user_namespaces*(7)* обзор Linux user_namespaces

ОПИСАНИЕ

Обзор пространств имён смотрите в **namespaces**(7).

Пользовательские пространства имён изолируют идентификаторы и атрибуты безопасности, в частности ID пользователя и ID группы (смотрите **credentials**(7)), корневой каталог, ключи (смотрите **keyctl**(2)) и мандаты (смотрите **capabilities**(7)). Идентификаторы пользователя и группы процесса могут отличаться внутри и снаружи пользовательского пространства имён. В частности, процесс может иметь обычный бесправный пользовательский ID снаружи и ID равный 0 внутри пространства имён; другими словами, процесс имеет доступ ко всем операциям внутри пользовательского пространства имён, но не имеет доступа к привилегированным операциям вне пространства имён.

Вложенные пространства имён, членство пространств имён

Пользовательские пространства имён могут быть вложенными, то есть каждое пользовательское пространство имён — за исключением первого («корневого») — имеет родительское пространство имён и может иметь ноль или более дочерних пространств имён. Родительское пространство имён — это пользовательское пространство имён процесса, которое создаётся с помощью вызова **unshare**(2) или **clone**(2) с флагом **CLONE_NEWUSER**.

Ядро ограничивает (начиная с версии 3.11) глубину вложенности пользовательских пространств имён 32 уровнями. Вызовы **unshare**(2) или **clone**(2), которые бы превысили это ограничение, завершаются с ошибкой **EUSERS**.

Каждый процесс является членом только одного пользовательского пространства имён. Процесс, созданный с помощью **fork**(2) или **clone**(2) без флага **CLONE_NEWUSER**, является членом того же пользовательского пространства имён что и его родитель. Однонитевой процесс может перейти в другое пользовательское пространство имён с помощью **setns**(2), если в этом пространстве у него есть мандат **CAP_SYS_ADMIN**; после перехода он получает полный набор мандатов в этом пространстве имён.

Вызов **clone**(2) или **unshare**(2) с флагом **CLONE_NEWUSER** делает новый дочерний (для **clone**(2)) или вызвавший (для **unshare**(2)) процесс членом нового пользовательского пространства имён, создаваемого вызовом.

Мандаты

Дочерний процесс, созданный **clone**(2) с флагом **CLONE_NEWUSER**, запускается в новом пользовательском пространстве имён с полным набором мандатов. Аналогично, процесс, создающий новое пользовательское пространство имён с помощью **unshare**(2) или переходящий в существующее пользовательское пространство имён с помощью **setns**(2), получает полный набор мандатов в этом пространстве имён. С другой стороны, этот процесс не имеет мандатов в родительском (в случае **clone**(2)) или предыдущем (в случае **unshare**(2)

и **setns**(2)) пользовательском пространстве имён, даже если новое пространство имён создано или переход осуществлялся суперпользователем (т. е., процесс с ID пользователя 0 в корневом пространстве имён).

Заметим, что вызов **execve**(2) приводит к пересчёту мандатов процесса обычным порядком (смотрите **capabilities**(7)), поэтому, если ID пользователя процесс не равно 0 внутри пространства имён или исполняемый файл имеет непустую маску наследования мандатов, то он теряет все мандаты. Смотрите описание отображения пользовательских и групповых ID далее.

Вызов **clone**(2), **unshare**(2) или **setns**(2) с флагом **CLONE_NEWUSER** устанавливает флаги «securebits» (смотрите **capabilities**(7)) в их значения по умолчанию (все флаги сброшены) в потомке (для **clone**(2)) или вызывающем (для **unshare**(2) или **setns**(2)). Заметим, что из-то того, что вызывающий больше не имеет мандатов в своём первоначальном пользовательском пространстве имён после вызова **setns**(2), невозможно у процесса сбросить его флаги «securebits», хотя удержать своё членство в пользовательском пространстве имён можно с помощью пары вызовов **setns**(2) — сначала переместиться в другое пользовательское пространство имён и затем вернуться в своё первоначальное пользовательское пространство имён.

