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](file:///D:\BNTU\BNTU\OOP\Course_Project_OOP.docx#_Toc125733098)

Течническое задание 4

Вариант задания 4

[Теоретическая часть](file:///D:\BNTU\BNTU\OOP\Course_Project_OOP.docx#_Toc125733102) 4

[Практическая часть](file:///D:\BNTU\BNTU\OOP\Course_Project_OOP.docx#_Toc125733108) 11

[Краткое описание программы 11](file:///D:\BNTU\BNTU\OOP\Course_Project_OOP.docx#_Toc125733109)

[Исходный код программы 23](file:///D:\BNTU\BNTU\OOP\Course_Project_OOP.docx#_Toc125733110)

[Компиляция и запуск программы 34](file:///D:\BNTU\BNTU\OOP\Course_Project_OOP.docx#_Toc125733111)

[ЗАКЛЮЧЕНИЕ 35](file:///D:\BNTU\BNTU\OOP\Course_Project_OOP.docx#_Toc125733112)

[ЛИТЕРАТУРА 36](file:///D:\BNTU\BNTU\OOP\Course_Project_OOP.docx#_Toc125733113)

# **ВВЕДЕНИЕ**

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

С++ является мощным языком, унаследовав от Си богатые возможности по работе с памятью. Поэтому нередко С++ 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 С++

С++ является компилируемым языком, а это значит, что компилятор транслирует исходный код на С++ 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 С++ 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 С++ 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 С++.

В 1985 году была выпущена первая коммерческая версия языка С++, 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 Organiztion for Standartization). Первый стандарт получил название ISO/IEC 14882:1998 или сокращенно С++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**

****

**Теоретическая часть**

**Системные службы (демоны) в Linux**

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

- 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 **0\_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. Закрыть 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 ограниченно.

**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** устаналивает 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.

