Работа с компилятором GCC, отладчиком GBD и создание Makefile.

Компиляция системных приложений.

GCC компилятор (*GNU C Compiler, GNU Compiler Collection*) – одна из основных составляющих частей системного программирования в операционной системе Linux.

Работа *GCC* включает три этапа:

обработка препроцессором, компиляция и компоновка (или линковка).

Препроцессор включает в основной файл содержимое всех заголовочных файлов, указанных в директивах *#include.* В заголовочных файлах обычно находятся объявления функций, используемых в программе, но не определенных в тексте программы. Их определения находятся где - то в другом месте: или в других файлах с исходным кодом или в бинарных библиотеках (ключ -*E*).

Вторая стадия – компиляция. Она заключается в превращении текста программы на языке *C/C++* в набор машинных команд. Результат сохраняется в объектном файле. Разумеется, на машинах с разной архитектурой процессора двоичные файлы получаются в разных форматах, и на одной машине невозможно запустить бинарный файл, собранный на другой машине (разве только, если у них одинаковая архитектура процессора и одинаковые операционные системы). Поэтому программы для *UNIX*-подобных систем распространяются в виде исходных кодов: они должны быть доступны всем пользователям, независимо от того, у кого какой процессор и какая операционная система (ключ -*c*).

Последняя стадия – компоновка. Она заключается в связывании всех объектных файлов проекта в один, связывании вызовов функций с их определениями, и присоединением библиотечных файлов, содержащих функции, которые вызываются, но не определены в проекте. В результате формируется запускаемый файл – наша конечная цель. Если какая-то функция в программе используется, но компоновщик не найдет место, где эта функция определена, он выдаст сообщение об ошибке, и откажется создавать исполняемый файл (ключ -*o*).

Если создается объектный файл из исходника, уже обработанного препроцессором (например, такого, какой мы получили выше), то мы должны обязательно указать явно, что компилируемый файл является файлом исходного кода, обработанный препроцессором, и имеющий теги препроцессора. В противном случае он будет обрабатываться, как обычный файл *C++*, без учета тегов препроцессора, а значит связь с объявленными функциями не будет устанавливаться. Для явного указания на язык и формат обрабатываемого файла служит ключ -*x*. Файл *C++*, обработанный препроцессором обозначается *cpp-output.*

gcc -x cpp-output -c proga.cpp

Компиляция при помощи GCC может быть проведена с различными ключами. Наиболее распространенные из ключей следующие:

- *-c* компиляция без компоновки — создаются объектные файлы *file.o;*
- *-o file-name* задать имя *file-name* создаваемому файлу;
- *-g* поместить в файл (объектный или исполняемый) отладочную информацию для отладчика *gdb*;
- *-MM* вывести зависимости от заголовочных файлов *C* и/или *C++* программ в формате, подходящем для утилиты *make*; при этом объектные или исполняемые файлы не будет созданы;
- *-Wall* вывод на экран сообщений об ошибках, возникших во время компиляции;
- *-O* задать уровень оптимизации компиляции *<0,1,2,3,s>*, где 0 – без оптимизации; 1 – 3 уровни оптимизации по скорости работы; *s* – оптимизация по размеру;
- *-pipe* ускоряет процесс компиляции за счет использования конвейера (*pipe*) вместо временных файлов в течение разных стадий компиляции, которые используют большее количество памяти.

Задание 1.

1. Создайте отдельный каталог *hello*. Это будет каталог нашего первого проекта. В нем создайте текстовый файл *hello.c* со следующим текстом:

```
#include <stdio.h>
int main(void){
printf("Hello world!\n");
return(0);
}
```

2. Затем в консоли зайдите в каталог проекта. Наберите команду

```
gcc hello.c
```

3. Теперь в каталоге появился новый файл

```
a.out
```

4. Это и есть исполняемый файл. Запустим его.

```
./a.out
```

5. Компилятор *gcc* по умолчанию присваивает всем созданным исполняемым файлам имя «*a.out*». Если хотите назвать его по-другому, нужно к команде на компиляцию добавить флаг -o и имя, которым вы хотите его назвать:

```
gcc hello.c -o hello
```

6. Проверьте, что в каталоге появился исполняемый файл с названием *hello.*

7. Запустим его.

