13

Процессы-демоны

13.1. Введение

Демоны – это долгоживущие процессы. Зачастую они запускаются во время загрузки системы и завершают работу вместе с ней. Так как они не имеют управляющего терминала, говорят, что они работают в фоновом режиме. В системе UNIX демоны решают множество повседневных задач.

В этой главе мы рассмотрим структуру процессов-демонов и покажем, как они создаются. Так как демоны не имеют управляющего терминала, нам необходимо будет выяснить, как демон может вывести сообщение об ошибке, если что-то идет не так, как надо.

> Обсуждение истории термина *демон* применительно к компьютерным системам вы найдете в [Raymond 1996].

13.2. Характеристики демонов

Давайте рассмотрим некоторые наиболее распространенные системные демоны и их отношения с группами процессов, управляющими терминалами и сессиями. Команда ps(1) выводит информацию о процессах в системе. Эта команда имеет множество опций, дополнительную информацию о них вы найдете в справочном руководстве. Мы запустим команду

```
ps -axj
```

под управлением BSD-системы и будем использовать полученную от нее информацию при дальнейшем обсуждении. Ключ `-a` используется для вывода процессов, которыми владеют другие пользователи, ключ `-x` – для вывода процессов, не имеющих управляющего терминала, и ключ `-j` – для вывода дополнительных сведений, имеющих отношение к заданиям: идентификатора сессии, идентификатора группы процессов, управляющего терминала и идентификатора группы процессов терминала.

Для систем, основанных на System V, аналогичная команда выглядит как ps -efjc. (В целях безопасности некоторые версии UNIX не допускают просмотр процессов, принадлежащих другим пользователям, с помощью команды ps.) Вывод команды ps выглядит примерно следующим образом:

```
 PPID   PID  PGID   SID TTY TPGID UID   COMMAND
    0     1     0     0 ?      -1   0   init
    1     2     1     1 ?      -1   0   [keventd]
    1     3     1     1 ?      -1   0   [kapmd]
    0     5     1     1 ?      -1   0   [kswapd]
    0     6     1     1 ?      -1   0   [bdflush]
    0     7     1     1 ?      -1   0   [kupdated]
    1  1009  1009  1009 ?      -1  32   portmap
    1  1048  1048  1048 ?      -1   0   syslogd -m 0
    1  1335  1335  1335 ?      -1   0   xinetd -pidfile /var/run/xinetd.pid
    1  1403     1     1 ?      -1   0   [nfsd]
    1  1405     1     1 ?      -1   0   [lockd]
 1405  1406     1     1 ?      -1   0   [rpciod]
    1  1853  1853  1853 ?      -1   0   crond
    1  2182  2182  2182 ?      -1   0   /usr/sbin/cupsd
```

Из данного примера мы убрали несколько колонок, которые не представляют для нас особого интереса. Здесь показаны следующие колонки, слева направо: идентификатор родительского процесса, идентификатор процесса, идентификатор группы процессов, идентификатор сессии, имя терминала, идентификатор группы процессов терминала (группы процессов переднего плана, связанной с управляющим терминалом), идентификатор пользователя и строка команды.

> Система, на которой была запущена эта команда (Linux), поддерживает понятие идентификатора сессии, который мы упоминали при обсуждении функции setsid в разделе 9.5. Идентификатор сессии – это просто идентификатор процесса лидера сессии. Однако в системах, основанных на BSD, будет выведен адрес структуры session, соответствующей группе процессов, которой принадлежит данный процесс (раздел 9.11).

Перечень системных процессов, который вы увидите, во многом зависит от реализации операционной системы. Обычно это будут процессы с идентификатором родительского процесса 0, запускаемые ядром в процессе загрузки системы. (Исключением является процесс init, так как это команда уровня пользователя, которая запускается ядром во время загрузки.) Процессы ядра – это особые процессы, они существуют все время, пока работает система. Эти процессы обладают привилегиями суперпользователя и не имеют ни управляющего терминала, ни строки команды.

Процесс с идентификатором 1 – это обычно процесс init, о чем уже говорилось в разделе 8.2. Это системный демон, который, кроме всего прочего, отвечает за запуск различных системных служб на различных уровнях загрузки. Как правило, эти службы также реализованы в виде демонов.