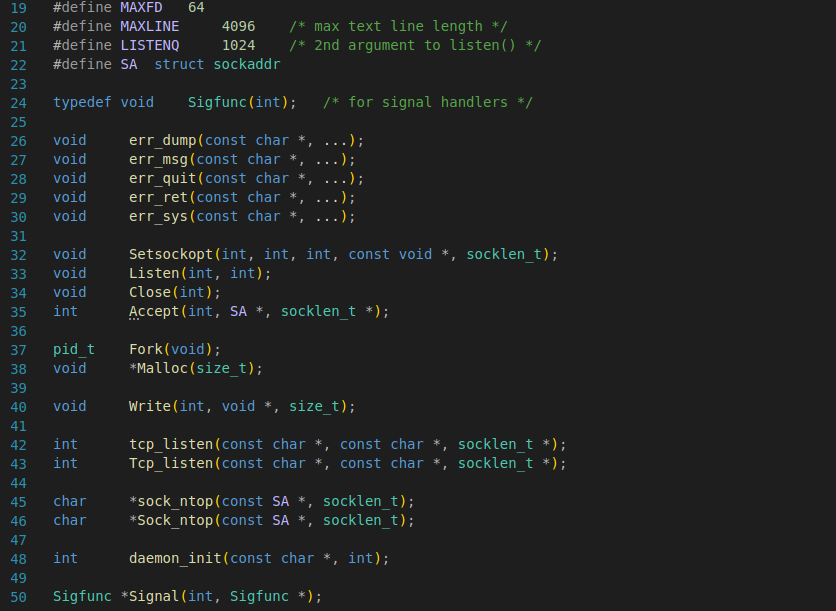
**Практическая часть**

**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:**



*Рис.1 Прототипы функций.*

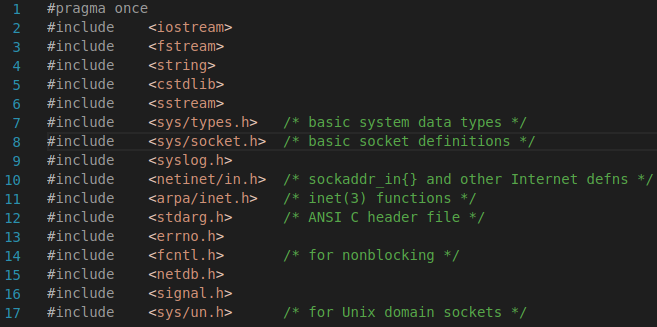
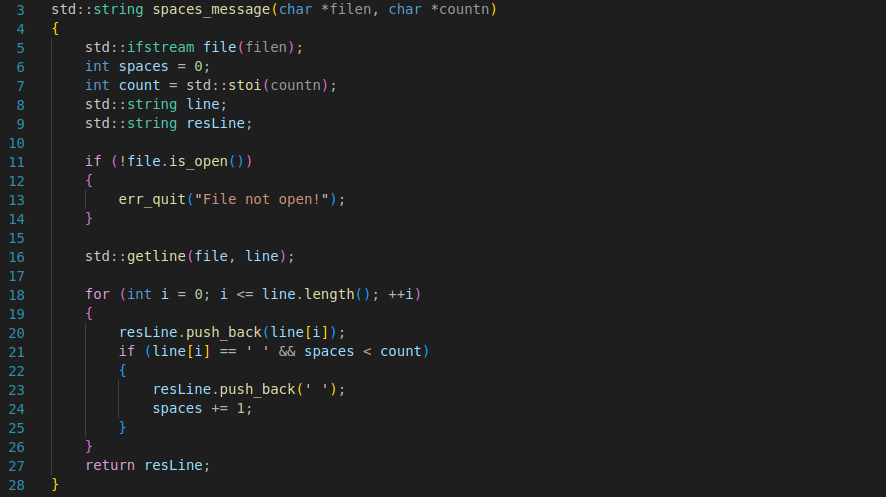


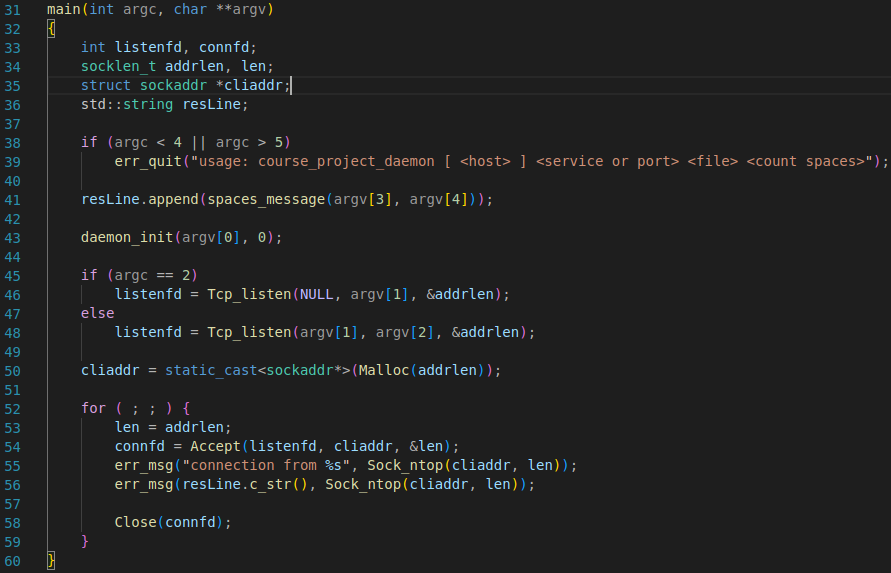
Рис.2 Подключение необходимых библиотек.