   ./hello.

Задание 2. Создание проекта в GCC

1. Создайте папку для проекта.
2. Создайте в командной строке фалы *hello.c, hello.h, main.c*. Например, используйте редактор *vi / vim.*
3. Файлы *hello.c* и *hello.h* должен содержать функцию вывод на экран фразы *Hello World !*, функция должна вызываться из *main.c.*
4. Скомпилируйте т.н. объектный код для файлов с расширением *«.c»* с использованием *gcc*

   например, *gcc -o hello.o hello.c*
5. Скомпилируйте программу на основе объектного кода

   например, *gcc -o hello hello.o main.o*
6. Проверьте работоспособность скомпилированной программы *(./hello)*

Создание библиотек при помощи GCC

Компилятор *GCC* позволяет работать со статическими и динамическими библиотеками. Стандартное расположение файлов библиотек–каталоги */usr/lib* и */usr/local/lib* (при желании можно добавить дополнительные путь). Если библиотечный файл имеет расширение «*.a*», то это статическая библиотека, то есть при компоновке весь ее двоичный код включается в исполняемый файл. Если расширение *«.so»*, то это динамическая библиотека. Это значит в исполняемый файл программы помещается только ссылка на библиотечный файл, а уже из него и запускается функция.

При компоновке любой программы компилятор *gcc* по умолчанию включает в запускаемый файл библиотеку *libc*. Это стандартная библиотека языка *C*. Она содержит рутинные функции, необходимые абсолютно во всех программах, написанных на *C*. Поскольку библиотека *libc* нужна во всех программах, она включается по умолчанию, без необходимости давать отдельное указание на ее включение.

Для линковки остальных библиотек они должны быть или в стандартных каталогах, или при компоновке приложения необходимо указать на них непосредственно.

Названия всех библиотек для *GCC* должно начинаться с буквосочетания *lib-*. Для их явного включения в исполняемый файл, нужно добавить к команде *gcc* опцию *-l*, к которой слитно прибавить название библиотеки без *lib-*. Например, чтобы включить библиотеку *libvga* надо указать опцию *-lvga.*

Задание 3.

1. Скопируйте файлы из задания 2 в новый проект и удалите объектные файлы.
2. Создайте динамическую библиотеку с расширением *«.so»*
   libHello (gcc -o libHello.so -shared -fPIC hello.c)
3. Проверьте наличие в библиотеке созданной функции при помощи утилиты nm (*nm libHello.so*, тип *«T»* - текст программы).
4. Скомпилируйте исполняемый файл на основе динамической библиотеки
   gcc main.c -L. -lHello -o hello

Флаг *-L.* – текущий каталог, *-lHello* - специальный формат имени библиотеки.

Для запуска программы с динамической библиотекой необходимо зарегистрировать ее в сервисе *ld* – специальном сервисе для подтягивания библиотек. Для этого есть несколько способов:

положить библиотеку в известный для *ld* каталог.

Зарегистрировать каталог в *ld.*

Использовать при компиляции специальную переменную *ldlibrarypath* – для того, чтобы *ld* знал где искать библиотеку конкретно под данную программу.

Использовать переменную *ldpreload* для того, чтобы *ld* заранее предзагрузил библиотеки.

5. Для работы с переменной *ldlibrarypath* нужно вызвать переменную окружения *export LD_LIBRARY_PATH=*.
6. Затем попробуйте запустить программу.

Создание файла автоматической сборки проектов.

В ОС *Linux* предусмотрены средства автоматизации сборки файлов – т.н. *Makefile.* Основная идея *makefile* заключается в объединении всех исходных файлов и всех команд для сборки программы в отдельный текстовый файл. Для того, чтобы потом считывать их оттуда одной командой.