В ОС Linux демон keventd предоставляет контекст процесса для запуска задач из очереди планировщика. Демон kapmd обеспечивает поддержку расши-

ренного управления питанием, которое доступно в некоторых компьютерных системах. Демон kswapd известен также как демон выгрузки страниц. Этот демон поддерживает подсистему виртуальной памяти, в фоновом режиме записывая на диск страницы, измененные со времени их чтения с диска (dirty pages), благодаря чему они могут быть использованы снова.

Сбрасывание кэшированных данных на диск в ОС Linux производится с помощью двух дополнительных демонов – bdflush и kupdated. Демон bdflush начинает сбрасывать измененные буферы из кэша на диск, когда объем свободной памяти уменьшается до определенного уровня. Демон kupdated сбрасывает измененные буферы из кэша на диск через регулярные интервалы времени, снижая тем самым риск потери данных в случае краха системы.

Демон portmap осуществляет преобразование номеров программ RPC (Remote Procedure Call – удаленный вызов процедур) в номера сетевых портов. Демон syslogd может использоваться программами для вывода системных сообщений в журнал для просмотра оператором. Сообщения могут выводиться на консоль, а также записываться в файл. (Более подробно syslogd рассматривается в разделе 13.4.)

В разделе 9.3 мы уже говорили о демоне inetd (xinetd). Этот демон ожидает поступления из сети запросов к различным сетевым серверам. Демоны nfsd, lockd и rpciod обеспечивают поддержку сетевой файловой системы (NFS – Network File System).

Демон cron (crond) производит запуск команд в определенное время. Множество административных задач выполняется благодаря регулярному запуску программ с помощью демона cron. Демон cupsd – это сервер печати, он обслуживает запросы к принтеру.

Обратите внимание: большинство демонов обладают привилегиями суперпользователя (имеют идентификатор пользователя 0). Ни один из демонов не имеет управляющего терминала – вместо имени терминала стоит знак вопроса, а идентификатор группы переднего плана равен –1. Демоны ядра запускаются без управляющего терминала. Отсутствие управляющего терминала у демонов пользовательского уровня – вероятно, результат вызова функции setsid. Все демоны пользовательского уровня являются лидерами групп и лидерами сессий, и они являются единственными процессами в своих группах процессов и сессиях. И, наконец, обратите внимание на то, что родительским для большинства демонов является процесс init.

13.3. Правила программирования демонов

При программировании демонов во избежание нежелательных взаимодействий следует придерживаться определенных правил. Сначала мы перечислим эти правила, а затем продемонстрируем функцию daemonize, которая их реализует.

1. Прежде всего нужно вызвать функцию umask, чтобы сбросить маску режима создания файлов в значение 0. Маска режима создания файлов, насле-

дуемая от запускающего процесса, может маскировать некоторые биты прав доступа. Если предполагается, что процесс-демон будет создавать файлы, может потребоваться установка определенных битов прав доступа. Например, если демон создает файлы с правом на чтение и на запись для группы, маска режима создания файла, которая выключает любой из этих битов, воспрепятствовала бы этому.

2. Вызвать функцию fork и завершить родительский процесс. Для чего это делается? Во-первых, если демон был запущен как обычная команда оболочки, то завершив родительский процесс, мы заставим командную оболочку думать, что команда была выполнена. Во-вторых, дочерний процесс наследует идентификатор группы процессов от родителя, но получает свой идентификатор процесса; таким образом мы гарантируем, что дочерний процесс не будет являться лидером группы, а это необходимое условие для вызова функции setsid, который будет произведен далее.

3. Создать новую сессию, обратившись к функции setsid. При этом (вспомните раздел 9.5) процесс становится (а) лидером новой сессии, (б) лидером новой группы процессов и (в) лишается управляющего терминала.

   Для систем, основанных на System V, некоторые специалисты рекомендуют в этой точке повторно вызвать функцию fork и завершить родительский процесс, чтобы второй потомок продолжал работу в качестве демона. Такой прием гарантирует, что демон не будет являться лидером сессии, и это препятствует получению управляющего терминала в System V (раздел 9.6). Как вариант, чтобы избежать обретения управляющего терминала, при любом открытии терминального устройства следует указывать флаг O_NOCTTY.

4. Сделать корневой каталог текущим рабочим каталогом. Текущий рабочий каталог, унаследованный от родительского процесса, может находиться на смонтированной файловой системе. Поскольку демон, как правило, существует все время, пока система не будет перезагружена, то в подобной ситуации, когда рабочий каталог демона находится в смонтированной файловой системе, ее невозможно будет отмонтировать.

