

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Международный институт дистанционного образования

Кафедра «Информационные системы и технологии»

КУРСОВОЙ ПРОЕКТ

по учебной дисциплине

«ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ»

Исполнитель: студент 3 курса,
группы 41703120 Реут В.Л.

Руководитель: Русак Л. В.

Минск 2023

ОГЛАВЛЕНИЕ

| | |
|-------------------------------------|-----------|
| ВВЕДЕНИЕ..... | 3 |
| Техническое задание..... | 4 |
| Вариант задания | 4 |
| Теоретическая часть..... | 4 |
| Практическая часть..... | 11 |
| Краткое описание программы | 11 |
| Исходный код программы..... | 23 |
| Компиляция и запуск программы | 34 |
| ЗАКЛЮЧЕНИЕ | 35 |
| ЛИТЕРАТУРА | 36 |

ВВЕДЕНИЕ

Язык программирования C++ представляет высокоуровневый компилируемый язык программирования общего назначения со статической типизацией, который подходит для создания самых различных приложений. На сегодняшний день C++ является одним из самых популярных и распространенных языков.

C++ является мощным языком, унаследовав от Си богатые возможности по работе с памятью. Поэтому нередко C++ находит свое применение в системном программировании, в частности, при создании операционных систем, драйверов, различных утилит, антивирусов и т.д. К слову сказать, ОС Windows большей частью написана на C++

C++ является компилируемым языком, а это значит, что компилятор транслирует исходный код на C++ в исполняемый файл, который содержит набор машинных инструкций. Но разные платформы имеют свои особенности, поэтому скомпилированные программы нельзя просто перенести с одной платформы на другую и там уже запустить. Однако на уровне исходного кода программы на C++ по большей степени обладают переносимостью, если не используются какие-то специфичные для текущей ОС функции. А наличие компиляторов, библиотек и инструментов разработки почти под все распространенные платформы позволяет компилировать один и тот же исходный код на C++ в приложения под эти платформы.

В 1979-80 годах Бьерн Страуструп разработал расширение к языку Си - "Си с классами". В 1983 язык был переименован в C++.

В 1985 году была выпущена первая коммерческая версия языка C++, а также первое издание книги "Языка программирования C++", которая представляла первое описание этого языка при отсутствии официального стандарта.

В 1989 была выпущена новая версия языка C++ 2.0, которая включала ряд новых возможностей. После этого язык развивался относительно медленно вплоть до 2011 года. Но при этом в 1998 году была предпринята первая попытка по стандартизации языка организацией ISO (International Organization for Standardization). Первый стандарт получил название ISO/IEC 14882:1998 или сокращенно C++98. В дальнейшем в 2003 была издана новая версия стандарта C++03.

Техническое задание.

1. Цель работы: получить навыки создания и журналирования работы программ-демонов на языке C в операционных системах семейства Linux.

2. Краткие теоретические сведения: см. дополнительный материал к курсовой работе.

3. Методические указания.

3.1. Проект может быть реализован в виде консольного приложения в среде ОС Ubuntu, Fedora, CentOS или других средствами компилятора gcc версии не ниже 4.

3.2. Проект является модификацией программ связанных с формированием серверных реализаций (при условии, что проект выполнялся с применением каналов FIFO).

3.3. Проект должен предусматривать обработку исключительных ситуаций (отсутствие или неверное количество входных параметров, ошибки открытия входного и/или выходного файла, ошибки чтения и записи).

3.4. Проект рекомендуется реализовать в 2 этапа.

3.4.1. Преобразование операций вывода (включая сообщения об ошибках) в серверном приложении в операции с журналом **syslogd**.

3.4.2. Преобразование серверного приложения в демона.

4. Порядок выполнения работы.

4.1. Написать серверное приложение, преобразуя операции вывода в консоль в операции с журналом **syslogd**.

4.2. Модифицировать серверное приложение в демона, продемонстрировать работоспособность программы преподавателю.

4.3. Результатом выполнения работы считается демонстрация работы клиентского приложения с демоном и журнала работы демона.

Вариант задания

| | | |
|---|-------------------------------|--|
| 5 | Заменить каждый пробел на два | 1. Имя входного файла 2. Количество замен |
|---|-------------------------------|--|

Теоретическая часть

Системные службы (демоны) в Linux

Демон (англ. *daemon*) – это процесс, обладающий следующим свойством.

- Имеет длинный жизненный цикл. Часто демоны создаются во время загрузки системы и работают до момента ее выключения.

- Выполняется в фоновом режиме и не имеет контролирующего терминала.

Последняя особенность гарантирует, что ядро не сможет генерировать для такого процесса никаких сигналов, связанных с терминалом или управлением заданиями (таких, как **SIGINT**, **SIGHUP**).

Демоны создаются для выполнения специфических задач. Например:

- **cron** – демон, который выполняет команды в запланированное время;
- **sshd** – демон защищенной командной оболочки, который позволяет входить в систему с удаленных компьютеров, используя безопасный протокол;
- **httpd** – демон HTTP-сервера (Apache), который обслуживает веб-страницы;

Многие стандартные демоны работают в качестве привилегированных процессов (то есть их действующий пользовательский идентификатор равен 0), поэтому при их написании следует руководствоваться рекомендациями по написанию безопасных программ с повышенными привилегиями.

Создание демона

Для того чтобы стать демоном, программа должна выполнить следующие шаги.

1. Сделать вызов **fork()**, после которого родитель завершается, а потомок продолжает работать (в результате этого демон становится потомком процесса **init**). Этот шаг делается по двум следующим причинам.

- Исходя из того, что демон был запущен в командной строке, за вершение родителя будет обнаружено командной оболочкой, которая вслед за этим выведет новое приглашение и позволит потомку выполняться в фоновом режиме.

- Потомок гарантированно не станет лидером группы процессов, поскольку он наследует идентификатор группы программ **PGID** от своего родителя и получает свой уникальный идентификатор, который отличается от унаследованного **PGID**. Это необходимо для успешного выполнения следующего шага.

2. Дочерний процесс вызывает **setsid()**, чтобы начать новую сессию и разорвать любые связи с контролирующим терминалом.

Контролирующий терминал – это тот, который устанавливается при первом открытии устройства терминала лидером сессии. Любой контролирующий терминал может быть связан не более чем с одной сессией. *Сессия* – это набор групп процессов. Членство процесса в сессии определяется идентификатором **SID** (session identifier – *идентификатор сессии*), который по аналогии с **PGID** (process group identifier – *идентификатор группы процессов*) является числом типа **pid_t**. *Лидером сессии* является процесс, который ее создал и чей идентификатор используется в качестве **SID**. Новый процесс наследует идентификатор **SID** своего родителя. Этот вызов возвращает идентификатор новой сессии или **-1**, если случилась ошибка:

```
#include <unistd.h>
pid_t setsid(void);
```

Создание новой сессии системным вызовом **setsid()** происходит следующим образом:

- вызывающий процесс становится лидером новой сессии и новой группы процессов внутри нее. Идентификаторы **PGID** и **SID** нового процесса получают то же значение, что и сам процесс;

- вызывающий процесс не имеет контролирующего терминала.

