Unix как царство тьмы. Все, что видно - это черный экран с небольшой серой надписью - командной строкой, что представляет в нем тебя самого. Он населен скрытыми в недрах командами, о существовании несметного количества которых можно только догадываться. И в нем живут демоны.

Так может показаться сначала. На самом же деле, unix-подобные операционные системы довольно дружелюбны и миролюбивы, даже несмотря на то, что демоны там все-таки есть. А догадались вы правильно. Именно их мы сегодня и будем писать.

Так получилось, что два года подряд на одном из моих предыдущих мест работы я, можно сказать, специализировался на программах такого рода, за что меня называли "демонологом". Демоны оказывались подходящими решениями для массы задач, поэто меня в последнее время все больше и больше одолевало желание поделиться накопленным опытом с широкой аудиторией. Эта статья призвана поведать о том, какими полезными для народного хозяйства существами являются демоны.

Курс молодого демонолога

Итак, демон - это программа, не имеющая стандартного ввода и вывода, и при этом работающая в фоновом режиме. Практически все системные сервисы в unix суть демоны: cron, sendmail, telnet, ssh, ftp, talk, и прочие несметные полчища нечисти. Как видно из этого списка, демон совсем не обязательно должен обслуживать входящие TCP/IP соединения - стандартный cron выполняет исключительно локальные задачи.

Любой хоть мало-мальски уважающий себя демон начинает работу с вызова fork(2), который "раздваивает" процесс. В итоге получаются два совершенно идентичных процесса, отличающихся только PID и значением, возвращенным в ним оным вызовом. Процесс-родитель получит PID ребенка, а ребенок - 0 по возвращении из вызова fork(2). Сразу после старта это делается затем, чтобы запущенная программа вышла и не задерживала работу пользователя в командной строке, или какого-нибудь скрипта, что запускает демонов автоматически. К тому же, теперь гарантируется, что оставшийся кусок демона не будет лидером группы процессов. А это, в свою очередь, нам нужно, чтобы предпринять следующий шаг по демонизации программы.

POSIX вызов setsid(2), в случае, если текущий процесс не является лидером группы, создает новую группу процессов, и на момент возврата только процесс, вызвавший setsid(2) будет единственным членом группы. Также, у процесса больше нет контролирующего терминала.

Расскажу немного, для чего нужны группы процессов. Прежде всего, по группам происходит рассылка сигналов операционной системы. С помощью вызова kill(2) можно послать сигнал не только одному процессу, как может показаться сначала. Передав оному в качестве первого аргумента 0, можно послать сигнал всем процессам в группе, к которой принадлежит вызвавший его процесс. Если же передать отрицательное значение, то сигнал будет послан всем членам заданной группы. Это очень удобно, так как, скажем, все процессы-родственники, порожденные с помощью fork(2), являются по умолчанию членами одной и той же группы, можно с легкостью послать сигнал всем сразу.

Также есть понятие контролирующего терминала. Самое слово "контролирующий" здесь нужно воспринимать буквально. Процессы или группы процессов, связанные с каким-либо терминалом, принадлежат к сессии. Контроль производится на уровне терминала, и подконтрольной является вся сессия. Скажем, если терминал "отвалился" - закрылось TCP/IP или соединение по асинхронному порту, ОС посылает сигнал SIGHUP (hangup) всем процессам и группам процессов сессии.

Теперь понятно, что чтобы ликвидировать зависимость от терминала и вызывающей оболочки, мы и занимаемся всем этим непотребством с fork(2) и setsid(2). Теперь демону предстоит сделать два последних шага. Будучи гордой программой, он не только не желает зависеть от чего-либо, но хочет минимизировать неудобства, причиняемые окружающим. В частности, ему следует сделать своим текущим каталогом корневой, дабы не возникало проблем с монтированными файловыми системами. Действительно, если демон был запущен из каталога файловой системы, отличной от корневой, и тут же

ушел в фоновый режим, команда umount будет ругаться о занятости ресурса одним из процессов до тех пор, пока работа демона не завершится.

И, наконец, с закрытием файловых дескрипторов стандартного ввода и вывода, демон уходит в свободный полет и может приступать к выполнению своей основной задачи.

Что позволено цезарю - не позволено быку

Мы рассмотрели процесс превращения программы в демона. Это - средство, а цель обычно бывает в другом. В моей практике я часто сталкивался с необходимостью написания демонов, которые обслуживали сетевые соединения. В самом деле, это очень широкая область, в которой программы такого рода могут здорово пригодиться.