   Как вариант, некоторые демоны могут устанавливать собственный текущий рабочий каталог, в котором они производят все необходимые действия. Например, демоны печати в качестве текущего рабочего каталога часто выбирают буферный каталог, куда помещаются задания для печати.

5. Закрыть все ненужные файловые дескрипторы. Это предотвращает удержание в открытом состоянии некоторых дескрипторов, унаследованных от родительского процесса (командной оболочки или другого процесса). С помощью нашей функции open_max (листинг 2.4) или с помощью функции getrlimit (раздел 7.11) можно определить максимально возможный номер дескриптора и закрыть все дескрипторы вплоть до этого номера.

6. Некоторые демоны открывают файловые дескрипторы с номерами 0, 1 и 2 на устройстве /dev/null – таким образом, любые библиотечные функции, которые пытаются читать со стандартного устройства ввода или писать на стандартное устройство вывода или сообщений об ошибках, не будут ока-

зывать никакого влияния. Поскольку демон не связан ни с одним терминальным устройством, он не сможет взаимодействовать с пользователем в интерактивном режиме. Даже если демон изначально был запущен в рамках интерактивной сессии, он все равно переходит в фоновый режим, и начальная сессия может завершиться без воздействия на процесс-демон. С этого же терминала в систему могут входить другие пользователи, и демон не должен выводить какую-либо информацию на терминал, да и пользователи не ждут того, что их ввод с терминала будет прочитан демоном.

Пример

В листинге 13.1 приводится функция, которую может вызывать приложение, желающее стать демоном.

Листинг 13.1. Инициализация процесса-демона

```
#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>

void
daemonize(const char *cmd)
{
    int i, fd0, fd1, fd2;
    pid_t pid;
    struct rlimit rl;
    struct sigaction sa;

    /*
     * Сбросить маску режима создания файла.
     */
    umask(0);

    /*
     * Получить максимально возможный номер дескриптора файла.
     */
    if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
        err_quit("%s: невозможно получить максимальный номер дескриптора ",
                 cmd);

    /*
     * Стать лидером новой сессии, чтобы утратить управляющий терминал.
     */
    if ((pid = fork()) < 0)
        err_quit("%s: ошибка вызова функции fork", cmd);
    else if (pid != 0) /* родительский процесс */
        exit(0);
    setsid();

    /*
     * Обеспечить невозможность обретения управляющего терминала в будущем.
     */
```

```
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: невозможно игнорировать сигнал SIGHUP");
    if ((pid = fork()) < 0)
        err_quit("%s: ошибка вызова функции fork", cmd);
    else if (pid != 0) /* родительский процесс */
        exit(0);

    /*
     * Назначить корневой каталог текущим рабочим каталогом,
     * чтобы впоследствии можно было отмонтировать файловую систему.
     */
    if (chdir("/") < 0)
        err_quit("%s: невозможно сделать текущим рабочим каталогом /");

    /*
     * Закрыть все открытые файловые дескрипторы.
     */
    if (rl.rlim_max == RLIM_INFINITY)
        rl.rlim_max = 1024;
    for (i = 0; i < rl.rlim_max; i++)
        close(i);

    /*
     * Присоединить файловые дескрипторы 0, 1 и 2 к /dev/null.
     */
    fd0 = open("/dev/null", O_RDWR);
    fd1 = dup(0);
    fd2 = dup(0);

    /*
     * Инициализировать файл журнала.
     */
    openlog(cmd, LOG_CONS, LOG_DAEMON);
    if (fd0 !=·0 || fd1 != 1 || fd2 != 2) {
        syslog(LOG_ERR, "ошибочные файловые дескрипторы %d %d %d",
               fd0, fd1, fd2);
        exit(1);
    }
}
```

Если функция daemonize **будет вызвана из программы, которая затем приостанавливает работу, мы сможем проверить состояние демона с помощью команды** ps**:**

```
$ ./a.out
$ ps -axj
 PPID    PID  PGID   SID TTY TPGID UID   COMMAND
    1   3346  3345  3345 ?      -1 501   ./a.out
$ ps -axj | grep 3345
    1   3346  3345  3345 ?      -1 501   ./a.out
```

С помощью команды ps можно убедиться в том, что в системе нет активного процесса с идентификатором 3345. Это означает, что наш демон относится к осиротевшей группе процессов (раздел 9.10) и не является лидером сессии, поэтому он не имеет возможности обрести управляющий терминал. Это результат второго вызова функции fork в функции daemonize. Как видите, наш демон был инициализирован вполне корректно.