Любое соединение с контролирующим терминалом, установленное ранее, разрывается.

3. Если после этого демон больше не открывает никаких терминальных устройств, мы можем не волноваться о том, что он восстановит соединение с контролирующим терминалом. В противном случае нам необходимо сделать так, чтобы терминальное устройство не стало контролирующим. Это можно сделать двумя нижеописанными способами.

- указывать флаг **O_NOCTTY** для любых вызовов **open()**, которые могут открыть терминальное устройство.

- более простой вариант: после **setsid()** можно еще раз сделать вызов **fork()**, опять позволив родителю завершиться, а потомку (правнуку) – продолжить работу. Это гарантирует, что потомок не станет лидером сессии, что делает невозможным повторное соединение с контролирующим терминалом (это соответствует процедуре получения контролирующего терминала, принятой в System V).

4. Очистить атрибут **umask** процесса, чтобы файлы и каталоги, созданные демоном, имели запрашиваемые права доступа.

5. Поменять текущий рабочий каталог процесса (обычно на корневой – /). Это необходимо, поскольку демон обычно выполняется вплоть до выключения системы. Если файловая система, на которой находится его текущий рабочий каталог, не является корневой, она не может быть отключена. Как вариант, в качестве рабочего каталога демон может задействовать то место, где он выполняет свою работу, или воспользоваться значением в конфигурационном файле; главное, чтобы файловая система, в которой находится этот каталог, никогда не нуждалась в отключении. Например, **cron** применяет для этого

/var/spool/cron.

6. Закрывать все открытые файловые дескрипторы, которые демон унаследовал от своего родителя (возможно, некоторые из них необходимо оставить открытыми, поэтому данный шаг является необязательным и может быть откорректирован). Это делается по целому ряду причин. Поскольку демон потерял свой контролирующий терминал и работает в фоновом режиме, ему больше не нужно хранить дескрипторы с номерами 0, 1 и 2 (они ссылаются на терминал). Кроме того, мы не можем отключить файловую систему, на которой долгоживущий демон удерживает открытыми какие-либо файлы. И,

следуя обычным правилам, мы должны закрывать неиспользуемые файловые дескрипторы, поскольку их число ограничено.

Запись в журнал сообщений и ошибок с помощью системы **syslog**

При написании демона одной из проблем является вывод сообщений об ошибках. Поскольку демон выполняется в фоновом режиме, он не может выводить информацию в терминале, как это делают другие программы. В качестве альтернативы сообщения можно записывать в отдельный журнальный файл программы.

Для записи сообщений в журнал любой процесс может воспользоваться библиотечной функцией **syslog()**. На основе переданных ей аргументов она создает сообщение стандартного вида и помещает его в сокет **/dev/log**, где оно будет доступно для **syslogd**.

Программный интерфейс **syslog** состоит из трех основных функций.

1. Функция **openlog()** устанавливает настройки, которые по умолчанию применяются ко всем последующим вызовам **syslog()**.

Она не является обязательной. Если ею не воспользоваться, соединение с системой ведения журнала устанавливается при первом вызове **syslog()** на основе стандартных настроек.

2. Функция **syslog()** записывает сообщения в журнал.

3. Функция **closelog()** вызывается после окончания записи сообщений, чтобы разорвать соединение с журналом.

Ни одна из этих функций не возвращает значение статуса. Частично это продиктовано тем, что системное журналирование должно быть всегда доступным (если оно перестанет работать, системный администратор должен быстро это заметить). Кроме того, если при ведении журнала произошла ошибка, приложение обычно мало что может сделать, чтобы об этом сообщить.

Функция **closelog()** закрывает описатель, используемый для записи данных в журнал. Использование **closelog()** необязательно.

Функция **openlog()** при необходимости устанавливает соединение с системным средством ведения журнала и задает настройки, которые будут применяться по умолчанию ко всем последующим вызовам **syslog()**.

```
#include <syslog.h>
void openlog (const char *ident, int log_options,
int facility);
```

Аргумент **ident** является указателем на строку, которая добавляется в каждое сообщение, записываемое с помощью **syslog()**; обычно это название программы. Стоит отметить, что **openlog()** всего лишь копирует значение этого указателя. Продолжая использовать вызовы **syslog()**, приложение должно следить за тем, чтобы строка, на которую ссылается данный аргумент, не изменилась.

Если в качестве аргумента **ident** указать **NULL**, интерфейс **syslog** из состава **glibc**, как и некоторые другие реализации, будет автоматически подставлять вместо него название программы. Однако такое поведение не предусмотрено стандартом SUSv3 и не выполняется в некоторых системах, поэтому переносимые приложения не должны на него полагаться.

Аргумент **log_options** для вызова **openlog()** представляет собой битовую маску, состоящую из любых комбинаций следующих констант, к которым применяется побитовое ИЛИ.

- **LOG_CONS** – если в системный журнал приходит ошибка, она записывается в системную консоль (**/dev/console**).

- **LOG_NDELAY** – соединение с системой ведения журнала (то есть с сокетом домена UNIX, **/dev/log**) устанавливается немедленно. По умолчанию (**LOG_ODELAY**) это происходит, только когда (и если) первое сообщение попадает в журнал с помощью вызова **syslog()**. Флаг **LOG_NDELAY** может пригодиться в программах, которым нужно контролировать момент выделения файлового дескриптора для **/dev/log**. Например, это может быть приложение, которое вызывает **chroot()**; после этого вызова путь **/dev/log** перестает быть доступным, поэтому, если вы вызываете функцию **openlog()** с флагом **LOG_NDELAY**, это нужно делать до **chroot()**. Примером программы, которая использует флаг **LOG_NDELAY** таким образом, может служить демон **tftpd** (*Trivial File Transfer*).

- **LOG_NOWAIT** – вызов **syslog()** не ждет дочерний процесс, который мог быть создан для записи сообщения в журнал. Этот флаг нужен в приложениях, в которых для записи сообщений используются отдельные дочерние процессы. Он позволяет вызову **syslog()** избежать ожидания потомков, уже утилизированных родителем, который тоже их ожидал. В Linux флаг **LOG_NOWAIT** ни на что не влияет, так как в этой системе при записи сообщений в журнал дочерние процессы не создаются.

- **LOG_ODELAY** – противоположность флагу **LOG_NDELAY**. Соединение с системой ведения журнала откладывается до тех пор, пока не будет записано первое сообщение. Этот флаг используется по умолчанию, и его не нужно указывать отдельно.

- **LOG_PID** – включать PID в каждое сообщение.

Аргумент **facility** устанавливает значение по умолчанию, если не указываются соответствующие параметры при вызовах **syslog()**.

Аргумент **facility** используется для указания типа программы, записывающей сообщения. Это позволяет файлу конфигурации указывать, что сообщения от различных программ будут по-разному обрабатываться.