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:**

****

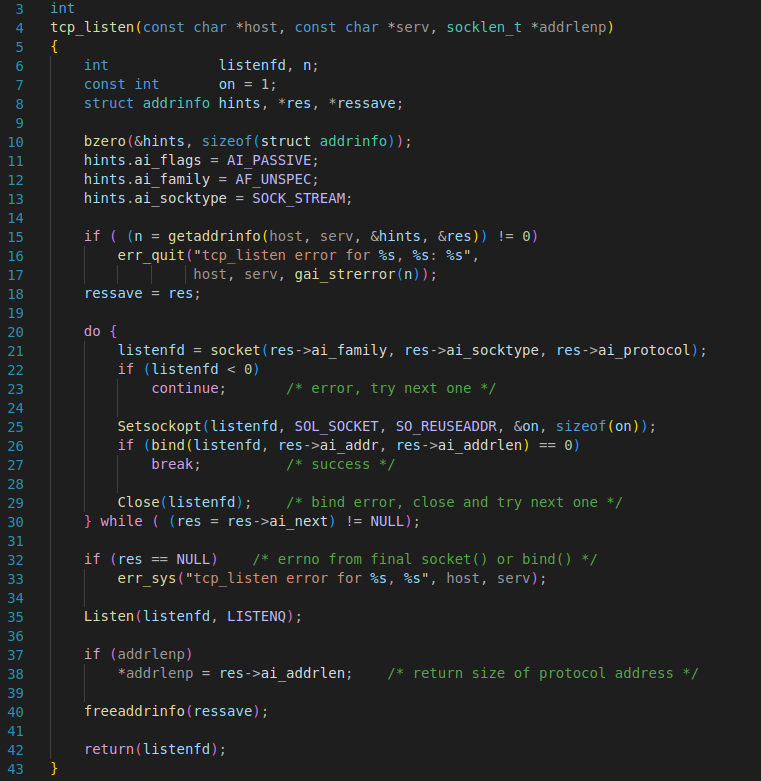
*Рис.3 Функция читает с файла и ставит доп. пробел.*



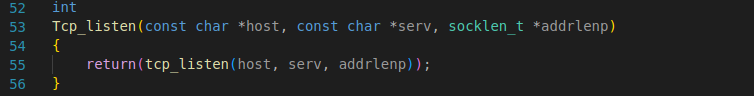
*Рис.4 Функция* ***main****.*

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

**tcp\_listen.cpp:**

****

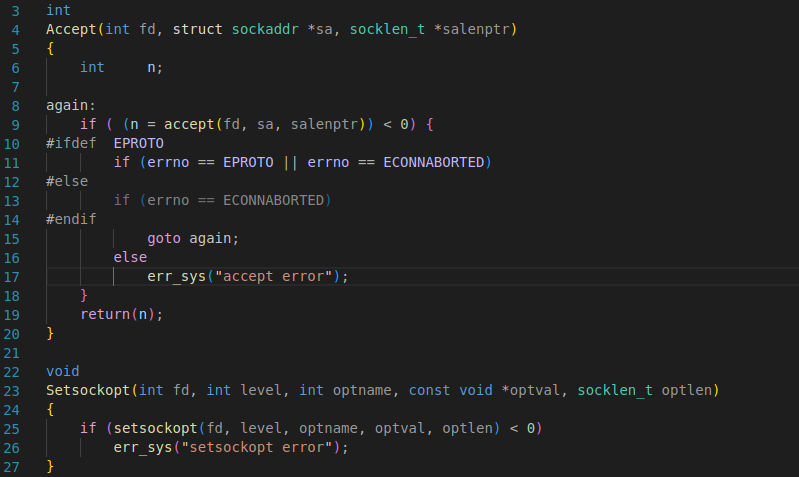
*Рис.5 Функция* ***tcp\_listen****.*



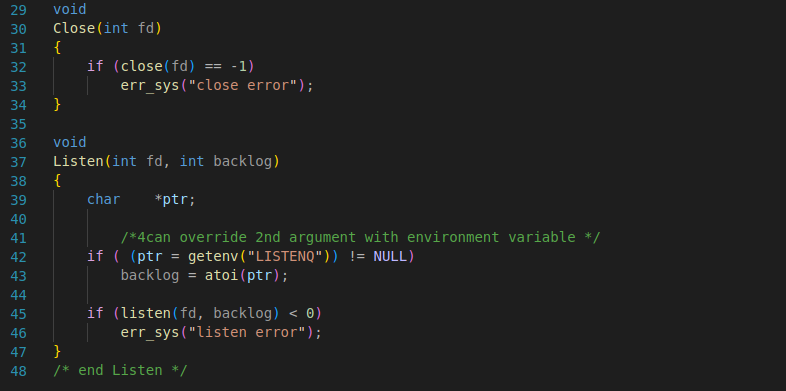
*Рис.6 Функция-обертка для функции* ***tcp\_listen****.*

