15. Демоны

Демонами в мире Unix традиционно называются процессы, которые не взаимодействуют с пользователем напрямую. У процесса-демона нет управляющего терминала и нет, соответственно, пользовательского интерфейса. Для управления демонами приходится использовать другие программы.

Само название «демоны» возникло благодаря тому, что многие процессы этого типа большую часть времени проводят в ожидании какого-то события. Когда это событие наступает, демон активизируется (выпрыгивает, как чертик из табакерки), выполняет свою работу и снова засыпает в ожидании события.

Следует отметить, что многие демоны, такие как, например, Web-сервер или сервер баз данных, могут отбирать на себя практически все процессорное время и другие ресурсы системы. Такие демоны гораздо больше работают, чем спят.

Вам, вряд ли придется заниматься созданием собственного демона, поскольку круг задач, для которых может понадобиться демон, не так уж и велик. Область применения демонов создание таких приложений, которые могут, и должны, выполняться без участия пользователя. Обычно это разного рода серверы.

Тем не менее, демоны задействуют многие важные элементы системы и понимание принципов работы демонов способствует пониманию принципов работы Unix/Linux в целом (помимо прочего, добрый демон поможет нам изучить некоторые особенности применения сигналов, с которыми мы ранее не сталкивались).

В качестве примера демона мы рассмотрим простой сетевой сервер aahzd, способный принимать запросы клиентов и возвращать ответы.

Исходный код нашего сервера aahzd.c представляет собой доработанный исходный код демонстрационного демона, написанного Давидом Жилье (David Gillies), ниже приведен исходный код (функции main() нашего демона):

```
volatile sig_atomic_t   gGracefulShutdown=0;
volatile sig_atomic_t   gCaughtHupSignal=0;
int               gLockFileDesc=-1;
int               gMasterSocket=-1;
const int         gaahzdPort=30333;
const char *constgLockFilePath = "/var/run/aahzd.pid";
int main(int argc,char *argv[])
{
      int                                 result;
      pid_t                               daemonPID;
      if (argc > 1) {
            int fd, len;
            pid_t pid;
            char pid_buf[16];
            if ((fd = open(gLockFilePath, O_RDONLY)) < 0)
            {
              perror("Lock file not found. May be the server is not running?");
              exit(fd);
            }
            len = read(fd, pid_buf, 16);
            pid_buf[len] = 0;
            pid = atoi(pid_buf);
            if(!strcmp(argv[1], "stop")) {
                  kill(pid, SIGUSR1);
                  exit(EXIT_SUCCESS);
            }
            if(!strcmp(argv[1], "restart")) {
                  kill(pid, SIGHUP);
                  exit(EXIT_SUCCESS);
            }
```

```
            printf ("usage %s [stop|restart]\n", argv[0]);
            exit (EXIT_FAILURE);
      }
      if((result = BecomeDaemonProcess(gLockFilePath, "aahzd",
            LOG_DEBUG, &gLockFileDesc, &daemonPID))<0) {
                  perror("Failed to become daemon process");
                  exit(result);
            }
      if((result = ConfigureSignalHandlers())<0) {
            syslog(LOG_LOCAL0|LOG_INFO, "ConfigureSignalHandlers failed,
                  errno=%d", errno);
            unlink(gLockFilePath);
            exit(result);
      }
      if((result = BindPassiveSocket(INADDR_ANY, gaahzdPort,
                  &gMasterSocket))<0) {
            syslog(LOG_LOCAL0|LOG_INFO, "BindPassiveSocket failed,
                  errno=%d", errno);
            unlink(gLockFilePath);
            exit(result);
      }
      do {
            if(AcceptConnections(gMasterSocket)<0) {
                  syslog(LOG_LOCAL0|LOG_INFO,"AcceptConnections failed,
                        errno=%d",errno);
                  unlink(gLockFilePath);
                  exit(result);
            }
            if((gGracefulShutdown==1)&&(gCaughtHupSignal==0))
                  break;
            gGracefulShutdown=gCaughtHupSignal=0;
      } while(1);
      TidyUp();
      return 0;
}
```

Сейчас мы пропустим блок операторов if (argc > 1) {...} (мы вернемся к нему позже) и рассмотрим основные этапы работы демона. Функция BecomeDaemonProcess() превращает обычный консольный процесс Linux в процесс-демон.

Функция ConfigureSignalHandlers() настраивает обработчики сигналов процесса-демона, а функция BindPassiveSocket() открывает определеный порт TCP/IP для прослушивания входящих запросов.