- **LOG_AUTH** – сообщения о безопасности/авторизации (рекомендуется использовать вместо него **LOG_AUTHPRIV**);

- **LOG_AUTHPRIV** – сообщения о безопасности/авторизации (частные);

- **LOG_CRON** – демон часов (**cron** и **at**);

- **LOG_DAEMON** – другие системные демоны;

- **LOG_KERN** – сообщения ядра;
- **LOG_LOCAL0** до **LOG_LOCAL7** – зарезервированы для определения пользователем;
- **LOG_LPR** – подсистема принтера;
- **LOG_MAIL** – почтовая подсистема;
- **LOG_NEWS** – подсистема новостей USENET;
- **LOG_SYSLOG** – сообщения, генерируемые **syslogd**;
- **LOG_USER** (по умолчанию) – общие сообщения на уровне пользователя;
- **LOG_UUCP** – подсистема UUCP.

Функция **syslog()** изначально разрабатывалась для BSD, в настоящее время она предоставляется большинством производителей систем UNIX.

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

syslog() создает сообщение для журнала, которое передается демону **syslogd**. **priority** получается при логическом сложении **facility**, описанном выше, и **level**, описанном ниже. Аргументы **format** такие же, как и в **printf()**, кроме того, что сочетание **%m** будет заменено сообщением об ошибке **strerror(errno)** и будет добавлен завершающий символ новой строки.

Параметр **level** определяет степень важности сообщения. Далее значения приводятся по понижению степени их важности:

- **LOG_EMERG** – система остановлена;
- **LOG_ALERT** – требуется немедленное вмешательство;
- **LOG_CRIT** – критические условия;
- **LOG_ERR** – ошибки;
- **LOG_WARNING** – предупреждения;
- **LOG_NOTICE** – важные рабочие условия;
- **LOG_INFO** – информационные сообщения;
- **LOG_DEBUG** – сообщения об отладке.

Функция **setlogmask()** может использоваться для ограничения доступа на указанные уровни.

Назначение аргументов **facility** и **level** в том, чтобы все сообщения, которые посылаются процессами определенного типа (то есть с одним значением аргумента **facility**), могли обрабатываться одинаково в файле **/etc/syslog.conf** или чтобы все сообщения одного уровня (с одинаковым значением аргумента **level**) обрабатывались одинаково. Например, демон может сделать следующий вызов, когда вызов функции **rename** неожиданно оказывается неудачным:

```
syslog(LOG_INFO/LOG_LOCAL2, "rename(%s, %s): %m",
file1, file2);
```

Конфигурационный файл **/etc/syslog.conf** определяет поведение демона **syslogd**. Он состоит из правил и комментариев (последние начинаются с символа #). Правила в общем случае имеют следующий вид:

категория.приоритет действие

Сочетание *категории* и *приоритета* называют *селектором*, поскольку они позволяют выбрать сообщения, к которым применяется правило. Виды категорий (**facility**) и приоритетов (**level**) описаны выше, но в файле конфигурации применяются без префикса **LOG_**.

Под *действием* подразумевается место назначения сообщений, которые соответствуют *селектору*. *Селектор* и *действие* разделены пробельными символами. Ниже показан пример нескольких правил:

```
*.err /dev/tty10  
auth.notice root  
*.debug;mail.none;news.none -/var/log/messages
```

Согласно первому правилу сообщения всех категорий (*) с приоритетом **err** (**LOG_ERR**) или выше должны передаваться консольному устройству **/dev/tty10**. Второе правило делает так, что сообщения, связанные с авторизацией (**LOG_AUTH**) и имеющие приоритет **notice** (**LOG_NOTICE**) или выше, должны отправляться во все консоли или терминалы, в которых работает пользователь **root**. Это, например, позволит администратору немедленно получать все сообщения о неудачных попытках повышения привилегий (вызове команды **su**).

В последней строчке демонстрируются некоторые продвинутые аспекты синтаксиса для описания правил. В ней перечислено сразу несколько селекторов, разделенных точкой с запятой. Первый селектор относится к сообщениям любой категории (*) с приоритетом **debug** (самым низким) и выше, т. е. это затрагивает все сообщения. В Linux, как и в большинстве других UNIX-систем, вместо **debug** можно указать символ *, который будет иметь то же значение, однако данная возможность поддерживается не всеми реализациями **syslog**. Если правило содержит несколько селекторов, оно обычно охватывает сообщения, соответствующие любому из них. Но если в качестве приоритета указать значение **none**, то сообщения, принадлежащие к заданной категории, будут отбрасываться. Таким образом, это правило передает все сообщения (кроме тех, которые имеют категории **mail** и **news**) в файл **/var/log/messages**. Символ «тильда» (~) перед именем этого файла говорит о том, что сбрасывание данных на диск будет происходить не при каждой передаче сообщения. Это приводит к увеличению скорости записи, но в случае сбоя системы сообщения, пришедшие недавно, могут быть утеряны.

При каждом изменении файла **syslog.conf** демону следует отправлять сигнал, чтобы он смог заново себя инициализировать:

```
$killall -HUP syslogd //Отправляем сигнал SIGHUP  
демону syslogd
```

Синтаксис файла syslog.conf позволяет создавать куда более сложные правила, чем те, что были показаны.

Практическая часть

Краткое описание программы

Далее мы рассмотрим приложение состоящее из нескольких файлов, содержимое которых представляет собой в основном функции-обертки для работы с сетевыми протоколами.

Ниже представлен хедер со всеми необходимыми объявлениями (Рис.2), а так же прототипами функций (Рис.1).

server_daemon_header.h:

```
19 #define MAXFD 64
20 #define MAXLINE 4096 /* max text line length */
21 #define LISTENQ 1024 /* 2nd argument to listen() */
22 #define SA struct sockaddr
23
24 typedef void Sigfunc(int); /* for signal handlers */
25
26 void err_dump(const char *, ...);
27 void err_msg(const char *, ...);
28 void err_quit(const char *, ...);
29 void err_ret(const char *, ...);
30 void err_sys(const char *, ...);
31
32 void Setsockopt(int, int, int, const void *, socklen_t);
33 void Listen(int, int);
34 void Close(int);
35 int Accept(int, SA *, socklen_t *);
36
37 pid_t Fork(void);
38 void *Malloc(size_t);
39
40 void Write(int, void *, size_t);
41
42 int tcp_listen(const char *, const char *, socklen_t *);
43 int Tcp_listen(const char *, const char *, socklen_t *);
44
45 char *sock_ntop(const SA *, socklen_t);
46 char *Sock_ntop(const SA *, socklen_t);
47
48 int daemon_init(const char *, int);
49
50 Sigfunc *Signal(int, Sigfunc *);
```

Рис.1 Прототипы функций.

```

1  #pragma once
2  #include    <iostream>
3  #include    <fstream>
4  #include    <string>
5  #include    <cstdlib>
6  #include    <sstream>
7  #include    <sys/types.h>    /* basic system data types */
8  #include    <sys/socket.h>  /* basic socket definitions */
9  #include    <syslog.h>
10 #include    <netinet/in.h>  /* sockaddr_in{} and other Internet defns */
11 #include    <arpa/inet.h>   /* inet(3) functions */
12 #include    <stdarg.h>     /* ANSI C header file */
13 #include    <errno.h>
14 #include    <fcntl.h>      /* for nonblocking */
15 #include    <netdb.h>
16 #include    <signal.h>
17 #include    <sys/un.h>     /* for Unix domain sockets */

```

Рис.2 Подключение необходимых библиотек.

