Системное программирование

Лекция 11

Псевдотерминалы

Сеансы и группы процессов

Демоны

Журнал событий

Терминалы и псевдотерминалы

Терминал – отдельное устройство, предназначенное для организации взаимодействия пользователя с ЭВМ.

Псевдотерминал – программная абстракция, имитирующая терминал.

Т.к. в UNIX «все есть файл», терминалы и псевдотерминалы представляются в виде файлов. Управляющий терминал процесса доступен в виде файла **/dev/tty**.

Для псевдотерминалов выделяют *master*- и *slave*-концы. Сеансы пользователей подключаются к slave-концам псевдотерминала. К master-концам подключаются программы, осуществляющие отображение или передачу данных (эмулятор терминала, sshd/telnetd и пр.).

В Linux используются 2 реализации псевдотерминалов: BSD (файлы **/dev/tty*** *и* **/dev/pty***.) и System V (файлы **/dev/pts*** и **/dev/ptmx**)

Дисциплина линии

Считается, что с (псевдо)терминалом связаны входная и выходная очереди. Во входную очередь помещается пользовательский ввод, в выходную очередь помещаются данные для отображения на терминале. За управление очередями отвечает модуль ОС, называемый **дисциплиной линии.**

Символы ASCII из диапазона 0x00-1F и символ 0x7F являются управляющими. При возникновении данных символов во входной очереди дисциплина линии связи выполняет дополнительные действия. Например, символ 0x08 (BS, Backspace) приводит к удалению 1 символа из входной очереди. Большинство управляющих символов поглощаются дисциплиной линии и не передаются приложению, исключение – символ 0x0A (LF, Line Feed, ‘\n’).

Некоторые управляющие символы заставляют дисциплину линии послать подключенному процессу некоторый сигнал. Например, символ 0x03 (ETX, End-of-Text, Control-C) превращается в сигнал SIGINT.

Правила обработки специсимволов могут быть изменены функцией `tcsetattr()`.

По умолчанию дисциплина линии связи работает в *каноническом режиме* – данные считываются построчно (отправляются считывающему процессу по спецсимволу LF). Приложение может перевести линию в *неканонический режим*, в котором данные могут считываться произвольным образом.

Группа процессов

Группа процессов – набор из одного или более процессов.

- Группа процессов имеет свой идентификатор (PGID), который равен идентификатору **лидера группы процессов**.
- Процесс, созданный `fork()`, изначально находится в той же группе, что и процесс-родитель.
- Процессы можно перемещать между группами процессов одного сеанса.
- Группам процессов можно массово рассылать и ожидать завершения всех процессов группы (см. pid<0 для `kill()` и `waitpid()`).

Основные вызовы для групп процессов

```
/* получить PGID указанного процесса*/
pid_t getpgid(pid_t pid);

/* получить PGID текущего процесса*/
pid_t getpgrp();


/* приосоединить процесс к указанной группе*/
/* setpgid(0,0) создаст новую группу с текущим процессом*/
int setpgid(pid_t pid, pid_t pgid);
```

Группы переднего и заднего плана

В каждом сеансе есть единственная **группа переднего плана**. Все остальные группы являются группами заднего плана.

Номер группы переднего плана совпадает с номером группы терминала, которую можно получить и изменить следующими вызовами.

```
pid_t tcgetpgrp(int fd); /*получить PGID терминала*/
int tcsetpgrp(int fd, pid_t pgrp); /*изменить PGID терминала*/
```

Дескриптор fd должен быть дескриптором управляющего терминала (/dev/tty).

Процессы группы переднего плана могут производить ввод/вывод в терминал, а также получают от терминала сигналы (Ctrl-C = SIGINT группе переднего плана).

Сигналы SIGTTIN и SIGTTOU

Если процесс группы заднего плана пытается прочитать данные из терминала, всеем процессам группы посылается сигнал SIGTTIN. Если процесс-читатель игнорирует или блокирует этот сигнал, попытка чтения завершается с ошибкой EIO.

Аналогичное поведение возникает при попытке ввода, с той разницей, что посылается сигнал SIGTTOU. Кроме того, вывод от групп заднего плана можно разрешить (см. *stty -tostop* или `tcsetattr(), TOSTOP`).

Кроме того, сигнал SIGTTOU посылается, если процесс заднего плана выполняет вызов `tcsetpgrp()` – т.е., пытается изменить номер группы переднего плана.

И SIGTTIN, и SIGTTOU по умолчанию останавливают процесс.

Группы процессов и оболочка

Группы процессов нужны оболочке для 2-х вещей:

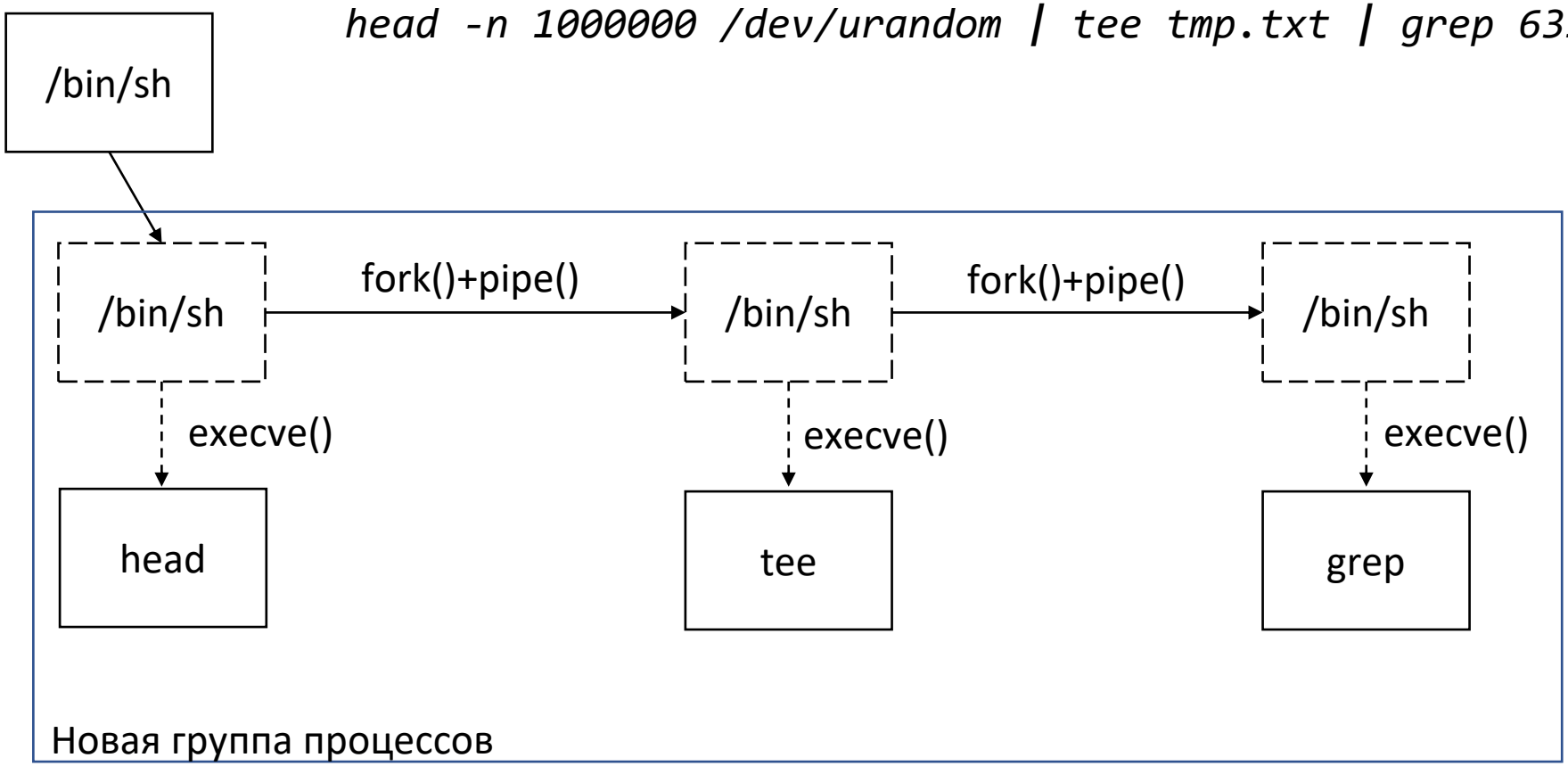
1. Для обработки конвейеров (a|b|c) – чтобы посылать сигналы на завершение всем процессам разом.
2. Для обработки фоновых заданий (a|b|c&)

Запуск команд оболочкой

1. Оболочка создает свою копию вызовом fork() и вызывает `setpgid(`*`child_pid`*`,`*`child_pid`*`)`.
2. Копия оболочки вызывает `setpgid(0,0)` и , если команда не заканчивается на &, временно блокирует сигнал SIGTTOU и забирает управляющий терминал вызовом `tcsetpgrp(0, getpgrp())`.
3. Исходная оболочка вызывает `waitpid(child_id, …, WUNTRACED)`.
4. Если команда не является последним элементом конвейера:
   1. Копия оболочки создает канал вызовом pipe();
   2. Копия оболочки выполняет fork();
   3. «Старая» копия вызовом dup2() дублирует дескриптор входного конца канала в дескриптор 1.
   4. «Новая» копия вызовом dup2() дублирует дескриптор выходного конца канала в дескриптор 0.
   5. Дескрипторы, возвращенные pipe(), закрываются.
5. Если конвейер не кончился – goto 4.
6. При наличии перенаправлений ввода-вывода (символы >,<), копия оболочки открывает файлы и переназначает дескрипторы 0,1,2 вызовом dup2().
7. Копия оболочки выполняет exec() и начинает выполнение программы.
8. Исходная оболочка дожидается изменения состояния (завершения либо остановки) процессов группы, после чего возвращает управление терминалом себе.

Конвейеры

```
head -n 1000000 /dev/urandom | tee tmp.txt | grep 6311 > result.txt
```



Задания

Современные оболочки поддерживают задания – наборы команд, выполняющиеся одновременно.

Чтобы создать фоновое задание, в конце сроки с командой ставится символ &.

```
$ head -n 1000000 /dev/urandom | grep 6311 > result_bg.txt &
$ head -n 1000000 /dev/urandom | grep 6312 > result_fg.txt
```

Переключение между заданиями осуществляется командой *fg*.

Для того, чтобы отправить текущее задание в фон, нужно остановить его (например, через Ctrl-Z = SIGTSTP), а затем выполнить команду *bg*.

