Тема 4 - Разграничение ресурсов ОС Linux

Table of Contents

Безопасность операционных систем

Осень 2025 г.

Веричев Александр Владимирович
@xanchelavederiver alexandervерichev@gmail.com

1. Контроль потребления ресурсов с помощью ulimit

1.1. История развития механизма ulimit

Unix (1970–1980-е)

- UNIX разрабатывалась как многопользовательская и многозадачная система
- Распределение ресурсов между процессами пользователей очевидным образом оказалось одной из важнейших задач
- Помимо этого серьёзной проблемой оказалось истощение ресурсов — один некорректный процесс мог занять память, CPU или файловые дескрипторы
- В качестве решения в ОС Unix появились **лимиты ресурсов на процесс** и понятие приоритета процесса (и обратная величина `nice`)

Эволюция BSD и System V

- Появились понятия **мягкие** и **жёсткие** лимиты
- Добавлены структуры `RLIMIT` в ядре
- Введены команды оболочки (`ulimit`, `limit`) для управления лимитами

Стандартизация POSIX (1990-е)

- Стандарт POSIX унифицировал интерфейсы `ulimit`
- В стандарте определены системные вызовы:
  - `getrlimit()`
  - `setrlimit()`
- Командные интерпретаторы (в частности, `bash`) внедрили POSIX-совместимые механизмы ограничения ресурсов в систему команд оболочки

Linux

- В современных дистрибутивах Linux добавлен `/etc/security/limits.conf` для управления пользовательскими лимитами
- PAM-модуль `pam_limits.so` применяет лимиты при входе в систему - конфигурация сессии
- Появление механизма `cgroups` расширило возможности контроля ресурсов, но **ulimit остаётся актуальным**:
  - лёгкие ограничения на уровне процесса
  - совместимость со старыми системами
  - ограничение ресурсов в скриптах

1.2. Назначение механизма ulimit

Назначение механизма: предотвратить чрезмерное потребление системных ресурсов

одним процессом или пользователем.

Основные задачи `ulimit`:

1. Поддержка **стабильности системы**
2. Обеспечение **честного распределения ресурсов**
3. Предоставление **контроля администраторам**
4. Поддержка **тестирования приложений**

Решаемые проблемы:

1. **Истощение памяти**
2. **Перепроизводство процессов (fork bomb)**
3. **Исчерпание файловых дескрипторов**
4. **Огромные дампы памяти**
5. **Бесконтрольное использование CPU**
6. **Неограниченный рост размера файлов**

1.3. Управление лимитами ресурсов с помощью ulimit

`ulimit` - команда командного интерпретатора, используемая для просмотра и управления лимитами потребления ресурсов текущим **пользователем**.

1.3.1. Виды ресурсов

Аргумент ulimit	Ресурс	Описание
-c	`core`	ограничивает размер core-файла (в КБ)
-d	`data`	максимальный размер сегмента данных (в КБ)
-f	`fsize`	максимальный размер файла (в КБ)
-l	`memlock`	максимальный объём заблокированного в памяти адресного пространства (в КБ)
-n	`nofile`	максимальное число открытых файлов
-m	`rss`	максимальный размер резидентного набора (в КБ)
-s	`stack`	максимальный размер стека (в КБ)
-t	`cpu`	максимальное время работы CPU (в минутах)
-u	`nproc`	максимальное число процессов (см. примечание ниже)
-v	`as`	лимит адресного пространства (в КБ)

Аргумент ulimit	Ресурс	Описание
—	`maxlogins`	максимальное количество одновременных входов для данного пользователя
—	`maxsyslogins`	максимальное число одновременных входов в систему
—	`priority`	приоритет, с которым запускаются процессы пользователя
—	`locks`	максимальное количество файловых блокировок, которое может удерживать пользователь
—	`sigpending`	максимальное количество ожидающих сигналов
—	`msgqueue`	максимальный объём памяти, используемый очередями POSIX-сообщений (в байтах)
-e	`nice`	максимальный «nice»-приоритет, на который пользователь может повысить значение (диапазон [-20, 19])
-r	`rtprio`	максимальный приоритет реального времени

1.3.2. Мягкие и жёсткие лимиты

- **Soft limit** (**мягкий лимит**) обычно устанавливается на более низком уровне и действует как предупреждение для пользователя. Это не окончательный предел, но при его достижении система может предупредить пользователя, что ресурсы приближаются к исчерпанию.

  Рекомендуемое значение для soft limit может варьироваться, но часто устанавливается в диапазоне от 1024 до 4096.

- **Hard limit** (**жесткий лимит**) устанавливает абсолютный максимум, который не может быть превышен ни при каких обстоятельствах. Этот лимит служит для защиты системы от перегрузки.

  Рекомендуемое значение для hard limit обычно значительно выше и может быть установлено в пределах от 4096 до 65535, в зависимости от требований системы и приложений.

1.3.3. Как ядро применяет лимиты

```
if (usage + request > rlimit[RESOURCE].rlim_cur)
    return -EPERM
```

1.3.4. Управления лимитами в оболочке

Как задаются лимиты

- Команды оболочки (`ulimit`)
- PAM и `/etc/security/limits.conf`
- Значения по умолчанию ядра
- Программно через `setrlimit()`

Команда оболочки ulimit

Временная настройка лимитов с помощью команды `ulimit`

```
$ ulimit -c 0      # Запрещаем создавать core файлы
$ ulimit -d 48000  # Ограничиваем максимальный размер сегмента данных в 48 MB
$ ulimit -s 8192   # Ограничиваем максимальный размер стэка в 8 MB
$ ulimit -m 48000  # Ограничиваем максимальный размер резидентной части процесса
(находящейся в ОЗУ) в 48 MB
$ ulimit -u 64     # Ограничиваем максимальное число запущенных этим пользователем
процессов.
$ ulimit -n 128    # Ограничиваем максимальное число открытых файлов.
$ ulimit -f 100000 # Ограничиваем максимальный размер создаваемого файла в 100 MB
$ ulimit -v 100000 # Ограничиваем максимальный размер используемой виртуальной памяти
в 100 MB
```

Так установленные лимиты будут **жить до перезагрузки (сессии)**.