Полученные мандаты внутри пользовательского пространства имён позволяют процессу выполнять операции (требующие прав) только с ресурсами, которые регулируются этим пространством имён. Правила получения процессом мандатов в определённом пользовательском пространстве имён следующие:

1. Процесс имеет мандат внутри пользовательского пространства имён, если он является членом этого пространства имён и имеет мандат в своём наборе эффективных мандатов. Процесс может получить мандаты в своём наборе эффективных мандатов различными способами. Например, он может запустить программу с битом set-user-ID или исполняемый файл, имеющий мандаты файла. Также процесс может получить мандаты при выполнении **clone**(2), **unshare**(2) или **setns**(2), как описывалось ранее.
2. Если процесс имеет мандат в пользовательском пространстве имён, то он также имеет этот мандат во всех дочерних (и позже удалённых потомках) пространствах имён.
3. При создании пользовательского пространства имён ядро записывает эффективный пользовательский ID создающего процесса как «владельца» пространства имён. Процесс, располагающийся в родительском пространстве имён пользовательского пространства имён и чей эффективный пользовательский ID совпадает с владельцем пространства имён, имеет все мандаты в пространстве имён. Предыдущее правило означает, что у процесса также есть все мандаты во всех в последствии удалённых потомках пользовательских пространств имён.

Взаимодействие между пользовательскими и другими типами пространств имён

Начиная с Linux 3.8, непривилегированные процессы могут создавать пользовательские пространства имён, а для создания пространства имён монтирования, PID, IPC, network и

UTS достаточно наличия только одного мандата **CAP_SYS_ADMIN** в пользовательском пространстве имён вызывающего.

После создания не пользовательского пространства имён оно принадлежит пользовательскому пространству имён, в котором создающий процесс являлся членом при создании пространства имён. Для действий с не пользовательским пространством имён требуются мандаты в соответствующем пользовательском пространстве имён.

Если вместе с флагами **CLONE_NEW*** указан флаг **CLONE_NEWUSER** в вызове **clone**(2) или **unshare**(2), то пользовательское пространство имён гарантированно создаётся первым, давая потомку (**clone**(2)) или вызывающему (**unshare**(2)) права на остальные пространства имён, создаваемые вызовом. Даже бесправный вызывающий может задать такую комбинацию флагов.

При создании нового пространства имён IPC, монтирования, network, PID или UTS посредством **clone**(2) или **unshare**(2), ядро записывает пользовательское пространство имён создающего процесса вместе с новым пространством имён (эту связь нельзя изменить). Когда процесс в новом пространстве имён в дальнейшем выполняет привилегированные операции, которые работают с глобальными ресурсами, изолированными пространством имён, выполняется проверка прав согласно мандатам процесса в пользовательском пространстве имён, которое ядро связало с новым пространством имён.

Ограничения у пространств имён монтирования

Отметим следующие моменты относительно пространств имён монтирования:

* Владельцем пространства имён монтирования является владелец пользовательского пространство имён. Пространство имён монтирования, чей владелец пользовательского пространства имён отличается от владельца пользовательского пространства имён родительского пространства имён монтирования, считается менее привилегированным пространством имён монтирования.
* При создании менее привилегированного пространства имён монтирования количество общих точек монтирования сокращаются до списка точек монтирования slave. Это гарантирует, что отображения, выполняемые в менее привилегированном пространстве имён монтирования, не распространятся в более привилегированные пространства имён монтирования.
* Точки монтирования, которые появились как единый блок из более привилегированного монтирования, объединяются и не могут быть разделены в менее привилегированном пространстве имён монтирования (операция **unshare**(2) **CLONE_NEWNS** переносит все точки монтирования из исходного пространства имён монтирования единым блоком и рекурсивные монтирования, которые передаются в нескольких пространствах имён монтирования, также единым блоком).
* Значения флагов **MS_RDONLY**, **MS_NOSUID**, **MS_NOEXEC** у **mount**(2) и флагов «atime» (**MS_NOATIME**, **MS_NODIRATIME**, **MS_RELATIME)** блокируются при передаче из более привилегированного в менее привилегированное пространство имён монтирования, и не могут быть изменены в менее привилегированном пространстве

имён монтирования.

