2022

Лабораторная работа #3

Автоматическая сборка много-файловых проектов

(Краткая теория, задиния)

РАЗРАБОТАЛ

СЕРГЕЙ СТАНКЕВИЧ (SERHEY STANKEWICH, MINSK, BELARUS)

Operating Systems and System Programming

ЛАБОРАТОРНАЯ РАБОТА #3

Компиляция и отладка простейшего приложения в Linux. Автоматическая сборка многофайловых проектов

Цель работы

Изучить встроенный инструментарий для разработки приложений под семейство ОС Linux и фундаментальные основы системного программирования с использованием компиляторов ***gcc/g++***, отладчика ***gdb*** и других для проектирования, компиляции, отладки и запуска приложений на языке программирования C/C++.

Научиться грамотно проектировать и разрабатывать многофайловые проекты на языке программирования C/C++ с использованием архитектурного шаблона проектирования и разработки гибких и масштабируемых приложений Model-View-Controller (MVC).

Научиться эффективно использовать специальные средства для автоматизации процесса компиляции, сборки и запуска многофайловых проектов.

Краткие теоретические сведения.

Создание программы (технический уровень)

Создание любой программы начинается с постановки задачи, проектирования и написания исходного кода (source code). Обычно исходный код программы записывается в один или несколько файлов, которые называют *исходными файлами* или *исходниками*.

Исходные файлы обычно создаются и набираются в текстовом редакторе.

Чтобы запустить программу, ее необходимо сначала перевести с понятного человеку исходного кода в понятный компьютеру исполняемый код. Такой перевод называется компиляцией (compilation).

Компиляция исходных текстов на Си в исполняемый файл происходит в несколько этапов.