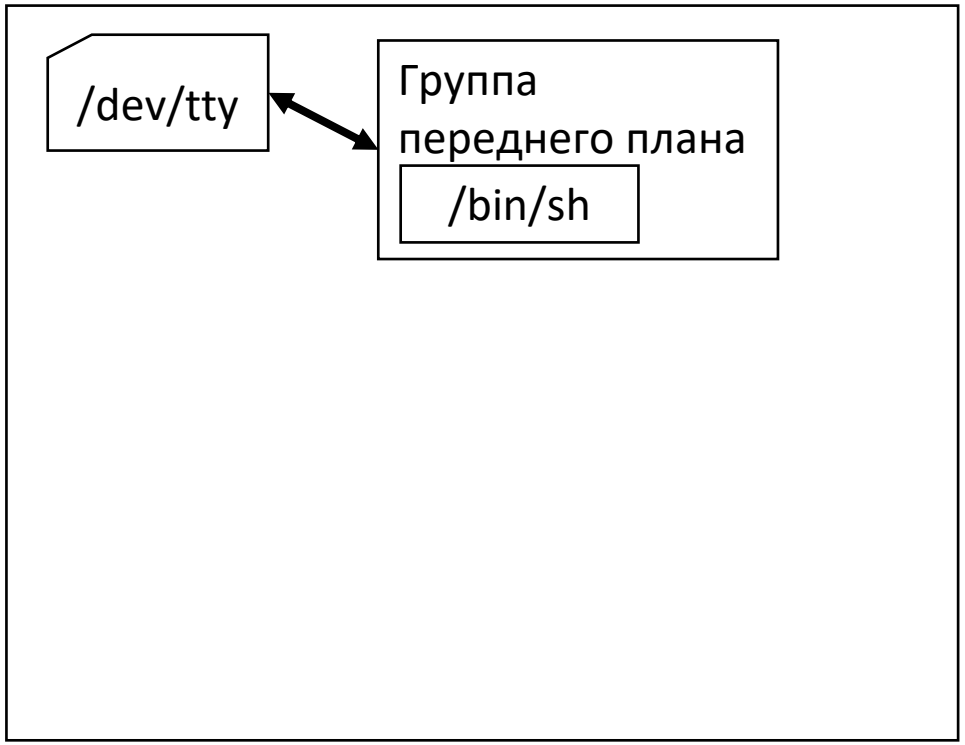
Получить список заданий можно командой *jobs*.

Задания

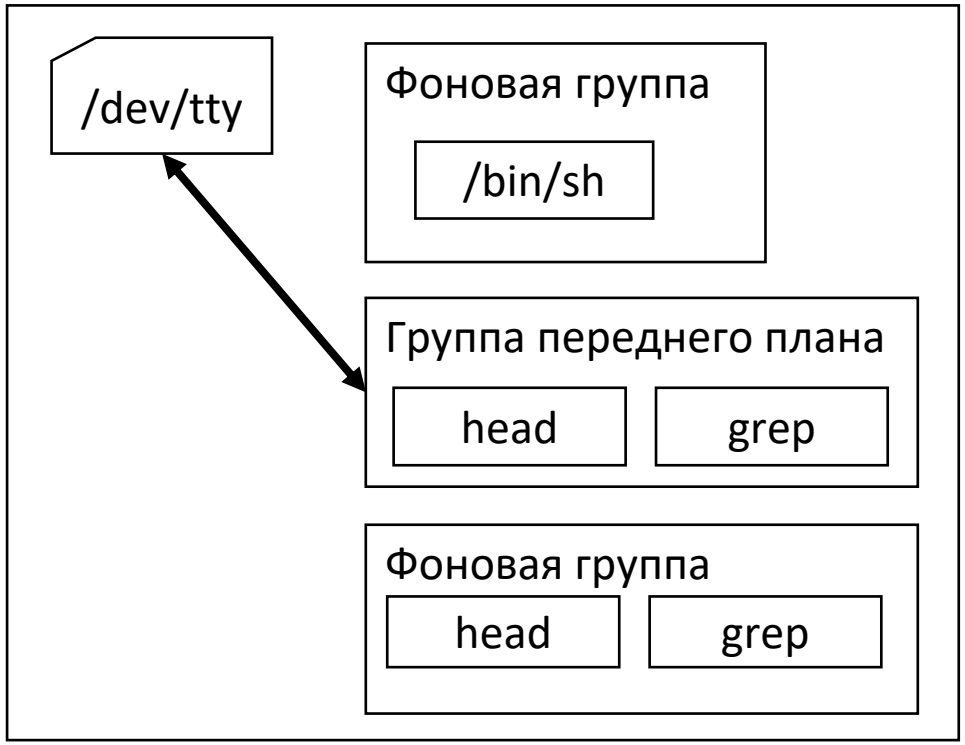
```
$ head -n 1000000 /dev/urandom | grep 6311 > result_bg.txt &

$ head -n 1000000 /dev/urandom | grep 6312 > result_fg.txt
```

Сеанс до выполнения команд



Сеанс во время выполнения команд



Осиротевшие группы

Осиротевшей группой процессов является группа, в которой у каждого процесса родитель либо находится в этой же группе процессов, либо находится в другом сеансе.

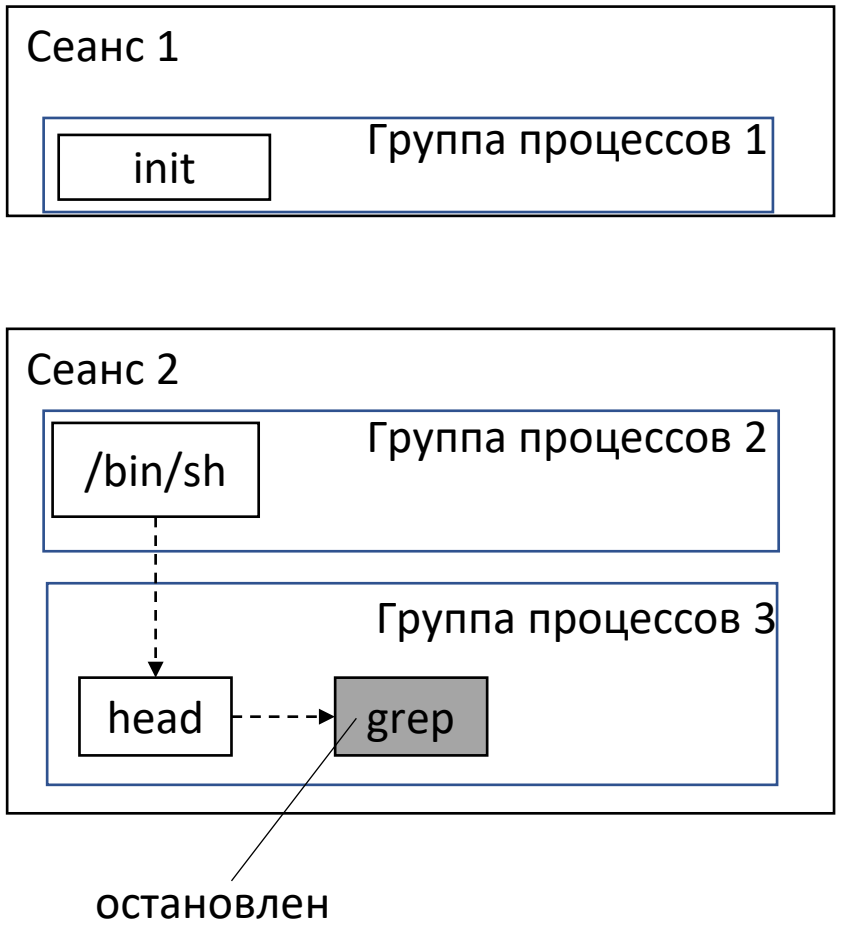
Если группа процессов становится осиротевшей, и в этой группе есть остановленные процессы, то ОС посылает всем процессам группы сигнал SIGHUP и SIGCONT.