Приведу один любопытный пример. В мою бытность программистом в одной харьковской фирме, к нам поступил заказ на написание баннерной системы. Задача была очень неслабой. Структура базы данных не умещалась на столе, математический аппарат - экспертную систему - разрабатывал преподаватель военного ВУЗа, и так далее. Самым интересным местом заказа были требования к быстродействию. Система должна была быть в состоянии отдавать порядка 500 баннеров в секунду. И вот, дело дошло до тестирования производительности CGI программ. Когда мы убедились, что 500 там никак нет, начался профайлинг - мы замеряли производительность отдельных участков кода. Самым узким местом оказался логгинг - на каждый запрошенный баннер нужно было добавлять в таблицу лога базы данных системы одну запись. Добавление работало быстро, но когда дело доходило до commit - сохранения изменений, начинались тормоза. Без лога экспертная система не смогла бы получать данных для анализа. Мы были в раздумьях.

Тогда я вспомнил о своих любимых питомцах - демонах, и меня осенило. Я предложил вынести операции insert и commit в отдельный модуль, который мог бы работать даже на другой машине, дабы не создавать лишних задержек. Мной была написана связка из демона и клиентской библиотеки на C++, которые занимались тривиальнейшим делом: клиент (функция из библиотеки) "заворачивала" запись лога в UDP пакет, и отсылала демону. Демон же имел встроенную очередь, в которой накапливались данные из пакетов. Когда размер очереди превышал заданное в конфигурации системы число, порождался новый процесс. При этом, очередь в процессе-накопителе очищалась, и он продолжал принимать новые данные. В порожденном же процессе открывалось соединение к базе данных, для всех записей производился insert, потом commit, после чего порожденный процесс благополучно завершался.

Как известно, UDP протокол не требует предварительного установления соединения, как TCP/IP, что делает его очень удобным для мгновенной передачи данных в небольших объемах. Тот факт, что он в теории не гарантирует доставку каждого из посланных пакетов, нас не очень смущал, так как на практике 100% данных проходили без проблем в пределах локальной сети, да и статистика - то, для чего нам был нужен логгинг, терпит погрешности. Решение оказалось очень подходящим, и мои питомцы оправдали свое существование в очередной раз.

UDP-демон

Теперь напишем простенький UDP демон и клиент к нему. По сути, надо дописать программу в отпочковывающейся ее части. Приведенный исходник реализует описанный выше механизм. Основной процесс накапливает получаемые пакеты, после чего порождает другой, который сохраняет их содержимое в текущем каталоге в файлах со случайными именами. Этот пример реализован на C++, так как мне было удобно использовать некоторые STL элементы, такие, как vector и string.

Здесь, пожалуй, внимание нужно уделить механизму работы с UDP сокетами и вызову fork(2), который порождает commit-процесс. Сначала проходит обыкновенная процедура создания и подготовки "слушающего" сокета, а затем цикл начинает накапливать данные, при необходимости порождая "записывающие" процессы. Плюс ко всему, этот демон обладает несколько продвинутой диагностикой.

Сообщения о порождении новых процессов он пишет с помощью вызова syslog(3) в стандартный сервис лога.

Кроме того, в этом же примере иллюстрируется правило, которого следует придерживаться в любой программе, потенциально могущей порождать массу процессов. Это обязательный обработчик сигнала SIGCHLD. Упомянутый сигнал посылается программе всегда, когда один из процессов-детей завершается. Наш обработчик состоит из одного вызова waitpid(2), который используется, преследуя следующую целью. Здесь наша терминология пополнится еще одним леденящим душу словом - "зомби" (zombie), которое в данном контексте, конечно, ничего общего с ходячим мертвым телом иметь не будет. Так в UNIX называются "тени" процессов, которые завершились и память за которыми уже освобождена, однако, в таблице процессов записи остаются на тот случай, если программа-родитель захочет вычитать код завершения. Накапливание зомби зачастую приводит к невозможности породить процесс в системе в целом (впрочем, зависит от установок ulimit), поэтому удалять зомби очень желательно. Таким образом, если бы нам нужен был механизм обработки такого кода, то, во-первых, в методе udpdaemon::commit() должно было присутствовать некоторое разнообразие значений, передаваемых вызову exit(3). Во-вторых, для обработки этих кодов udpdaemon::sighandler() передавал бы в waitpid(2) в качестве второго параметра ссылку на переменную типа int, в которую и помещалось бы желаемое значение. В нашем же примере вызов waitpid(2) просто убивает запись о только что завершившемся commit-процессе.