* Файл или каталог, являющийся точкой монтирования в одном пространстве имён, и не являющийся в другом, может быть переименован, отсоединён (unlinked) или удалён (**rmdir**(2)) в пространстве имён монтирования, в котором он не является точкой монтирования (выполняются обычные проверки прав доступа).
Раньше, попытка переименовать, отсоединить или удалить файл или каталог, который являлся точкой монтирования в другом пространстве имён монтирования, приводила к ошибке **EBUSY**. Такое поведение вызывало технические проблемы в работе (например, NFS) и позволяло выполнять атаку отказа в обслуживании более привилегированных пользователей (т. е., не давало обновлять отдельные файлы посредством монтирования поверх их).

Отображение идентификаторов пользователей и групп: uid_map и gid_map

В новом созданном пользовательском пространстве имён отсутствует отображение пользовательских ID (ID групп) в родительское пользовательское пространство. Файл */proc/[pid]/uid_map* и */proc/[pid]/gid_map* (доступны начиная с Linux 3.5) предоставляют отображения пользовательских и групповых ID внутри пользовательского пространства имён для процесса *pid*. Эти файлы можно читать для просмотра отображений в пользовательском пространстве имён и писать (однократно) для определения отображений.

В следующих параграфах объясняется формат *uid_map*; *gid_map* имеет тот же формат, но каждый экземпляр «ID пользователя» заменяется на «ID группы».

Файл *uid_map* предоставляет отображение пользовательских ID из пользовательского пространства имён процесса *pid* в пользовательское пространство имён процесса, который открыл *uid_map* (но смотрите уточнение далее). Другими словами, процессы, которые находятся в разных пользовательских пространствах имён, возможно будут видеть разные значения при чтении соответствующего файла *uid_map*, в зависимости от отображений пользовательских ID у пользовательских пространств имён читающего процесса.

Каждая строка в файле *uid_map* определяет отображение 1-в-1 непрерывного диапазона пользовательских ID между двумя пользовательскими пространствами имён (при создании пользовательского пространства имён этот файл пуст). В каждой строке содержится три числа через пробел. Первые два числа определяют начальный пользовательский ID в каждом из двух пользовательских пространств имён. Третье число определяет длину отображаемого диапазона. Эти поля рассматриваются так:

(1) Начало диапазона пользовательских ID в пользовательском пространстве имён процесса *pid*.

(2) Начало диапазона пользовательских ID, на который отображаются пользовательские ID, указанные в первом поле. Интерпретация второго поля зависит от того, находится ли процесс, открывший *uid_map*, и процесс *pid*, в одном пользовательском пространстве имён:

a)

Если два процесса находятся в разных пользовательских пространствах имён: поле два — начало диапазона пользовательских ID в пользовательском пространстве имён процесса, который открыл *uid_map*.

б) Если два процесса находятся в одном пользовательском пространстве имён: поле два — начало диапазона пользовательских ID в родительском пользовательском пространстве имён процесса *pid*. Это позволяет открывшему *uid_map* (обычно открывают */proc/self/uid_map*) видеть отображение пользовательских ID в пользовательском пространстве имён процесса, создавшего это пользовательское пространство имён.

(3) Длина диапазона пользовательских ID, выполняющего отображение между двумя пользовательскими пространствами имён.

Системные вызовы, возвращающие пользовательские ID (ID групп), например, **getuid**(2), **getgid**(2), и мандатные поля в структуре, возвращаемой **stat**(2), возвращают пользовательский ID (ID группы), отображённый в пользовательском пространстве имён вызывающего.