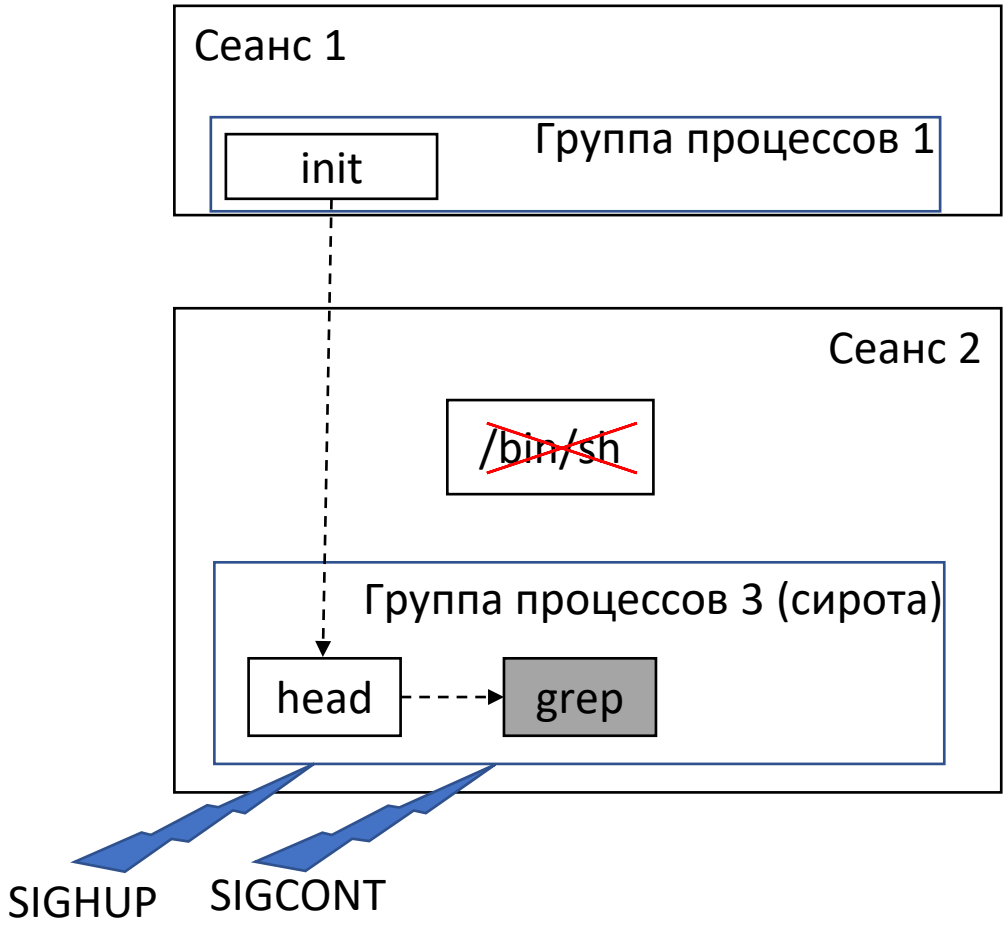
Сигнал SIGHUP означает потерю связи с управляющим терминалом, и *по умолчанию уничтожает процесс*.

Сигнал SIGCONT позволяет процессам, пережившим SIGHUP, продолжить выполнение, если они до этого были остановлены.

Логика проста – если группа процессов осиротела, то (скорее всего) некому разбудить остановленные процессы. Поэтому их надо либо уничтожить, либо разбудить.

Осиротевшие группы





Сеансы

Сеанс (session)– набор групп процессов, ассоциированных с не более чем 1 управляющим терминалом.

- Сеанс имеет свой идентификатор (SID).
- В сеансе есть как минимум одна группа процессов.
- Процесс, создавший сеанс, становится *лидером сеанса*. Идентификатор сеанса равен PGID лидера сеанса.
- С сеансом может быть связан 1 управляющий терминал, либо вообще не связано ни одного терминала.
- Управляющий терминал доступен всем процессам сеанса как файл **/dev/tty**. Чтение из /dev/tty равносильно чтению из терминала, запись - выводу в терминал.
- При потере связи с управляющим терминалом (например, из-за закрытия окна эмулятора терминала) лидеру сеанса посылается сигнал SIGHUP.
- При завершении лидера сеанса всем процессам группы переднего плана посылается сигнал SIGHUP.

Управление сеансами

Сеанс (session)– набор групп процессов, ассоциированных с не более чем 1 управляющим терминалом.

```
/*получить SID процесса */
pid_t getsid(pid_t pid);

/*создать новый сеанс (если процесс - не лидер группы)*/
pid_t setsid();
```

Процесс, вызвавший `setsid()`, становится новым лидером сеанса без терминала (управляющий терминал теряется). *Процесс должен быть готов к сигналу SIGHUP*.

Если процесс – лидер группы, то вызов завершится с ошибкой.