13.4. Журналирование ошибок

Одна из проблем, присущих демонам, связана с обслуживанием сообщений об ошибках. Демон не может просто выводить сообщения на стандартное устройство вывода сообщений об ошибках, поскольку он не имеет управляющего терминала. Мы не можем требовать от демона, чтобы он выводил сообщения на консоль, поскольку на большинстве рабочих станций в консоли запускается многооконная система. Мы также не можем требовать, чтобы демон хранил свои сообщения в отдельном файле. Это стало бы источником постоянной головной боли для системного администратора, который будет вынужден запоминать, какой демон в какой файл пишет свои сообщения. Необходим некий централизованный механизм регистрации сообщений об ошибках.

> Механизм syslog для BSD-систем был разработан в Беркли и получил широкое распространение, начиная с 4.2BSD. Большинство систем, производных от BSD, поддерживают syslog.
>
> До появления SVR4 ОС System V не имела централизованного механизма регистрации сообщений об ошибках.
>
> Функция syslog была включена в стандарт Single UNIX Specification как расширение XSI.

Механизм syslog для BSD-систем широко используется, начиная с 4.2BSD. Большинство демонов используют именно этот механизм. На рис. 13.1 показана его структура.

Существует три способа регистрации сообщений:

1. Процедуры ядра могут обращаться к функции log. Эти сообщения могут быть прочитаны любым пользовательским процессом, который может открыть и прочитать устройство /dev/klog. Мы не будем рассматривать эту функцию, поскольку написание процедур ядра не представляет для нас интереса.
2. Большинство пользовательских процессов (демонов) для регистрации сообщений вызывают функцию syslog(3). Порядок работы с ней мы рассмотрим позже. Эта функция отправляет сообщения через сокет домена UNIX – /dev/log.
3. Пользовательский процесс, который выполняется на данном или каком-либо другом компьютере, соединенном с данным компьютером сетью TCP/IP, может отправлять сообщения по протоколу UDP на порт 514. Обратите внимание, что функция syslog никогда не генерирует дейтаграммы UDP – данная функциональность требует, чтобы сама программа поддерживала сетевые взаимодействия.

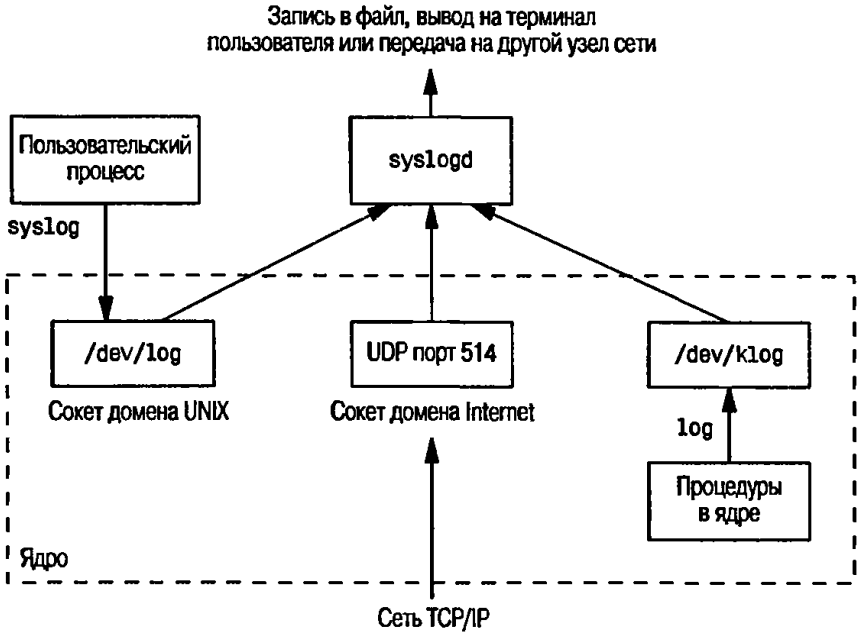

Рис. 13.1. Механизм syslog для BSD-систем

За дополнительной информацией о сокетах домена UNIX обращайтесь к [Stevens, Fenner, and Rudoff 2004].

Обычно демон `syslogd` понимает все три способа регистрации сообщений. На запуске этот демон считывает конфигурационный файл (как правило, это `/etc/syslog.conf`), в котором определяется, куда должны передаваться различные классы сообщений. Например, срочные сообщения могут выводиться на консоль системного администратора (если он находится в системе), тогда как сообщения класса предупреждений могут записываться в файл.