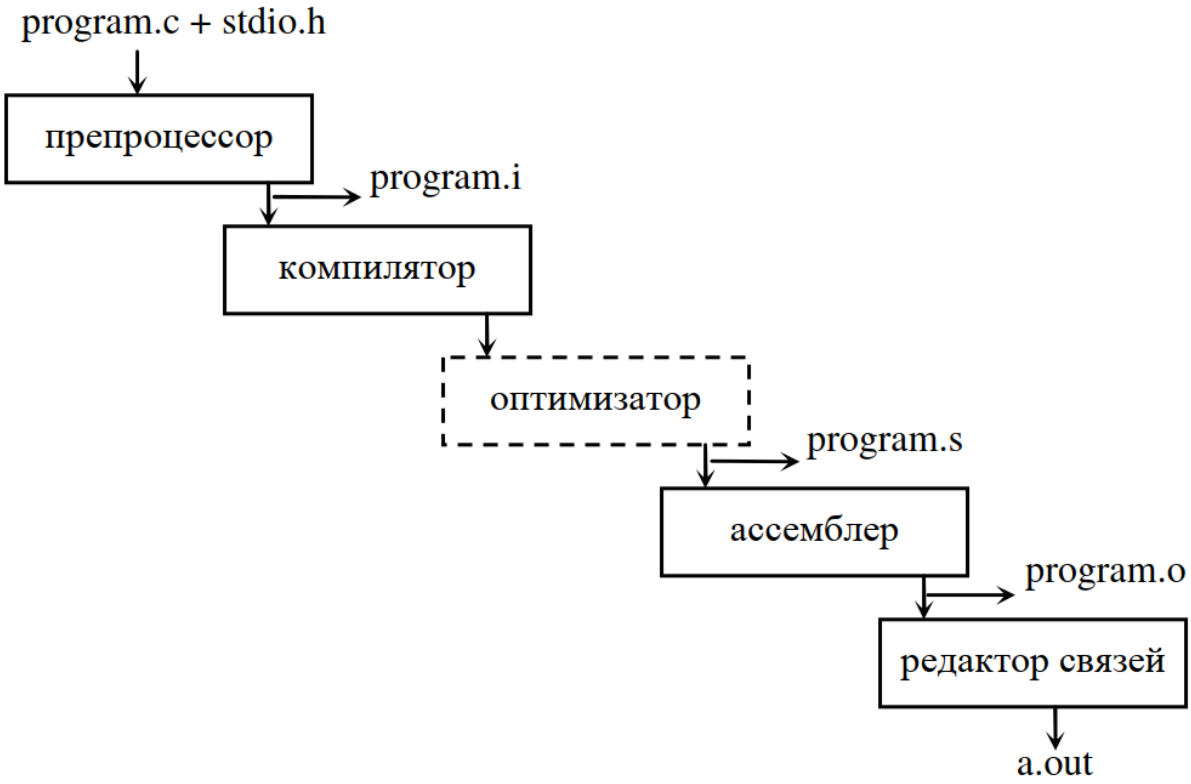

Рисунок 1 – Процесс генерации исполняемого файла по исходному тексту

В результате получается исполняемый (бинарный) код. Файл, содержащий исполняемый код, обычно называют *исполняемым файлом* или *бинарником* (binary).

Инструментарий

Прохождение каждого такого этапа требует наличия определенных инструментов, совокупный набор которых называется *инструментарием*. Базовым и наиболее часто используемым инструментарием Linux-программиста являются: *текстовый редактор*, *компилятор* и *компоновщик*.

Си-компилятор является основным средством для создания программ в системе UNIX. Переоценить его значение для системы невозможно. Достаточно сказать, что около 90% ядра и почти 100% утилит и библиотек системы UNIX написаны на языке Си.

В системе UNIX Си-компилятор, как правило, состоит из трех программ: *препроцессора*, *синтаксического анализатора* и *генератора кода*.

Результатом его работы является программа на языке ассемблера, которая затем транслируется в объектный файл, компонуемый с другими модулями загрузчиком. В итоге образуется выполняемая программа.

Исходные файлы

К исходным файлам существуют некоторые требования.

Каждый файл должен быть целостным, т. е. не должен содержать незавершенных конструкций. Функции и структуры не должны разрываться. Если в рамках проекта предполагается создание исполняемой программы, то в одном из исходных файлов должна присутствовать функция main().

В программах на языке C, C++ рекомендуется использование заголовочных файлов. У заголовочных файлов особая роль: они устанавливают соглашения по использованию общих идентификаторов (имен) в различных частях программы. Если, например, функция **func()** реализована в файле **a.c**, а вызывается в файле **b.c**, то в оба файла требуется включить директивой #include заголовочный файл, содержащий объявление (прототип) нашей функции.

Технически можно обойтись и без заголовочных файлов, но в этом случае функцию можно будет вызвать с произвольными аргументами, и компилятор, за отсутствием соглашений, не выведет ни одной ошибки. Подобный "слепой" подход потенциально опасен и в большинстве случаев свидетельствует о плохом стиле программирования.

Также заголовочные файлы могут добавляться в пакеты распространения программ, так как заголовочные файлы содержат определение функций без их реализации (аналог интерфейсов в других языках), а также комментарии к ним.

Аргументы программы

Наилучшим способом разработки и поддержки больших программ является конструирование программы из небольших частей или *модулей*. Простейшие модули в C (C++) называются *функциями*. Функции могут принимать или выдавать значения. Эти значения называются параметрами функций. *Параметры функции* – это значения переменных, которые позволяют функциям обмениваться информацией. Типы входных параметров определены аргументами функций при определении функций. Таким образом *аргументами функций* являются определения типов передаваемых значений, а параметрами функций являются непосредственно передаваемые значения.