При этом можно настроить конфигурацию лимитов сессии в файлах наподобие `~/.bashrc` что фактически будет означать настройку постоянных лимитов.

Просмотр установленных лимитов

```
$ ulimit -a
```

PAM

Подсистема PAM позволяет задавать постоянные лимиты для пользователей и групп.

Постоянные лимиты задаются в файле `/etc/security/limits.conf` системы PAM.

Модуль `pam_limits.so` вызывается в группе управления (type) `session` (т.к. настраивает сессию) из конфигурационных файлов:

- `/etc/pam.d/login`
- `/etc/pam.d/sshd`

Включение pam_limits в PAM-стеке

Для настройки лимитов необходимо подключить модуль `pam_limits.so` в соответствующих

PAM-конфигурациях.

SSH

Файл: `/etc/pam.d/sshd`

```
session    required     pam_limits.so
```

Локальный терминал

Файл: `/etc/pam.d/login`

```
session    required     pam_limits.so
```

su

Файл: `/etc/pam.d/su`

```
session    required     pam_limits.so
```

Настройка лимитов в /etc/security/limits.conf

В файле задаются мягкие и жёсткие лимиты для пользователей и групп.

Формат записи: `<пользователь | @группа> <soft|hard> <лимит> <значение>`.

Пример конфигурации:

```
# Ограничение числа процессов
*               soft    nproc           200
*               hard    nproc           300

# Ограничение числа открытых файлов
*               soft    nofile          1024
*               hard    nofile          4096

# Ограничение виртуальной памяти
*               soft    as              500000
*               hard    as              700000

# Лимиты для конкретного пользователя
alice           soft    cpu             5
alice           hard    cpu             10

# Лимиты для группы (admins)
@admins         soft    nofile          8192
@admins         hard    nofile          16384

# Запрет более чем одного одновременного входа
```

```
bob            hard    maxlogins       1
```

1.4. Примеры применения ulimit

1.4.1. Защита от исчерпания файловых дескрипторов (ulimit -n)

Проблема: система может потерять возможность открывать файлы и принимать сетевые соединения.

С лимитом:

```
$ ulimit -n 1024
```

Попытка открытия файлов в цикле:

```
$ python3 -c "[open(f'/tmp/file_{i}', 'w') for i in range(1024+10)]"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
OSError: [Errno 24] Too many open files: '/tmp/file_1021'
```

1.4.2. Блокировка fork bomb (ulimit -u)

Проблема: форкобомба может съесть всю память / процессорное время бесконтрольным созданием дочерних процессов

Установка лимита:

```
$ ulimit -u 200
```

Результат:

```
$ :(){ :|:& };:
-bash: fork: retry: Resource temporarily unavailable
-bash: fork: retry: Resource temporarily unavailable
-bash: fork: retry: Resource temporarily unavailable
...
```

1.4.3. Предотвращение заполнения диска (ulimit -f)

Проблема: вредоносный код может создавать огромные мусорные файлы, чтобы забить дисковое пространство

Установить лимит размера одного файла 10 МБ:

```
$ ulimit -f 10240
```

Результат:

```
$ dd if=/dev/urandom bs=11M count=1 of=/tmp/dd
File size limit exceeded (core dumped)
```

1.4.4. Ограничение использования оперативной памяти (ulimit -v, ulimit -m)

Проблема: чрезмерное потребление RSS и виртуальной памяти одним процессом

Установить лимит:

```
$ ulimit -v 102400
```

Результат:

```
$ python3 -c 'import os; os.urandom(100 * 1024 * 1024)'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
MemoryError

$ dd if=/dev/urandom bs=92M count=1 of=/dev/null
dd: memory exhausted by input buffer of size 96468992 bytes (92 MiB)
```

1.4.5. Ограничение CPU (ulimit -t)

Проблема: долго работающий процесс съедает процессорное время

Установить лимит:

```
$ ulimit -t 5
```

Результат:

```
$ python3 -c 'while True: pass'
Killed
$ echo $?
137
```

1.4.6. Ограничение лимитов с помощью PAM-модуля pam_limits.so

Установим ограничение на максимальный размер файла, создаваемого пользователем x:

```
$ sudo nano /etc/security/limits.conf
...
x    hard    fsize    1024
```

В результате любой процесс, запущенный от пользователя x, при попытке создать файл, размером превышающий 1024 байта, будет аварийно завершен:

Пробуем создать большой файл:

```
$ ssh x@127.42.42.42
x@127.42.42.42's password:
$ cd /tmp
$ dd if=/dev/urandom bs=1024 count=32 of=/tmp/random_file
$ echo $?
0
$ python3 -c "import os; open('/tmp/random_file2', 'wb+').write(os.urandom(2**10 *
2**11))"
$ Traceback (most recent call last):
  File "<string>", line 1, in <module>
OSError: [Errno 27] File too large echo $?
```

Файл `random_file` размером 32Кб будет создан, а файл `random_file2` размером 2Мб – нет.

2. Иерархическая система распределения ресурсов Control Group (cgroup)

2.1. История разработки cgroup(s)

cgroup / **cgroups** (сокращение от **control group(s)**) - механизм ядра ОС Linux, предназначенный для управления (ограничения, учёта) и изоляции потребления ресурсов машины (CPU, память, дисковый ввод-вывод и т.д.) [1][2].

Cgroup выполняет сходные функции с `ulimit` + `/etc/security/limits.conf` и `nice(1)`, но более гибко - на уровне процесса.

История развития механизма

Разработка `cgroups` была начата в 2006 году в Google, изначально назывались «контейнеры процессов» (process containers) [1]. Изначальный замысел был скромен: усовершенствовать механизм `cpuset`, предназначенный для распределения процессорного времени и памяти между задачами.

В январе 2008 года механизм `cgroups` был добавлен в ядро Linux (версия 2.6.24) под

названием "контрольные группы". Целью переименования была попытка избежать путаницы, вызванной множественным значением термина "контейнер" в контексте ядра Linux. Основные функции релиза `cgroup` в ядре:

- три контроллера ресурсов (все связанные с CPU);
- введена специальная файловая система cgroups - `cgroupfs`;
- в виртуальной файловой системе `/proc` появились новые файлы: `/proc/{pid}/cgroup` для каждого процесса и `/proc/cgroups` для системы в целом.