Далее следует цикл, в котором сервер обрабатывает запросы. Многие сетевые серверы, получив запрос, создают дочерний процесс для его обработки. Таким образом достигается возможность параллельной обработки запросов. Некоторые серверы используют для параллельной обработки запросов потоки.

Что касается нашего сервера, то из соображений простоты он обрабатывает запросы в последовательном (блокирующем) режиме. Нормальный выход из цикла обработки запросов происходит при получении процессом сигнала SIGUSER1. После выхода из цикла процесс вызывает функцию TidyUp() и завершает работу.

Мы, безусловно, можем завершить процесс-демон, послав ему сигнал SIGKILL (SIGTERM и некоторые другие), но пользовательский сигнал SIGUSER1 гарантирует вежливое завершение нашего демона. «Вежливое завершение» означает, что сервер ответит на текущий запрос перед тем, как завершиться и удалит свой pid-файл.

Рассмотрим теперь подробнее функцию BecomeDaemonProcess(), благодаря которой обычный процесс Linux становится демоном. Мы делаем корневую директорию текущей директорией процесса-демона.

```
chdir("/");
```

Будучи запущен, наш демон может работать вплоть до перезагрузки системы, поэтому его

текущая директория должна принадлежать файловой системе, которая не может быть размонтирована.
Каждый процесс-демон создает так называемый pid-файл (или файл блокировки). Этот файл обычно содержится в директории /var/run и имеет имя daemon.pid, где “daemon” соответствует имени демона:

```
lockFD = open(lockFileName, O_RDWR|O_CREAT|O_EXCL, 0644);
```

Файл блокировки содержит значение PID процесса демона. Этот файл важен по двум причинам. Во-первых, его наличие позволяет установить, что в системе уже запущен один экземпляр демона. Большинство демонов, включая наш, должны выполняться не более чем в одном экземпляре (это логично, если учесть, что демоны часто обращаются к неразделяемым ресурсам, таким, как сетевые порты).
Завершаясь, процесс-демон удаляет pid-файл, указывая тем самым, что можно запустить другой экземпляр процесса. Однако, работа демона не всегда завершается нормально, и тогда на диске остается pid-файл несуществующего процесса.
Это, казалось бы, может стать непреодолимым препятствием для повторного запуска демона, но на самом деле, демоны успешно справляются с такими ситуациями. В процессе запуска демон проверяет наличие на диске pid-файла с соответствующим именем. Если такой файл существует, демон считывает из него значение PID и с помощью функции kill(2) проверяет, существует ли в системе процесс с указанным PID.
Если процесс существует, значит, пользователь пытается запустить демон повторно. В этом случае программа выводит соответствующее сообщение и завершается. Если процесса с указанным PID в системе нет, значит pid-файл принадлежал аварийного завершенному демону.
В этой ситуации программа обычно советует пользователю удалить pid-файл (ответственность в таких делах всегда лучше переложить на пользователя) и попытаться запустить ее еще раз.
Может, конечно, случиться и так, что после аварийного завершения демона на диске останется его pid-файл, а затем какой-то другой процесс получит тот же самый PID, что был у демона. В этой ситуации для вновь запускаемого демона все будет выглядеть так, как будто его копия уже работает в системе, и запустить демон повторно вы не сможете. К счастью, описанная ситуация крайне маловероятна.
Вторая причина, по которой файл блокировки считается полезным, заключается в том, что с помощью этого файла мы можем быстро выяснить PID демона, не прибегая к команде ps.
Далее наш демон вызывает функцию fork(3), которая создает копию его процесса. Родительский процесс при этом завершается:

```
curPID=fork();
switch(curPID) {
      case 0: /* мы в дочернем процессе */
              break;
      case -1: /* ошибка fork() - случилось что-то страшное */
              fprintf(stderr,"Error: initial fork failed: %s\n",
                        strerror(errno));
              return -1;
              break;
      default: /* мы в родительском процессе, завершаем его */
              exit(0);
              break;
}
```

Делается это для того чтобы процесс-демон отключился от управляющего терминала.
С каждым терминалом Unix связан набор групп процессов, именуемый сессией. В каждый момент времени только одна из групп процессов, входящих в сессию, имеет доступ к терминалу (то есть, может выполнять ввод/вывод с помощью терминала). Эта группа именуется foreground (приоритетной).
В каждой сессии есть процесс-родоначальник, который называется лидером сессии. Если

процесс-демон запущен с консоли, он, естественно, становится частью приоритетной группы процессов, входящих в сессию соответствующего терминала.
Для того чтобы отключиться от терминала, демон должен начать новую сессию, не связанную с каким-либо терминалом. Для того чтобы демон мог начать новую сессию, он сам не должен быть лидером какой-либо другой сессии. Вызов fork() создает дочерний процесс, который заведомо не является лидером сессии. Далее дочерний процесс, полученный с помощью fork(), начинает новую сессию с помощью вызова функции setsid(2). При этом процесс становится лидером (и единственным участником) новой сессии.

```
if(setsid()<0)
      return -1;
```