Файл *Makefile* является списком правил. Каждое правило начинается с указателя, называемого «Цель». После него стоит двоеточие, и далее через пробел указываются зависимости. После зависимостей пишутся команды. Каждая команда должна находиться на отдельной строке, и отделяться от начала строки клавишей «*Tab*». Структура правил *Makefile* может быть очень сложной. Там могут присутствовать переменные, конструкции ветвления, цикла.

Если выполнение какой-либо цели *Makefile* требует выполнения других целей, то последние называются зависимыми целями. Такие цели должны быть указаны для основной цели. Тогда зависимые цели будут выполнены прежде основной.

По мимо обычных целей, в файле *Makefile* могут быть псевдоцели (несвязанные с компиляцией). Такие цели могут представлять собой аналог скриптов на языке *Bash*. Часто все цели и псевдоцели называют правилами.

Следует отметить, что существует, как минимум, три различных наиболее распространенных варианта утилиты *make*: *GNU make, System V make и Berkeley make*. Во многих случаях *makefile* полностью генерируется специальной программой. Например, для разработки процедур сборки используются программы *autoconf/automake*. Однако в некоторых программах может потребоваться непосредственное создание Файла *makefile* без использования процедур автоматической генерации.

Задание 4.

1. Для упрощения процедуры создания исполняемого файла создадим *Makefile* – цель использования данного файла автоматизация сборки. *Makefile* может содержать следующий текст:

```
all: exe lib
exe:    hello.h main.c lib
        gcc main.c -fPIC -L. -lHello -o hello
lib: hello.h hello.c
        gcc -o libHello.so -shared -fPIC hello.c
clean:
        -rm hello libHello.so
```

   Обратите внимание на строки, введённые с отступом от левого края. Этот отступ получен с помощью клавиши «*Tab*». Если будете использовать клавишу «Пробел», команды не будут исполняться.

   В файле *make* первая строка *all* — это главная цель, она указывает компилятору откуда начинать сборку. В главной цели есть ссылка на зависимые цели, переходя по таким ссылкам компилятор выполняет соответствующие действия и таким образом собирает проект.

   В дополнении в *make* файле есть цель *clean* – эта цель не содержит зависимостей, она нужна для очистки проекта.

2. Запустите *Makefile* файл при помощи команды *make.*

Отметим, что при вызове, первая цель файла будет считаться главной целью. В ином случае главную цель необходимо указать вручную.

3. Проверьте работоспособность программы.
4. Удалите *libHello.so* и запустите команду *make lib,* проверьте работоспособность и объясните, что произошло.
5. Для удаления проекта используйте команду *make clean.*
6. Дополните *Makefile* правилами *install* и *uninstall,* так, чтобы правило *install* перемещало созданное приложение в стандартный каталог */usr/local/bin/,* а правило *uninstall* удаляло его.
7. Скомпилируйте и установите (*install*) созданную программу в режиме суперпользователя.
8. Проверьте, что программа запускается из домашней директории.
9. Удалите программу и проверьте ее отсутствие в стандартном каталоге.

Отметим, что если необходимо проинспектировать проект на предмет того, что он использует, то можно воспользоваться утилитой *ldd.* Например, для созданного проекта *ldd hello*. Динамический компоновщик *LDD* ищет библиотеки только в известных ему каталогах. Для того, чтобы добавить директорию с библиотекой в список известных директорий, надо редактировать файл */etc/ld.so.conf* или отредактировать переменную *LD_LIBRARY_PATH* (*LD_LIBRARY_PATH* = . для добавления текущего каталога).

Задание 5.

1. Создайте *Makefile* с использованием объектных файлов из задания 2.
2. Проверьте комплириуемость и работоспособность созданной программы.

Отладка приложений в командной строке.