Со временем различные команды разработчиков добавляли контроллеры для распределения прочих ресурсов: памяти, дискового ввода/вывода и доступа к иным устройствам, ввод/вывод сети и т. п. Однако годы разработки без единого руководства привели к созданию излишне сложной, внутренне противоречивого механизма [3].

Работы по созданию 2-й версии (`cgroup2`) начались практически сразу [3]:

- разработка началась в 2012 году;
- уже в 2015 подсистема `systemd` ввела поддержку `cgroup2`;
- официальный релиз 2-й версии состоялся в 2016 году, однако на момент релиза функционал был не сопоставим с 1-й версией, в частности, не поддерживался контроллер ресурсов CPU;
- контроллер CPU был добавлен в январе 2018;
- в октябре 2019 Fedora 31 стала первым дистрибутивом, официально установившим 2-ю версию `cgroup2` в качестве основной (включенной по умолчанию);
- в 2020 поддержка `cgroup2` внедрена в Docker;
- начиная с 2021 большинство дистрибутивов также стали использовать `cgroup2` в качестве основного механизма.

Важным изменением в `cgroup2` является введение единой иерархии контроллеров, в отличие от множества разрозненных иерархий `v1`.
Именно поэтому изначальная версия механизма именуется `cgroup*S*` - **control groups** (множество групп), а 2-я версия - `cgroup` (одна группа).

Термин `cgroup` означает "контрольная группа" и никогда не пишется с заглавной буквы. Единственное число используется для обозначения всего механизма [2].

Проверить используемую версию можно с помощью следующих команд:

```
$ mount | grep cgroup
cgroup2 on /sys/fs/cgroup type cgroup2
(rw,nosuid,nodev,noexec,relatime,nsdelegate,memory_recursiveprot)  ①
$ grep -c cgroup /proc/mounts
1  ②
```

Наличие одной иерархии и типа файловой системы `cgroup2` свидетельствует

об использовании `cgroup2`.

Отключение 1-й версии механизма `cgroups` возможно:

- с помощью параметра запуска ядра `cgroup_no_v1=all` (в `grub`);
- с помощью параметра загрузки `systemd.unified_cgroup_hierarchy`.

2.2. Основы механизма cgroup2

`cgroup2` — это механизм иерархической организации процессов и распределения системных ресурсов по иерархии контролируемым и настраиваемым образом.

Основными компонентами `cgroup2` являются [2][3]:

1. **ядро** - механизм построения иерархии **групп управления** процессами;
2. набор **контроллеров** - объектов ядра, управляющих потреблением ресурсов процессов группы управления.

Группой управления (**control group**, `cgroup`) называют множество процессов, объединяемых для управления их совокупным потреблением ресурсов.

Группы управлений формируют иерархию (единую в `cgroup2`). Каждый процесс в системе принадлежит **только к одной** группе управления. Все потоки процесса также принадлежат той же `cgroup`. При создании все процессы помещаются в `cgroup`, к которой на данный момент принадлежит родительский процесс. Процесс может быть перенесен в другую `cgroup`, однако перенос процесса не влияет на уже существующие дочерние процессы [2].

Процессы *nix-подобных ОС организованы в виде древовидной структуры. В корне располагается процесс `init` с `pid=0`, являющийся родительским для всех остальных процессов, все дочерние процессы образуют поддерево процессов.

Группы управления разбивают множество всех процессов на непересекающиеся подмножества, образованные подобными поддеревьями, и накладывают ограничения на общее потребление ресурсов всеми процессами группы.

Однако важно понимать, что иерархия групп управления в общем случае может отступать от иерархии процессов: например, один дочерний процесс (и только он) может быть перемещён из `cgroup`, в которой состоит родительский процесс, в любую другую группу управления!

Контроллер ресурсов (**controller**) обычно отвечает за распределение конкретного типа системных ресурсов по иерархии (контроллер CPU управляет распределением процессорного времени, контроллер памяти управляет потреблением оперативной памяти и т.п.). Также существуют утилитарные контроллеры, которые служат для других целей - например, логирование (мониторинг) потребления.

Контроллеры ресурсов также иногда называют *подсистемой* (*subsystem*), однако этот термин слишком многозначен и потому рекомендуется избегать его употребления.

Отметим некоторые особенности функционирования контроллеров в иерархии групп управления [2][3]:

- Следуя определенным структурным ограничениям, контроллеры могут быть выборочно включены или выключены в `cgroup`.
- Группы правления формируют иерархию: помимо процессов в конкретную `cgroup` могут входить дочерние (под)группы управления.
- Дочерние группы `cgroups` наследуют параметры управления ресурсами родительской `cgroup`: если контроллер включен в `cgroup`, он влияет на все процессы, которые принадлежат дочерним `cgroups` его суб-иерархии.
- Когда контроллер включен во вложенной `cgroup`, он всегда еще больше ограничивает распределение ресурсов. Ограничения, установленные ближе к корню иерархии, не могут быть переопределены в группах-потомках.

Интерфейс `cgroup2` предоставляется посредством *псевдофайловой системы* (*pseudo-filesystem*) `cgroup2`. В дистрибутивах, использующих подсистему `systemd`, псеводфайловая система `cgroup2` монтируется `systemd` в точку `/sys/fs/cgroup` или `/sys/fs/cgroup/unified`.

Псеводфайловой системе 1-й версии `cgroups` назначен тип `cgroup`.

Как следствие, конфигурирование `cgroups` сводится к выполнению файловых операций следующими способами:

- из командного интерпретатора;
- программно;
- подсистемой наподобие `systemd`;
- системой управления контейнерами (LXC, Docker, etc.).

Структура директории, ассоциированной с группой управления, определяет её параметры управления, а также множество отнесённых в неё процессов и подгрупп.

Подгруппы создаются с помощью mkdir(2) / rmdir(2) и аналогичных системных функций. При создании подгруппы автоматически создаются интерфейсные файлы предназначенные для управления как собственно группой, так и для конфигурации каждого из включенных контроллеров. В частности, интерфейсные файлы используются для:

- задания множества процессов, включаемых в данную `cgroup`;
- настройки контроллеров ресурсов;
- вывода статистики потребления ресурсов.