Получение псевдотерминала

Для открытия свободного псевдотерминала используется функция posix_openpt():

```
int posix_openpt(int flags);
```

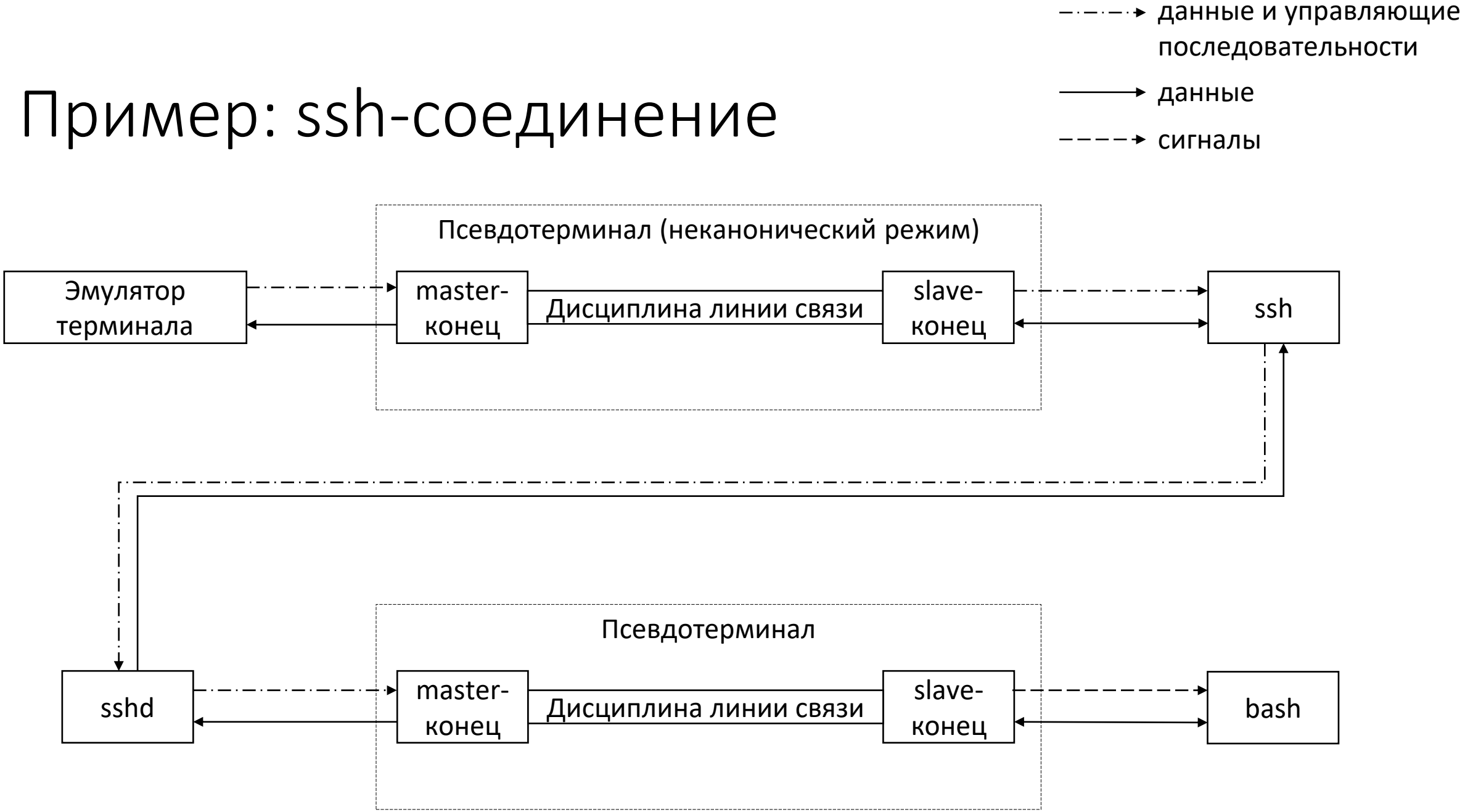
В параметре flags могут передаваться флаги O_RDWR (открыть терминал на чтение/запись) и O_NOCTTY (открыть терминал, но не делать его управляющим) или их комбинация.

Функция возвращает дескриптор, связанный с master-концом псевдотерминала. Slave-конец создается автоматически. Для получения имени slave-конца псевдотерминала по дескриптору master-конца используется функция `char* ptsname(int fd)`.

Сразу после создания slave-конец заблокирован. Для разблокировки slave-конца необходимо вызвать функции `int grantpt(int fd)` и `int unlockpt(int fd)`, передав им дескриптор master-конца. После этого slave-конец псевдотерминала будет доступен всем процессам пользователя, а также всем членам специальной группы пользователей *tty* и может быть открыт вызовом open().

Если процесс является лидером сеанса без управляющего терминала, то открытие slave-конца свободного псевдотерминала автоматически сделает данный терминал управляющим для сеанса, если при открытии не указан флаг O_NOCTTY .

Пример: ssh-соединение



Процессы-демоны

Процесс-демон (daemon) – процесс из сеанса, не связанного с управляющим терминалом.

Демонами обычно являются системные процессы, управляющие работой некоторого компонента (например, сети).

- Процесс-демон не имеет терминала -> обычная коммуникация с демоном невозможна.
- Параметры демона хранятся в конфигурационном файле. По общему правилу, конфигурационный файл должен иметь расширение *.conf* и располагаться в каталоге */etc*. Конфигурационный файл читается в момент запуска или перезапуска демона.
- Сообщения от демона выводятся в системный журнал либо в лог-файл.
- Поскольку демон не имеет управляющего терминала, сигнал SIGHUP для него имеет специальное значение - он вызывает перезапуск демона и чтение конфигурационного файла.

Самым известным демоном является процесс init.

Назначение демонов

Часто демоны являются процессами, управляющими некоторым системным компонентом или ресурсом.