Итак, на данном этапе мы имеем процесс, не связанный с каким-либо терминалом. Далее рекомендуется вызвать fork() еще раз, чтобы новый процесс осиротел и его усыновил (точнее удочерил) init.
Стоит отметить, что теперь наш демон получил новый PID, который мы снова должны записать в pid-файл демона. Мы записываем значение PID в файл в строковом виде (а не как переменную типа pid_t). Делается это для удобства пользователя, чтобы значение PID из pid-файла можно было прочитать с помощью cat. Например:
kill `cat /var/run/aahzd.pid `
Нашему демону удалось разорвать связь с терминалом, с которого он был запущен, но он все еще может быть связан с другими процессами и файловыми системами через файловые дескрипторы, унаследованные от родительских процессов. Для того чтобы разорвать и эту связь, мы закрываем все файловые дескрипторы, открытые в нашем процессе:

```
numFiles = sysconf(_SC_OPEN_MAX);
for(i = numFiles-1; i >= 0; --i) {
      if(i != lockFD)
      close(i);
}
```

Функция sysconf() с параметром _SC_OPEN_MAX возвращает максимально возможное количество дескрипторов, которые может открыть наша программа. Мы вызываем функцию close() для каждого дескриптора (независимо от того, открыт он или нет), за исключением дескриптора pid-файла, который должен оставаться открытым.
Во время работы демона дескрипторы стандартных потоков ввода, вывода и ошибок также должны быть открыты, поскольку они необходимы многим функциям стандартной библиотеки. В то же время, эти дескрипторы не должны указывать на какие-либо реальные потоки ввода/вывода.
Для того, чтобы решить эту задачу, мы закрываем первые три дескриптора, а затем снова открываем их, указывая в качестве имени файла /dev/null:

```
stdioFD = open("/dev/null", O_RDWR);
dup(stdioFD);
dup(stdioFD);
```

Теперь мы можем быть уверены, что демон не получит доступа к какому-либо терминалу. Тем не менее, у демона должна быть возможность выводить куда-то сообщения о своей работе. Традиционно для этого используются файлы журналов (log-файлы). Файлы журналов для демона подобны черным ящикам самолетов. Если в работе демона произошел какой-то сбой, пользователь может проанализировать файл журнала, чтобы установить причину сбоя. Ничто не мешает нашему демону открыть свой собственный файл журнала, но это не очень удобно. Большинство демонов пользуются услугами утилиты syslog, ведущей журналы множества системных событий. Мы открываем доступ к журналу syslog с помощью функции openlog(3):

```
openlog(logPrefix, LOG_PID|LOG_CONS|LOG_NDELAY|LOG_NOWAIT, LOG_LOCAL0);
(void) setlogmask(LOG_UPTO(logLevel));
```

Первый параметр функции openlog() – префикс, который будет добавляться к каждой записи в системном журнале. Далее следуют различные опции syslog. Функция setlogmask(3)

позволяет установить уровень приоритета сообщений, которые записываются в журнал событий.
При вызове функции BecomeDaemonProcess() мы передаем в параметре logLevel значение LOG_DEBUG. В сочетании с макросом LOG_UPTO это означает, что в журнал будут записываться все сообщения с приоритетом, начиная с наивысшего и заканчивая LOG_DEBUG.
Последнее, что нам нужно сделать для «демонизации» процесса – вызывать функцию setpgrp();
Этот вызов создает новую группу процессов, идентификатором которой является идентификатор текущего процесса. На этом работа функции BecomeDaemonProcess() завершается, так как теперь процесс стал настоящим демоном.
Функция ConfigureSignalHandlers() настраивает обработчики сигналов. Сигналы, которые получит наш демон, можно разделить на три группы: игнорируемые, «фатальные» и обрабатываемые. Вызывая функцию signal(SIGUSR2, SIG_IGN) мы указываем, что наш демон должен игнорировать сигнал SIGUSR2.
Аналогично мы поступаем с сигналами SIGPIPE, SIGALRM, SIGTSTP, SIGPROF, SIGCHLD. Сигналы SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGIOT, SIGBUS, SIGFPE, SIGSEGV, SIGSTKFLT, SIGCONT, SIGPWR и SIGSYS относятся к категории «фатальных». Мы не можем их игнорировать, но и продолжать выполнение процесса-демона после получения одного из этих сигналов нежелательно. Мы назначаем всем этим сигналам обработчик FatalSigHandler, например:

```
signal(SIGQUIT, FatalSigHandler);
```