Когда процесс обращается к файлу, его ID пользователя и группы отображаются в начальном пользовательском пространстве имён с целью проверки прав доступа и назначенного ID при создании файла. Когда процесс получает ID пользователя и группы файла через **stat**(2), то ID отображаются в обратном направлении, для создания значений, относительно отображений ID пользователя и группы процесса.

Начальное пользовательское пространство имён не имеет родительского пространства имён, но для однородности, для него ядро предоставляет фиктивные файлы отображения ID пользователей и групп. Посмотрим на файл *uid_map* (в *gid_map* тоже самое) из оболочки в начальном пространстве имён:

```
$ cat /proc/$$/uid_map
         0          0 4294967295
```

Данное отображение показывает, что диапазон начинающийся с пользовательского ID 0 в этом пространстве имён, отображается в диапазон, начинающийся, с 0, в (несуществующее) родительское пространство имён, и длина диапазона равна самому большому 32-битному беззнаковому целому (значение 4294967295 (32-битное знаковое значение -1) сознательно оставлено без отображения. Предназначение: *(uid_t) -1* используется в некоторых интерфейсах (например, **setreuid**(2)) для указания «отсутствия ID пользователя». Оставление *(uid_t) -1* без отображения и его не использование гарантирует, что при использовании этих интерфейсов не будет проблем).

Отображение идентификаторов пользователей и групп: запись в uid_map и gid_map

После создания нового пользовательского пространства имён в файл *uid_map один* из процессов в пространстве имён может выполнить *однократную* запись для определения отображения пользовательских ID в новом пользовательском пространстве имён. Повторная попытка записи в файл *uid_map* в пользовательском пространстве имён завершится с

ошибкой **EPERM**. Эти же правила применимы к файлам *gid_map*.

Записываемые в *uid_map* (*gid_map*) строки должны соответствовать следующим правилам:

* В трёх полях должны быть корректные числа и последнее поле должно быть больше 0.
* Строки заканчиваются символами новой строки.
* Есть (произвольное) ограничение на количество строк в файле. В Linux 3.8 ограничение равно пятью строкам. Также количество байт, записываемых в файл, должно быть меньше размера системной страницы, и запись должна выполняться в начало файла (т. е., после выполнения **lseek**(2) и **pwrite**(2) с ненулевым смещением запись в файл невозможна).
* Диапазон пользовательских ID (групповых ID), указанный в каждой строке, не должен перекрываться с диапазонами в других строках. В первой реализации (Linux 3.8) это требование удовлетворялось простейшим способом, который задавал другое требование: значения в полях 1 и 2 следующих одна за одной строк, должны увеличиваться, что не давало создавать некоторые корректные отображения. В Linux 3.9 и новее это ограничение было снято, и допустим любой набор не перекрывающихся отображений.
* В файл должна быть записана, как минимум, одна строка.

Попытки записи, нарушающие перечисленные выше правила, завершаются с ошибкой **EINVAL**.

Чтобы процесс мог записывать в файл */proc/[pid]/uid_map* (*/proc/[pid]/gid_map*) должны быть удовлетворены все условия:

1. Записывающий процесс должен иметь мандат **CAP_SETUID** (**CAP_SETGID**) в пользовательском пространстве имён процесса *pid*.
2. Записывающий процесс должен находиться в пользовательском пространстве имён процесса *pid* или внутри родительского пользовательского пространства имён процесса *pid*.
3. Отображаемые пользовательские ID (групповые ID) должны иметь соответствующее отображение в родительском пользовательском пространстве имён.
4. Одно из следующего должно быть верно:
    * Данные, записываемые в *uid_map* (*gid_map*), состоят из одной строки, которая отображает пользовательский ID (групповой ID) файловой системы записывающего процесса в родительском пользовательском пространстве имён в пользовательский ID (групповой ID) в пользовательском пространстве имён. Обычно, это одна строка, предоставляющая отображение пользовательского ID процесса, который создан в пространстве имён.
    * Открывающий процесс имеет мандат **CAP_SETUID** (**CAP_SETGID**) в

родительском пользовательском пространстве имён. То есть привилегированный процесс может создавать отображения в произвольные пользовательские ID (групповые ID) в родительском пользовательском пространстве имён.

