Разработка системного сервиса и консольного приложения для взаимодействия с ним для FreeBSD в среде KDevelop

Лабораторная работа

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

Дата последней модификации: 24.11.2010 15:41:00

История изменений

08.11.2010 – Версия 0.1. Первичный документ. Влад Ковтун

Содержание

Лабораторная работа 13. Разработка системного сервиса и клиентского консольного приложения для взаимодействия с ним для FreeBSD в среде Kdevelop

Вопросы

- Постановка задачи.
- Методические рекомендации.

Постановка задачи

Цель

1. Разработать системный сервис (демон).

2. Разработать клиентское приложение для взаимодействия пользователя с демоном.

Задачи

1. Ознакомиться с особенностями построения системного сервиса в ОС FreeBSD.

2. Ознакомиться с особенностями установки системного сервиса в ОС FreeBSD.

3. Ознакомиться с особенностями межпроцессного взаимодействия в ОС FreeBSD.

4. Разработать демон для ОС FreeBSD для ведения текстовых логов клиентского консольного приложения.

4.1. В текстовом файле в формате UTF-8 необходимо фиксировать следующую информацию:

- IP адрес, с которого передано сообщение.
- Дата и время получения сообщения.
- Имя пользователя, который запустил клиентское приложение.
- Марка процессора, на котором выполняется клиентское приложение.
- Текст самого сообщения.

5. Разработать клиентское консольное приложение для ОС FreeBSD, которое взаимодействует с демоном логирования.

6. Оценить производительность демона (какое количество записей он сможет фиксировать в секунду).

7. Оформить отчет к лабораторной работе. В отчете обязательно указать, каким образом производится тестирование на корректность.

Задание

Управление демоном осуществляется посредством следующих команд:

- -lf:<logfilepath> – полный путь и имя файла лога, куда будет производиться запись сообщений.
- -p:<pidfilepath> - полный путь и имя файла, который будет использоваться для синхронизации процессов – демона, что позволит запустить лишь единственную версию демона.
- -start – запустить на выполнение демон.
- -stop – остановить функционирующий демон.
- -restart – перезапустить демон.
- -? - вывод информации о допустимых ключах командной строки.

Управление клиентским консольным приложением осуществляется посредством команд:

- -m <message> - текст сообщения, которое следует занести в лог-файл.
- -? - вывод информации о допустимых ключах командной строки.

Пример

Пример запуска демона, который хранит принятые сообщения в файле `log.txt`, а файл синхронизации называется `logger.pid`:

```
[vladk@pcbsd-2329]->loggerd -if:tmp\logs\log.txt -p:logger.pid
```

Пример запуска клиентского консольного приложения с внесением в лог-файл сообщения «hello world!»:

```
[vladk@pcbsd-2329]->loggerc -m:hello world!
```

Вывод

Во время работы приложения, рекомендуется выводить информацию о статусе приложения, а также о корректности его работы на консоль.

Требования

- Архитектура приложения строится по модульному принципу.
- За основу принимается стандартная библиотека C++ (в случае разработки на C++).
- Рекомендуется использовать защищенные ресурсы и указатели.
- Обязательным является обработка исключений.
- Исходный код обязан быть комментирован. Для C++ следует использовать нотацию doxygen [5].

Тестирование

Тестирование приложения осуществляется в несколько этапов:

- Воспользоваться тестовыми пакетными файлами для проверки корректности функционирования.

Методические указания для самостоятельной работы

Теоретические сведения

Создание демона для ОС FreeBSD

Демоны играют очень важную роль в работе ОС. Достаточно будет сказать, что тему, доступ по сети, использование системы печати и электронной почты, — все это обеспечивается соответствующими демонами — неинтеракативными программами, составляющими собственные сеансы (и группы) и не принадлежащими ни одному из пользовательских сеансов (групп).