Функция-обработчик FatalSigHandler() записывает в журнал событий информацию о полученном сигнале, и затем завершает процесс, вызвав перед этим функции closelog() и TidyUp(), которые высвобождают все занятые процессом ресурсы:

```
void FatalSigHandler(int sig) {
#ifdef _GNU_SOURCE
      syslog(LOG_LOCAL0|LOG_INFO,"caught signal: %s – exiting",strsignal(sig));
#else
      syslog(LOG_LOCAL0|LOG_INFO,"caught signal: %d – exiting",sig);
#endif
      closelog();
      TidyUp();
      _exit(0);
}
```

На те три сигнала, которые относятся к категории обрабатываемых, – SIGTERM, SIGUSR1 и SIGHUP, демон реагирует по-разному:

```
sigtermSA.sa_handler = TermHandler;
sigemptyset(&sigtermSA.sa_mask);
sigtermSA.sa_flags = 0;
sigaction(SIGTERM, &sigtermSA, NULL);
sigusr1SA.sa_handler = Usr1Handler;
sigemptyset(&sigusr1SA.sa_mask);
sigusr1SA.sa_flags = 0;
sigaction(SIGUSR1, &sigusr1SA, NULL);
sighupSA.sa_handler = HupHandler;
sigemptyset(&sighupSA.sa_mask);
sighupSA.sa_flags = 0;
sigaction(SIGHUP, &sighupSA, NULL);
```

Обработчик TermHandler() вызывает функцию TidyUp() и завершает процесс. Обработчик Usr1Handler() делает в системном журнале запись о вежливом завершении процесса и присваивает переменной gGracefulShutdown значение 1 (что, как вы помните, приводит к выходу из цикла обработки запросов, когда цикл будет готов к этому). Обработчик сигнала HupHandler() также делает запись в системном журнале, после чего присваивает значение 1 переменным gGracefulShutdown и gCaughtHupSignal.

В реальной жизни получение сигнала SIGHUP приводит к перезапуску демона, который сопровождается повторным прочтением файла конфигурации (который обычно читается демоном именно во время запуска) и переустановкой значений записанных в нем параметров. Именно необходимость прочесть повторно файл конфигурации является наиболее частой причиной перезапуска демонов. У нашего демона файла конфигурации нет, так что в процессе перезапуска делать ему особенно нечего.
Обратите внимание на тип переменных gGracefulShutdown и gCaughtHupSignal. С типом sig_atomic_t мы раньше не встречались. Применение этого типа гарантирует, что чтение и запись данных в переменные gGracefulShutdown и gCaughtHupSignal будет выполняться атомарно, одной инструкцией процессора, которая не может быть прервана.
Атомарность при работе с переменными gGracefulShutdown и gCaughtHupSignal важна потому, что к ним могут одновременно получить доступ и обработчики сигналов, и главная функция программы. По этой же причине мы помечаем указанные переменные ключевым словом volatile.
Функция BindPassiveSocket() открывает для прослушивания порт сервера (в нашем случае это порт 30333) на всех доступных сетевых интерфейсах и возвращает соответствующий сокет:

```
int BindPassiveSocket(const int portNum,
        int *const boundSocket)
{
      struct sockaddr_in sin;
      int   newsock, optval;
      size_t optlen;
      memset(&sin.sin_zero, 0, 8);
      sin.sin_port = htons(portNum);
      sin.sin_family = AF_INET;
      sin.sin_addr.s_addr = htonl(INADDR_ANY);
      if((newsock= socket(PF_INET, SOCK_STREAM, 0))<0)
      return -1;
      optval = 1;
      optlen = sizeof(int);
      setsockopt(newsock, SOL_SOCKET, SO_REUSEADDR, &optval, optlen);
      if(bind(newsock, (struct sockaddr*) &sin, sizeof(struct sockaddr_in))<0)
      return -1;
      if(listen(newsock,SOMAXCONN)<0)
      return -1;
      *boundSocket = newsock;
      return 0;
}
```