В нашем случае взаимодействие с этим механизмом осуществляется посредством функции `syslog`.

```
#include <syslog.h>

void openlog(const char *ident, int option, int facility);

void syslog(int priority, const char *format, ...);

void closelog(void);

int setlogmask(int maskpri);
```

Возвращает предыдущее значение маски приоритета журналируемых сообщений

Можно и не вызывать функцию `openlog`. Если перед первым обращением к функции `syslog` функция `openlog` не вызывалась, то она будет вызвана автоматически. Обращаться к функции `closelog` также необязательно – она просто закрывает файловый дескриптор, который использовался для взаимодействия с демоном `syslogd`.

Функция openlog позволяет определить в аргументе *ident* строку идентификации, которая обычно содержит имя программы (например, cron или inetd). Аргумент *option* представляет собой битовую маску, которая определяет различные способы вывода сообщений. В табл. 13.1 приводятся значения, которые могут быть включены в маску. В столбце XSI отмечены те из них, которые стандарт Single UNIX Specification включает в определение функции openlog.

Таблица 13.1. Возможные значения, которые могут быть включены в аргумент option функции openlog

option	XSI	Описание
LOG_CONS	•	Если сообщение не может быть передано через сокет домена UNIX, оно будет выведено на консоль.
LOG_NDELAY	•	Сразу открыть сокет домена UNIX для взаимодействия с демоном syslogd, не дожидаясь, пока будет отправлено первое сообщение. Как правило, сокет открывается только тогда, когда отправлено первое сообщение.
LOG_NOWAIT	•	Не ждать завершения дочерних процессов, которые могли быть созданы в процессе регистрации сообщения. Это предотвращает возникновение конфликтов для тех приложений, которые перехватывают сигнал SIGCHLD, так как приложение уже могло получить код завершения дочернего процесса к моменту, когда syslog вызвал функцию wait.
LOG_ODELAY	•	Отложить установление соединения с демоном syslogd до появления первого сообщения.
LOG_PERROR		Вывести сообщение на стандартное устройство вывода сообщений об ошибках и дополнительно передать его демону syslogd (эта опция недоступна в ОС Solaris).
LOG_PID	•	Записывать идентификатор процесса вместе с текстом сообщения. Эта опция предназначена для демонов, которые порождают дочерние процессы для обработки различных запросов (в противоположность демонам, таким как syslogd, которые никогда не вызывают функцию fork).

Возможные значения аргумента *facility* приводятся в табл. 13.2. Обратите внимание, что стандарт Single UNIX Specification определяет лишь часть значений, обычно доступных на конкретной системе. Аргумент *facility* позволяет определить, как должны обрабатываться сообщения из разных источников. Если программа не вызывает функцию openlog или передает ей в аргументе *facility* значение 0, то указать источник сообщения можно с помощью функции syslog, определив его как часть аргумента *priority*.

Функция syslog вызывается для передачи сообщения. Аргумент *priority* представляет собой комбинацию значения для аргумента *facility* (табл. 13.2) и уровня важности сообщения (табл. 13.3). Уровни важности приведены в табл. 13.3 в порядке убывания, от высшего к низшему.

Таблица 13.2. Возможные значения аргумента facility функции openlog

facility	XSI	Описание
LOG_AUTH		Программы авторизации: login, su, getty, ...
LOG_AUTHPRIV		То же самое, что и LOG_AUTH, но журналирование идет в файл с ограниченными правами доступа
LOG_CRON		cron и at
LOG_DAEMON		Системные демоны: inetd, routed, ...
LOG_FTP		Демон FTP (ftpd)
LOG_KERN		Сообщения, сгенерированные ядром
LOG_LOCAL0	•	Определяется пользователем
LOG_LOCAL1	•	Определяется пользователем
LOG_LOCAL2	•	Определяется пользователем
LOG_LOCAL3	•	Определяется пользователем
LOG_LOCAL4	•	Определяется пользователем
LOG_LOCAL5	•	Определяется пользователем
LOG_LOCAL6	•	Определяется пользователем
LOG_LOCAL7	•	Определяется пользователем
LOG_LPR		Система печати: lpd, lpc, ...
LOG_MAIL		Система электронной почты
LOG_NEWS		Система новостей Usenet
LOG_SYSLOG		Демон syslogd
LOG_USER	•	Сообщения от других пользовательских процессов (по умолчанию)
LOG_UUCP		Система UUCP

Таблица 13.3. Уровни важности сообщений (в порядке убывания)