Некоторые демоны работают постоянно, наиболее яркий пример такого демона — процесс `init(1M)`, являющийся прародителем всех прикладных процессов в системе. Другими примерами являются `cron(1M)`, позволяющий запускать программы в определенные моменты времени, `inetd(1M)`, обеспечивающий доступ к сервисам системы из сети, и `sendmail(1M)`, обесапечивающий получение и отправку электронной почты.

При описании взаимодействия процессов с терминалом и пользователем, отмечалось особое место демонов, которые не имеют управляющего терминала. Теперь в отношении демонов можно сформулировать ряд правил, определяющих их нормальное функционирование, которые необходимо учитывать при разработке таких программ:

1. Демон не должен реагировать на сигналы управления заданиями, посылаемые ему при попытке операций ввода/вывода с управляющим терминалом. Начиная с некоторого времени, демон снимает ассоциацию с управляющим терминалом, но на начальном этапе запуска ему может потребоваться вывести то или иное сообщение на экран.

2. Необходимо закрыть все открытые файлы (файловые дескрипторы), особенно стандартные потоки ввода/вывода. Многие из этих файлов представляют собой терминальные устройства, которые должны быть закрыты, например, при выходе пользователя из системы. Предполагается, что демон остается работать и после того, как пользователь «покинул» UNIX.

3. Необходимо снять его ассоциацию с группой процессов и управляющим терминалом. Это позволит демону избавиться от сигналов, генерируемых терминалом (`SIGINT` или `SIGHUP`), например, при нажатии определенных клавиш или выходе пользователя из системы.

4. Сообщения о работе демона следует направлять в специальный журнал с помощью функции `syslog(3)`, — это наиболее корректный способ передачи сообщений от демона.

5. Необходимо изменить текущий каталог на корневой. Если этого не сделать, а текущий каталог, допустим, находится на смонтированной файловой системе, последнюю нельзя будет размонтировать. Самым надежным выбором является корневой каталог, всегда принадлежащий корневой файловой системе.

```
#include <stdio.h>
#include <syslog.h>
#include<signal.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/resource.h>

main(int argc, char **argv)
{
int fd;
struct rlimit flim;
/* Если родительский процесс — init, можно не беспокоиться за терминальные
сигналы. Если нет — необходимо игнорировать сигналы, связанные с
вводом/выводом на терминал фонового процесса: SIGTTOU, SIGTTIN, SIGTSTP */
if (getppid() != 1)
{
    signal(SIGTTOU, SIG_IGN);
    signal (SIGTTIN, SIG_IGN);
    signal (SIGTSTP, SIG_IGN);

/*Теперь необходимо организовать собственную группу и сеанс, не имеющие
управляющего терминала. Однако лидером группы и сеанса может стать процесс,
если он еще не является лидером. Поскольку предыстория запуска данной
программы неизвестна, необходима гарантия, что наш процесс не является
лидером. Для этого порождаем дочерний процесс. Т. к. его PID уникален, то ни
группы, ни сеанса с таким идентификатором не существует, а значит нет и
лидера. При этом родительский процесс немедленно завершает выполнение,
поскольку он уже не нужен.
Существует еще одна причина необходимости порождения дочернего процесса. Если
демон был запущен из командной строки командного интерпретатора shell не в
фоновом режиме, последний будет ожидать завершения выполнения демона, и таким
образом, терминал будет заблокирован. Порождая процесс и завершая выполнение
родителя, имитируем для командного интерпретатора завершение работы демона,
после чего shell выведет свое приглашение.*/

    if (fork () !=0)
       exit(0);
/*Родитель заканчивает работу*/
/*Дочерний процесс с помощью системного вызова setsid(2) становится лидером
новой группы, сеанса и не имеет ассоциированного терминала14*/
    setsid ();
}
/*Теперь необходимо закрыть открытые файлы. Закроем все возможные файловые
дескрипторы. Максимальное число открытых файлов получим с помощью функции
getrlimit(2)*/
getrlimit(RLIMIT_NOFILE, &flim);
for (fd = 0; fd < flim.rlimjnax; d++)
    close(fd); /*Сменим текущий каталог на корневой*/
chdir("/");
/*Заявим о себе в системном журнале. Для этого сначала установим опции
ведения журнала: каждая запись будет предваряться идентификатором PID демона,
при невозможности записи в журнал сообщения будут выводиться на консоль,
источник сообщений определим как "системный демон" (см. комментарии к
функциям ведения журнала ниже).* /
openlog("Скелет демона" , LOG_PID | LOG_CONS, LOG_DAEMON); /*Отметимся*/
syslog(LOG_INFO, "Демон начал плодотворную работу...");
closelog();

/*Далее следует текст программы, реализующий полезные функции демона. Эта
часть предоставляется для собственной разработки.*/
```

```
}
```