Далее представлен файл `main` в котором содержится функция содержащая в себе всю логику технического задания (Рис.3) а так же основная функция **main** (Рис.4).

main.cpp:

```

3  std::string spaces_message(char *filen, char *countn)
4  {
5      std::ifstream file(filen);
6      int spaces = 0;
7      int count = std::stoi(countn);
8      std::string line;
9      std::string resLine;
10
11     if (!file.is_open())
12     {
13         err_quit("File not open!");
14     }
15
16     std::getline(file, line);
17
18     for (int i = 0; i <= line.length(); ++i)
19     {
20         resLine.push_back(line[i]);
21         if (line[i] == ' ' && spaces < count)
22         {
23             resLine.push_back(' ');
24             spaces += 1;
25         }
26     }
27     return resLine;
28 }

```

Рис.3 Функция читает с файла и ставит доп. пробел.

```

31 main(int argc, char **argv)
32 {
33     int listenfd, connfd;
34     socklen_t addrlen, len;
35     struct sockaddr *cliaddr;
36     std::string resLine;
37
38     if (argc < 4 || argc > 5)
39         err_quit("usage: course_project_daemon [ <host> ] <service or port> <file> <count spaces>");
40
41     resLine.append(spaces_message(argv[3], argv[4]));
42
43     daemon_init(argv[0], 0);
44
45     if (argc == 2)
46         listenfd = Tcp_listen(NULL, argv[1], &addrlen);
47     else
48         listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
49
50     cliaddr = static_cast<sockaddr*>(Malloc(addrlen));
51
52     for ( ; ; ) {
53         len = addrlen;
54         connfd = Accept(listenfd, cliaddr, &len);
55         err_msg("connection from %s", Sock_ntop(cliaddr, len));
56         err_msg(resLine.c_str(), Sock_ntop(cliaddr, len));
57
58         Close(connfd);
59     }
60 }

```

Рис.4 Функция *main*.

Ниже файл **tcp_listen** в котором реализована одноименная функция делающая наш сервер независимый от протокола и возвращает прослушиваемый сокет (Рис.5), а так же функцию-обертку данной функции (Рис.6).

tcp_listen.cpp:

```

3  int
4  tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
5  {
6      int          listenfd, n;
7      const int     on = 1;
8      struct addrinfo hints, *res, *ressave;
9
10     bzero(&hints, sizeof(struct addrinfo));
11     hints.ai_flags = AI_PASSIVE;
12     hints.ai_family = AF_UNSPEC;
13     hints.ai_socktype = SOCK_STREAM;
14
15     if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
16         err_quit("tcp_listen error for %s, %s: %s",
17                 host, serv, gai_strerror(n));
18     ressave = res;
19
20     do {
21         listenfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
22         if (listenfd < 0)
23             continue;          /* error, try next one */
24
25         Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
26         if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
27             break;             /* success */
28
29         Close(listenfd);        /* bind error, close and try next one */
30     } while ( (res = res->ai_next) != NULL);
31
32     if (res == NULL)            /* errno from final socket() or bind() */
33         err_sys("tcp_listen error for %s, %s", host, serv);
34
35     Listen(listenfd, LISTENQ);
36
37     if (addrlenp)
38         *addrlenp = res->ai_addrlen;    /* return size of protocol address */
39
40     freeaddrinfo(ressave);
41
42     return(listenfd);
43 }

```

Рис.5 Функция *tcp_listen*.

```

52  int
53  Tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
54  {
55      return(tcp_listen(host, serv, addrlenp));
56  }

```

Рис.6 Функция-обертка для функции *tcp_listen*.

Далее следует рассмотреть файл **sock_wrap** содержащий функции-обертки для функций **accept**, **setsockopt** (Рис.7), **close** и **listen** (Рис.8).

sock_wrap.cpp:

```

3  int
4  Accept(int fd, struct sockaddr *sa, socklen_t *salenptr)
5  {
6      int      n;
7
8      again:
9      if ( (n = accept(fd, sa, salenptr)) < 0) {
10 #ifdef  EPROTO
11         if (errno == EPROTO || errno == ECONNABORTED)
12 #else
13         if (errno == ECONNABORTED)
14 #endif
15             goto again;
16         else
17             err_sys("accept error");
18     }
19     return(n);
20 }
21
22 void
23 Setsockopt(int fd, int level, int optname, const void *optval, socklen_t optlen)
24 {
25     if (setsockopt(fd, level, optname, optval, optlen) < 0)
26         err_sys("setsockopt error");
27 }

```

Рис.7 Функции-обертки для функций *accept* и *setsockopt*.

```

29 void
30 Close(int fd)
31 {
32     if (close(fd) == -1)
33         err_sys("close error");
34 }
35
36 void
37 Listen(int fd, int backlog)
38 {
39     char      *ptr;
40
41     /*4can override 2nd argument with environment variable */
42     if ( (ptr = getenv("LISTENQ")) != NULL)
43         backlog = atoi(ptr);
44
45     if (listen(fd, backlog) < 0)
46         err_sys("listen error");
47 }
48 /* end Listen */

```

Рис.8 Функции-обертки для функций *close* и *listen*.

Функции-обертки в своей основе берут на себя обработку событий, сигналов и ошибок. Ниже приведен файл **error_wrap** который содержит функции реагирующие на те самые ошибки, сигналы и события (Рис.9-11).

error_wrap.cpp:

```

4  int      daemon_proc;      /* set nonzero by daemon_init() */
5
6  static void err_doit(int, int, const char *, va_list);
7
8  /* Nonfatal error related to system call
9   * Print message and return */
10
11 void
12 err_ret(const char *fmt, ...)
13 {
14     va_list     ap;
15
16     va_start(ap, fmt);
17     err_doit(1, LOG_INFO, fmt, ap);
18     va_end(ap);
19     return;
20 }
21
22 /* Fatal error related to system call
23  * Print message and terminate */
24
25 void
26 err_sys(const char *fmt, ...)
27 {
28     va_list     ap;
29
30     va_start(ap, fmt);
31     err_doit(1, LOG_ERR, fmt, ap);
32     va_end(ap);
33     exit(1);
34 }

```

Рис.9 Файл *error_wrap*.


```

39 void
40 err_dump(const char *fmt, ...)
41 {
42     va_list    ap;
43
44     va_start(ap, fmt);
45     err_doit(1, LOG_ERR, fmt, ap);
46     va_end(ap);
47     abort();      /* dump core and terminate */
48     exit(1);      /* shouldn't get here */
49 }
50
51 /* Nonfatal error unrelated to system call
52  * Print message and return */
53
54 void
55 err_msg(const char *fmt, ...)
56 {
57     va_list    ap;
58
59     va_start(ap, fmt);
60     err_doit(0, LOG_INFO, fmt, ap);
61     va_end(ap);
62     return;
63 }