Пример

В качестве примера, следуя [3], продемонстрируем управление количеством процессов в

группе.

Ограничение количества запущенных процессов в группе позволяет защититься от т.н. форкобомб, см. Блокировка fork bomb (ulimit -u).

Для управления количеством процессов в группе используется контроллер `pids` (*process number*).

Создадим новую `cgroup`:

```
$ sudo mkdir /sys/fs/cgroup/demo-cgroup  ①
$ sudo ls /sys/fs/cgroup/demo-cgroup  ②
cgroup.controllers      cpu.stat.local       memory.reclaim
cgroup.events           cpu.uclamp.max       memory.stat
cgroup.freeze           cpu.uclamp.min       memory.swap.current
cgroup.kill             cpu.weight           memory.swap.events
cgroup.max.depth        cpu.weight.nice      memory.swap.high
cgroup.max.descendants  io.pressure          memory.swap.max
cgroup.pressure         memory.current       memory.swap.peak
cgroup.procs            memory.events        memory.zswap.current
cgroup.stat             memory.events.local  memory.zswap.max
cgroup.subtree_control  memory.high          memory.zswap.writeback
cgroup.threads          memory.low           pids.current
cgroup.type             memory.max           pids.events
cpu.idle                memory.min           pids.events.local
cpu.max                 memory.numa_stat     pids.max
cpu.max.burst           memory.oom.group     pids.peak
cpu.pressure            memory.peak
cpu.stat                memory.pressure

$ cat /sys/fs/cgroup/demo-cgroup/cgroup.procs  ③
$ cat /sys/fs/cgroup/demo-cgroup/cgroup.controllers  ④
cpu memory pids
$ cat /sys/fs/cgroup/demo-cgroup/cgroup.stat  ⑤
nr_descendants 0
nr_subsys_cpuset 0
nr_subsys_cpu 1
nr_subsys_io 0
nr_subsys_memory 1
...
nr_dying_subsys_dmem 0
```

① Создаём группу управления `demo-cgroup`

② Автоматически создаются интерфейсные файлы

③ Процессы созданной группы - пока ни один процесс не добавлен в группу `demo-cgroup`

④ В созданной группе включены три контроллера: `cpu` (процессор), `memory` (оперативная память), `pids` (количество процессов)

⑤ Структура группы

Вместе с новой группой управления `demo-cgroup` создаётся множество конфигурационных файлов группы и контроллеров. Отметим некоторые интерфейсные файлы:

- `cgroup.procs` - список процессов группы;
- `cgroup.controllers` - список включенных контроллеров ресурсов;
- `cgroup.stat` - файл, отражающий структуру группы управления;
- `cpu.*` - конфигурационные файлы контроллера `cpu`;
- `memory.*` - конфигурационные файлы контроллера `memory`;
- `pids.*` - конфигурационные файлы контроллера `pids`.

Конфигурационные файлы контроллера `pids`:

```
$ sudo ls /sys/fs/cgroup/demo-cgroup | grep pids
pids.current
pids.events
pids.events.local
pids.max
pids.peak
```

В рамках настоящего примера нас будут интересовать два из них:

1. `pids.current` - количество запущенных процессов в группе;
2. `pids.max` - максимально возможное количество процессов в группе.

Добавим текущий процесс `bash`, запущенный в рабочем терминале, в группу управления `demo-cgroup`:

```
$ echo $$  ①
5020
$ cat /proc/$$/cgroup
0::/user.slice/user-1001.slice/session-159.scope  ⑤

$ echo $$ | sudo tee -a /sys/fs/cgroup/demo-cgroup/cgroup.procs  ②
5020
$ cat /sys/fs/cgroup/demo-cgroup/cgroup.procs  ③
5020
14789
$ cat /sys/fs/cgroup/demo-cgroup/pids.current  ④
2

$ cat /proc/$$/cgroup
0::/demo-cgroup  ⑤
$ cat /sys/fs/cgroup/user.slice/user-1001.slice/session-159.scope/cgroup.procs
14798  ⑤
14872
```

```
14873
```

① В рабочем терминале в интерактивном режиме работает `bash` с `PID` 5020

② Добавляем текущий процесс 5020 в группу `demo-cgroup`

③ С помощью `cgroup.procs` видим два процесса в группе: 5020 и 14789 (см. комментарий ниже)

④ Файл `pids.current` также показывает два запущенных процесса в группе

⑤ Процесс 5020 был отвязан от исходной группы `user.slice/user-1001.slice/session-159.scope` и добавлен в `demo-cgroup`

Добавление процесса в группу выполняется путём дописывания его `PID` в файл `cgroup.procs`. Чтение этого файла даёт текущий список запущенных процессов группы. Количество запущенных процессов можно получить с помощью чтения файла `pids.current`.

В примере выше мы видим два процесса, хотя добавляли один. Причина этого проста: все процессы, запускаемые добавленным в `cgroup` родительским процессом, автоматически добавляются в `cgroup`. Так как чтение файла выполняется из `bash`, создаваемый для этого процесс `cat` также добавляется в группу `demo-cgroup`.

Что будет, если выполнить чтение файла из другой консоли?

Проверить, к какой группе отнесён конкретный процесс по его `PID`, можно посредством интерфейса *procfs* - с помощью чтения псевдофайла `/proc/<pid>/cgroup`.

В примере выше изначальная группа процесса `bash` была `user.slice/user-1001.slice/session-159.scope` - группа, создаваемая для всех процессов, запускаемых в рамках интерактивной сессии пользователя `user` (`UID` 1001). После добавления 5020 в `demo-cgroup` процесс `bash` удаляется из предыдущей группы автоматически.

Добавление процесса в какую-либо `cgroup` автоматически удаляет его из его текущей группы.

В `cgroup2` процесс может быть отнесёт только к одной группе единой иерархии, так что в выводе команды чтения всегда содержится одна строка. В примере выше показано, что `bash` действительно включен в группу `demo-cgroup`.