Уровень	Описание
LOG_EMERG	Аварийная ситуация (система остановлена) (наивысший приоритет)
LOG_ALERT	Требуется немедленное вмешательство
LOG_CRIT	Критическая ситуация (например, ошибка жесткого диска)
LOG_ERR	Ошибка
LOG_WARNING	Предупреждение
LOG_NOTICE	Обычная ситуация, которая не является ошибочной, но, возможно, требует специальных действий
LOG_INFO	Информационное сообщение
LOG_DEBUG	Отладочное сообщение (низший приоритет)

Аргумент *format* и все последующие аргументы передаются функции vsprintf для создания строки сообщения. Символы %m в строке формата заменяются сообщением об ошибке (strerror), которое соответствует значению переменной errno.

Функция setlogmask может использоваться для установки маски приоритетов сообщений процесса. Эта функция возвращает предыдущее значение маски. Если маска приоритетов установлена, сообщения, уровень приоритета которых не содержится в маске, не будут журналироваться. Обратите внимание: из вышесказанного следует, что если маска имеет значение 0, то журналироваться будут все сообщения.

Во многих системах имеется программа logger(1), которая может передавать сообщения механизму syslog. Некоторые реализации позволяют передавать программе необязательные аргументы, в которых указывается источник сообщения (*facility*), уровень важности и строка идентификации (*ident*), хотя стандарт System UNIX Specification не определяет дополнительные аргументы. Команда logger предназначена для использования в сценариях на языке командной оболочки, которые исполняются в неинтерактивном режиме и нуждаются в механизме журналирования сообщений.

Пример

В (гипотетическом) демоне печати вы можете встретить следующие строки:

```
openlog("lpd", LOG_PID, LOG_LPR);
syslog(LOG_ERR, "open error for %s: %m", filename);
```

Обращение к функции openlog устанавливает строку идентификации, в которую записывается имя программы, указывает, что идентификатор процесса обязательно должен добавляться к сообщению, и оговаривает, что источником сообщений будет демон системы печати. В вызове функции syslog указан уровень важности сообщения и само сообщение. Если опустить вызов функции openlog, то вызов функции syslog мог бы выглядеть так:

```
syslog(LOG_ERR | LOG_LPR, "open error for %s: %m", filename);
```

Здесь в аргументе *priority* мы скомбинировали указание на источник сообщения и уровень важности сообщения.

Кроме функции syslog, большинство платформ предоставляют ее разновидность, которая принимает дополнительные аргументы в виде списка переменной длины.

```
#include <syslog.h>
#include <stdarg.h>

void vsyslog(int priority, const char *format, va_list arg);
```

Все четыре платформы, обсуждаемые в данной книге, предоставляют функцию vsyslog, но она не входит в состав стандарта Single UNIX Specification.

Большинство реализаций syslogd для сокращения времени обработки запросов от приложений помещают поступившие сообщения в очередь. Если в это время демону поступит два одинаковых сообщения, то в журнал будет записано только одно. Но в конец такого сообщения демоном будет добавлена строка примерно такого содержания: «last message repeated *N* times» (последнее сообщение было повторено *N* раз).

13.5. Демоны в единственном экземпляре

Некоторые демоны реализованы таким образом, что допускают одновременную работу только одной своей копии. Причиной такого поведения может служить, например, требование монопольного владения каким-либо ресурсом. Так, если бы демон cron допускал одновременную работу нескольких своих копий, то каждая из них пыталась бы по достижении запланированного времени запустить одну и ту же операцию, что наверняка привело бы к ошибке.

Если демон требует наличия доступа к устройству, то некоторые действия по предотвращению открытия устройства несколькими программами может выполнить драйвер устройства. Это ограничит количество одновременно работающих экземпляров демона до одного. Однако, если не предполагается обращения демона к подобным устройствам, то нам самим придется выполнить всю необходимую работу по наложению ограничений.

Одним из основных механизмов, обеспечивающих ограничение количества одновременно работающих копий демона, являются блокировки файлов и записей. (Блокировки файлов и записей в файлах мы рассмотрим в разделе 14.3.) Если каждый из демонов создаст файл и попытается установить для этого файла блокировку для записи, то система разрешит установить только одну такую блокировку. Все последующие попытки установить блокировку для записи будут терпеть неудачу, сообщая тем самым остальным копиям демона о том, что демон уже запущен.

