# 1 Работа с процессами в POSIX-системах

Понятие «процесс» наряду с понятием «файл» относится к основным понятиям операционной системы. Под процессом можно понимать программу в стадии выполнения. С процессом в системе связано много атрибутов, например, страницы памяти, занимаемые процессом, или идентификатор пользователя. Процесс использует ресурсы системы, поэтому одна из основных задач ядра операционной системы — распределение ресурсов между процессами.

Каждый процесс имеет своё адресное пространство. Если в системе в текущий момент есть несколько готовых к выполнению процессов, они работают параллельно. На системе с одним процессором эти процессы разделяют время одного процессора, а на системе с несколькими процессорами каждый процесс может выполняться на своём процессоре, и они будут работать действительно параллельно. Параллелизм выполнения вносит принципиально новые моменты по сравнению с обычным последовательным программированием и делает параллельные программы значительно более трудоёмкими в разработке и отладке.

## 1.1 Создание процесса

Новый процесс создаётся с помощью системного вызова **fork**.

```
#include <unistd.h>

pid_t fork(void);
```

Вновь созданный (сыновний) процесс отличается только своим идентификатором процесса pid и идентификатором родительского процесса ppid. Кроме того, у нового процесса сбрасываются счётчики использования ресурсов, блокировки файлов и ожидающие сигналы. Сыновний процесс продолжает выполнять тот же код, что и родительский процесс. Отличить новый процесс от родительского можно только по возвращаемому системным вызовом `fork` значению. В сыновний процесс возвращается 0, а в родительский процесс возвращается идентификатор сыновнего процесса. Кроме того, функция `fork` возвращает число -1, когда новый процесс не может быть создан из-за нехватки ресурсов, либо из-за превышения максимального разрешённого числа процессов для пользователя или всей системы. Обычно функция `fork` используется следующим образом:

```
if ((pid = fork())) < 0) { /* сигнализировать об ошибке */ }
else if (!pid) {
  /* этот фрагмент кода выполняется в сыновнем процессе */
  /* ... */
  _exit(0);
} else {
  /* а этот фрагмент выполняется в родительском процессе */
}
```

Функции **getpid**, **getppid** позволяют получить идентификатор текущего или родительского процесса.

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

Функция getpid возвращает идентификатор текущего процесса, а функция getppid возвращает идентификатор родительского процесса. Если родительский процесс завершил своё выполнение раньше сыновнего, «родительские права» передаются процессу с номером 1 — процессу init. Идентификатор процесса — это положительное число в интервале от 1 до (как правило) 32000. При создании нового процесса система назначает ему новый идентификатор. Когда pid достигнет максимального допустимого значения, счёт начнётся снова с 1.

Каждый процесс относится к некоторой группе процессов. В группу могут объединяться процессы, выполняющие с точки зрения программиста одно задание. Например, интерпретатор команд может объединить в одну группу процессов все процессы, связанные конвейером. В простейшем случае группа процессов может состоять из единственного процесса. Группа процессов идентифицируется положительным целым числом, совпадающим с идентификатором одного из процессов в этой группе. Даже после того, как этот процесс завершит работу, идентификатор группы процессов будет действительным до момента, пока не завершит работу последний процесс этой группы процессов.

Группа процессов может выступать как одно целое в системных вызовах отправления сигнала (они будут подробно рассмотрены ниже). Отрицательный идентификатор процесса в них обозначает, что сигнал посылается не одному процессу, а всей группе процессов, идентификатор группы которой совпадает с модулем аргумента. При работе с терминалом выделяются основная и фоновые группы процессов, и нажатие на комбинацию клавиш Ctrl-C посылает сигнал SIGINT сразу всей основной группе процессов. Другие сигналы, генерируемые при работе с терминалом (SIGTSTP, SIGQUIT и др.) так же посылаются всей основной группе процессов.

Идентификатор группы процесса сохраняется (наследуется) при создании нового процесса с помощью fork. Но, в отличие от идентификатора процесса, идентификатор группы процессов можно изменить.

```
#include <sys/types.h>
#include <unistd.h>
int   setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);
```

Функция getpgid возвращает идентификатор группы процессов процесса с заданным идентификатором pid. Если аргумент pid равен 0, возвращается информация для текущего процесса. В случае ошибки возвращается -1.