Попытки записи, нарушающие перечисленные выше правила, завершаются с ошибкой **EPERM**.

Неотображённые пользовательские и групповые ID

Есть несколько мест, где в пользовательском пространстве могут появиться неотображённые пользовательские ID (групповые ID). Например, первый процесс в новом пользовательском пространстве имён может вызвать **getuid**() до определения отображения пользовательских ID для пространства имён. В большинстве случаев, неотображённый пользовательский ID преобразуется в пользовательский ID (групповой ID) переполнения (overflow); значение по умолчанию для пользовательского ID (группового ID) переполнения равно 65534. Смотрите описание */proc/sys/kernel/overflowuid* и */proc/sys/kernel/overflowgid* в **proc**(5).

Случаи, где неотображённые ID отображаются в таком виде, относятся к системным вызовам, которые возвращают пользовательские ID (**getuid**(2), **getgid**(2) и подобные), мандаты, передаваемые через доменный сокет UNIX, мандаты, возвращаемые **stat**(2), **waitid**(2) и System V IPC «ctl»-операциями **IPC_STAT**, мандаты, показываемые в */proc/PID/status* и файлах в */proc/sysvipc/**, мандаты, возвращаемые в поле *si_uid* структуры *siginfo_t*, полученной по сигналу (смотрите **sigaction**(2)), мандаты, записываемые в файл учёта процесса (смотрите **acct**(5)), и мандаты, возвращаемые с уведомлениями очереди сообщений POSIX (смотрите **mq_notify**(3)).

Есть один известный случай, где неотображённый пользовательский и групповой ID *не* преобразуется в соответствующее значение ID переполнения. Если при просмотре файла *uid_map* или *gid_map* обнаруживается, что для второго поля нет отображения, то поле отображается как 4294967295 (-1 для беззнакового целого);

Программы с установленными битами set-user-ID и set-group-ID

Когда процесс внутри пользовательского пространства имён выполняет программу с установленным битом set-user-ID (set-group-ID), то эффективный ID пользователя (группы) внутри пространства имён изменяется на значение, отображённое для ID пользователя (группы) файла. Однако, если ID пользователя *или* группы файла не имеет отображения внутри пространства имён, то бит set-user-ID (set-group-ID) просто игнорируется: выполняется новая программа, но эффективный ID пользователя (группы) остаётся не изменённым (такое поведение зеркально семантике выполнения программы с set-user-ID или set-group-ID, располагающейся в файловой системе, которая была смонтирована с флагом **MS_NOSUID**, как описано в **mount**(2)).

Разное

Когда ID пользователя и группы процесса передаются через доменный сокет UNIX в процесс в другом пользовательском пространстве имён (смотрите описание **SCM_CREDENTIALS** в **unix**(7)), то они транслируются в соответствующие значения согласно отображению ID пользователя и группы принимающего процесса.

СООТВЕТСТВИЕ СТАНДАРТАМ

Пространства имён есть только в Linux.

ЗАМЕЧАНИЯ

За эти годы в ядро Linux добавлено много свойств, которые были доступны только привилегированным пользователям, так как их возможности слишком велики, чтобы наделять ими приложения с set-user-ID. В целом, становится безопасно разрешать пользователю root в пользовательском пространстве имён использовать эти свойства, так как будучи в пользовательском пространстве имён, он не может получить больше прав, чем имеет root в пользовательском пространстве имён.