Использование вызова `setsid(2)` справедливо для UNIX System V. Для BSD UNIX процесс должен создать группу, лидером которой он становится, а затем открыть управляющий терминал и с помощью команды `ioctrl(2)` отключиться от него.

В программе использовалось еще не обсуждавшаяся возможность системного журнала сообщений выполняющихся программ. Функцией генерации сообщений является `syslog(3)`, отправляющая сообщение демону системного журнала `syslogd(lM)`, который в свою очередь либо дописывает сообщения в системный журнал, либо выводит на их консоль, либо перенаправляет в соответствии со списком пользователей данной или удаленной системы. Конкретный пункт назначения определяется конфигурационным файлом `/etc/syslog.conf`. Функция имеет определение:

```
#include <syslog.h>
void syslog(int priority, char *logstring, /* параметры*/...);
```

Каждому сообщению `logstring` назначается приоритет, указанный параметром `priority`. Возможные значения этого параметра приведены в таблице 1.

Таблица 1. Возможные приоритеты записей системного лога

Значение	Описание
`LOG_EMERG`	Идентифицирует состояние "паники" в системе. Обычно рассылается всем пользователям.
`LOG_ALERT`	Идентифицирует ненормальное состояние, которое должно быть исправлено немедленно, например, нарушение целостности системной базы данных.
`LOG_CRIT`	Идентифицирует критическое событие, например, ошибку дискового устройства.
`LOG_ERR`	Идентифицирует различные ошибки.
`LOG_WARNING`	Идентифицирует предупреждения
`LOG_NOTICE`	Идентифицирует события, которые не являются ошибками, но требуют внимания.
`LOG_INFO`	Идентифицирует информационные сообщения.
`LOG_DEBUG`	Идентифицирует сообщение, обычно используемое только при отладке программы.

Практическое задание

Введение: что есть демон?

Демон (или служба) является фоновым процессом, который разработан специально для автономной работы, с минимальным вмешательством пользователя или вообще без него. HTTP-демон (httpd) веб-сервера Apache является одним из примеров таких процессов. Он работает в фоновом режиме, слушая специфичные порты и обслуживая страницы или выполняя скрипты, в зависимости от вида запроса.

Создание демона в UNIX включает в себя соблюдение набора специфических правил в определенном порядке. Знание того как они работают поможет вам понять как демоны работают в пространстве пользователя UNIX, но при этом также могут использовать вызовы ядра. На самом деле, совсем немного демонов взаимодействует с модулями ядра, которые работают с аппаратными устройствами, такими как платы внешних контроллеров, принтеров и КПК. Они являются одним из фундаментальных строительных блоков в UNIX и дают невероятные гибкость и производительность системе в целом.

В процессе повествования будет разработан очень простой демон на языке C. По мере продвижения по документу будет добавляться все больше кода, показывающего правильный порядок разработки, необходимый для запуска и работы демона.

Планирование вашего демона

Постановка задачи

Демон должен делать только одну вещь и делать ее хорошо. Эта одна вещь может быть такой же сложной, как управление сотнями почтовых ящиков во множестве доменов или такой же простой как формирование отчета и вызов `sendmail` для оправки этого отчета администратору.