Функция setpgid помещает процесс pid в группу процессов pgid. И pid, и pgid могут быть равны 0, что означает идентификатор текущего процесса. При необходимости создаётся новая группа процессов. Если pid не совпадает с идентификатором самого процесса, это должен быть идентификатор одного из сыновних процессов, который ещё не выполнил к этому моменту вызов exec. В случае успешного завершения функция возвращает 0. При неудаче функция возвращает -1, а переменная errno устанавливается в код ошибки.

| | |
|---|---|
| EACCES | Сыновний процесс, заданный аргументом pid уже выполнил exec. |
| EINVAL | Недопустимое значение pgid. |
| ENOSYS | Система не поддерживает управление заданиями. |
| EPERM | Процесс, заданный аргументом pid, является лидером сессии, не находится в той же самой сессии, что текущий процесс, значение pgid не соответствует группе процессов в той же самой сессии, что текущий процесс. |

## 1.2 Замещение тела процесса

Чаще всего новый процесс создаётся для того, чтобы запустить на выполнение какую-либо другую программу. В системах POSIX другую программу можно запустить, только заменив области памяти текущего процесса. Для этого служат функции семейства **exec**.

```
#include <unistd.h>

extern char **environ;

int execve(const char *filename, char *const argv[],
           char *const envp[]);

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int  execle(const char *path, const char *arg, ...,
            char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Основной функцией является системный вызов `execve`. Все остальные функции используют системный вызов `execve`.

Системный вызов **execve** запускает программу, заданную аргументом `filename`. Программа должна быть либо двоичной исполняемой программой, либо скриптом, начинающимся со строки вида

```
#! interpreter [arg]
```

В этом случае интерпретатор `interpreter` должен быть правильным путём к исполняемому двоичному файлу. Тогда программа будет запущена на выполнение строкой:

```
interpreter [arg] filename
```

`argv` — это массив строк — аргументов командной строки, передаваемый новой программе. `envp` — массив строк вида `key=value`, который передаётся как окружение в новую программу. И `argv` и `envp` должны завершаться нулевым указателем. Аргументы командной строки и переменные окружения доступны из функции `main` вызванной программы (если это — программа на Си), которая определяется следующим образом:

```
int main(int argc, char *argv[], char *envp[]);
```

Системный вызов `execve` не возвращает управление вызвавшей его программе в случае успеха, поскольку страницы кода, данных и стека процесса замещаются страницами из загружаемой программы. Запускаемая программа наследует идентификатор процесса pid и все открытые файловые дескрипторы, кроме тех, которые явно помечены флагом `FD_CLOEXEC`. Очищаются сигналы, ожидающие доставки. Обработчики всех неигнорируемых и не обрабатываемых по умолчанию сигналов сбрасываются в значения по умолчанию.

Если текущий процесс трассируется с помощью системного вызова `ptrace`, после успешного вызова `execve` ему посылается сигнал `SIGTRAP`.

Если у файла запускаемой программы установлен бит `SUID`, эффективный идентификатор пользователя изменяется на идентификатор владельца данного файла. Аналогично, если

у файла запускаемой программы установлен бит SGID, эффективный идентификатор группы изменяется на идентификатор группы данного файла.

Если исполняемый файл использует динамические библиотеки, система вызывает динамический загрузчик, чтобы он подгрузил в адресное пространство процесса необходимые динамические библиотеки.

В случае ошибки execve возвращает -1, и переменная errno устанавливается в код ошибки. Возможные ошибки приведены в таблице.

| | |
|---|---|
| EACCES | 1) Запускаемый файл или интерпретатор скрипта не является регулярным файлом.<br>2) Нет прав на выполнение запускаемого файла или интерпретатора скрипта.<br>3) Файловая система не допускает выполнение файлов.<br>4) Нет права поиска в каком-либо каталоге, входящем в путь к filename или интерпретатору скрипта. |
| EPERM | Попытка запустить файл с установленным битом SUID или SGID в трассируемом процессе. |
| E2BIG | Список аргументов слишком велик. |
| ENOEXEC | Неверный формат исполняемого файла. |
| EFAULT | Какой-либо из аргументов — недопустимый адрес. |
| ENAMETOOLONG | Аргумент filename слишком длинный. |
| ENOENT | Файл filename или интерпретатор скрипта не существует. |
| ENOMEM | Недостаточно памяти ядра. |
| ENOTDIR | Некоторая компонента пути к файлу filename или интерпретатору скрипта не является каталогом. |
| ELOOP | Слишком много символических ссылок при прослеживании файла filename или интерпретатора скрипта. |
| ETXTBUSY | Файл filename или интерпретатор скрипта открыт на запись другим процессом. |
| EIO | Ошибка ввода/вывода. |
| ENFILE | Достигнуто максимальное количество открытых файлов в системе. |
| EMFILE | Достигнуто максимальное количество открытых файлов для процесса. |

Таблица 1: Коды ошибок системного вызова execve

Функции **execvp**, **execv**, **execle**, **execlp**, **execl** являются оболочками над execve.

Аргумент arg и последующие аргументы в функциях execl, execlp, execle можно рассматривать как arg0, arg1, ..., argn. Все вместе они описывают список указателей на строки, представляющий собой список аргументов для вызываемой программы. По соглашению первый аргумент должен содержать имя запускаемого файла. Список аргументов **должен** завершаться нулевым указателем NULL.

Функции execv и execvp принимают массив указателей на строки, которые будут переданы как аргументы в вызываемую программу. По соглашению первый аргумент должен содержать имя запускаемого файла. Массив аргументов **должен** завершаться нулевым указателем NULL.

Функция execle позволяет задавать окружение для запускаемой программы. Массив указателей на строки, задающие переменные окружения, **должен** завершаться нулевым указателем NULL. Адрес массива должен идти следующим параметром функции execle после нулевого указателя, обозначающего конец списка аргументов, передаваемых вызываемой программе. Все другие функции берут переменные окружения из глобальной переменной

environ текущего процесса. Эта переменная содержит все переменные, переданные текущему процессу, и новые переменные окружения, добавленные с помощью вызова putenv. Таким образом, все другие функции наследуют переменные окружения текущего процесса.

Функции execlp и execvp ищут файл file в пути поиска, если строка file не содержит символ '/'. Путь поиска определяется переменной окружения PATH, а если эта переменная не задана, используется путь по умолчанию :/bin:/usr/bin. Кроме того, эти функции по-особому реагируют на некоторые ошибки.

Если в доступе к файлу отказано (вызов execve вернул EACCES), эти функции продолжают поиск в оставшейся части пути поиска. Но если другого файла не найдено, функции возвращаются с кодом ошибки EACCES).

Если формат файла не был распознан (вызов execve вернул ENOEXEC), эти функции вызовут командный интерпретатор с путём к файлу в качестве первого аргумента. Если вызов закончится неудачей, дальнейший поиск в пути поиска не производится.

Таблица свойств функций семейства exec приведена ниже.

| Имя | передача списка аргументов в параметрах | поиск в пути | задание нового окружения |
|---|---|---|---|
| execve | нет | нет | да |
| execvp | нет | да | нет |
| execv | нет | нет | нет |
| execle | да | нет | да |
| execlp | да | да | нет |
| execl | да | нет | нет |

Таблица 2: Свойства функций семейства exec

## 1.3 Завершение работы процесса