Доступность

Для использования пользовательских пространств имён ядро должно быть собрано с параметром **CONFIG_USER_NS**. Пользовательские пространства имён требуют поддержки во многих подсистемах ядра. Если в ядре задействована неподдерживаемая подсистема, то включить поддержку пользовательских пространств имён невозможно.

В Linux 3.8 самые важные подсистемы поддерживают пользовательские пространства имён, но значительное количество файловых систем не имеют инфраструктуры для отображения пользовательских и групповых ID между пользовательскими пространствами имён. В Linux 3.9 добавлена требуемая поддержка инфраструктуры во многие неподдерживаемые файловые системы (Plan 9 (9P), Andrew File System (AFS), Ceph, CIFS, CODA, NFS и OCFS2). В Linux 3.11 добавлена поддержка в последние основные файловые системы (XFS).

ПРИМЕР

Представленная далее программа разработана для экспериментов с пользовательскими пространствами имён. Она создаёт пространства имён согласно параметрам командной строки и затем выполняет команду внутри этих пространств имён. В комментариях и функции *usage()* предоставлено полное описание программы. Следующий сеанс оболочки показывает её работу.

Сначала, посмотрим на окружение выполнения:

```
$ uname -rs     # требуется Linux 3.8 или новее
Linux 3.8.0
$ id -u         # работа от непривилегированного пользователя
1000
$ id -g
```

```
1000
```

Теперь запустим новую оболочку в новых пользовательском (*-U*), монтирования (*-m*) и PID (*-p*) пространствах имён с пользовательским (*-M*) и групповым ID (*-G*) 1000, отображающимся в 0 внутри пользовательского пространства имён:

```
$ ./userns_child_exec -p -m -U -M '0 1000 1' -G '0 1000 1' bash
```

У оболочки PID равен 1, так как это первый процесс в новом пространстве имён PID:

```
bash$ echo $$
1
```

Внутри пользовательского пространства имён идентификаторы пользователя и группы оболочки равны 0, и она имеет полный набор разрешённых и эффективных мандатов:

```
bash$ cat /proc/$$/status | egrep '^[UG]id'
Uid:    0       0       0       0
Gid:    0       0       0       0
bash$ cat /proc/$$/status | egrep '^Cap(Prm|Inh|Eff)'
CapInh: 0000000000000000
CapPrm: 0000001fffffffff
CapEff: 0000001fffffffff
```

Смонтируем новую файловую систему */proc* и просмотрим все процессы, видимые в новом пространстве имён PID; убедимся, что оболочка не видит ни одного процесса вне своего пространства имён PID:

```
bash$ mount -t proc proc /proc
bash$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 pts/3    S      0:00 bash
   22 pts/3    R+     0:00 ps ax
```