Ограничим максимально возможное количество запущенных процессов группы и попробуем запустить группу процессов, количество которых превышает лимит:

```
$ echo 5 | sudo tee /sys/fs/cgroup/demo-cgroup/pids.max  ①
5
$ cat /sys/fs/cgroup/demo-cgroup/pids.max
5
$ for a in $(seq 1 5); do sleep 10 & done  ②
[1] 14968
[2] 14969
[3] 14970
```

```
[4] 14971
-bash: fork: retry: Resource temporarily unavailable  ③
-bash: fork: retry: Resource temporarily unavailable
-bash: fork: retry: Resource temporarily unavailable
-bash: fork: retry: Resource temporarily unavailable
-bash: fork: Interrupted system call
[3]-  Done                    sleep 10  ④
[1]   Done                    sleep 10
[2]-  Done                    sleep 10
[4]+  Done                    sleep 10
```

① Устанавливаем лимит максимального числа запущенных процессов

② Пробуем запустить пять процессов

③ Четыре процесса запускаются успешно, пятый же запустить не удаётся: после серии попыток `bash` получает `SIGINT` и прекращает запуск
N.B. Почему 5-й процесс не был запущён, хотя лимит как раз 5?

④ Четыре успешно запущенных процесса завершают работу

Как видно в листинге выше, лимит на количество процессов препятствует запуску новых процессов в группе управления.

Удаление группы выполняется путём удаления соответствующей директории. Удаление разрешено только в случае отсутствия у группы процессов и дочерних групп (т.е. "пустой" группы):

```
$ cd /sys/fs/cgroup
$ sudo rmdir demo-cgroup/  ①
rmdir: failed to remove 'demo-cgroup/': Device or resource busy

$ echo $$ | sudo tee -a /sys/fs/cgroup/user.slice/user-1001.slice/session-
159.scope/cgroup.procs  ②
5020
$ cat /sys/fs/cgroup/user.slice/user-1001.slice/session-159.scope/cgroup.procs
14798
14872
14873
5020
15002
$ sudo rmdir demo-cgroup/  ③
$ ll | grep demo
```

① Попытка удаления непустой группы управления `demo-cgroup` завершается неудачей

② Переносим процесс `bash` с `OID` 5020 в его исходную группу управления

③ Удаление группы `demo-cgroup` выполнено успешно

2.3. Управление ресурсами с помощью cgroup2

2.3.1. Монтирование псевдофайловой системы

В отличие от `cgroups v1`, в `cgroup2` определена одна иерархия [2]. Файловой системе `cgroup2` задана сигнатура (*magic-number*) `0x63677270` (`"cgrp"`).

Иерархию `cgroup2` можно смонтировать с помощью следующей команды:

```
# mount -t cgroup2 none $MOUNT_POINT
```

В современных дистрибутивах, использующих подсистему инициализации и управления службами `systemd`, монтирование выполняет `systemd` во время загрузки системы. Также большинство дистрибутивов используют 2-ю версию `cgroup2` по умолчанию.

Для гарантии отключения `cgroups v1` возможно задание параметра загрузки ядра `cgroup_no_v1=all` или параметра `systemd.unified_cgroup_hierarchy`.

`cgroup2` поддерживает ряд опций монтирования, предназначенных для настройки некоторых подсистем механизма (подробнее см. [2], раздел "Mounting").

2.3.2. Интерфейсные файлы

2.3.3. Организация иерархии групп процессов и потоков

Создание группы управления

Изначально, после запуска ОС, существует единственная корневая группа (и, возможно, несколько подгрупп, см. [section-cgroup_cgroup-and-systemd]), которой принадлежат все процессы [2].

Дочерняя `cgroup` может быть создана путём создания поддиректории в точке монтирования псеводфайловой системы `cgroup2`:

```
# cd $MOUNT_POINT
# mkdir $CGROUP_NAME
```

Группа управления может содержать несколько дочерних `cgroup`, формируя древовидную структуру - часть единой иерархии.

Добавление и удаление процессов

В каждой `cgroup` создаётся доступный для чтения и записи интерфейсный файл `cgroup.procs`. При его чтении вывод содержит список `PID` всех процессов, принадлежащих данной `cgroup` - по одному идентификатору на строку. `PID` в списке не упорядочены.

Процесс может быть перенесён в группу управления путём записи его `PID` в файл

`cgroup.procs` целевой `cgroup`. Одним вызовом write(2) можно перенести только один процесс. Если процесс состоит из нескольких потоков, запись `PID` любого из потоков переносит все потоки процесса.

При создании дочерних процессов новый процесс помещается в ту же `cgroup`, к которой принадлежал родительский процесс **на момент выполнения запуска**.

Дочерний процесс может быть перемещён в другую группу управления после создания - в этом случае родительский и дочерний процессы окажутся в разных `cgroup`.

После завершения процесс остаётся ассоциированным с `cgroup`, к которой он принадлежал **на момент завершения**, до тех пор, пока не будет "зачищен" (*reaped*).

Зомби-процесс (*zombie*, *defunct*) не отображается в `cgroup.procs` и, следовательно, не может быть перемещён в другую `cgroup`, поэтому остаётся ассоциированным с группой управления.

Для определения принадлежности конкретного процесса по его `PID` удобно использовать псевдофайл `/proc/$PID/cgroup`, содержащий для `cgroup2` одну строку формата `0::/$PATH`, где `$PATH` - относительный путь к группе от корня иерархии:

```
# cat /proc/$PID/cgroup
0::/demo-cgroup/test-cgroup-nested
```

В случае использования `cgroups v1` в выводе может содержаться несколько строк - по одной для каждой иерархии.

`Cgroup`, у которой нет дочерних групп или активных процессов, может быть удалена путём удаления каталога - с помощью rmdir(2)).

```
# cd $MOUNT_POINT
# rmdir $CGROUP_NAME
```

`Cgroup` без дочерних групп, ассоциированная только с зомби-процессами, считается пустой и может быть удалена.

Если процесс становится зомби, а связанная с ним `cgroup` впоследствии удаляется, к пути добавляется (`deleted`):

```
# cat /proc/$PID/cgroup
0::/demo-cgroup (deleted)
```

Управление ресурсами на уровне потоков

`Cgroup2` поддерживает управление ресурсами с большей гранулярностью - на уровне потока (*thread*) [2].