Программа, созданная на языке C (C++), может получать данные извне, посредством чтения передаваемых параметров аргументами функции **main()**. Аргументы функции определяются в прототипах функций. Прототип (объявление) функции main() можно представить следующим образом:

```
int main (int argc, char ** argv)
```

Благодаря этому программа может принимать и обрабатывать переданные ей данные из консольной строки или другой программы (процесса). Не следует отождествлять понятия "аргументы программы" и "аргументы командной строки". Дело в том, что программа получает свои аргументы от родительского процесса, в качестве которого не всегда выступает командная оболочка. Таким процессом может оказаться, например, браузер, в котором никакой "командной строки" нет. Обмен данными между процессами мы основательно изучим при рассмотрении большой темы «**Межпроцессное взаимодействие**» (IPC).

Если в программе не предусмотрено чтение аргументов, то функцию main() можно объявить еще проще:

```
int main (void)
```
или даже так
```
int main ()
```

Возможность подобных вариаций с прототипами обусловлена отсутствием единого объявления для функции main(). Работая с ее аргументами, компилятор рассчитывает лишь на "здравый рассудок" программиста.

Для того чтобы программа могла принимать параметры, передаваемые из командной строки как правило применяется следующая конструкция:

```
int main (int argc, char ** argv)
{ ...
        if (argc < 2) {
                fprintf (stderr, "Too few arguments\n");
                return 1;
} ...
```

Вместо "исторических" имен argc и argv можно выбрать любые другие. Но имена argc и argv настолько укоренились в культуре программирования, что любой опытный программист наверняка скажет, что использование других имен означает дурной стиль программирования.

Процесс компиляции

Процесс компиляции состоит из следующих этапов:

1. **Лексический анализ**. Последовательность символов исходного файла преобразуется в последовательность лексем.
2. **Синтаксический анализ**. Последовательность лексем преобразуется в дерево разбора.
3. **Семантический анализ**. Дерево разбора обрабатывается с целью установления его семантики (смысла) – например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т.д.
4. **Оптимизация**. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла.
5. **Ассемблирование**. Промежуточное представление кода на языке ассемблера.
6. **Генерация кода**. Из промежуточного представления порождается объектный код. Результатом генерации является *объектный код*.
7. **Связывание**. Это последний этап, который либо ведет к получению исполняемого файла.

Препроцессор

Первым этапом работы компилятора является обработка исходного кода препроцессором языка C. Препроцессор выполняет 3 операции: **текстовые замены**, **вырезание комментариев** и **включение файлов**.

Текстовые замены и включения файлов запрашиваются программистом с помощью директив препроцессора. Директивы препроцессора – это строки, начинающиеся с символа "#".

Директивы препроцессора **#include** это постоянно используемая директива. Она создает копию указанного файла, которая включается в программу вместо директивы. Существует две формы использования директивы **#include**:

```
#include <filename>
#include “filename”
```

Разница между ними заключается в том, где компилятор будет искать файлы, которые необходимо включить. Если имя заключено в кавычки (“ и ”), система ищет его в том же каталоге, что и компилируемый файл. Такую запись обычно используют для включения определенных пользователем заголовочных файлов. Если же имя файла заключено в угловые скобки (< и >), это используется

для файлов стандартной библиотеки, то поиск будет вестись в зависимости от конкретной реализации компилятора, обычно в предопределенных операционной системой каталогах.

Рассмотрим пример:

```
#include <stdio.h>

// Это комментарий.

#define STRING "This is a test"
#define COUNT (5)

int main ()
{
  int i;

  for (i=0; i<COUNT; i++)
  {
        puts(STRING);
  }

  return 1;
}
```

Первая директива в нашем примере подключает стандартный заголовочный файл **stdio.h**, его содержимое подставляется в наш исходный файл. Вторая и третья директивы заменяют строки в нашем коде.

Запустив gcc с ключом "-E", можно остановить его работу после первого этапа и увидеть результаты работы препроцессора над нашим кодом.

`gcc -E test.c -o test.i` так `gcc -E test.c > test.txt` или так `gcc -i test.c`

флаги -i, -ii зависят от языка программирования, C или C++ соответственно.