```

Рис.10 Файл *error_wrap*.

```

68 void
69 err_quit(const char *fmt, ...)
70 {
71     va_list    ap;
72
73     va_start(ap, fmt);
74     err_doit(0, LOG_ERR, fmt, ap);
75     va_end(ap);
76     exit(1);
77 }
78
79 /* Print message and return to caller
80  * Caller specifies "errnoflag" and "level" */
81
82 static void
83 err_doit(int errnoflag, int level, const char *fmt, va_list ap)
84 {
85     int        errno_save, n;
86     char        buf[MAXLINE + 1];
87
88     errno_save = errno;    /* value caller might want printed */
89 #ifdef HAVE_VSNPRINTF
90     vsnprintf(buf, MAXLINE, fmt, ap);    /* safe */
91 #else
92     vsprintf(buf, fmt, ap);                /* not safe */
93 #endif
94     n = strlen(buf);
95     if (errnoflag)
96         snprintf(buf + n, MAXLINE - n, ": %s", strerror(errno_save));
97     strcat(buf, "\n");
98
99     if (daemon_proc) {
100         syslog(level, buf);
101     } else {
102         fflush(stdout);    /* in case stdout and stderr are the same */
103         fputs(buf, stderr);
104         fflush(stderr);
105     }
106     return;
107 }

```

Рис.11 Функция *err_doit*.

Каждая из функций вызывается в зависимости от ожидаемой реакции, одни прерывают выполнение программы, другие выводят сообщение и дают приложению функционировать дальше.

В каждой функции вызывается функция **err_doit** (Рис.11, с 82 строки). Она отвечает за вывод информации об ошибке, либо информации с заданного файла в консоль или в системный журнал используя функцию **syslog**.

Следующий файл **sock_ntop**, который содержит одноименную кастомную функцию (Рис.13-14, а так же ее функцию-обертку (Рис.12)), которая решает проблему функции **inet_ntop**, зависимость от протокола.

sock_ntop.cpp:

```

78 char *
79 sock_ntop(const struct sockaddr *sa, socklen_t salen)
80 {
81     char    *ptr;
82
83     if ( (ptr = sock_ntop(sa, salen)) == NULL)
84         err_sys("sock_ntop error"); /* inet_ntop() sets errno */
85     return(ptr);
86 }

```

Рис.12 Обертка функции *sock_ntop*.

```

3  #ifndef HAVE_SOCKADDR_DL_STRUCT
4  #include <net/if_dl.h>
5  #endif
6
7  /* include sock_ntop */
8  char *
9  sock_ntop(const struct sockaddr *sa, socklen_t salen)
10 {
11     char    portstr[8];
12     static char str[128]; /* Unix domain is largest */
13
14     switch (sa->sa_family) {
15     case AF_INET: {
16         struct sockaddr_in *sin = (struct sockaddr_in *) sa;
17
18         if (inet_ntop(AF_INET, &sin->sin_addr, str, sizeof(str)) == NULL)
19             return(NULL);
20         if (ntohs(sin->sin_port) != 0) {
21             snprintf(portstr, sizeof(portstr), ":%d", ntohs(sin->sin_port));
22             strcat(str, portstr);
23         }
24         return(str);
25     }
26     /* end sock_ntop */
27
28     #ifdef IPV6
29     case AF_INET6: {
30         struct sockaddr_in6 *sin6 = (struct sockaddr_in6 *) sa;
31
32         str[0] = '[';
33         if (inet_ntop(AF_INET6, &sin6->sin6_addr, str + 1, sizeof(str) - 1) == NULL)
34             return(NULL);
35         if (ntohs(sin6->sin6_port) != 0) {
36             snprintf(portstr, sizeof(portstr), "]:%d", ntohs(sin6->sin6_port));
37             strcat(str, portstr);
38             return(str);
39         }
40         return (str + 1);
41     }
42     #endif

```

Рис.13 Функция *sock_ntop* (окончание на следующем скрине).

```

44 #ifdef AF_UNIX
45     case AF_UNIX: {
46         struct sockaddr_un *unp = (struct sockaddr_un *) sa;
47
48         /* OK to have no pathname bound to the socket: happens on
49          * every connect() unless client calls bind() first. */
50         if (unp->sun_path[0] == 0)
51             strcpy(str, "(no pathname bound)");
52         else
53             snprintf(str, sizeof(str), "%s", unp->sun_path);
54         return(str);
55     }
56 #endif
57
58 #ifdef HAVE_SOCKADDR_DL_STRUCT
59     case AF_LINK: {
60         struct sockaddr_dl *sdl = (struct sockaddr_dl *) sa;
61
62         if (sdl->sdl_nlen > 0)
63             snprintf(str, sizeof(str), "%*s (index %d)",
64                      sdl->sdl_nlen, &sdl->sdl_data[0], sdl->sdl_index);
65         else
66             snprintf(str, sizeof(str), "AF_LINK, index=%d", sdl->sdl_index);
67         return(str);
68     }
69 #endif
70     default:
71         snprintf(str, sizeof(str), "sock_ntop: unknown AF_xxx: %d, len %d",
72                  sa->sa_family, salen);
73         return(str);
74     }
75     return (NULL);
76 }

```

Рис.14 Функция *sock_ntop* (продолжение).

Файл **signal_wrap** который содержит функцию **signal** для обработки сигналов, и ее обертку (Рис.15).

signal_wrap.cpp:

```

3  Sigfunc *
4  signal(int signo, Sigfunc *func)
5  {
6      struct sigaction  act, oact;
7
8      act.sa_handler = func;
9      sigemptyset(&act.sa_mask);
10     act.sa_flags = 0;
11     if (signo == SIGALRM) {
12 #ifdef  SA_INTERRUPT
13         act.sa_flags |= SA_INTERRUPT;  /* SunOS 4.x */
14 #endif
15     } else {
16 #ifdef  SA_RESTART
17         act.sa_flags |= SA_RESTART;    /* SVR4, 44BSD */
18 #endif
19     }
20     if (sigaction(signo, &act, &oact) < 0)
21         return(SIG_ERR);
22     return(oact.sa_handler);
23 }
24 /* end signal */
25
26 Sigfunc *
27 Signal(int signo, Sigfunc *func)  /* for our signal() function */
28 {
29     Sigfunc *sigfunc;
30
31     if ( (sigfunc = signal(signo, func)) == SIG_ERR)
32         err_sys("signal error");
33     return(sigfunc);
34 }

```

Рис.15 Функция *signal* и ее обертка.