В любом случае вы должны хорошо представлять себе то, что должен делать ваш демон. Если планируется взаимодействие с другими демонами, разрабатываемыми и вами, и не вами, то это тоже необходимо учитывать при планировании.

Насколько интерактивные?

Демоны не должны общаться с пользователем напрямую через терминал. На самом деле, демон вообще не должен напрямую общаться с пользователем. Все общение должно производиться через определенный интерфейс (который может позволять, а может и не позволять запись), который может быть сложным как GTK+ GUI или простым как набор сигналов.

Базовая структура демона

Когда демон запускается, он выполняет некоторую низкоуровневую работу для подготовки себя к основной работе. Первая включает в себя несколько шагов:

- Отделение (ответвление, `fork`) от родительского процесса.
- Изменение файловой маски (`umask`).
- Открытие любых журналов на запись.
- Создание уникального ID сессии (SID).
- Изменение текущего рабочего каталога на безопасное место.
- Закрытие стандартных файловых дескрипторов.
- Переход к коду собственно демона.

Отделение от родительского процесса

Демон запускается либо самой системой, либо пользователем в терминале или скрипте. Во время запуска его процесс ничем не отличается от любого другого процесса в системе. Чтобы сделать его по-настоящему автономным, нужно создать дочерний процесс, в котором будет выполняться код демона. Для этого используется функция `fork()`:

```
pid_t pid;
/* отделяемся от родительского процесса */
pid = fork();
if (pid < 0) {
    exit(EXIT_FAILURE);
}
/* Если с PID'ом все получилось, то родительский процесс можно завершить.*/
if (pid > 0) {
    exit(EXIT_SUCCESS);
}
```

Отметим здесь проверку успешного завершения вызова `fork()`. При разработке демона необходимо делать код максимально стабильным. На самом деле, большую часть всего кода демона составляют именно проверки на ошибки.

Функция `fork()` возвращает либо id дочернего процесса (PID, не равный нулю), либо -1 в случае ошибки. Если процесс не может породить потомка, то демон должен завершиться прямо здесь.

Если получение PID от fork() совершилось успешно, то родительский процесс должен изящно завершиться. Это может показаться странным для всех, кто такого еще не видел, но после ответвления дочерний процесс продолжает выполнение остального кода с этого места.

Изменение файловой маски (Umask)

Чтобы иметь возможность писать в любые файлы (включая журналы), созданные демоном, файловая маска (`umask`) должна быть изменена так, чтобы они могли быть записаны или прочитаны правильным образом. Это похоже на выполнение `umask` из командной строки, но мы прагматично делаем это здесь при помощи функции `umask()`:

```
pid_t pid, sid;

/* Ответвляемся от родительского процесса */

pid = fork();

if (pid < 0) {

    /* Фиксируем ошибку (через syslog при возможности) */

    exit(EXIT_FAILURE);

}

/* Если с PID'ом все получилось, то родительский процесс можно завершить. */

if (pid > 0) {

    exit(EXIT_SUCCESS);

}

/* Изменяем файловую маску */

umask(0);
```

Через установку `umask` в 0 мы получим полный доступ к файлам, созданным демоном. Даже если вы не планируете использовать какие-либо файлы вообще, установка `umask` остается хорошей идеей просто на случай доступа к файлам на файловой системе.

Открытие журналов на запись

Эта действие опционально, но все-таки рекомендуется открыть где-нибудь в системе файл журнала на запись. Это можно сделать даже только для того, чтобы имелась возможность посмотреть на отладочную информацию от демона.