В среде *GNU/Linux* наибольшее распространение получил отладчик *GDB* (*GNU Debugger*). Работа *GDB* осуществляется в командном режиме, однако существуют надстройки, интегрирующие *GDB* в *IDE* (например, *Eclipse*). Отладчик *GDB* может запустить любую программу, записанную в формат*ах «.out, COFF, ECOFF, XCOFF, ELF, SOM»*. Запустив отладку пользователь имеет возможность посмотреть состояние ресурсов, а также получить информацию об исполняемом коде программы, по крайней мере в виде ассемблерных инструкций (если в отладочной информации не указан исходник). Отметим, что для указания исполняемого файла при компиляции *GCC* следует указать опцию *–g* (или *g<0,1,2,3>* -с различными режимами информация для отладки, *-g3* –наиболее детальная).

Также отладочную информацию можно выделить из приложения при помощи утилиты *objcopy*.

Отладчик *GDB* обладает широким перечнем возможностей. Для запуска отладчика необходимо воспользоваться утилитой *GDB*, например,

- для отладки приложения *hello*:

  GDB ./hello;

- если *GDB* запускается с флагом *–c core* – то загрузка приложения происходит в определенный момент (описанный в файле *core file (core dump)*, для создания *core dump* можно воспользоваться утилитой *ulimit (флаг –c unlimited));*
- если *GDB* запускается с флагом *–pid <process-id>*, то можно подключиться к действующему процессу по его *id*.

Находясь в консоли *GDB* можно использовать специальные команды. Например,

- команда *run (r)* запускает процесс;
- *c (continue)* продолжить,
- *q(quit)* заверишь.
- для движения по запущенному процессу можно использовать команды *step; next; until;*
- для работы с точками остановки используется команд *break*, например, *break myfunction*; *break + (break -)* или число линий – для относительной установки точек;
- для просмотра листинга используется команда *list, list +-n* – тогда n строк от текущей.

Задание 6.

1. Создайте проект со следующим кодом:

```
#include <stdio.h>
int *f(){
    return (int *)12;
}
void print (int *v){
    printf("value=%d\n",*v);
}
int main(void){
    printf("Hello WORLD!\n");
    print(f());
}
```

2. Включите утилиту *ulimit –c unlimited* для создания *core files*.
3. Скомпилируйте программу с флагом *g3* и проверьте ее работоспособность.
4. Проверьте созданный файл *core*.
5. Запустите отладчик *gdb –c core ./<project>* и проверьте где остановилась программа, затем выйдите из *gdb*.
6. Запустите *gdb ./<project>* и поставьте точку остановки на функции *print*.
   например, команда *break print*.
7. Проверьте, что у вас поставлена точка (например, команда *info breakpoints*).
8. Запустите процесс, отметьте, где он остановился.
9. Проверьте откуда был последний вызов при помощи команды *frame (f(1))*.
10. Поставьте точки остановки перед крахом приложения, перезапустите приложение, перейдите к точке остановки и пройдите от нее далее по шагам.
11. Завершите процесс отладки командой *kill*.

Задание 7.

Проверьте разницу между компиляцией *Cu* и *Cpp*

1. Создайте новый проект с файлами *hello.c, hello.h* и *main.c.*
2. Переименуйте *hello.c* в *hello.cpp.*
3. Скомпилируйте библиотеку и проверьте ее содержание.

Отметим, что в C++ соглашения об именах включают указание типа.

Для того, чтобы узнать прототип полученного имени можно воспользоваться утилитой c++filt

4. Найдите получившиеся имя функции и ее прототип.
5. Для того, чтобы функция вызывалась как прототип в C++ нужно добавить *extern "C"* перед объявлением ее прототипа в head файле. Попробуйте собрать и запустить проект с использованием *extern "C".*

Задание 8.

1. Разобраться с тем, что делает следующий код, написать комментарии.

```
#include <stdio.h>
#include <stdbool.h>
#include <stddef.h>
#include <dlfcn.h>
void (*hello_message)(const char*);
bool init_library(){
        void *hdl = dlopen("./libHello.so",RTLD_LAZY);
```

```
        if (NULL==hdl)
                return false;
        hello_message = (void (*)(const char *))dlsym(hdl, "hello_message");
        if (NULL== hello_message)
                return false;
        return true;
}
int main(void){
        if (init_library())
                hello_message("Hello World \n");
        else
                printf("Library not found \n");
        return 0;
}
```