```
#include <stdlib.h>
void exit(int status);
void _exit(int status);
void abort(void);
```

Процесс может завершить свою работу одним из следующих способов:

- Вызовом библиотечной функции exit(status);, где n — код завершения процесса.
- Возвратом из функции main. Этот способ эквивалентен предыдущему. В качестве кода завершения процесса используется значение, возвращаемое из main.
- Вызовом системного вызова _exit(status);, где n — код завершения процесса.
- Получением необрабатываемого, неигнорируемого и неблокируемого сигнала, который вызывает по умолчанию нормальное или аварийное завершение процесса.

Код возврата процесса — это некоторое целое число, обычно в интервале от 0 до 255 (один байт). Оно может использоваться родительским процессом, чтобы определить, завершился ли процесс успешно или нет. Соглашение о взаимодействии процессов предполагает, что код завершения 0 означает успешное завершение процесса, а все прочие коды —

| **Атрибут** | **Наследование при `fork`** | **Наследование при `exec`** |
|---|---|---|
| Страницы кода программы | да, разделяются | нет |
| Страницы данных программы | да, копируются при записи | нет |
| Переменные окружения | да | возможно |
| Аргументы программы | да | возможно |
| Идентификатор пользователя (uid) | да | да |
| Идентификатор группы (gid) | да | да |
| Эффективный идентификатор пользователя (euid) | да | да, если не установлен бит SUID |
| Эффективный идентификатор группы (egid) | да | да, если не установлен бит SGID |
| Идентификатор процесса (pid) | нет | да |
| Идентификатор группы процессов (pgid) | да | да |
| Идентификатор родительского процесса (ppid) | нет | да |
| Приоритет процесса (nice) | да | да |
| Маска прав при создании файлов (umask) | да | да |
| Ограничения процессов (limits) | да | да |
| Счётчики использования ресурсов | нет | да |
| Сигналы, обрабатываемые по умолчанию | да | да |
| Игнорируемые сигналы | да | да |
| Перехватываемые сигналы | да | нет |
| Сигналы, ожидающие доставки | нет | нет |
| Файловые дескрипторы | да | да, если для дескриптора не установлен флаг `FD_CLOEXEC` |
| Блокировки файлов | нет | да |
| Рабочий каталог | да | да |
| Корневой каталог | да | да |

Таблица 3: Наследование атрибутов при вызовах `fork`, `exec`

неуспешное завершение. Процесс может выработать код неуспешного завершения, если, например, он не смог открыть файл, необходимый для работы, или произошла другая ошибка, из-за которой процесс не смог выполнить ожидаемые от него действия. В случае незначительных ошибок, не влияющих значительно на выполнение процессом ожидаемых действий, следует вырабатывать код завершения 0.

Библиотечная функция `exit` отличается от системного вызова `_exit` тем, что функция `exit` выполнит корректное закрытие всех открытых дескрипторов потока и вызовет обработчики завершения работы процесса, зарегистрированные с помощью функции `atexit`. Системный вызов `_exit` вызывает немедленное завершение процесса, при этом содержимое незаписанных буферов файловых дескрипторов теряется. Ни одна из этих функций никогда не возвращает управление в программу.

Когда процесс завершается из-за получения сигнала, никакой код завершения не формируется, но процесс-родитель может узнать, что данный процесс завершился из-за сигнала, и узнать номер сигнала, вызвавшего завершение работы процесса. Обработчики завершения

работы процесса и открытые дескрипторы потока не закрываются. Некоторые сигналы по умолчанию вызывают аварийное завершение процесса. Это — сигналы, генерируемые при некоторых фатальных ошибках работы процесса, например, при доступе к адресам, не отображённым в адресное пространство процесса. При аварийном завершении ядро записывает содержимое адресного пространства процесса и содержимое регистров центрального процессора в файл (обычно называемый `core`) в текущем каталоге. Это файл может потом использоваться для «посмертной отладки».

Функция `abort` вызывает фатальное завершение работы процесса (как будто бы он получил сигнал `SIGABRT`). Эта функция может использоваться, когда сама программа диагностировала состояние, когда она не может продолжить выполнение из-за ошибки в самой программе. `abort` не должна использоваться, когда процесс завершает работу из-за ошибки во входных данных или в параметрах командной строки.

В любом случае все ресурсы, связанные с процессом освобождаются, но запись в таблице процессов не удаляется для того, чтобы процесс-родитель смог прочитать статус завершения процесса. Процесс в таком состоянии, когда все ресурсы уже освобождены, и осталась только запись в таблице процессов, называется «зомби». Если процесс-родитель «не интересуется» судьбой сыновних процессов, они останутся зомби до тех пор, пока процесс-родитель не завершится. Тогда их родителем станет процесс 1 (init). Как только родительский процесс прочитает статус завершения сыновнего процесса-зомби, запись в таблице процессов уничтожается, и процесс окончательно прекращает своё существование.

## 1.4 Ожидание завершения сыновнего процесса

Простейшая функция, с помощью которой можно узнать состояние процесса, — функция **`wait`**.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *pstatus)
```

Функция `wait` приостанавливает выполнение текущего процесса до тех пор, пока какой-либо сыновний процесс не завершит своё выполнение, либо пока в процесс не поступит сигнал, который вызовет обработчик сигнала или завершит выполнение процесса. Если какой-либо сыновний процесс уже завершил выполнение («зомби»), функция возвращается немедленно, и процесс окончательно уничтожается.

Если указатель `pstatus` не равен `NULL`, функция `wait` записывает информацию о статусе завершения по адресу `pstatus`. Для получения информации о статусе завершения процесса могут использоваться следующие макросы. Они принимают в качестве параметра сам статус завершения процесса (типа **`int`**), а не указатель на него.

**`WIFEXITED(status)`** — принимает ненулевое значение, если процесс завершил своё выполнение нормально.

**`WEXITSTATUS(status)`** — этот макрос может быть использован, только если вызов макроса `WIFEXITED` дал ненулевое значение. Макрос выдаёт код возврата процесса, который был задан как аргумент функции `exit` или в операторе `return` функции `main`.