По умолчанию все потоки процесса принадлежат одной `cgroup`, которая также выступает в качестве домена ресурсов, объединяющего потребление ресурсов, не специфичных для конкретного процесса или потока. **Режим потоков** (**thread mode**) позволяет распределять потоки по поддереву, сохраняя при этом для них общий домен ресурсов.

Контроллеры, поддерживающие режим потоков, называются **потоковыми контроллерами** (**threaded controllers**). Не поддерживающие потоковый режим работы контроллеры называются **доменными контроллерами** (**domain controllers**).

Текущий режим работы или тип `cgroup` отображается в файле `cgroup.type`. Содержимое файла определяет режим работы группы управления:

- `domain` - обычная `cgroup`, которая может содержать процессы и подгруппы.
- `domain threaded` - корень т.н. *threaded subtree*; в подгруппы такой `cgroup` могут помещаться потоки многопоточных приложений для управления ресурсами на уровне потоков.
- `threaded` - `cgroup`, управляющая потреблением ресурсов потоками.
- `domain invalid` - `cgroup` находится в некорректном состоянии (*invalid state*).

Пример типов групп:

```
$ cd /sys/fs/cgroup
$ cat user.slice/cgroup.type
domain
$ cat system.slice/cgroup.type
domain

$ mkdir my_threaded_group
$ cat my_threaded_group/cgroup.type
domain
$ echo threaded sudo tee my_threaded_group/cgroup.type
threaded
$ cat my_threaded_group/cgroup.type
threaded
```

Следующие контроллеры являются потоковыми и могут быть включены в потоковой `cgroup`:

- cpu;
- cpuset;
- perf_event;
- pids.

Подробнее про `thread mode` см. [2], раздел "Threads".

2.3.4. Управление контроллерами

Доступность контроллера в группе управления

Для управления (включения и отключения) контроллеров группы управления используют два интерфейсных файла [3]:

- `cgroup.controllers` - список доступных для включения в данной `cgroup` контроллеров;
- `cgroup.subtree_control` - список включенных в данной `cgroup` контроллеров.

Конфигурация файлов позволяет настраивать управление различными ресурсами на различных уровнях иерархии.

Контроллер доступен в группе управления `cgroup` только в если он [2]:

1. поддерживается ядром (т.е. скомпилирован с ядром);
2. не отключён и не привязан к иерархии `cgroups v1`;
3. перечислен в файле `cgroup.controllers` группы управления;
4. перечислен в файле `cgroup.controllers` всех родительских групп.

Некоторые контроллеры (*implicit controllers*) доступны для включения всегда и не указываются в `cgroup.controllers` (например, `freezer` или `perf_event`).

Пример списка доступных контроллеров:

```
# cat cgroup.controllers
cpu io memory
```

Доступность означает, что интерфейсные файлы контроллера представлены в каталоге группы управления, позволяя наблюдать или управлять распределением целевого ресурса внутри этой `cgroup`.

Включение и отключение контроллера

Чтение интерфейсного файла группы управления `cgroup.subtree_control` возвращает список включенных контроллеров, разделённых пробелами:

```
$ cat cgroup.subtree_control
cpuset cpu io memory hugetlb pids rdma misc dmem
```

Включение и отключение контроллеров осуществляется путём записи в интерфейсный файл `cgroup.subtree_control`. В отличие от добавления процесса в группу управления, возможно выполнение включения / отключения нескольких контроллеров с помощью одной операции записи.

Формат записываемых данных схематично можно представить следующим образом:

```
[+|-]cnt1 [+|-]cnt2 [+|-]cnt3 ...
```

где `cnt{n}` - название очередного контроллера, `+` обозначает включение, `-` обозначает отключение контроллера.

Пример включения контроллеров `cpu` и `memory` и отключения `io`:

```
# echo "+cpu +memory -io" > cgroup.subtree_control
```

По умолчанию ни один контроллер не включен.

При включении / отключении нескольких контроллеров одновременно, как в примере выше, либо все операции выполняются успешно, либо все завершаются ошибкой. Если для одного контроллера указано несколько операций, применяется последняя.

Все доступные для включения контроллеры перечислены в файле `cgroup.controllers`, создаваемом для каждой группы управления.

Таким образом, невозможно включить контроллер, не указанный в `cgroup.controllers`, а также множество включенных контроллеров `cgroup.subtree_control` является подмножеством контроллеров, указанных в `cgroup.controllers`.

Иерархическое управление ресурсами

Включение контроллера в группе управления `cgroup` [3]:

1. означает включение контроля распределения целевого ресурса между процессами её дочерних групп;
2. влечёт создание интерфейсных файлов контролера во всех дочерних группах управления.

Интерфейсные файлы дочерних групп используются процессом, управляющим родительской группой, для настройки распределения ресурсов между дочерними группами.

Рассмотрим следующее поддерево (в скобках указаны включенные контроллеры) [2]:

```
A(cpu,memory) - B(memory) - C()
                          \ D()
```

- Поскольку у `A` включены `cpu` и `memory`, `A` будет контролировать распределение процессорного времени и памяти между процессами своих дочерних групп, в данном случае группы `B`.
- Так как у `B` включен контроллер `memory` и отключен контроллер `cpu`, группы `C` и `D` будут свободно конкурировать за циклы CPU (предоставляемые квотой от группы `A`), но

распределение доступной от `B` памяти будет контролироваться согласно заданным правилам.

Именно потому, что контроллер управляет распределением целевого ресурса между процессами дочерних групп, его включение создаёт интерфейсные файлы контроллера во всех дочерних `cgroups`.

В примере выше включение контроллера `cpu` в `B` создаёт интерфейсные файлы `cpu.*` в группах `C` и `D`. Аналогично, отключение `memory` в `B` удаляет интерфейсные файлы `memory.*` из `C` и `D`.

Фактически такое поведение означает, что интерфейсные файлы контроллера принадлежат родительской `cgroup`, а не самой `cgroup`.

Необходимо выделить следующие особенности иерархического управления ресурсами [2].

1. **Ограничение сверху вниз (Top-down Constraint)**

   Ресурсы распределяются в направлении сверху вниз (*top-down*): группа управления может распределять ресурс только если он был распределён ей от родительской группы. Это означает, что интерфейсный файл `cgroup.subtree_control` любой не корневой группы может содержать только контроллеры, включённые в файле `cgroup.subtree_control` родителя.