Создание уникального ID сессии (SID)

С этого места для нормальной работы дочерний процесс должен получить уникальный SID от ядра. Иначе дочерний процесс станет сиротой. Тип `pid_t`, объявленный в предыдущем разделе, также используется для создания нового SID для дочернего процесса:

```
pid_t pid, sid;

/* Ответвляемся от родительского процесса */

pid = fork();

if (pid < 0) {

    exit(EXIT_FAILURE);

}

/* Если с PID'ом все получилось, то родительский процесс можно завершить. */

if (pid > 0) {

    exit(EXIT_SUCCESS);

}

/* Изменяем файловую маску */

umask(0);

/* Здесь можно открывать любые журналы */

/* Создание нового SID для дочернего процесса */

sid = setsid();

if (sid < 0) {

    /* Журналируем любой сбой */
```

```
    exit(EXIT_FAILURE);
}
```

Как видим, функция `setsid()` возвращает данные того же типа что и `fork()`. Чтобы проверить что функция создала SID для дочернего процесса мы можем использовать аналогичную процедуру проверки на ошибки.

Изменение рабочего каталога

Текущий рабочий каталог нужно сменить на некоторое место, гарантированно присутствующее в системе. Поскольку многие дистрибутивы UNIX не полностью следуют стандарту иерархии файловой системы UNIX (FHS, Filesystem Hierarchy Standard), то в системе гарантированно присутствует только корень файловой системы (/). Сменить каталог можно при помощи функции `chdir()`:

```
pid_t pid, sid;
/* Ответвляемся от родительского процесса */
pid = fork();
if (pid < 0) {
    exit(EXIT_FAILURE);
}
/* Если с PID'ом все получилось, то родительский процесс можно завершить. */
if (pid > 0) {
    exit(EXIT_SUCCESS);
}
/* Изменяем файловую маску */
umask(0);
/* Здесь можно открывать любые журналы */
/* Создание нового SID для дочернего процесса */
sid = setsid();
if (sid < 0) {
    /* Журналируем любой сбой */
    exit(EXIT_FAILURE);
}
/* Изменяем текущий рабочий каталог */
if ((chdir("/")) < 0) {
    /* Журналируем любой сбой */
    exit(EXIT_FAILURE);
}
```

И снова мы видим здесь код обработки ошибок. Функция `chdir()` при ошибке возвращает -1, так что не забывайте проверять возвращаемое ею значение после смены каталога.

Закрытие стандартных файловых дескрипторов

Одним из последних шагов в стартовой настройке демона является закрытие стандартных файловых дескрипторов (`STDIN`, `STDOUT`, `STDERR`). Поскольку демон не может использовать терминал, эти файловые дескрипторы излишни и создают угрозу безопасности. Закрыть их можно при помощи функции `close()`:

```
pid_t pid, sid;
/* Ответвляемся от родительского процесса */
pid = fork();
if (pid < 0) {
    exit(EXIT_FAILURE);
```

```
}
/* Если с PID'ом все получилось, то родительский процесс можно завершить. */
if (pid > 0) {
    exit(EXIT_SUCCESS);
}
/* Изменяем файловую маску */
umask(0);
/* Здесь можно открывать любые журналы */
/* Создание нового SID для дочернего процесса */
sid = setsid();
if (sid < 0) {
    /* Журналируем любой сбой */
    exit(EXIT_FAILURE);
}
/* Изменяем текущий рабочий каталог */
if ((chdir("/")) < 0) {
    /* Журналируем любой сбой */
    exit(EXIT_FAILURE);
}
/* Закрываем стандартные файловые дескрипторы */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
```

Использование определенных для файловых дескрипторов констант является хорошей идеей, поскольку улучшает переносимость.

Разработка кода демона

Инициализация

На данном этапе вы уже в основном сообщили UNIX о том что вы демон, так что сейчас самое время написать код самого демона. Инициализация является первым шагом на этом пути. Поскольку возможно существование множества разных функций, которые могут быть вызваны здесь для настройки вашего демона, я не буду углубляться в этот вопрос.

Важный момент здесь в том, что при инициализации чего-нибудь в демоне необходимо применять те же методы обнаружения и обработки ошибок. Будьте болтливы ("verbose") насколько это возможно при записи в syslog или в ваши собственные журналы. Отладка демона может затрудниться при недостатке информации о его состоянии.