**`WIFSIGNALED(status)`** — принимает ненулевое значение, если процесс завершил своё выполнение в результате получения необрабатываемого сигнала, который по умолчанию вызывает завершение работы процесса.

**WTERMSIG(status)** — этот макрос может быть использован, только если вызов макроса WIFSIGNALED дал ненулевое значение. Макрос выдаёт номер сигнала, который вызвал завершение работы процесса.

Функция возвращает идентификатор завершившегося процесса, либо -1 в случае ошибки. В этом случае переменная errno устанавливается в код ошибки. Возможные коды ошибки приведены в таблице.

Больше возможностей предоставляет функция **wait4**, определённая следующим образом:

```
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait4(pid_t pid, int *pstatus, int options,
            struct rusage *prusage)
```

Функция wait4 приостанавливает выполнение процесса до тех пор, пока сыновний процесс с заданным идентификатором pid не завершит выполнение, либо пока процесс не получит сигнал, который вызовет обработчик сигнала или завершение работы процесса. Если процесс с заданным идентификатором pid уже завершил работу к моменту вызова функции wait4 (процесс-зомби), функция завершает работу немедленно.

Аргумент pid может принимать следующие значения:

| | |
|---|---|
| < -1 | ожидать завершения любого сыновнего процесса, идентификатор группы процессов которого совпадает с абсолютным значением pid |
| -1 | ждать любой сыновний процесс (как функция wait) |
| 0 | ждать любой сыновний процесс, идентификатор группы процессов которого совпадает с идентификатором группы процессов текущего процесса |
| > 0 | ждать сыновний процесс с заданным идентификатором процесса pid |

Значение options образуется как побитовое «или» одной или нескольких следующих констант, либо может быть равен 0.

**WNOHANG** — завершить работу немедленно, если нет сыновних процессов, завершивших выполнение.

**WUNTRACED** — возвращать информацию и о процессах, которые были приостановлены, и статус которых ещё не был сообщён.

Если pstatus не равен NULL, информация о статусе завершения процесса записывается в область памяти, на которую указывает pstatus.

Помимо макросов, позволяющих получить статус завершения процесса, описанных выше, могут использоваться следующие макросы.

**WIFSTOPPED(status)** — возвращает ненулевое значение, если процесс, который вызвал завершение работы функции wait4 приостановлен. Это возможно, только если wait4 была вызвана с установленным флагом WUNTRACED.

**WSTOPSIG(status)** — возвращает номер сигнала, который вызвал приостановку выполнения процесса. Этот макрос может использоваться, только если WIFSTOPPED дал ненулевое значение.

Если prusage не равен NULL, информация об использовании процессом ресурсов системы записывается по адресу, указанному аргументом prusage.

Структура struct rusage определена в операционной системе Linux следующим образом:

```
struct rusage
```

```
{
        struct timeval ru_utime; /* время в режиме пользователя */
        struct timeval ru_stime; /* время в режиме ядра */
        long ru_maxrss;          /* максимальный размер в памяти */
        long ru_ixrss;      /* размер всех разделяемых страниц */
        long ru_idrss;      /* размер всех неразделяемых страниц */
        long ru_isrss;      /* размер страниц стека */
        long ru_minflt;   /* сбоев страницы без подкачки */
        long ru_majflt;   /* сбоев страницы с подкачкой */
        long ru_nswap;    /* количество полных откачек */
        long ru_inblock;        /* блочных операций ввода */
        long ru_oublock;        /* блочных операций вывода */
        long ru_msgsnd;         /* послано сообщений */
        long ru_msgrcv;         /* получено сообщений */
        long ru_nsignals;       /* получено сигналов */
        long ru_nvcsw;    /* "добровольных" переключений контекста */
        long ru_nivcsw;   /* "недобровольных" переключений */
};
```

Функция `wait4` возвращает идентификатор процесса, который вызвал завершение работы функции, либо -1 при ошибке (например, не существует сыновних процессов заданной категории), либо 0, если был использован флаг `WNOHANG`, и нет процессов, завершивших выполнение. В последних двух случаях переменная `errno` будет содержать код ошибки.