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

**sock\_wrap.cpp:**



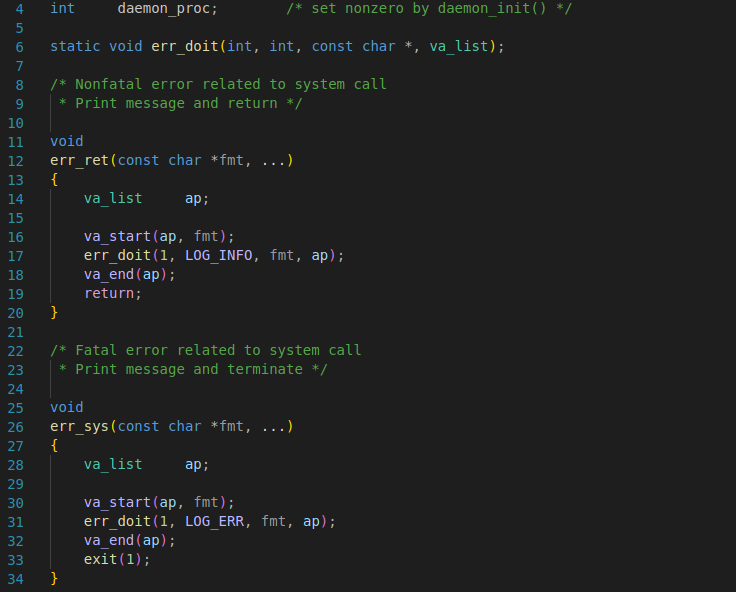
*Рис.7 Функции-обертки для функций* ***accept*** *и* ***setsockopt****.*



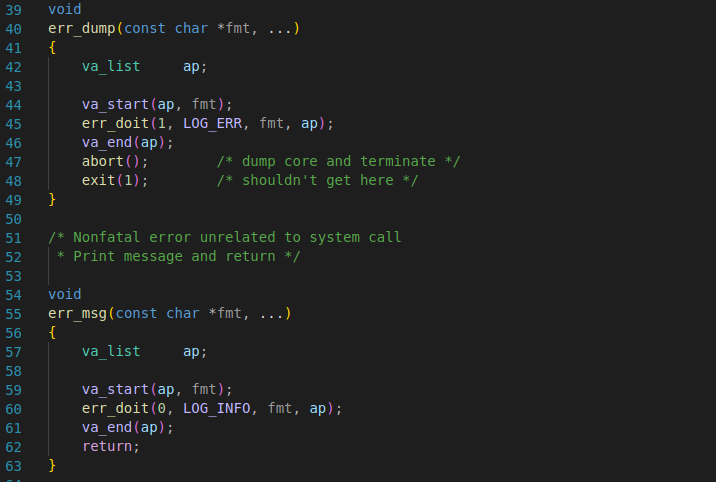
*Рис.8 Функции-обертки для функций* ***close*** *и* ***listen****.*

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).