Исходный код программы

```
/* userns_child_exec.c
   Лицензируется на условиях Универсальной общественной лицензии
   GNU версии 2 и новее
   Создаёт дочерний процесс, который запускает командную оболочку
   в новых пространствах имён; может выполнять отображение UID и GID,
   если они указаны при создании пользовательского пространства имён.
*/
#define _GNU_SOURCE
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <signal.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <errno.h>
/* Простая функция обработки ошибок: выводит сообщение об ошибке согласно
   значению в «errno» и завершает вызвавший процесс */
#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)
struct child_args {
```

```
    char **argv;        /* команда, выполняемая потомком с параметрами */
    int    pipe_fd[2];  /* канал для синхронизации родителя и потомка */
};
static int verbose;
static void
usage(char *pname)
{
    fprintf(stderr, "Использование: %s [параметры] кмд [арг…]\n\n", pname);
    fprintf(stderr, "Создаёт дочерний процесс, который запускает командную "
            "оболочку в новом пользовательском пространстве имён,\n"
            "и, возможно, также в других новых пространствах имён.\n\n");
    fprintf(stderr, "Параметры:\n\n");
#define fpe(str) fprintf(stderr, "    %s", str);
    fpe("-i          Новое пространство имён IPC\n");
    fpe("-m          Новое пространство имён монтирования\n");
    fpe("-n          Новое сетевое пространство имён\n");
    fpe("-p          Новое пространство имён PID\n");
    fpe("-u          Новое пространство имён UTS\n");
    fpe("-U          Новое пользовательское пространство имён\n");
    fpe("-M uid_map  карта UID для пользовательского пространства имён\n");
    fpe("-G gid_map  карта GID для пользовательского пространства имён\n");
    fpe("-z          Отображать пользовательский UID и GID в 0 в
пользовательском пространстве имён\n");
    fpe("            (эквивалентно: -M '0 <uid> 1' -G '0 <gid> 1')\n");
    fpe("-v          показывать дополнительные сообщения\n");
    fpe("\n");
    fpe("Если указан -z, -M или -G, то требуется -U.\n");
    fpe("Нельзя указывать -z вместе с -M или -G.\n");
    fpe("\n");
    fpe("Строка карты для -M и -G состоит из записей вида:\n");
    fpe("\n");
    fpe("    ID-внутри-ns   ID-вне-ns   длина\n");
    fpe("\n");
    fpe("Строка карты может содержать несколько записей через запятую;\n");
    fpe("запятые замещаются на символы новой строки перед записью"
        " в файлы карт.\n");
    exit(EXIT_FAILURE);
}
/* Обновляем файл отображения «map_file» значением из
   «mapping» — строкой, в которой определены отображения UID или GID.
   Отображения UID или GID состоят из одной или более записей
   (разделённых символом новой строки) вида:
       ID-внутри-ns    ID-снаружи-ns   длина
   Требовать от пользователя указывать строку с символами новой строки
   в командной строке неприемлемо. Поэтому мы позволим использовать
   для разделения записей запятые и заменим их символами новой строки
   перед записью строки в файл. */
static void
update_map(char *mapping, char *map_file)
{
    int fd, j;
    size_t map_len;     /* длина «mapping» */
    /* Заменяем запятые на символы новой строки в строке отображения */
    map_len = strlen(mapping);
    for (j = 0; j < map_len; j++)
        if (mapping[j] == ',')
            mapping[j] = '\n';
    fd = open(map_file, O_RDWR);
    if (fd == -1) {
        fprintf(stderr, "ОШИБКА: open %s: %s\n", map_file,
                strerror(errno));
        exit(EXIT_FAILURE);
    }
    if (write(fd, mapping, map_len) != map_len) {
```

```
        fprintf(stderr, "ОШИБКА: write %s: %s\n", map_file,
                strerror(errno));
        exit(EXIT_FAILURE);
    }
    close(fd);
}
static int              /* Начальная функция клонированного потомка */
childFunc(void *arg)
{
    struct child_args *args = (struct child_args *) arg;
    char ch;
    /* Ждём пока родитель обновит отображения UID и GID.
       Смотрите комментарий в main(). Мы ждём конца файла в канале,
       который будет закрыт родительским процессом после обновления
       отображений. */
    close(args->pipe_fd[1]);    /* закрываем наш дескриптор для записи
                                   конца канала для того, чтобы мы
                                   увидели EOF, когда родитель закроет
                                   свой дескриптор */