   Контроллер может быть включён, только если он включён у родительской группы, и контроллер не может быть отключён, если он включён у одной или нескольких дочерних групп управления.

2. **Ограничение "без внутренних процессов" (No Internal Process Constraint)**

   Не корневые группы управления могут распределять ресурсы своим потомкам только тогда в случае отсутствия у них собственных процессов.

   Другими словами, только `cgroup` без отнесённых к ней т.н. внутренних процессов (*internal processes*) может включать контроллеры.

   Такой подход гарантирует, что все процессы части иерархии, потреблением ресурсов которой управляет контроллер, всегда располагаются только на листьях - терминальных группах, не содержащих дочерние `cgoups`. Это исключает ситуации, когда процессы дочерних `cgroup` конкурируют с внутренними процессами родителя.

   Исключением является только корневая группа иерархии, т.к. она содержит процессы и временные группы для анонимного потребления ресурсов, которые не могут быть связано с другими `cgroup`.

Ограничение "без внутренних процессов" применяется только в случае наличия включенных контроллеров группы и поэтому не мешает созданию дочерних групп группе управления с внутренними процессами.

> Такое поведение позволяет создать дочерние группы, переместить все внутренние процессы в них и включить необходимые контроллеры с помощью интерфейсного файла `cgroup.subtree_control`.

2.3.5. Уведомления об изменении списка процессов группы

В каждой не корневой группе управления создаётся файл `cgroup.events`, содержащий поле `populated`, которое указывает, имеются ли в поддереве `cgroup` активные процессы. Его значение равно 0, если в `cgroup` и её дочерних группах нет активных процессов; в противном случае - 1 [2].

При изменении этого значения порождаются события `poll` и `[id]notify`. Уведомления об этих событиях можно использовать, например, для запуска операции удаления временной группы (*transient*) после завершения всех её процессов. Обновления состояния списка процессов и, соответственно, уведомления, являются рекурсивными.

Рассмотрим следующее поддерево, где числа в скобках обозначают количество процессов в каждой `cgroup`:

```
A(4) - B(0) - C(1)
          \ D(0)
```

Поля `populated` для `A`, `B` и `C` будут равны 1, а для `D` — 0. После завершения одного процесса в `C`, поля `populated` для `B` и `C` изменятся на 0 и будут сгенерированы *события изменения файла* (*file modified events*) на файлах `cgroup.events` обеих `cgroup`.

2.4. Модели распределения ресурсов

Контроллеры `cgroup` реализуют различные схемы распределения ресурсов в зависимости от типа ресурса и предполагаемых сценариев использования [2].

Пропорциональное распределение (Weights)

Ресурс родителя распределяется путём суммирования весов всех активных потомков и предоставления каждому доли, соответствующей отношению его веса к сумме. Поскольку в распределении участвуют только те потомки, которые могут использовать ресурс в данный момент, модель позволяет обеспечить полную загрузку - распределение всего доступного ресурса. Из-за динамической природы эта модель обычно используется для ресурсов без сохранения состояния.

Веса находятся в диапазоне `[1, 10000]`, по умолчанию используется значение 100 (такое значение позволяет описывать доли в процентах).

Пока веса находятся в допустимом диапазоне, любые комбинации значений допустимы и не отклоняются ядром.

Пример использования модели весов: - `cpu.weight`, пропорционально распределяющий процессорное время между активными процессами дочерних групп.

Абсолютные ограничения (Limits)

В модели лимитов ограничений процессы дочерней группы могут потреблять не более заданного количества ресурса.

Лимиты находятся в диапазоне `[0, max]`, по умолчанию - `"max"` (строка), что интерпретируется как отсутствие ограничений (вся доля, выделенная родительской группе управления).

Величина `max` устанавливается с помощью интерфейсных файлов `res.max`, где `res` - название ресурса.

Лимиты могут быть *перегружаемыми* (*over-committed*) - сумма лимитов потомков может превышать количество ресурса, доступного родителю. Как следствие, любые конфигурации лимитов допустимы и не отклоняются ядром.

Пример использования модели лимитов: `cpu.max`, ограничивающий максимальное потребление процессорного времени процессами дочерних групп.

Защита (Protections)

`Cgroup` защищена до заданного количества ресурса, пока использование всех её предков не превышает их защищённых уровней. Защита может быть строгой гарантией (*hard guarantee*) или мягкой границей (*soft boundary*) по принципу "наилучшего усилия" (*best effort*). Защита также может быть *перегружаемой*, и в этом случае среди потомков защищается только до того количества ресурса, которое доступно родителю.

Защита находится в диапазоне `[0, max]`, по умолчанию - 0, что означает отсутствие защиты.

Поскольку защита может быть *перегружаемой*, все комбинации конфигураций допустимы, и нет причин отклонять изменения конфигурации или миграции процессов.

Пример использования модели лимитов: `memory.low`, реализующий защиту памяти по принципу "наилучшего усилия".

Бронирование ресурса (Allocations)

В модели бронирования ресурса группе управления `cgroup` предоставляется в монопольное использование определённое количество конечного ресурса. Бронирование не может быть перегружаемым - сумма резервов потомков не может превышать количество ресурса, доступного родителю.

Выделения находятся в диапазоне `[0, max]`, по умолчанию - 0, что интерпретируется как отсутствие ресурса.

Поскольку бронирования не могут быть перегружаемыми, некоторые комбинации конфигураций недопустимы и отклоняются ядром. Кроме того, если ресурс обязателен для выполнения процессов, изменения резервов могут быть отклонены.

Пример использования модели резервирования: `cpu.rt.max`, жёстко выделяющий кванты процессорного времени.