Отметим одну интересную деталь — если предыдущий, уже закрытый, сокет, связанный с данным портом, находится в состоянии TIME_WAIT, между закрытием старого и открытием нового сокета может произойти задержка до двух минут. Для того, чтобы при повторном запуске демона нам не пришлось ждать, мы используем функцию setsockopt() с параметром SO_REUSEADDR.
Функция AcceptConnections() обрабатывает запросы последовательно, используя блокирующий вызов accept():

```
int AcceptConnections(const int master)
{
      int   proceed = 1, slave, retval = 0;
      struct sockaddr_in      client;
      socklen_t   clilen;
      while((proceed==1)&&(gGracefulShutdown==0)) {
            clilen = sizeof(client);
            slave = accept(master,(struct sockaddr *)&client,&clilen);
            if(slave<0) {     /* ошибка accept() */
                  if(errno == EINTR)
                        continue;
```

```
                        syslog(LOG_LOCAL0|LOG_INFO,"accept() failed: %m\n");
                        proceed = 0;
                        retval = -1;
                }
                else
                {
                        retval = HandleConnection(slave);
                        if(retval)
                                proceed = 0;
                }
                close(slave);
        }
        return retval;
}
```

Это не лучший образ поведения демона, но на данном этапе нет смысла усложнять данный демон. Переменная proceed, совместно с переменной gGracefulShutdown, указывает, должна ли программа продолжать обрабатывать запросы. Если очередной вызов connect() или HandleConnection() вернул сообщение об ошибке, этой переменной присваивается 0 и обработка запросов прекращается. Новый сокет, полученный в результате вызовы accept(), передается функции HandleConnection().

```
int HandleConnection(const int slave)
{
        char readbuf[1025];
        size_t      bytesRead;
        const size_t      buflen=1024;
        int   retval;
        retval = ReadLine(slave, readbuf, buflen, &bytesRead);
        if(retval==0)
                WriteToSocket(slave, readbuf, bytesRead);
        return retval;
}
```

Функция HandleConnection() считывает переданную клиентом строку и тут же возвращает ее клиенту. Затем функция AcceptConnections() закрывает соединение, открытое в результате вызова accept(). Функции ReadLine() и WriteToSocket() тривиальны, и рассматривать их мы не будем. Если где-то в цепочке вызовов AcceptConnections(), HandleConnection(), ReadLine() и WriteToSocket() возникла ошибка, информация об ошибке будет передаваться вверх по цепочке до тех пор, пока не достигнет функции main(). В функции main() эта информация приведет к немедленному завершению работы демона с соответствующей записью в журнал системных сообщений.

Рассмотрим, наконец, функцию TidyUp(), к которой обращаются многие функции сервера перед тем, как завершить его работу.

```
void TidyUp(void)
{
        if(gLockFileDesc!=-1) {
                close(gLockFileDesc);
                unlink(gLockFilePath);
                gLockFileDesc=-1;
        }
        if(gMasterSocket!=-1) {
                close(gMasterSocket);
                gMasterSocket=-1;
        }
}
```

Задача функции TidyUp() – «прибрать мусор» за процессом-демоном. В принципе, без этой функции можно обойтись, так как после завершения процесса система сама закроет все его дескрипторы, но правила хорошего тона требуют явного высвобождения всех ресурсов, выделенных явным образом.

Если вы скомпилируете программу-демон с помощью команды

```
$ gcc aahzd.c -o aahzd
```

То сможете запустить демон командой:

```
# ./aahzd
```

Поскольку демон нуждается в доступе к директории /var/run, запускать его нужно в режиме root. Сразу после запуска программы вы снова увидите приглашение командной строки, что для демонов совершенно нормально.

Если бы сервер aahzd выполнял что-нибудь полезное, команду его запуска можно было бы прописать в одном из сценариев запуска системы в директории /etc/init.d, но мы этого делать не будем. После того как сервер запущен, вы можете дать команду:

```
$ telnet localhost 30333
```

В результате будет установлено соединение с сервером. Напечатайте какую-нибудь последовательность символов в консоли telnet и нажмите «Ввод». В качестве ответа сервер возвратит напечатанную строку и закроет соединение.

Вернемся теперь к начальным строкам функции main(). Хотя мы можем получить PID демона из его pid-файла и управлять демоном с помощью команды kill, такой вариант нельзя назвать очень удобным. Часто для управления демоном используется сам исполнимый файл демона, запускаемый со специальными аргументами командной строки. Наш демон понимает две команды: stop (завершение работы демона) и restart (перезапуск).

Посмотрим, как поведет себя демон, запущенный с аргументами командной строки. В этом случае в начале программы демон пытается считать значение PID из своего pid-файла. Если открыть pid-файл не удается, значит, скорее всего, демон не запущен, и управляющему режиму просто нечего делать. Если значение PID получено, процесс, управляющий демоном, посылает демону соответствующий сигнал с помощью функции kill().

Демоны не рассчитаны на то, чтобы получать какую-либо информацию от пользователя в интерактивном режиме. Собственную информацию они передают другим программам либо записывают в журналы системных событий.