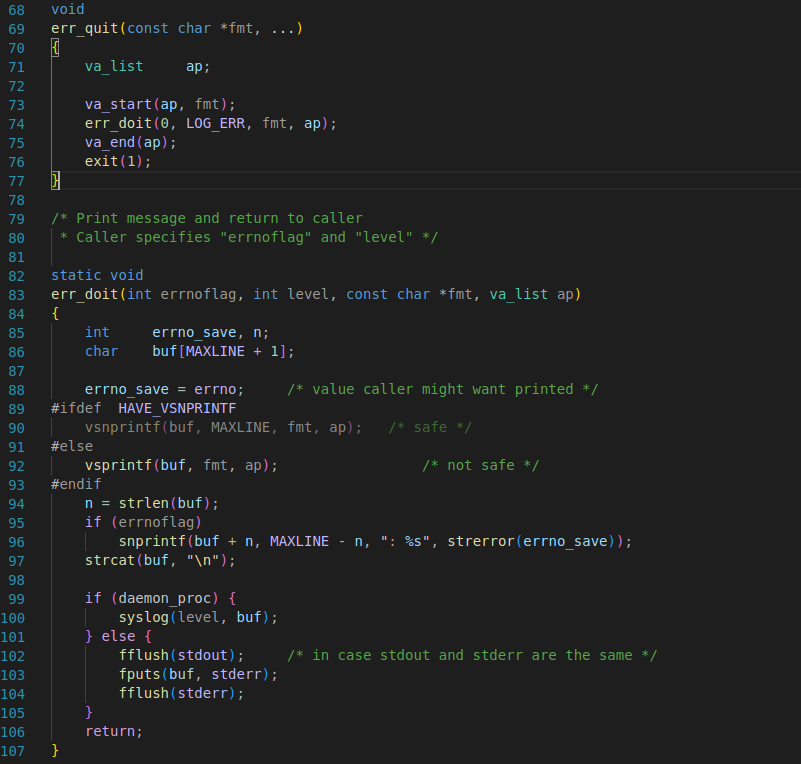
**error\_wrap.cpp:**

****

*Рис.9 Файл* ***error\_wrap****.*



*Рис.10 Файл* ***error\_wrap****.*



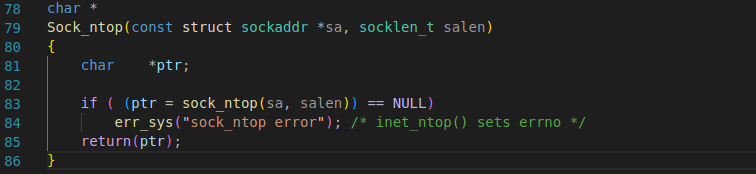
*Рис.11 Функция* ***err\_doit****.*

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

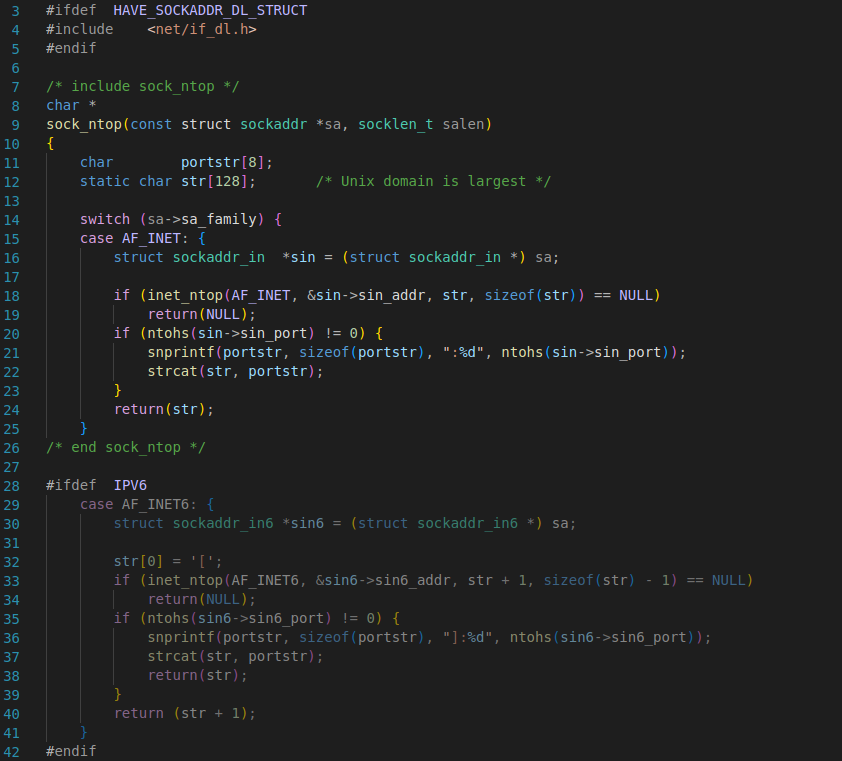
В каждой функции вызывается функция **err\_doit** (Рис.11, с 82 строки). Она отвечает за вывод информации об ошибке, либо информации с заданного файла в консоль или в системный журнал используя функцию **syslog**.