Завершить работу такого демона можно, послав сигнал SIGTERM процессу с PID, который программа сообщит при старте. Это можно сделать как с помощью вызова kill(2), так и при помощь одноименной команды.

$ kill <PID>

Или же по имени процесса, воспользовавшись командой killall(1).

$ killall daemon2

Вообще говоря, хорошим тоном является прямо там же, в демоне, иметь механизм остановки. Обычно это делается следующим образом. При старте PID созданного при помощи fork(2) "слушающего" процесса записывается куда-нибудь в файл, а потом, при подаче в командной строке какого-либо специального ключика, этот номерок достается и ему посылается SIGTERM.

Исходник клиента тоже предельно прост. Здесь всего лишь создается клиентский сокет и заданное количество раз в цикле посылаются данные. Для простоты предполагается, что демон всегда работает на той же машине (в качестве адреса назначения используется localhost:1666).

Выше я упоминал одно неприятное свойство UDP протокола, заключающееся в невозможности убедиться в удачной доставке пакета. Конечно, проблему эту можно решить, отсылая свой собственный пакет подтверждения на каждый полученный демоном кусок данных, однако, есть еще и любопытная возможность - расширение, предоставляемое пока, правда, только Linux, и то с версии ядра 2.3. Это - флаг MSG_CONFIRM, который может быть передан функциям семейства send(2).

TCP-демон

С UDP разобрались без проблем. В рамках этой статьи стоит также рассмотреть работу с протоколом TCP, который является более "продвинутым", так как ответственнее относится к данным, и для передачи оных требует открытия отдельного соединения для каждого клиента. Разумеется, демоны, расчитанные на TCP, будут предназначаться уже для других задач. Для простого накапливания данных, как в предыдущем примере, TCP слишком расточителен, так как нужно тратить время на открытие соединения, занимать отдельный сокет, количество которых, кстати говоря, в системе не бесконечное, и так далее. Однако, когда клиенты индивидуализированы, и есть понятие четко ограниченной сессии,

длящейся какое-то время, самое подходящее решение - это серверная программа, работающая по TCP протоколу.

Обычно демоны, обслуживающие TCP, используют механизм fork(2) для порождения процессов для каждого из соединений. В каждый из порожденных процессов передается открытый сокет. Большинство интернет-протоколов, базирующихся на TCP, таких, скажем, как SMTP, HTTP, и NNTP, работают в режиме диалога при помощи команд. При этом на каждую из текстовых строк, посланных клиентом, демон отвечает предусмотренным некоторым механизмом ответа.

Точно так же, как и для демона UDP, здесь присутствует обработчик SIGCHLD, призванный убивать возникающие процессы-зомби. За обработку каждого из соединений отвечает метод tcpdaemon::operate().

Текстовые протоколы, базированные на TCP, удобны еще и тем, что обращаться к ним можно непосредственно "вручную", и клиентские программы писать иногда бывает совсем не обязательно. Демон из примера выше как раз такой протокол и использует. И действительно, если мы соединимся с ним при помощи команды

$ telnet localhost 1667

то без проблем увидим наш протокол "в разрезе". А набрав команду "hello", получим ответ, предусмотренный в исходнике. Команда "quit" приведет к разрыву соединения на стороне демона.

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
the simple tcp daemon is ready
hello
my daemonic greatings
quit
Connection closed by foreign host.
```

Для "облегчения" TCP демонов вместо fork(2) иногда используют потоки (threads), порождая каждый раз не новый процесс, а поток в текущем процессе. Это приводит к более экономному расходованию ресурсов, в частности - процессора и памяти, но требует и большей аккуратности в написании. Дело в том, что если в демоне с fork(2) "упадет" один из обработчиков соединений с ошибкой вроде segmentation fault, "слушающий" процесс без проблем продолжит работу. С потоками такого не случится, так как вместе с обработчиком "свалится" сразу вся программа.

Плюс ко всему, использование fork(2) имеет под собой и некоторые исторические подоплеки, так как потоки в большинстве UNIX систем появились довольно недавно.

И напоследок...

В заключение хотелось бы сказать, что "прикладная демонология", несмотря на существование более "продвинутых" клиент-серверных технологий, таких, как RPC и CORBA, до сих пор не потеряла свою актуальность, в силу, наверное, своей легкости и простоты реализации в проектах любого рода. Удачного демоностроения.