Далее следует файл **unix_wrap**. Он содержит функции-обертки для функций **fork**, **malloc** и **write** (Рис.16).

unix_wrap.cpp:

```

3  pid_t
4  Fork(void)
5  {
6      pid_t  pid;
7
8      if ( (pid = fork()) == -1)
9          err_sys("fork error");
10     return(pid);
11 }
12
13 void *
14 Malloc(size_t size)
15 {
16     void    *ptr;
17
18     if ( (ptr = malloc(size)) == NULL)
19         err_sys("malloc error");
20     return(ptr);
21 }
22
23 void
24 Write(int fd, void *ptr, size_t nbytes)
25 {
26     if (write(fd, ptr, nbytes) != nbytes)
27         err_sys("write error");
28 }

```

Рис.16 Файл *unix_wrap*.

Ну и наконец файл **daemon_init**, который содержит в себе одноименную функцию инициализирующую демона (Рис.17).

daemon_init.cpp:

```

3  extern int  daemon_proc;    /* defined in error.c */
4
5  int
6  daemon_init(const char *pname, int facility)
7  {
8      int      i;
9      pid_t    pid;
10
11     if ( (pid = Fork()) < 0)
12         return (-1);
13     else if (pid)
14         _exit(0);          /* parent terminates */
15
16     /* child 1 continues... */
17
18     if (setsid() < 0)        /* become session leader */
19         return (-1);
20
21     Signal(SIGHUP, SIG_IGN);
22     if ( (pid = Fork()) < 0)
23         return (-1);
24     else if (pid)
25         _exit(0);          /* child 1 terminates */
26
27     /* child 2 continues... */
28
29     daemon_proc = 1;        /* for err_XXX() functions */
30
31     chdir("/");             /* change working directory */
32
33     /* close off file descriptors */
34     for (i = 0; i < MAXFD; i++)
35         close(i);
36
37     /* redirect stdin, stdout, and stderr to /dev/null */
38     open("/dev/null", O_RDONLY);
39     open("/dev/null", O_RDWR);
40     open("/dev/null", O_RDWR);
41
42     openlog(pname, LOG_PID, facility);
43
44     return (0);             /* success */
45 }

```

Рис.17 Функция *daemon_init*.

Листинг программы

server_daemon_header.h:

```

#include <fstream>
#include <string>
#include <cstdlib>
#include <sstream>
#include <sys/types.h> /* basic system data types */
#include <sys/socket.h> /* basic socket definitions */
#include <syslog.h>
#include <netinet/in.h> /* sockaddr_in{ } and other Internet defs */
#include <arpa/inet.h> /* inet(3) functions */

```

```

#include    <stdarg.h>           /* ANSI C header file */
#include    <errno.h>
#include    <fcntl.h>           /* for nonblocking */
#include    <netdb.h>
#include    <signal.h>
#include    <sys/un.h>          /* for Unix domain sockets */

#define MAXFD      64
#define MAXLINE    4096 /* max text line length */
#define LISTENQ    1024 /* 2nd argument to listen() */
#define SA        struct sockaddr

typedef void  Sigfunc(int); /* for signal handlers */

void  err_dump(const char *, ...);
void  err_msg(const char *, ...);
void  err_quit(const char *, ...);
void  err_ret(const char *, ...);
void  err_sys(const char *, ...);

void  Setsockopt(int, int, int, const void *, socklen_t);
void  Listen(int, int);
void  Close(int);
int   Accept(int, SA *, socklen_t *);

pid_t  Fork(void);
void   *Malloc(size_t);

void  Write(int, void *, size_t);

int   tcp_listen(const char *, const char *, socklen_t *);
int   Tcp_listen(const char *, const char *, socklen_t *);

char   *sock_ntop(const SA *, socklen_t);
char   *Sock_ntop(const SA *, socklen_t);

int   daemon_init(const char *, int);

Sigfunc *Signal(int, Sigfunc *);

```

main.cpp:

```

#include "server_daemon_header.h"

std::string spaces_message(char *filen, char *countn)
{
    std::ifstream file(filen);
    int spaces = 0;
    int count = std::stoi(countn);
    std::string line;
    std::string resLine;

```



```

        if (!file.is_open())
        {
            err_quit("File not open!");
        }

        std::getline(file, line);

        for (int i = 0; i <= line.length(); ++i)
        {
            resLine.push_back(line[i]);
            if (line[i] == ' ' && spaces < count)
            {
                resLine.push_back(' ');
                spaces += 1;
            }
        }
        return resLine;
    }

int
main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t addrlen, len;
    struct sockaddr      *cliaddr;
    std::string resLine;

    if (argc < 4 || argc > 5)
        err_quit("usage: course_project_daemon [ <host> ] <service or port> <file>
<count spaces>");

    resLine.append(spaces_message(argv[3], argv[4]));

    daemon_init(argv[0], 0);

    if (argc == 2)
        listenfd = Tcp_listen(NULL, argv[1], &addrlen);
    else
        listenfd = Tcp_listen(argv[1], argv[2], &addrlen);

    cliaddr = static_cast<sockaddr*>(Malloc(addrlen));

    for ( ; ; ) {
        len = addrlen;
        connfd = Accept(listenfd, cliaddr, &len);
        err_msg("connection from %s", Sock_ntop(cliaddr, len));
        err_msg(resLine.c_str(), Sock_ntop(cliaddr, len));

        Close(connfd);
    }
}

```

tcp_listen.cpp:

```
#include "server_daemon_header.h"

int
tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)
{
    int                listenfd, n;
    const int          on = 1;
    struct addrinfo hints, *res, *ressave;

    bzero(&hints, sizeof(struct addrinfo));
    hints.ai_flags = AI_PASSIVE;
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
        err_quit("tcp_listen error for %s, %s: %s",
                  host, serv, gai_strerror(n));
    ressave = res;

    do {
        listenfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
        if (listenfd < 0)
            continue;                /* error, try next one */

        Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
        if (bind(listenfd, res->ai_addr, res->ai_addrlen) == 0)
            break;                    /* success */

        Close(listenfd);              /* bind error, close and try next one */
    } while ( (res = res->ai_next) != NULL);

    if (res == NULL)                  /* errno from final socket() or bind() */
        err_sys("tcp_listen error for %s, %s", host, serv);

    Listen(listenfd, LISTENQ);

    if (addrlenp)
        *addrlenp = res->ai_addrlen; /* return size of protocol address */

    freeaddrinfo(ressave);

    return(listenfd);
}
/* end tcp_listen */

/*
 * We place the wrapper function here, not in wraplib.c, because some
 * XTI programs need to include wraplib.c, and it also defines
 * a Tcp_listen() function.
 */
```

```
*/
```

```
int  
Tcp_listen(const char *host, const char *serv, socklen_t *addrlenp)  
{  
    return(tcp_listen(host, serv, addrlenp));  
}
```

sock_wrap.cpp:

```
#include "server_daemon_header.h"
```

```
int  
Accept(int fd, struct sockaddr *sa, socklen_t *salenptr)  
{  
    int n;  
  
again:  
    if ( (n = accept(fd, sa, salenptr)) < 0) {  
#ifdef EPROTO  
        if (errno == EPROTO || errno == ECONNABORTED)  
#else  
        if (errno == ECONNABORTED)  
#endif  
            goto again;  
        else  
            err_sys("accept error");  
    }  
    return(n);  
}  
  
void  
Setsockopt(int fd, int level, int optname, const void *optval, socklen_t optlen)  
{  
    if (setsockopt(fd, level, optname, optval, optlen) < 0)  
        err_sys("setsockopt error");  
}  
  
void  
Close(int fd)  
{  
    if (close(fd) == -1)  
        err_sys("close error");  
}  
  
void  
Listen(int fd, int backlog)  
{  
    char *ptr;
```

```
/*4can override 2nd argument with environment variable */
```

```

        if ( (ptr = getenv("LISTENQ")) != NULL)
            backlog = atoi(ptr);

        if (listen(fd, backlog) < 0)
            err_sys("listen error");
    }
/* end Listen */