Файл **stdio.h** довольно велик, поэтому опустим некоторые ненужные для нас строки.

```
# 1 "test.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 330 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 348 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 349 "/usr/include/sys/cdefs.h" 2 3 4
# 331 "/usr/include/features.h" 2 3 4
# 354 "/usr/include/features.h" 3 4
# 1 "/usr/include/gnu/stubs.h" 1 3 4

# 653 "/usr/include/stdio.h" 3 4
extern int puts (__const char *__s);
```

```
int main ()
{
    int i;

    for (i=0; i<5; i++)
    {
        puts("This is a test");
    }

    return 1;
}
```

Препроцессор дописал к нашей простой программе много новых строк. А сколько строк в файле, полученном в результате препроцессинга вашей программы?

Мы запросили у препроцессора включение заголовочного файла stdio.h в нашу программу. В свою очередь, stdio.h запросил включение других заголовочных файлов, и так далее. Препроцессор сделал отметки о том, включение какого файла и на какой строке было запрошено. Эта информация будет использована на следующих этапах компиляции. Так, строки

```
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
```

означают, что файл features.h был запрошен на строке 28 файла stdio.h. Препроцессор создает эту отметку перед соответствующим "интересным" местом, так что, если встретится ошибка, компилятор сможет нам сообщить, где именно произошла ошибка.

Теперь посмотрим на эти строки:

```
# 653 "/usr/include/stdio.h" 3 4
extern int puts (__const char *__s);
```

Здесь **puts()** объявлена как внешняя функция (extern), возвращающая целочисленной значение и принимающая массив постоянных символов в качестве параметра. Если бы случилась несостыковка, касающаяся этой функции, тогда компилятор смог бы сообщить нам, что данная функция была объявлена в файле stdio.h на строке 653. Интересно, что на данном этапе функция puts() не определена, а лишь объявлена. Здесь пока нет реального кода, который будет работать при вызове функции puts(). Определение функций будет происходить позже.

Также обратите внимание на то, что все комментарии были удалены препроцессором, и произведены все запрошенные текстовые замены. Можно сделать вывод, что препроцессор нужен для исследования кода программ и его зависимостей от других библиотек.

В данный момент программа готова к следующему этапу, трансляции на язык ассемблера.

Ассемблирование

Ассемблирование – это процесс преобразования программы на язык ассемблера. Результат трансляции можно увидеть с помощью ключа -S:

gcc -S test.c или так gcc -S test.i -o test.s

Будет создан файл с именем **test.s**, содержащий реализацию нашей программы на языке ассемблера.

```
        .file   "test.c"
        .section        .rodata
.LC0:
        .string "This is a test"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $20, %esp
        movl    $0, -8(%ebp)
        jmp     .L2
.L3:
        movl    $.LC0, (%esp)
        call    puts
        addl    $1, -8(%ebp)
.L2:
        cmpl    $4, -8(%ebp)
        jle     .L3
        movl    $1, %eax
        addl    $20, %esp
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.2.4 (Gentoo 4.2.4 p1.0)"
        .section        .note.GNU-stack,"",@progbits
```

Я не силен в языке ассемблера, однако некоторые моменты можно выделить сразу. Строка сообщения была перемещена в другую область памяти и стала называться .LC0. Основную часть кода занимают операции, от начала выполнения программы и до ее завершения. Очевидна реализация цикла for на метке .L2: это просто проверка (cmpl) и инструкция "переход, если меньше"

("Jump if Less Than", jle). Инициализация цикла осуществляется оператором movl перед меткой .L3. Между метками .L3 и .L2 очевиден вызов функции puts(). Ассемблер знает, что вызов функции puts() по имени здесь корректен, и что это не метка памяти, как например .L2. Обсудим этот механизм далее, когда будем говорить о заключительном этапе компиляции, связывании. Наконец, наша программа завершается операцией возвращения (ret).

Генерация объектного кода

Объектный код – это программа на языке машинных кодов с частичным сохранением символьной информации, необходимой в процессе сборки.