    if (read(args->pipe_fd[0], &ch, 1) != 0) {
        fprintf(stderr,
                "Ошибка в потомке: при чтении из канала получен != 0\n");
        exit(EXIT_FAILURE);
    }
    /* Запускаем командную оболочку */
    printf("O exec %s\n", args->argv[0]);
    execvp(args->argv[0], args->argv);
    errExit("execvp");
}
#define STACK_SIZE (1024 * 1024)
static char child_stack[STACK_SIZE];    /* место под стек в потомке */
int
main(int argc, char *argv[])
{
    int flags, opt, map_zero;
    pid_t child_pid;
    struct child_args args;
    char *uid_map, *gid_map;
    const int MAP_BUF_SIZE = 100;
    char map_buf[MAP_BUF_SIZE];
    char map_path[PATH_MAX];
    /* Разбираем параметры командной строки. Начальный символ «+» в
       последнем аргументе getopt() предотвращает подстановку параметров
       командной строки в стиле GNU. Это полезно, так как иногда
       «команда», выполняемая этой программой, сама имеет параметры
       командной строки. Мы не хотим, чтобы getopt() передала их
       нашей программе. */
    flags = 0;
    verbose = 0;
    gid_map = NULL;
    uid_map = NULL;
    map_zero = 0;
    while ((opt = getopt(argc, argv, "+imnpuUM:G:zv")) != -1) {
        switch (opt) {
        case 'i': flags |= CLONE_NEWIPC;        break;
        case 'm': flags |= CLONE_NEWNS;         break;
        case 'n': flags |= CLONE_NEWNET;        break;
        case 'p': flags |= CLONE_NEWPID;        break;
        case 'u': flags |= CLONE_NEWUTS;        break;
        case 'v': verbose = 1;                  break;
        case 'z': map_zero = 1;                 break;
        case 'M': uid_map = optarg;             break;
        case 'G': gid_map = optarg;             break;
        case 'U': flags |= CLONE_NEWUSER;       break;
```

```
            default:  usage(argv[0]);
            }
        }
        /* -M или -G без -U не имеют смысла */
        if (((uid_map != NULL || gid_map != NULL || map_zero) &&
                    !(flags & CLONE_NEWUSER)) ||
                (map_zero && (uid_map != NULL || gid_map != NULL)))
            usage(argv[0]);
        args.argv = &argv[optind];
        /* Мы используем канал для синхронизации родителя и потомка, чтобы
           родитель настроил отображения UID и GID до того, как потомок
           вызовет execve(). Это гарантирует, что потомок предъявит свои
           мандаты при execve(); обычно мы хотим отобразить эффективный
           пользовательский ID потомка в 0 в новом пользовательском
           пространстве имён. Без этой синхронизации потомок потерял
           бы свои мандаты при вызове execve() с ненулевым пользовательским
           ID (смотрите в справочной странице capabilities(7) подробности
           преобразования мандатов процесса при execve()). */
        if (pipe(args.pipe_fd) == -1)
            errExit("pipe");
        /* создаём потомка в новом пространстве имён */
        child_pid = clone(childFunc, child_stack + STACK_SIZE,
                          flags | SIGCHLD, &args);
        if (child_pid == -1)
            errExit("clone");
        /* предок попадает сюда */
        if (verbose)
            printf("%s: PID потомка, созданного clone(): %ld\n",
                    argv[0], (long) child_pid);
        /* обновляем отображения UID и GID в потомке */
        if (uid_map != NULL || map_zero) {
            snprintf(map_path, PATH_MAX, "/proc/%ld/uid_map",
                    (long) child_pid);
            if (map_zero) {
                snprintf(map_buf, MAP_BUF_SIZE, "0 %ld 1", (long) getuid());
                uid_map = map_buf;
            }
            update_map(uid_map, map_path);
        }
        if (gid_map != NULL || map_zero) {
            snprintf(map_path, PATH_MAX, "/proc/%ld/gid_map",
                    (long) child_pid);
            if (map_zero) {
                snprintf(map_buf, MAP_BUF_SIZE, "0 %ld 1", (long) getgid());
                gid_map = map_buf;
            }
            update_map(gid_map, map_path);
        }
        /* закрываем конец канала на стороне записи для сообщения потомку
           о том, что мы обновили отображения UID и GID */
        close(args.pipe_fd[1]);
        if (waitpid(child_pid, NULL, 0) == -1)      /* ждём потомка */
            errExit("waitpid");
        if (verbose)
            printf("%s: завершение\n", argv[0]);
        exit(EXIT_SUCCESS);
    }
```