Большой Цикл

Основной код демона обычно находится внутри бесконечного цикла. Технически это не бесконечный цикл конечно, но он организован как бесконечный:

```
pid_t pid, sid;
/* Ответвляемся от родительского процесса */
pid = fork();
if (pid < 0) {
    exit(EXIT_FAILURE);
}
/* Если с PID'ом все получилось, то родительский процесс можно завершить. */
```

```
if (pid > 0) {
 exit(EXIT_SUCCESS);
}
/* Изменяем файловую маску */
umask(0);
/* Здесь можно открывать любые журналы */
/* Создание нового SID для дочернего процесса */
sid = setsid();
if (sid < 0) {
    /* Журналируем любой сбой */
    exit(EXIT_FAILURE);
}
/* Изменяем текущий рабочий каталог */
if ((chdir("/")) < 0) {
    /* Журналируем любой сбой */
    exit(EXIT_FAILURE);
}
/* Закрываем стандартные файловые дескрипторы */
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
/* Специфичная для демона инициализация проходит тут */
/* Большой Цикл */
while (1) {
    /* Делаем тут чего-нибудь ... */
    sleep(30); /* ждем 30 секунд */
}
```

Типичный цикл обычно является циклом `while`, который имеет бесконечное условие завершения с вызовом `sleep` при необходимости выполнения через фиксированные интервалы.

Представьте себе это стуком сердца: когда ваше сердце сжимается, оно выполняет несколько задач, затем ожидает следующего сжатия. Многие демоны следуют этой методологии.

Собираем все вместе

Законченный пример

Приведенное ниже является законченным примером демона и иллюстрирует все шаги, необходимые для запуска и работы. Для его выполнения просто скимпилируйте при помощи KDeveloper и запустите на выполнение из командной строки. Для завершения выясните его PID и воспользуйтесь командой `kill`.

В примере задействованы необходимые заголовочные файлы для взаимодействия с syslog'ом, использование которого рекомендуется как минимум для записи в журнал информации о запуске/останове/паузе/завершении, в дополнение к использованию ваших собственных журналов через вызовы функций `fopen()/fwrite()/fclose()`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
```

```
#include <errno.h>
#include <unistd.h>
#include <syslog.h>
#include <string.h>

int main(void) {
/* Наши ID процесса и сессии */
pid_t pid, sid;
/* Ответвляемся от родительского процесса */
pid = fork();
if (pid < 0) {
   exit(EXIT_FAILURE);
}
/* Если с PID'ом все получилось, то родительский процесс можно завершить. */
    if (pid > 0) {
    exit(EXIT_SUCCESS);
}
/* Изменяем файловую маску */
umask(0);
/* Здесь можно открывать любые журналы */
/* Создание нового SID для дочернего процесса */
sid = setsid();
if (sid < 0) {
    /* Журналируем любой сбой */
    exit(EXIT_FAILURE);
}
/* Изменяем текущий рабочий каталог */
if ((chdir("/")) < 0) {
    /* Журналируем любой сбой */
    exit(EXIT_FAILURE);
}
/* Закрываем стандартные файловые дескрипторы */
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);
/* Специфичная для демона инициализация проходит тут */
/* Большой Цикл */
while (1) {
    /* Делаем здесь чего-нибудь ... */
    sleep(30); /* ждем 30 секунд */
}
exit(EXIT_SUCCESS);
}
```

Необходимо использовать этот скелет для разработки демонов. Не забывайте вести журналы событий (или используйте возможности `syslog`).

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

3. Робачевский А.М. Операционная система UNIX. –СПб.: БХВ-Петербург, 2002. -528 с.

4. Рочкинд М. Программирование для UNIX. 2-е изд. перераб. и доп. –Пер. с англ. –М.: Издательско-торговый дом «Русская редакция»; СПб.: БХВ-Петербург, 2005. -704 с.

5. Формирование документации к исходному коду с помощью средства doxygen. URL: www.nrjetix.com/r-and-d/lectures