Блокировки файлов и записей представляют собой удобный механизм взаимного исключения. Если демон установит для целого файла блокировку для записи, она будет автоматически снята по завершении демона. Это упрощает процедуру восстановления после ошибок, поскольку снимает необходимость удаления блокировки, оставшейся от предыдущей копии демона.

Пример

Функция, представленная в листинге 13.2, демонстрирует использование блокировок файлов и записей для того, чтобы обеспечить запуск единственного экземпляра демона.

Листинг 13.2. Функция, которая гарантирует запуск только одной копии демона

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

```
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

extern int lockfile(int);

int
already_running(void)
{
    int fd;
    char buf[16];

    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0) {
        syslog(LOG_ERR, "не возможно открыть %s: %s",
               LOCKFILE, strerror(errno));
        exit(1);
    }
    if (lockfile(fd) < 0) {
        if (errno == EACCES || errno == EAGAIN) {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "невозможно установить блокировку на %s: %s",
               LOCKFILE, strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf)+1);
    return(0);
}
```

Каждая копия демона будет пытаться создать файл и записать в него свой идентификатор процесса. Это поможет системному администратору идентифицировать процесс. Если файл уже заблокирован, функция `lockfile` завершается неудачей с кодом ошибки `EACCESS` или `EAGAIN` в переменной `errno`, и в вызывающую программу возвращается значение 1, которое указывает, что демон уже запущен. В противном случае функция усекает размер файла до нуля, записывает в него идентификатор процесса и возвращает значение 0.

Усечение размера файла необходимо по той причине, что идентификатор процесса предыдущей копии демона, представленный в виде строки, мог иметь большую длину. Предположим, например, что ранее запускавшаяся копия демона имела идентификатор процесса 12345, а текущая копия имеет идентификатор процесса 9999. Таким образом, когда этот демон запишет

в файл свой идентификатор, то в файле окажется строка 99995. Операция усечения файла удаляет информацию, которая относится к предыдущей копии демона.

13.6. Соглашения для демонов

В системе UNIX демоны придерживаются следующих соглашений.

- Если демон использует файл блокировки, то этот файл помещается в каталог /var/run. Однако, чтобы создать файл в этом каталоге, демон должен обладать привилегиями суперпользователя. Имя файла обычно имеет вид *name*.pid, где *name* – имя демона или службы. Например, демон cron создает файл блокировки с именем /var/run/crond.pid.
- Если демон поддерживает определение дополнительных настроек, то они обычно сохраняются в каталоге /etc. Имя конфигурационного файла, как правило, имеет вид *name*.conf, где *name* – имя демона или службы. Например, конфигурационный файл демона syslogd называется /etc/syslog.conf.
- Демоны могут запускаться из командной строки, но все-таки чаще всего запуск демонов производится из сценариев инициализации системы (/etc/rc* или /etc/init.d/*). Если после завершения демон должен автоматически перезапускаться, мы можем указать на это процессу init, добавив запись respawn в файл /etc/inittab.
- Если демон имеет конфигурационный файл, то настройки из него считываются демоном во время запуска, и затем он обычно не обращается к этому файлу. Если в конфигурационный файл были внесены изменения, то демон пришлось бы останавливать и перезапускать снова, чтобы новые настройки вступили в силу. Во избежание этого некоторые демоны устанавливают обработчики сигнала SIGHUP, в которых производится считывание конфигурационного файла и перенастройка демона. Поскольку демоны не имеют управляющего терминала и являются либо лидерами сессий без управляющего терминала, либо членами осиротевших групп процессов, у них нет причин ожидать сигнала SIGHUP. Таким образом, он может использоваться для других целей.

Пример

Программа, представленная листингом 13.3, демонстрирует один из способов заставить демон перечитать файл конфигурации. Программа использует функцию sigwait и отдельный поток для обработки сигналов, как описано в разделе 12.8.