Следующий файл **sock ntop**, который содержит одноименною кастомную функцию (Рис.13-14, а так же ее функцию-обертку (Рис.12)), которая решает проблему функции **inet\_ntop**, зависимость от протокола.

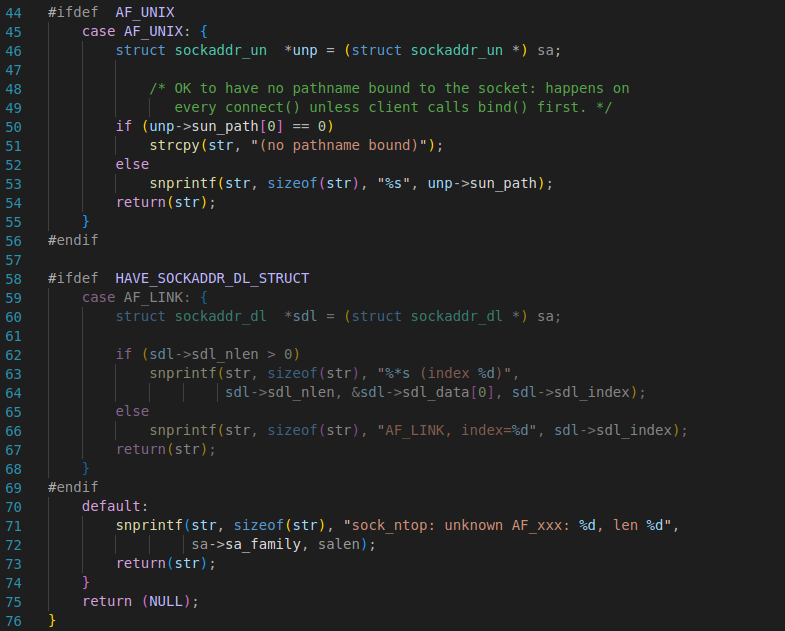
**sock\_ntop.cpp:**

****

*Рис.12 Обертка функции* ***sock\_ntop****.*

****

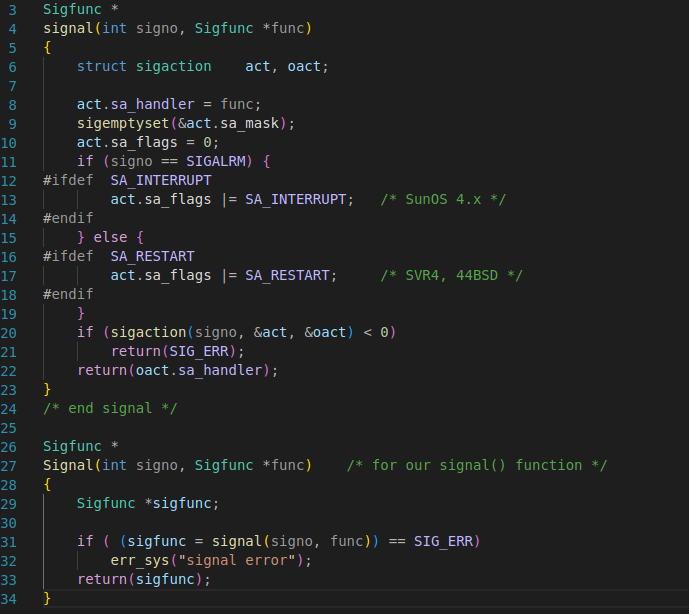
*Рис.13 Функция* ***sock\_ntop*** *(окончание на следующем скрине).*

