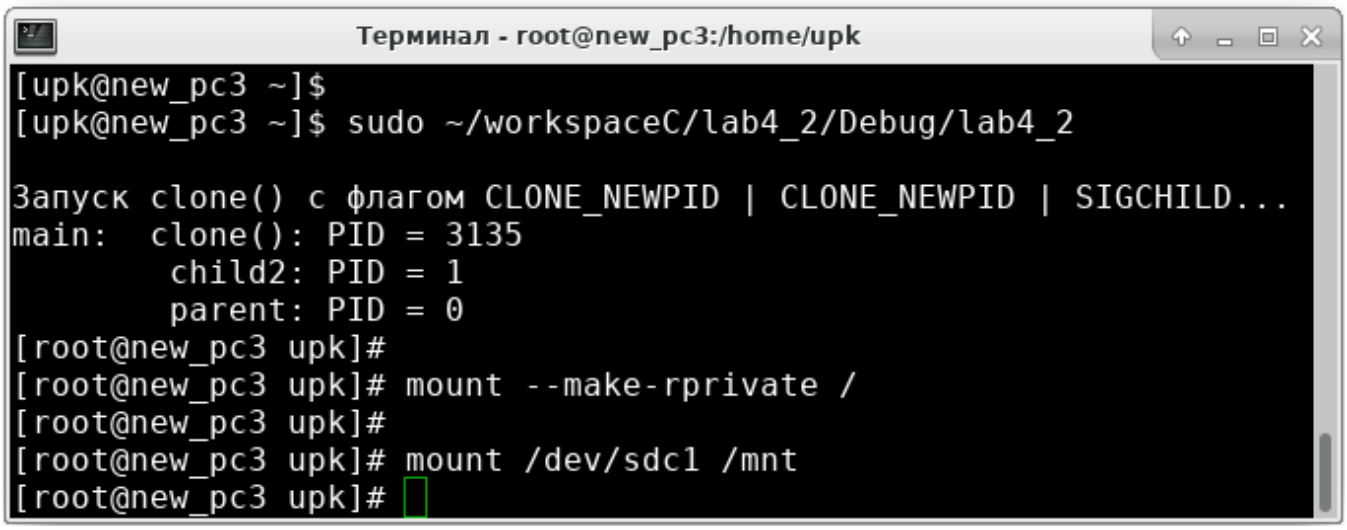

Рисунок 2.11 — Действия по алгоритму Шаг 1

Шаг 2. Проверим видимость монтирования файловых систем клонированным и обычным процессами:

- выполним команду ***mount*** в терминале клонированного процесса; результат показан на рисунке 2.12;
- запустим новый виртуальный терминал и выполним команду ***mount*** (для

обычного процесса); результат показан на рисунке 2.13;