Если доступ к ресурсу должен быть предоставлен другим процессам, то демон ресурса часто строится на основе клиент-серверной архитектуры, либо предоставляет другие способы взаимодействия

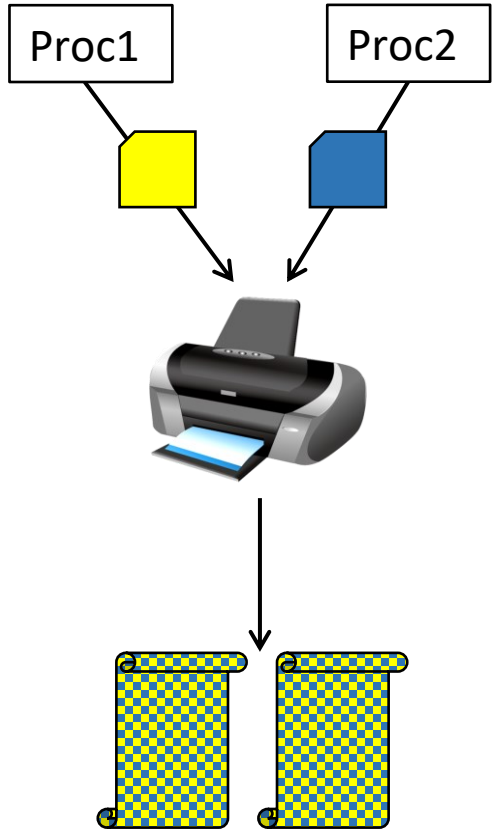
Примерами демонов являются демон виртуальной памяти (swapd), демон системного журнала (syslogd или journald), демон Bluetooth (bluetoothd) и пр.

Пример: простой демон печати

Принтер является общим ресурсом системы.

Если два процесса попытаются напечатать что-то, имея доступ к принтеру напрямую, то результат будет некорректным.

Поскольку принтер является ресурсом системного уровня, требуется наличие арбитра системного уровня, который будет управлять доступом к принтеру.

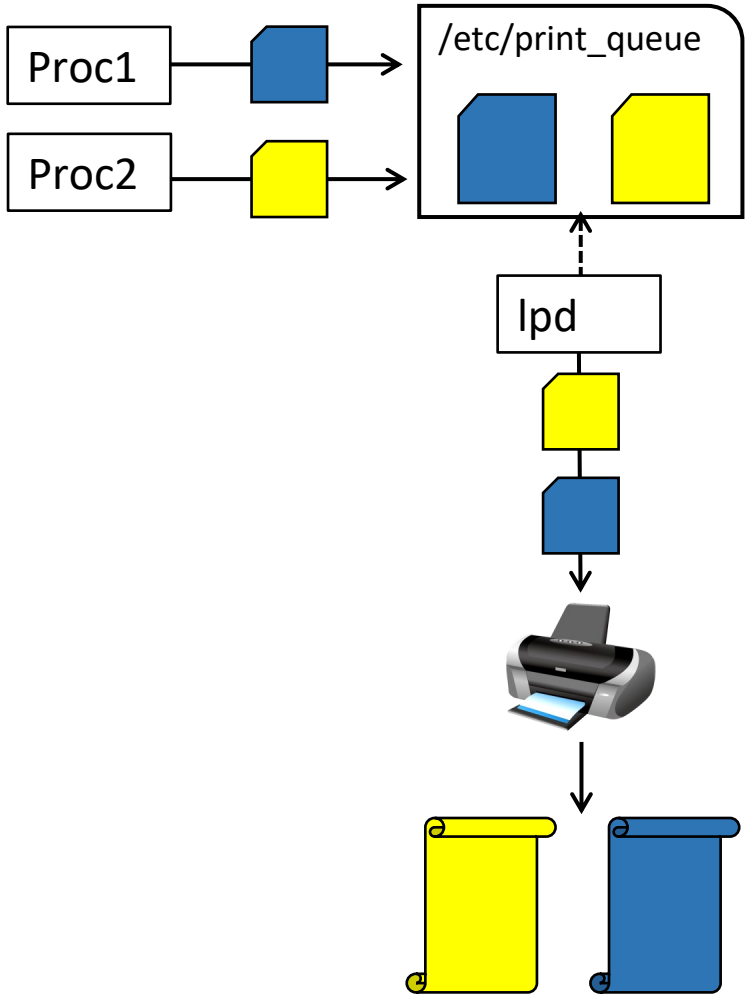


Пример: простой демон печати

Проблема решается, если к принтеру будет иметь доступ единственный процесс - демон принтера (line printer daemon, lpd).

В простом варианте для направления файла на печать он помещается в выделенный каталог (например, /etc/print_queue).

Демон принтера периодически сканирует этот каталог, и если в нем есть файл - печатает его и удаляет из каталога.