```

error_wrap.cpp:

```

#include "server_daemon_header.h"

int          daemon_proc;          /* set nonzero by daemon_init() */

static void   err_doit(int, int, const char *, va_list);

/* Nonfatal error related to system call
 * Print message and return */

void
err_ret(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, LOG_INFO, fmt, ap);
    va_end(ap);
    return;
}

/* Fatal error related to system call
 * Print message and terminate */

void
err_sys(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(1);
}

/* Fatal error related to system call
 * Print message, dump core, and terminate */

void
err_dump(const char *fmt, ...)

```

```

{
    va_list      ap;

    va_start(ap, fmt);
    err_doit(1, LOG_ERR, fmt, ap);
    va_end(ap);
    abort();      /* dump core and terminate */
    exit(1);      /* shouldn't get here */
}

/* Nonfatal error unrelated to system call
 * Print message and return */

void
err_msg(const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    err_doit(0, LOG_INFO, fmt, ap);
    va_end(ap);
    return;
}

/* Fatal error unrelated to system call
 * Print message and terminate */

void
err_quit(const char *fmt, ...)
{
    va_list      ap;

    va_start(ap, fmt);
    err_doit(0, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(1);
}

/* Print message and return to caller
 * Caller specifies "errnoflag" and "level" */

static void
err_doit(int errnoflag, int level, const char *fmt, va_list ap)
{
    int          errno_save, n;
    char         buf[MAXLINE + 1];

    errno_save = errno;      /* value caller might want printed */
#ifdef HAVE_VSNPRINTF
    vsnprintf(buf, MAXLINE, fmt, ap); /* safe */
#else
    vsprintf(buf, fmt, ap);          /* not safe */
#endif

```

```

#endif
    n = strlen(buf);
    if (errnoflag)
        snprintf(buf + n, MAXLINE - n, ": %s", strerror(errno_save));
    strcat(buf, "\n");

    if (daemon_proc) {
        syslog(level, buf);
    } else {
        fflush(stdout);          /* in case stdout and stderr are the same */
        fputs(buf, stderr);
        fflush(stderr);
    }
    return;
}

```

sock_ntop.cpp:

```

#include "server_daemon_header.h"

#ifdef HAVE_SOCKADDR_DL_STRUCT
#include <net/if_dl.h>
#endif

/* include sock_ntop */
char *
sock_ntop(const struct sockaddr *sa, socklen_t salen)
{
    char        portstr[8];
    static char str[128];          /* Unix domain is largest */

    switch (sa->sa_family) {
    case AF_INET: {
        struct sockaddr_in    *sin = (struct sockaddr_in *) sa;

        if (inet_ntop(AF_INET, &sin->sin_addr, str, sizeof(str)) == NULL)
            return(NULL);
        if (ntohs(sin->sin_port) != 0) {
            snprintf(portstr, sizeof(portstr), ":%d", ntohs(sin->sin_port));
            strcat(str, portstr);
        }
        return(str);
    }
    }
}
/* end sock_ntop */

#ifdef IPV6
case AF_INET6: {
    struct sockaddr_in6    *sin6 = (struct sockaddr_in6 *) sa;

    str[0] = '[';
    if (inet_ntop(AF_INET6, &sin6->sin6_addr, str + 1, sizeof(str) - 1) == NULL)

```

```

        return(NULL);
    if (ntohs(sin6->sin6_port) != 0) {
        snprintf(portstr, sizeof(portstr), "]:%d", ntohs(sin6->sin6_port));
        strcat(str, portstr);
        return(str);
    }
    return (str + 1);
}
#endif

#ifdef AF_UNIX
case AF_UNIX: {
    struct sockaddr_un    *unp = (struct sockaddr_un *) sa;

    /* OK to have no pathname bound to the socket: happens on
       every connect() unless client calls bind() first. */
    if (unp->sun_path[0] == 0)
        strcpy(str, "(no pathname bound)");
    else
        snprintf(str, sizeof(str), "%s", unp->sun_path);
    return(str);
}
#endif

#ifdef HAVE_SOCKADDR_DL_STRUCT
case AF_LINK: {
    struct sockaddr_dl    *sdl = (struct sockaddr_dl *) sa;

    if (sdl->sdl_nlen > 0)
        snprintf(str, sizeof(str), "%*s (index %d)",
                 sdl->sdl_nlen, &sdl->sdl_data[0], sdl->sdl_index);
    else
        snprintf(str, sizeof(str), "AF_LINK, index=%d", sdl->sdl_index);
    return(str);
}
#endif

default:
    snprintf(str, sizeof(str), "sock_ntop: unknown AF_xxx: %d, len %d",
             sa->sa_family, salen);
    return(str);
}
return (NULL);
}

char *
Sock_ntop(const struct sockaddr *sa, socklen_t salen)
{
    char    *ptr;

    if ( (ptr = sock_ntop(sa, salen)) == NULL)
        err_sys("sock_ntop error"); /* inet_ntop() sets errno */
    return(ptr);
}

```

```
}
```

signal_wrap.cpp:

```
#include "server_daemon_header.h"
```

```
Sigfunc *
```

```
signal(int signo, Sigfunc *func)
```

```
{
```

```
    struct sigaction      act, oact;
```

```
    act.sa_handler = func;
```

```
    sigemptyset(&act.sa_mask);
```

```
    act.sa_flags = 0;
```

```
    if (signo == SIGALRM) {
```

```
#ifdef SA_INTERRUPT
```

```
        act.sa_flags |= SA_INTERRUPT;    /* SunOS 4.x */
```

```
#endif
```

```
    } else {
```

```
#ifdef SA_RESTART
```

```
        act.sa_flags |= SA_RESTART;      /* SVR4, 44BSD */
```

```
#endif
```

```
    }
```

```
    if (sigaction(signo, &act, &oact) < 0)
```

```
        return(SIG_ERR);
```

```
    return(oact.sa_handler);
```

```
}
```

```
/* end signal */
```

```
Sigfunc *
```

```
Signal(int signo, Sigfunc *func)    /* for our signal() function */
```

```
{
```

```
    Sigfunc      *sigfunc;
```

```
    if ( (sigfunc = signal(signo, func)) == SIG_ERR)
```

```
        err_sys("signal error");
```

```
    return(sigfunc);
```

```
}
```

unix_wrap.cpp:

```
#include "server_daemon_header.h"
```

```
pid_t
```

```
Fork(void)
```

```
{
```

```
    pid_t  pid;
```

```
    if ( (pid = fork()) == -1)
```

```
        err_sys("fork error");
```



```

        return(pid);
    }

    void *
    Malloc(size_t size)
    {
        void *ptr;

        if ( (ptr = malloc(size)) == NULL)
            err_sys("malloc error");
        return(ptr);
    }

    void
    Write(int fd, void *ptr, size_t nbytes)
    {
        if (write(fd, ptr, nbytes) != nbytes)
            err_sys("write error");
    }