2.5. Демо

Подготавливаем иерархию:

```
$ cd /sys/fs/cgroup/demo-cgroup
$ cat ../cgroup.subtree_control
cpu memory pids
$ cat cgroup.controllers
cpu memory pids
$ cat cgroup.subtree_control
$ echo "+cpu" | sudo tee cgroup.subtree_control
+cpu
$ cat cgroup.subtree_control
cpu

$ sudo mkdir A
$ cat A/cgroup.controllers
cpu
$ cat A/cgroup.subtree_control

$ echo $$ | sudo tee A/cgroup.procs
5020
$ cat A/cgroup.procs
5020
60272

$ sudo mkdir A/{P,Q,R}
$ cat A/{P,Q,R}/cgroup.subtree_control


$ echo "+cpu" | sudo tee A/cgroup.subtree_control
+cpu
$ cat A/cgroup.subtree_control
cpu
$ echo "+cpu" | sudo tee A/P/cgroup.subtree_control
+cpu
tee: A/P/cgroup.subtree_control: Operation not supported  ①

$ echo $$ | sudo tee ../user.slice/user-1001.slice/session-159.scope/cgroup.procs
5020
$ echo "+cpu" | sudo tee A/P/cgroup.subtree_control
+cpu
$ cat A/P/cgroup.subtree_control
cpu  ②

$ echo $$ | sudo tee A/cgroup.procs
5020
$ cat A/cgroup.type
domain threaded
$ cat A/cgroup.subtree_control
```

```
cpu
$ cat A/P/cgroup.subtree_control
cpu
$ cat A/P/cgroup.type
domain invalid  ③

$ echo $$ | sudo tee A/P/cgroup.procs
5020
tee: A/P/cgroup.procs: Operation not supported
$ echo $$ | sudo tee ../user.slice/user-1001.slice/session-159.scope/cgroup.procs
5020
$ cat A/P/cgroup.type
domain
$ cat A/cgroup.type
domain
$ echo $$ | sudo tee A/P/cgroup.procs
5020
$ cat A/cgroup.type
domain
$ cat A/P/cgroup.type
domain threaded  ④
$ cat A/P/cgroup.procs
5020  ④
60494

$ echo $$ | sudo tee ../user.slice/user-1001.slice/session-159.scope/cgroup.procs
5020  ⑤

$ echo "+cpu" | sudo tee A/{Q,R}/cgroup.subtree_control  ⑥
```

Проверяем иерархию:

```
$ cat A/cgroup.subtree_control
cpu
$ cat A/{P,Q,R}/cgroup.subtree_control
cpu
cpu
cpu
```

Настраиваем квоты:

- A - 50%
- A/P - 10%
- A/Q - 15%
- A/R - 25%

```
$ cat A/cpu.max
```

```
max 100000

$ echo "50000 100000" | sudo tee A/cpu.max
50000 100000

$ echo "10000 100000" | sudo tee A/P/cpu.max
10000 100000
$ echo "15000 100000" | sudo tee A/Q/cpu.max
15000 100000
$ echo "25000 100000" | sudo tee A/R/cpu.max
25000 100000

$ cat A/cpu.max
50000 100000
$ cat A/{P,Q,R}/cpu.max
10000 100000
15000 100000
25000 100000
```

Запустим `stress-ng` в группах `P`, `Q`, `R`:

в рабочей консоли

```
$ sudo apt install stress-ng sysstat htop

$ stress-ng --metrics --verbose --cpu 1 --timeout 60s
```

в основной консоли

```
$ htop

$ _PID=`ps aux | grep stress | awk 'NR == 2 {print $2}'`; echo $_PID | sudo tee
A/P/cgroup.procs
$ htop
$ watch -n 1 cat A/P/cpu.pressure
```

Запустим три процесса `stress-ng` во всех группах `P`, `Q`, `R` одновременно:

в рабочей консоли

```
$ stress-ng --metrics --verbose --cpu 3 --timeout 120s
```

в основной консоли

```
$ htop

$ _PID=`ps aux | grep stress | awk 'NR == 2 {print $2}'`; echo $_PID | sudo tee
A/P/cgroup.procs
$ _PID=`ps aux | grep stress | awk 'NR == 3 {print $2}'`; echo $_PID | sudo tee
```

```
A/Q/cgroup.procs
$ _PID=`ps aux | grep stress | awk 'NR == 4 {print $2}'`; echo $_PID | sudo tee
A/R/cgroup.procs
$ htop
```

Запустим три процесса `stress-ng` в одной группе `R`:

в рабочей консоли

```
$ stress-ng --metrics --verbose --cpu 3 --timeout 120s
```

в основной консоли

```
$ htop

$ _PID=`ps aux | grep stress | awk 'NR == 2 {print $2}'`; echo $_PID | sudo tee
A/R/cgroup.procs
$ _PID=`ps aux | grep stress | awk 'NR == 3 {print $2}'`; echo $_PID | sudo tee
A/R/cgroup.procs
$ _PID=`ps aux | grep stress | awk 'NR == 4 {print $2}'`; echo $_PID | sudo tee
A/R/cgroup.procs
$ htop
```

Запустим три процесса `stress-ng` во группе `R` одновременно **на одном процессоре**:

в основной консоли (подготовка)

```
$ echo "+cpuset" | sudo tee ../cgroup.subtree_control  ①
cpuset
$ echo "+cpuset" | sudo tee ./cgroup.subtree_control
cpuset
$ echo "+cpuset" | sudo tee -a A/cgroup.subtree_control
cpuset
$ echo "+cpuset" | sudo tee -a A/{P,Q,R}/cgroup.subtree_control
cpuset

$ cat A/R/cpuset.cpus  ②
$ echo "0" | sudo tee A/R/cpuset.cpus  ③
0
$ cat A/R/cpuset.cpus
0
```

в рабочей консоли

```
$ stress-ng --metrics --verbose --cpu 3 --timeout 120s
```

в основной консоли

```
$ htop

$ _PID=`ps aux | grep stress | awk 'NR == 2 {print $2}'`; echo $_PID | sudo tee
A/R/cgroup.procs
$ _PID=`ps aux | grep stress | awk 'NR == 3 {print $2}'`; echo $_PID | sudo tee
A/R/cgroup.procs
$ _PID=`ps aux | grep stress | awk 'NR == 4 {print $2}'`; echo $_PID | sudo tee
A/R/cgroup.procs
$ htop
```

Источники

- [1] [Wikipedia] cgroups
- [2] [Kernel admin docs] Control Group v2
- [3] [NDC TechTown 2021] Michael Kerrisk. An introduction to control groups (cgroups) version 2 slides video
- [4] [NDC TechTown 2021] Michael Kerrisk. Diving deeper into control groups (cgroups) v2 slides video