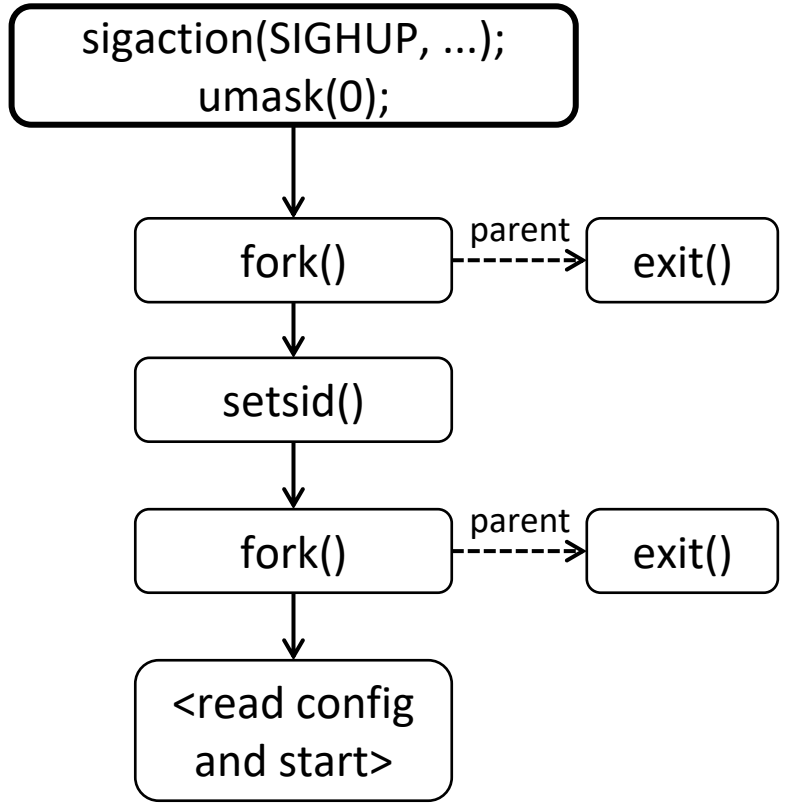


Создание демона

Для создания демона достаточно создать процесс, который находится в сеансе , не имеющем управляющего терминала.

На первом этапе временно игнорируется сигнал SIGHUP и, на всякий случай, сбрасывается маска прав.

На этом же этапе закрываются все открытые файловые дескрипторы.

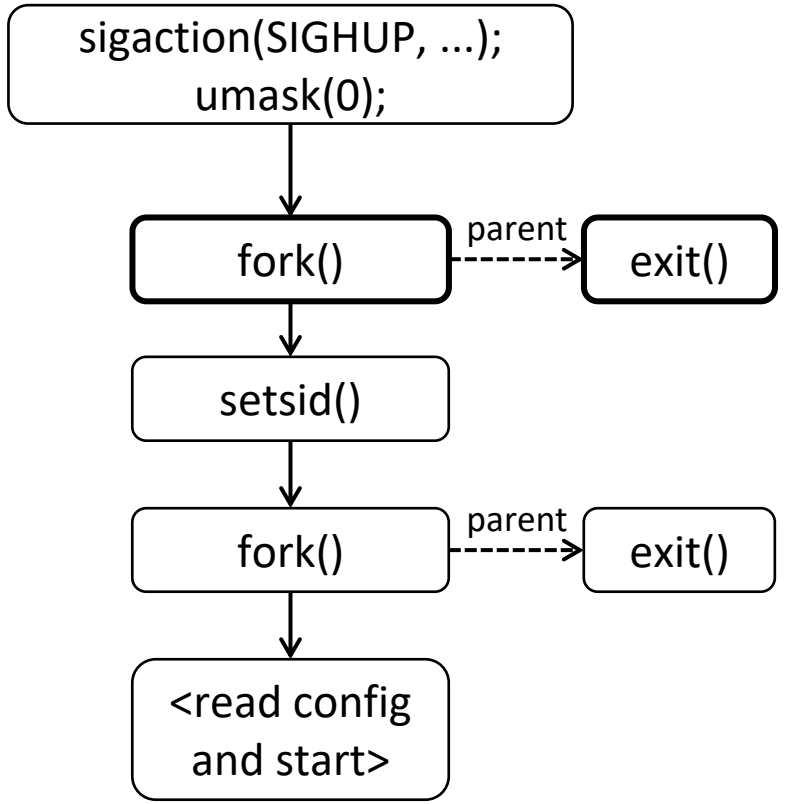


Создание демона

На втором этапе процесс выполняет `fork()`.

Выполнение `fork()` гарантирует, что дочерний процесс не будет лидером группы (иначе `setsid()` завершится с ошибкой).

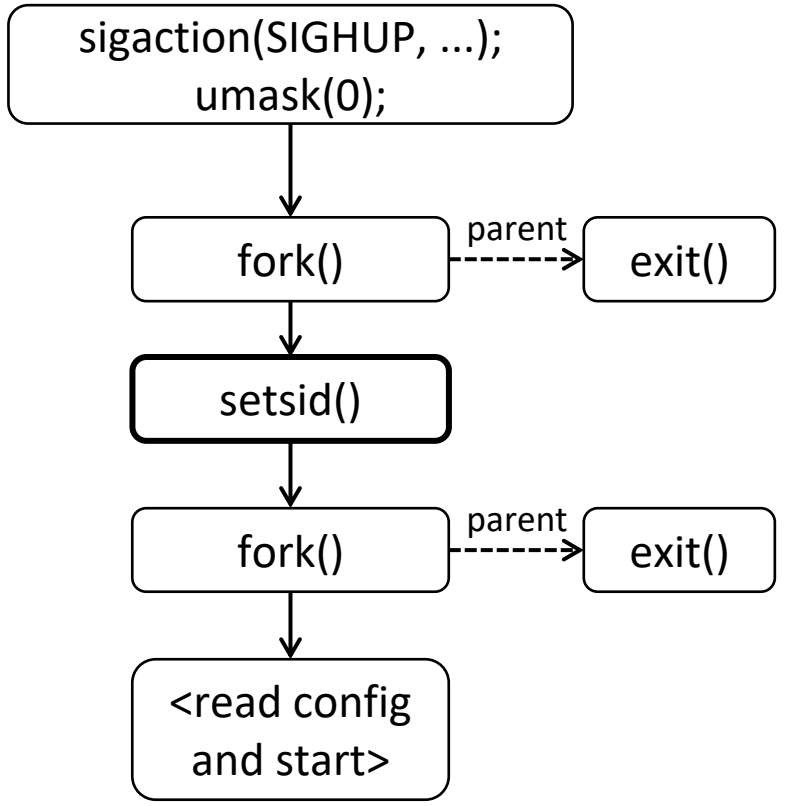
Родительский процесс сразу после `fork()` выполняет `exit()`.



Создание демона

На третьем этапе процесс выполняет вызов `setsid()` и становится лидером нового сеанса и лидером новой группы процессов внутри этого сеанса.

При этом процесс теряет связь с управляющим терминалом старого сеанса, однако процесс потенциально все еще может получить новый управляющий терминал.