При отладочной сборке возможно сохранение большого количества символьной информации (идентификаторов переменных, функций, а также типов).

Если указать компилятору, чтобы он пропустил этап связывания, с помощью ключа -c и -o:

```
gcc -c test.c -o test.o
```

тогда мы получим объектный файл размеров всего 888 байт. Если скомпилировать нашу программу с этапом связывания, мы получим исполняемый файл размером 6885 байт. Разницу составляет как раз код для запуска и завершения программы, а также вызов функции puts() из библиотеки libc.so.

Связывание (линковка)

Связывание ведет к получению исполняемого файла, либо объектного файла, который можно объединить с другим объектным файлом, и таким образом получить исполняемый файл. Проблема с вызовом функции puts() разрешается именно на этапе связывания. Помните, в stdio.h функция puts() была объявлена как внешняя функция? Это и означает, что функция будет определена (или реализована) в другом месте. Если бы у нас было несколько исходных файлов нашей программы, мы могли бы объявить некоторые функции как внешние и реализовать их в различных файлах; такие функции можно использовать в любом месте нашего кода, ведь они объявлены как внешние. До тех пор, пока компилятор не знает, откуда берется реализация такой функции, в получаемом коде лишь остается ее "пустой" вызов. Линковщик разрешит все эти зависимости и в процессе работы подставит в это "пустое" место реальный адрес функции.

Линковщик выполняет также и другую работу. Он соединяет нашу программу со стандартными процедурами, которые будут запускать нашу программу. К

примеру, есть стандартная последовательность команд, которая настраивает рабочее окружение, например, принимает аргументы командной строки и переменные системного окружения. В завершении программы должны также присутствовать определенные операции, чтобы помимо всего прочего программа могла возвратить код ошибки. Очевидно, эти стандартные операции порождают немалое количество кода.

Компиляция простой программы

Компилятором языка C в Linux обычно служит программа gcc (GNU C Compiler) из пакета компиляторов GCC (GNU Compiler Collection). Чтобы откомпилировать программу, следует вызвать gcc, указав в качестве аргумента имя исходного файла:

```
$ gcc myclock.c
```

Если компилятор не нашел ошибок в исходном коде, то в текущем каталоге появится файл **a.out**. Во времена компьютера PDP, a.out означало "assembler output" - "вывод ассемблера". В данный момент это означает просто старый формат исполняемого файла. В современных версиях Unix и Linux используется формат **ELF** (*Executable and Linkable Format*). Он намного более сложен. Хотя получившийся файл и носит имя "a.out", но на самом деле исполняемый файл имеет формат ELF.

Теперь, чтобы выполнить программу, требуется указать командной оболочке путь к исполняемому файлу. Поскольку текущий каталог обычно обозначается точкой, то запуск программы можно осуществить следующим образом:

```
$ ./a.out
Wed Nov  8 03:09:01 2006
```

Исполняемые файлы программ обычно располагаются в каталогах, имена которых перечислены через двоеточие в особой переменной PATH. Чтобы просмотреть содержимое этой переменной, введите следующую команду:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/lib/qt4/bin
```

Если бинарник находится в одном из этих каталогов, то для запуска программы достаточно ввести ее имя (например, ls). В противном случае потребуется указание пути к исполняемому файлу.

Имя a.out не всегда подходит для программы. Один из способов исправить положение – просто переименовать полученный файл:

```
$ mv a.out myclock
```

Но есть способ лучше. Можно запустить компилятор с опцией -o, которая позволяет явно указать имя файла на выходе:

```
$ gcc -o myclock myclock.c
```

В достаточно объемных программах исходный код обычно разделяется для удобства на несколько частей, которые компилируются отдельно, а затем соединяются воедино. Каждый такой "кусок" содержит объектный код и называется объектным модулем.

Объектные модули записываются в объектные файлы, имеющие расширение **.o**. В результате объединения объектных файлов могут получаться исполняемые файлы (обычные запускаемые бинарники), а также библиотеки, о которых пойдет речь в следующей лабораторной работе.

Для объединения объектных файлов служит *компоновщик* (*линковщик*), а сам процесс называют компоновкой или линковкой. В Linux имеется компоновщик GNU ld, входящий в состав пакета GNU binutils. Иногда компоновщик называют также *загрузчиком*.

Ручная компоновка объектных файлов — довольно неприятный процесс, требующий передачи программе ld большого числа параметров, зависящих от многих факторов. К счастью, компиляторы из коллекции GCC сами вызывают линковщик с нужными параметрами, когда это необходимо. Программист может самостоятельно передавать компоновщику дополнительные параметры через компилятор.

Сборка многофайловых проектов

Современные программные проекты не ограничиваются одним исходным файлом. Распределение исходного кода программы на несколько файлов имеет ряд существенных преимуществ перед однофайловыми проектами:

- Использование нескольких исходных файлов накладывает на репозиторий (рабочий каталог проекта) определенную логическую структуру. Такой код легче читать и модернизировать.
- В однофайловых проектах любая модернизация исходного кода влечет повторную компиляцию всего проекта. В многофайловых проектах,

напротив, достаточно откомпилировать только измененный файл, чтобы обновить проект. Это экономит массу времени.

- Многофайловые проекты позволяют реализовывать одну программу на разных языках программирования.
- Многофайловые проекты позволяют применять к различным частям программы разные лицензионные соглашения.

Обычно процесс сборки многофайлового проекта осуществляется по следующему алгоритму:

1. Создаются и подготавливаются исходные файлы.
2. Создаются и подготавливаются заголовочные файлы.
3. Каждый исходный файл отдельно компилируется с опцией -c. В результате появляется набор объектных файлов.
4. Полученные объектные файлы соединяются компоновщиком в одну исполняемую программу.

Для определения формата файла и его содержимого можно использовать команду file. Например,

```
$ file /etc/shadow
```

Принцип раздельной компиляции

Компиляция, алгоритмически сложный процесс, для больших программных проектов требующий существенного времени и вычислительных возможностей ЭВМ. Благодаря наличию в процессе сборки программы этапа компоновки (связывания) возникает возможность *раздельной компиляции*.

В модульном подходе программный код разбивается на несколько файлов **.c** (**.cpp**), каждый из которых компилируется отдельно от остальных. Такие стадии компиляции как препроцессинг, ассемблирование и создание объектных файлов транслируются отдельно для каждого исполняемого файла.

Это позволяет значительно уменьшить время перекомпиляции при изменениях, вносимых лишь в небольшое количество исходных файлов. Также это даёт возможность замены отдельных компонентов конечного программного продукта, без необходимости пересборки всего проекта.

Рассмотрим пример. Если необходимо скомпоновать несколько объектных файлов (OBJ1.o, OBJ2.o и т. д.), то применяют следующий простой шаблон:

```
$ gcc -o OUTPUT_FILE OBJ1.o OBJ2.o ...
```

Например, программа содержит несколько файлов: **print_up.c**, **main.c**, и заголовочный файл **print_up.h**.

Нужно собрать проект воедино. Сначала откомпилируем каждый файл с расширением **.c**:

```
$ gcc -c print_up.c
$ gcc -c main.c
```

В результате компиляции в репозитории программы должны появиться объектные файлы print_up.o и main.o, которые следует скомпоновать в один бинарный файл:

```
$ gcc -o printup print_up.o main.o
```

Если компилятор gcc вызывается с опцией -c, но без опции -o, то имя выходного файла получается заменой расширения **.c** на **.o**. Например, из файла **foo.c** получается файл **foo.o**.

Осталось только запустить и протестировать программу:

```
$ ./printup
```

Обратите внимание на то, что заголовочный файл print_up.h не компилируется. Заголовочные файлы вообще никогда отдельно не компилируются. Дело в том, что на стадии *препроцессирования* (условно первая стадия компиляции) все директивы **#include** заменяются на содержимое указанных в них файлов.

Автосборка

Сборкой называется процесс подготовки программы к непосредственному использованию. Простейший пример сборки — компиляция и компоновка. Более сложные проекты могут также включать в себя дополнительные промежуточные этапы (операции над файлами, конфигурирование и т. п.). Скриптовые языки программирования, позволяют запускать программы сразу после подготовки исходного кода, минуя стадию сборки. Примером такого языка является Python.

Собирать программы вручную неудобно, поэтому программисты, как правило, прибегают к различным приемам, позволяющим автоматизировать этот процесс. Самый простой способ – написать сценарий оболочки (shell-скрипт), который будет автоматически выполнять все то, что вы обычно вводите вручную.

Пример, как можно собрать многофайловый проект при помощи скрипта **make_printup**:

```
#!/bin/sh
gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Осталось только вызвать скрипт make_printup, и проект будет создан:

```
$ ./make_printup
```

Перечислим некоторые проблемы использования скриптов для сборки проектов:

- Скрипты оболочки статичны. Их работа не зависит от состояния текущей задачи. Даже если нужно заново откомпилировать только один файл, скрипт будет собирать проект "с нуля".
- В скриптах плохо просматриваются связи между различными элементами проекта.
- Скрипты не обладают возможностью самодиагностики.

К счастью, в наличии имеется достаточно большой арсенал специализированных средств для автоматической сборки программных проектов. Такие средства называют автосборщиками или утилитами автоматической сборки. Благодаря специализированным автосборщикам программист может сосредоточиться, собственно, на программировании, а не на процессе сборки.

Все утилиты автосборки в Linux можно разделить на несколько условных категорий.

- **Семейство make** — это различные реализации стандартного для Unix-подобных систем автосборщика make. Из представителей данного семейства наибольшей популярностью в Linux пользуются утилиты GNU make, imake, pmake, cmake, fastmake и tmake.
- **Надстройки над make** — утилиты семейства make работают с особыми файлами, в которых содержится вся информация о сборке проекта. Такие файлы называют make-файлами (makefiles). При работе с классическими make-утилитами эти файлы создаются и редактируются вручную. Надстройки над make автоматизируют процесс создания make-файлов. Наиболее популярные представители этого семейства в Linux — утилиты из пакета GNU Autotools (automake, autoconf, libtool и т. д.).

- **Специализированные автосборщики** — обычно такие утилиты создаются для удобства при работе с определенными проектами. Типичный представитель этого семейства — утилита qmake (Qt make) — автосборщик для проектов, использующих библиотеку Qt. Библиотека Qt была создана норвежской компанией Trolltech, но в настоящее время принадлежит корпорации Nokia.

Мы будем пользоваться самой популярной в Linux утилитой автосборки GNU make.

Утилита make

Утилита make (GNU make) — наиболее популярное и проверенное временем средство автоматической сборки программ в Linux. Даже "гигант" автосборки, пакет GNU Autotools, является лишь надстройкой над make.

Автоматическая сборка программы на языке C обычно осуществляется по следующему алгоритму.

1. Подготавливаются исходные и заголовочные файлы.

2. Подготавливаются make-файлы, содержащие сведения о проекте. Порой даже крупные проекты обходятся одним make-файлом. Вообще говоря, make-файл может называться, как угодно, однако обычно выбирают одно из трех стандартных имен (Makefile, makefile или GNUmakefile), которые распознаются автосборщиком автоматически.

3. Вызывается утилита make, которая собирает проект на основании данных, полученных из make-файла. Если в проекте используется нестандартное имя make-файла, то его нужно указать после опции -f при вызове автосборщика.

Разработчики GNU make рекомендуют использовать имя Makefile. В этом случае у вас больше шансов, что make-файл будет стоять обособленно в отсортированном списке содержимого репозитория.