****

*Рис.14 Функция* ***sock\_ntop*** *(продолжение).*

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

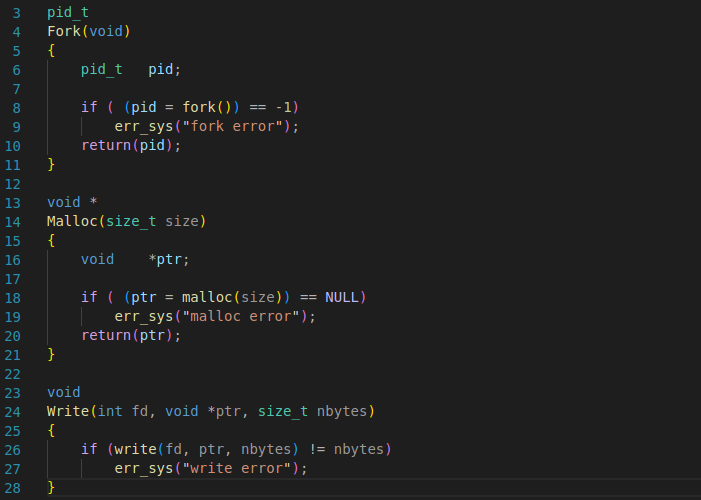
**signal\_wrap.cpp:**



*Рис.15 Функция* ***signal*** *и ее обертка.*

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

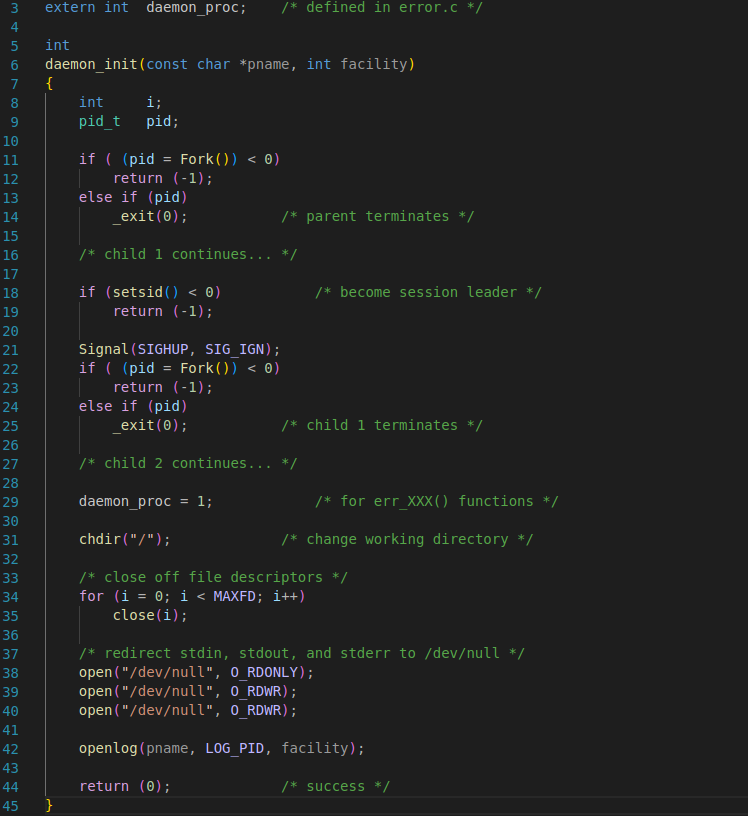
**unix\_wrap.cpp:**



*Рис.16 Файл* ***unix\_wrap****.*

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

**daemon\_init.cpp:**



*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 defns \*/

#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

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**

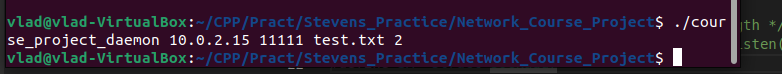
Компиляцию и запуск производим в системе Ubuntu 22, с использованием стандартного для этой системы терминала, а так же с использованием компилятора **gcc/g++**.

**Компиляция:**

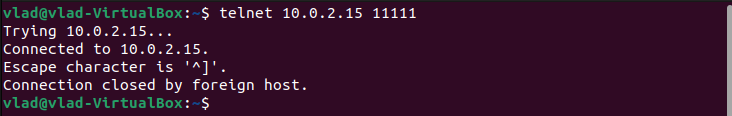
****

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

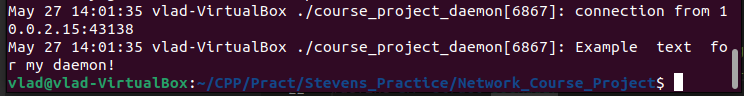
**Запуск программы:**

****

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



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

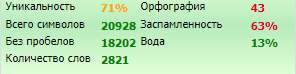
****

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 c 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**

****

****

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.