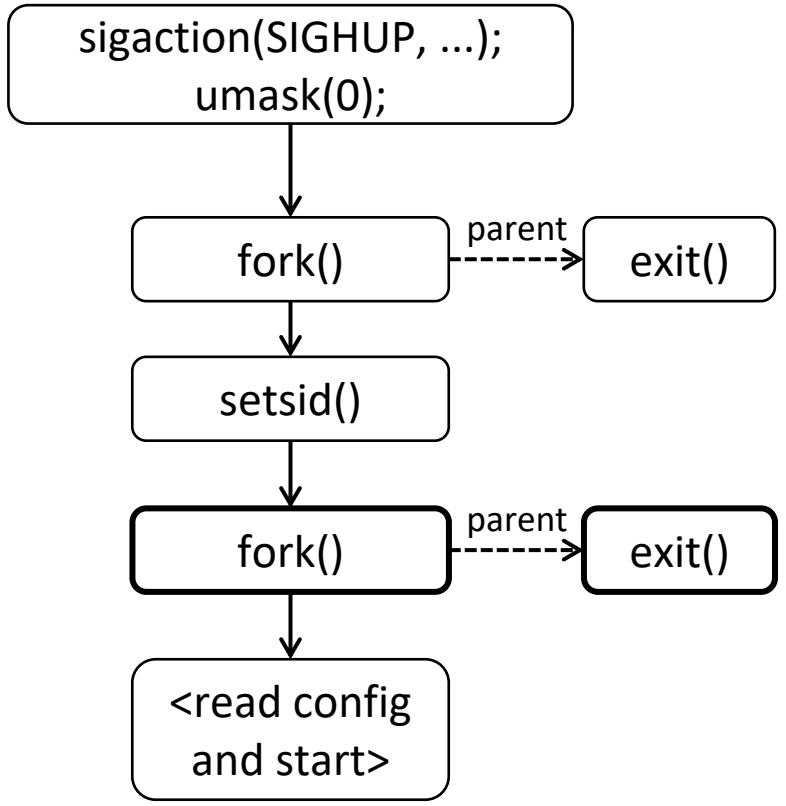


Создание демона

На четвертом этапе процесс опять выполняет `fork()`, после чего родительский процесс выполняет `exit()`.

Дочерний процесс уже не будет ни лидером сеанса, ни лидером группы. Как следствие, он не сможет получить новый управляющий терминал.

После 4 этапа процесс, при необходимости, устанавливает новый обработчик сигнала `SIGHUP`, читает файл конфигурации и начинает выполнение работы.



Запрет дубликатов демона

Формально, любую программу можно запустить в нескольких экземплярах. В случае с демонами это часто недопустимо.

Запрет дубликатов решается созданием файлов по фиксированному пути с последующей блокировкой данного файла с помощью `flock()`.

По общему соглашению, подобные файлы имеют имя *<daemon_name>.pid*, создаются в каталоге */var/run* и должны содержать PID процесса, держащего блокировку.

Если дубликат попытается запуститься, но не сможет захватить блокировку - он должен будет завершиться (обеспечить подобное поведение - задача программиста).

Запрет дубликатов демона

```
int fd = open("/var/run/mydaemon.pid", O_CREAT | O_RDWR, S_IRWXU);

if (flock(fd, LOCK_EX | LOCK_NB) != 0)
    exit(-1); //exit if failed to get the lock

char buf[17];
int len = sprintf(buf, "%ud", getpid()); //convert pid to str
ftruncate(fd, 0);
write(fd, buf, len); //write to .pid-file
```

Журнал событий

Практически в любой операционной системе ведется журнал событий.

В большинстве дистрибутивов Linux системный журнал ведется демоном journald (если дистрибутив использует systemd в качестве системы инициализации).

Данный демон использует сокеты домена UNIX для получения сообщений от остальных процессов.

В POSIX существует стандартный способ взаимодействия с системным журналом и, как следствие, с демоном журнала.

Журнал событий

Функция `openlog()` устанавливает связь с системным журналом.

```
#include <syslog.h>
void openlog(const char* ident, int option, int facility);
```

Параметры:

`ident` – префикс, с которого будут начинаться строки с описанием событий;
`option` – флаги [необязательно];
`facility` - тип источника записи.

Параметр `facility` неявно указывает итоговое место хранения записи. Рекомендуется передавать 0 (LOG_USER) в качестве значения. Допускается через ИЛИ передавать также уровень записи по умолчанию (e.g. LOG_USER|LOG_WARNING).

После завершения работы с журналом соединение может быть закрыто функцией
`void closelog();`
Вызов функции `closelog()` является необязательным.

Журнал событий

Функция `syslog()` создает новую запись в журнале.

```
#include <syslog.h>
void syslog(int priority, const char *format, ...);
```

Параметры:

`priority` - тип и серьезность причины (LOG_DEBUG-LOG_EMERG).

Дальнейшие параметры аналогичны `printf` (строка формата + аргументы).

Допускается использовать syslog() без предварительного `openlog` (будет вызван по умолчанию с `ident`=имя программы, `option`=0, `facility`=0).

Просмотр журнала

На большинстве дистрибутивов Linux, использующих systemd как систему инициализации, журнал можно посмотреть командой *journalctl*.

Данная команда служит как для просмотра журнала, так и для управления демоном журнала. Команда позволяет также проводить поиск и фильтрацию логов.

Например следующая команда выведет все записи, начиная с последней загрузки, с приоритетом не больше 3 (т.е. ошибки или серьезнее), содержащие подстроку NVIDIA

$ *journalctl --boot --priority 3 --grep NVIDIA*

Просмотр журнала

В дистрибутивах, которые используют стандартный init вместо systemd, журнал можно просмотреть, читая файл */var/log/syslog.*

Например, просмотреть записи за 14 декабря, связанные с ядром, можно командой

$ sudo cat /var/log/syslog | grep "Dec 14" | grep kernel