Листинг 13.3. Пример демона, который перечитывает конфигурационный файл по сигналу

```
#include "apue.h"
#include <pthread.h>
#include <syslog.h>
```

```
sigset_t mask;

extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0) {
            syslog(LOG_ERR, "ошибка вызова функции sigwait");
            exit(1);
        }
        switch (signo) {
        case SIGHUP:
            syslog(LOG_INFO, "Чтение конфигурационного файла");
            reread();
            break;
        case SIGTERM:
            syslog(LOG_INFO, "получен сигнал SIGTERM; выход");
            exit(0);
        default:
            syslog(LOG_INFO, "получен непредвиденный сигнал %d\n", signo);
        }
    }
    return(0);
}

int
main(int argc, char *argv[])
{
    int err;
    pthread_t tid;
    char *cmd;
    struct sigaction sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Перейти в режим демона.
     */
    daemonize(cmd);
```

```
    /*
     * Убедиться в том, что ранее не была запущена другая копия демона.
     */
    if (already_running()) {
        syslog(LOG_ERR, "демон уже запущен");
        exit(1);
    }

    /*
     * Восстановить действие по умолчанию для сигнала SIGHUP
     * и заблокировать все сигналы.
     */
    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: невозможно восстановить действие SIG_DFL для SIGHUP");
    sigfillset(&mask);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
        err_exit(err, "ошибка выполнения операции SIG_BLOCK");

    /*
     * Создать поток, который будет заниматься обработкой SIGHUP и SIGTERM.
     */
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "невозможно создать поток");

    /*
     * Остальная часть программы-демона.
     */
    /* ... */
    exit(0);
}
```

Для перехода в режим демона программа использует функцию daemonize из листинга 13.1. После возврата из нее вызывается функция already_running из листинга 13.2, которая проверяет наличие других запущенных копий демона. В этой точке сигнал SIGHUP все еще игнорируется, поэтому мы должны переустановить его диспозицию в значение по умолчанию, в противном случае функция sigwait никогда не сможет получить его.

Далее выполняется блокировка всех сигналов, поскольку это рекомендуется для многопоточных программ, и создается поток, который будет заниматься обработкой сигналов. Поток обслуживает только сигналы SIGHUP и SIGTERM. При получении сигнала SIGHUP функция reread перечитывает файл конфигурации, а при получении сигнала SIGTERM поток записывает сообщение в журнал и завершает работу процесса.

В табл. 10.1 указано, что действие по умолчанию для сигналов SIGHUP и SIGTERM состоит в завершении процесса. Поскольку эти сигналы заблокированы, демон не будет завершаться, если получит один из них. Вместо этого от-

дельный поток будет получать номера доставленных сигналов с помощью функции sigwait.

Пример

Как уже отмечалось в разделе 12.8, в ОС Linux потоки ведут себя по отношению к сигналам несколько иначе. Это осложняет идентификацию процесса, которому должен быть передан сигнал. Кроме того, нет никаких гарантий, что демон будет реагировать на сигнал так, как мы этого ожидаем, из-за различий в реализации.

Программа, представленная листингом 13.4, показывает, как демон может перехватить сигнал SIGHUP и выполнить повторное чтение конфигурационного файла, не используя для этого отдельного потока.

Листинг 13.4. Альтернативная реализация демона, который перечитывает конфигурационный файл по сигналу

```
#include "apue.h"
#include <syslog.h>
#include <errno.h>

extern int lockfile(int);
extern int already_running(void);

void
reread(void)
{
    /* ... */
}

void
sigterm(int signo)
{
    syslog(LOG_INFO, "получен сигнал SIGTERM; выход");
    exit(0);
}

void
sighup(int signo)
{
    syslog(LOG_INFO, "Чтение конфигурационного файла");
    reread();
}

int
main(int argc, char *argv[])
{
    char *cmd;
    struct sigaction sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
```

```
        cmd++;

    /*
     * Перейти в режим демона.
     */
    daemonize(cmd);

    /*
     * Убедиться в том, что ранее не была запущена другая копия демона.
     */
    if (already_running()) {
        syslog(LOG_ERR, "демон уже запущен");
        exit(1);
    }

    /*
     * Установить обработчики сигналов.
     */
    sa.sa_handler = sigterm;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGHUP);
    sa.sa_flags = 0;
    if (sigaction(SIGTERM, &sa, NULL) < 0) {
        syslog(LOG_ERR, "невозможно перехватить сигнал SIGTERM: %s",
               strerror(errno));
        exit(1);
    }

    sa.sa_handler = sighup;
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask, SIGTERM);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0) {
        syslog(LOG_ERR, "невозможно перехватить сигнал SIGHUP: %s",
               strerror(errno));
        exit(1);
    }

    /*
     * Остальная часть программы-демона.
     */
    /* ... */
    exit(0);
}
```

После инициализации демона устанавливаются обработчики сигналов SIGHUP и SIGTERM. У нас есть два варианта обработки сигнала SIGHUP: либо читать конфигурационный файл в функции-обработчике, либо просто установить в обработчике специальный флаг, а все необходимые действия выполнять в основном потоке программы.