```

daemon_init.cpp:

```

#include "server_daemon_header.h"

extern int      daemon_proc; /* defined in error.c */

int
daemon_init(const char *pname, int facility)
{
    int          i;
    pid_t        pid;

    if ( (pid = Fork()) < 0)
        return (-1);
    else if (pid)
        _exit(0);                                /* parent terminates */

    /* child 1 continues... */

    if (setsid() < 0)                               /* become session leader */
        return (-1);

    Signal(SIGHUP, SIG_IGN);
    if ( (pid = Fork()) < 0)
        return (-1);
    else if (pid)
        _exit(0);                                /* child 1 terminates */

    /* child 2 continues... */

    daemon_proc = 1;                               /* for err_XXX() functions */

```

```

chdir("/");                                /* change working directory */

/* close off file descriptors */
for (i = 0; i < MAXFD; i++)
    close(i);

/* redirect stdin, stdout, and stderr to /dev/null */
open("/dev/null", O_RDONLY);
open("/dev/null", O_RDWR);
open("/dev/null", O_RDWR);

openlog(pname, LOG_PID, facility);

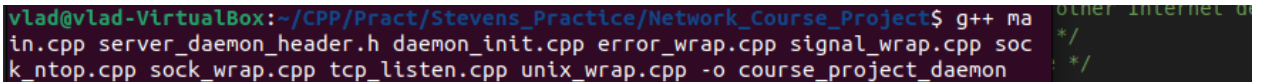
return (0);                                /* success */
}

```

Компиляция и запуск программы

Компиляцию и запуск производим в системе Ubuntu 22, с использованием стандартного для этой системы терминала, а так же с использованием компилятора **gcc/g++**.

Компиляция:



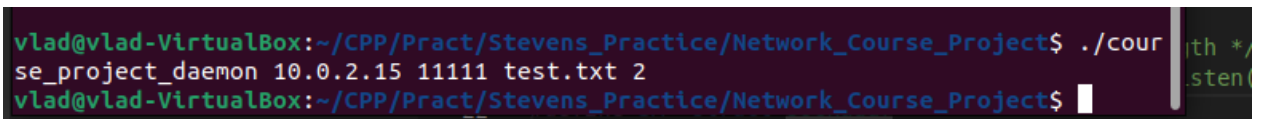
```

vlad@vlad-VirtualBox:~/CPP/Pract/Stevens_Practice/Network_Course_Project$ g++ main.cpp server_daemon_header.h daemon_init.cpp error_wrap.cpp signal_wrap.cpp socket_ntop.cpp sock_wrap.cpp tcp_listen.cpp unix_wrap.cpp -o course_project_daemon

```

В качестве аргументов командной строки программа принимает адрес либо имя хоста, номер порта, имя файла, количество пробелов которое нужно заменить на 2 пробела.

Запуск программы:

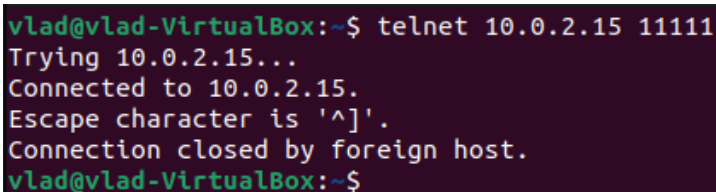


```

vlad@vlad-VirtualBox:~/CPP/Pract/Stevens_Practice/Network_Course_Project$ ./course_project_daemon 10.0.2.15 11111 test.txt 2
vlad@vlad-VirtualBox:~/CPP/Pract/Stevens_Practice/Network_Course_Project$

```

Далее связываемся с демоном с помощью **telnet**:



```

vlad@vlad-VirtualBox:~$ telnet 10.0.2.15 11111
Trying 10.0.2.15...
Connected to 10.0.2.15.
Escape character is '^]'.
Connection closed by foreign host.
vlad@vlad-VirtualBox:~$

```

Все прошло удачно. Теперь заглянем в **syslog**:

```
May 27 14:01:35 vlad-VirtualBox ./course_project_daemon[6867]: connection from 1
0.0.2.15:43138
May 27 14:01:35 vlad-VirtualBox ./course_project_daemon[6867]: Example text fo
r my daemon!
vlad@vlad-VirtualBox:~/CPP/Pract/Stevens_Practice/Network_Course_Project$
```

Видим сообщение о присоединении, а так же наш тестовый файл с 2 пробелами в двух местах, что соответствует указанным аргументам.

Выводы

В процессе написания программы для курсового проекта ознакомились и разобрались с таким понятием как процессы-демоны. Научились их реализации, а так же удалось поработать с системным журналом, закрепить ранее полученные навыки написания серверных приложений. Процесс разработки проходил в ОС Linux Ubuntu 22 с использованием языков программирования C/C++ и их стандартных библиотек. Так же для удобства разработки серверного приложения были созданы функции-обертки располагающиеся в отдельных файлах. Функции-обертки отвечающие за создание самого сервера расположены в файлах **tcp_listen.cpp**, **sock_wrap**, **sock_ntop.cpp**. Функции-обертки отвечающие за выделение динамической памяти, передачи байтов в сети, создание потоков и тд., находятся в файле **unix_wrap.cpp**. Функции-обертки отвечающие за обработку сигналов и ошибок находятся в **signal_wrap.cpp** и **error_wrap.cpp**. Инициализация процесса-демона в **daemon_init.cpp**.

Проверка на антиплагиат

| | | | |
|-----------------|-------|----------------|-----|
| Уникальность | 71% | Орфография | 43 |
| Всего символов | 20928 | Заспамленность | 63% |
| Без пробелов | 18202 | Вода | 13% |
| Количество слов | 2821 | | |

| | | | | | | |
|--------------------------|---|----------------------------|-------|------------------|--------|--|
| <input type="checkbox"/> | Network_Prog_Course_Project_Antipl.docx | БЕЛОРУССКИЙ НАЦИОНАЛЬНЫ... | 20806 | 11.06.2023 11:51 | 71.27% | |
|--------------------------|---|----------------------------|-------|------------------|--------|--|

Ссылка на результат:

<https://text.ru/antiplagiat/64858b412007a>

ЛИТЕРАТУРА

1. Linux API. Исчерпывающее руководство. Майкл Керриск.
2. UNIX: разработка сетевых приложений. 3-е издание. У.Р. Стивенс.