Возможные коды ошибок приведены в таблице.

| | |
|---|---|
| `ECHILD` | процесс, заданный аргументом `pid`, не существует или не является сыновним процессом, либо у процесса вообще нет сыновних процессов |
| `ERESTARTSYS` `EINTR` | флаг `WNOHANG` не был установлен, и процесс получил неблокируемый сигнал или `SIGCHLD`. |
| `EINVAL` | какой-либо из аргументов имеет недопустимое значение. |

Таблица 4: Коды ошибок для функций `wait`, `wait4`

## 1.5 Пример программы

Напишем реализацию функции `system` (стандартная функция, которая выполняет заданную команду).

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int system(char const *cmd)
{
  int pid, status;

  if ((pid = fork()) < 0) {
    /* ошибка */
    perror("fork");
    return -1;
```

```
  } else if (!pid) {
    /* child */
    execl("/bin/sh", "/bin/sh", "-c", cmd, NULL);
    /* ошибка */
    perror("execl");
    _exit(1);
  }

  /* parent */
  wait(&status);
  if (WIFSIGNALED(status)) return WTERMSIG(status) + 256;
  return WEXITSTATUS(status);
}
```

# 2 Перенаправление стандартных потоков

В начале работы процесса у него, как правило, уже открыты три файловых дескриптора с номерами 0, 1, 2 — стандартный ввод, стандартный вывод и стандартный поток ошибок. Как правило, эти потоки связаны с терминалом, с которым работает командный интерпретатор, вызвавший процесс. Однако можно связать со стандартными потоками произвольные объекты, работа с которыми ведётся с помощью файловых дескрипторов (каналы, сокеты и пр.).

Для копирования открытого файлового дескриптора используется функция **dup2**, определённая следующим образом:

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

Функция `dup2` создаёт копию файлового дескриптора `oldfd`. После этого использование двух файловых дескрипторов полностью эквивалентно. Оба файловых дескриптора разделяют блокировки файлов, указатель текущего положения в файле и флаги открытия файлов. Например, если текущая позиция в файле была изменена с помощью `lseek` у одного из дескрипторов, текущая позиция изменится и у другого дескриптора.

Флаг «закрытия при exec» не разделятся.

Функция `dup2` создаёт копию `oldfd` в дескрипторе с номером `newfd`, при необходимости предварительно закрывая `newfd`.

Функция возвращает номер нового файлового дескриптора или -1, если функция не смогла создать новый файловый дескриптор. Переменная `errno` в этом случае содержит код ошибки.

## 2.1 Пример программы

Напишем программу, которая печатает все процессы, идентификатор пользователя которых 0 (root). Для получения списка процессов будем использовать вызов программы `ps`. Предположим, что второе число в строке, печатаемой командой `ps` как раз содержит идентификатор пользователя.

```
#include <stdlib.h>
```

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

void pexit(char const *str)
{
  perror(str);
  exit(1);
}

int main(void)
{
  char tmppatt[] = "/tmp/XXXXXX";
  int  fd, pid, status;
  FILE *f;
  char  buf[1024];
  int   v1, v2;

  /* создаём временный файл */
  if ((fd = mkstemp(tmppatt)) < 0) pexit("mkstemp");
  /* сразу удаляем */
  if (unlink(tmppatt) < 0) pexit("unlink");
  /* создаём новый процесс */
  if ((pid = fork()) < 0) pexit("fork");

  if (!pid) { /* сын */
    /* перенаправляем стандартный вывод */
    if (dup2(fd, 1) != 1) { perror("dup2"); _exit(1); }
    /* закрываем старый дескриптор */
    close(fd);
    /* вызываем другой процесс */
    execlp("ps", "ps", "axl", 0);
    perror("execlp");
    _exit(1);
  }

  /* отец */
  /* ждём */
  wait(&status);
  /* ошибка выполнения сыновнего процесса */
  if (!WIFEXITED(status) || WEXITSTATUS(status) > 0) exit(1);
  /* перематываем файл на начало */
  if (lseek(fd, 0, SEEK_SET) < 0) pexit("lseek");
  /* связываем уже открытый файловый дескриптор с FILE * */
  if (!(f = fdopen(fd, "r"))) pexit("fdopen");
  while (fgets(buf, sizeof(buf), f)) {
    if (strlen(buf) >= sizeof(buf) - 1) {
      fprintf(stderr, "string␣too␣long\n");
      exit(1);
```

```
    }
    /* печатаем все процессы, владелец которых - root */
    if (sscanf(buf, "%d %d", &v1, &v2) == 2 && !v2)
      printf("%s", buf);
  }
  fclose(f);
  return 0;
}
```