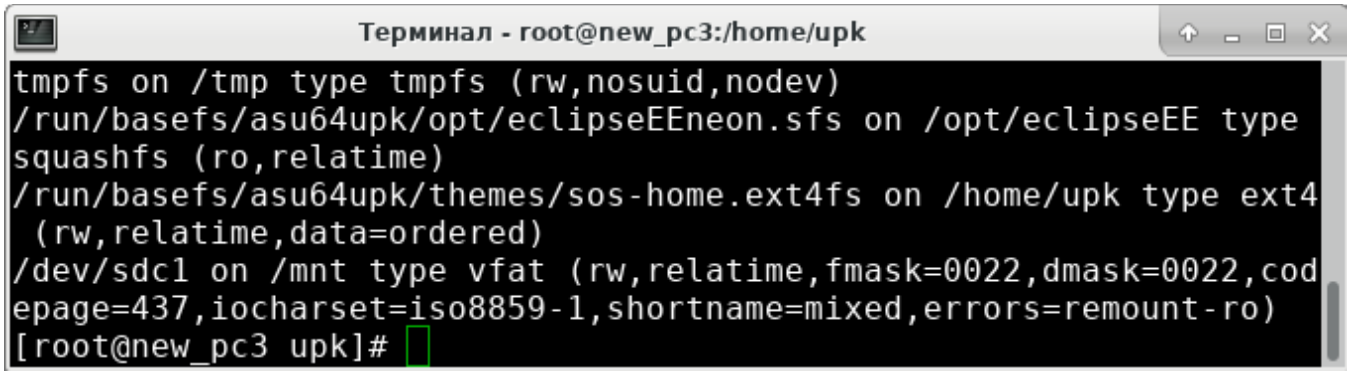

Рисунок 2.12 — Результат команды mount для клонированного процесса

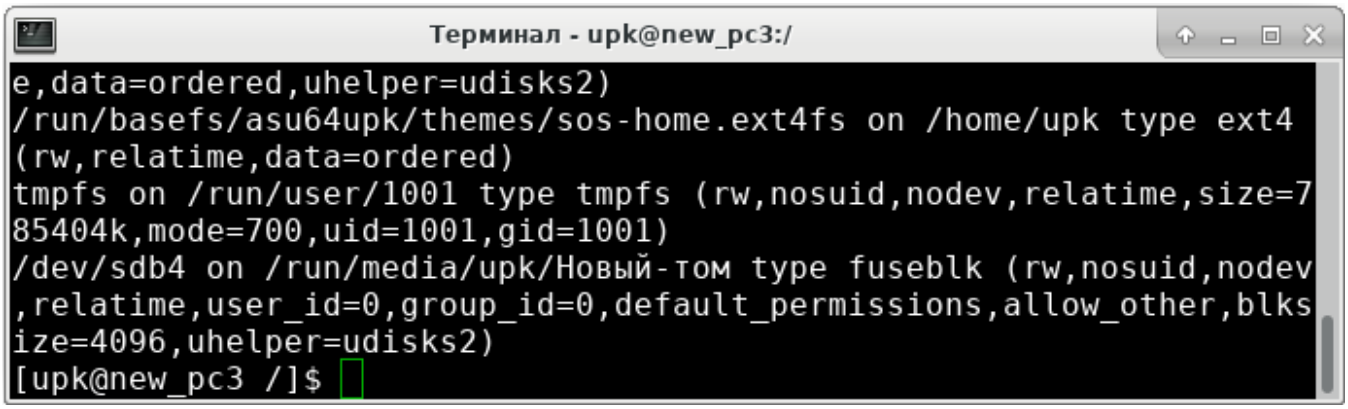

Рисунок 2.13 — Результат команды mount для обычного процесса

Хорошо видно, что клонированный процесс показывает монтирование flashUSB, а обычный процесс — нет.

Шаг 3:

- запустить в виртуальных терминалах файловый менеджер ***mc*** и визуально убедиться в различной видимости фаловой системы процессами;
- проверить, что значек устройства flashUSB, на рабочем столе ОС, показывает отсутствие монтирования устройства;
- описать содержимое и результаты исследования в личном отчете;
- с помощью необходимого количеста команд exit завершить работу клонированного процесса в терминале.

Замечание

Как описано в пункте 1.4.2, имеются и другие механизмы разделения пространств процессов, которые студент может изучить самостоятельно — по различным источникам в Интернет.

Список использованных источников

1 Резник В.Г. Современные операционные системы. Самостоятельная и индивидуальная работа студента. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 15 с.

2 Гордеев А.В. Операционные системы: учебное пособие для вузов. ─ СПб.: Питер, 2004. ─ 415с.

3 Таненбаум Э. Современные операционные системы. - СПб.: Питер, 2007. - 1037с.

4 Резник В.Г. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux. Учебно-методическое пособие. – Томск, ТУСУР, 2017. – 38 с.

5 Резник В. Г., Операционные системы: Учебное пособие для студентов направления 09.03.01, «Информатика и вычислительная техника» [Электронный ресурс] / Резник В. Г. — Томск: ТУСУР, 2016. — 183 с. — Режим доступа: https://edu.tusur.ru/publications/6261.

6 Резник В. Г. Операционные системы. Часть 2: Учебное пособие для студентов направления 09.03.01, «Информатика и вычислительная техника» [Электронный ресурс] / Резник В. Г. — Томск: ТУСУР, 2016. — 216 с. — Режим доступа: https://edu.tusur.ru/publications/6262.

Учебное издание

Резник Виталий Григорьевич

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебно-методическое пособие предназначено для изучения темы №4 по дисциплине «Современные операционные системы» для студентов уровня основной образовательной программы магистратура направления подготовки: 09.04.01 «Информатика и вычислительная техника».

Учебно-методическое пособие

Усл. печ. л. . Тираж . Заказ .
Томский государственный университет
систем управления и радиоэлектроники
634050, г. Томск, пр. Ленина, 40