Базовый синтаксис Makefile

Чтобы работать с make, необходимо создать файл с именем Makefile. В make-файлах могут присутствовать следующие конструкции:

- **Комментарии**. В make-файлах допустимы однострочные комментарии, которые начинаются символом # (решетка) и действуют до конца строки.

- **Объявления констант**. Константы в make-файлах служат для подстановки. Они во многом схожи с константами препроцессора языка C.
- **Целевые связки**. Эти элементы несут основную нагрузку в make-файле. При помощи целевых связок задаются зависимости между различными частями программы, а также определяются действия, которые будут выполняться при сборке программы. В любом make-файле должна быть хотя бы одна целевая связка.

В Makefile обязательны только целевые связки. Каждая целевая связка состоит из следующих компонентов:

- Имя цели. Если целью является файл, то указывается его имя. После имени цели следует двоеточие.
- Список зависимостей. Здесь просто перечисляются через пробел имена файлов или имена промежуточных целей. Если цель ни от чего не зависит, то этот список будет пустым.
- Инструкции. Это команды, которые должны выполняться для достижения цели.

Рассмотрим пример. Сначала создаем Makefile программы printup:

```
# Makefile for printup
printup: print_up.o main.o
	gcc -o printup print_up.o main.o

print_up.o: print_up.c print_up.h
	gcc -c print_up.c
main.o: main.c
	gcc -c main.c

clean:
	rm -f *.o
	rm -f printup
```

Строки после комментария, это целевые связки.

Первая связка отвечает за создание исполняемого файла printup и формируется следующим образом:

1. Сначала записывается имя цели (printup).

2. После двоеточия перечисляются зависимости (print_up.o и main.o).

3. На следующей строке после **знака табуляции** пишется правило для получения бинарника printup.

Аналогичным образом оформляются остальные целевые связки.

Вызовем утилиту make с указанием цели, которую нужно достичь. В нашем случае это будет выглядеть так:

```
$ make printup

gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Итак, проект собран.

Вообще говоря, при запуске make имя цели можно не указывать. Тогда основной целью будет считаться первая цель в Makefile. Следовательно, в нашем случае, чтобы собрать проект, достаточно вызвать make без аргументов:

```
$ make

gcc -c print_up.c
gcc -c main.c
gcc -o printup print_up.o main.o
```

Целевая связка (**clean**) требует особого рассмотрения:

1. Сначала указывается имя цели (clean).
2. После двоеточия следует пустой список зависимостей. Это значит, что данная связка не требует наличия каких-либо файлов и не предполагает предварительного выполнения промежуточных целей.
3. На следующих двух строках прописаны инструкции, удаляющие объектные файлы и бинарник.

Эта цель очищает проект от всех файлов, автоматически созданных при сборке. Итак, чтобы очистить проект, достаточно набрать следующую команду:

```
$ make clean

rm -f *.o
rm -f printup
```

Очистка проекта обычно выполняется в следующих случаях:

- при подготовке исходного кода к отправке конечному пользователю или другому программисту, когда нужно избавить проект от лишних файлов;

- при изменении или добавлении в проект заголовочных файлов;
- при изменении make-файла.

Иногда требуется вписать в make-файл нечто длинное, например инструкцию, не умещающуюся в одной строке. В таком случае строки условно соединяются символом \ (обратная косая черта):

```
gcc -Wall -pedantic -g -o my_very_long_output_file one.o two.o \
three.o four.o five.o
```

Автосборщик при обработке make-файла будет интерпретировать такую конструкцию как единую строку.

Константы make

В make-файлах для параметризации процесса сборки можно использовать константы. Для объявления и инициализации констант предусмотрен следующий шаблон:

```
NAME=VALUE
```

Здесь NAME — это имя константы, VALUE — ее значение. Имя константы не должно начинаться с цифры. Значение может содержать любые символы, включая пробелы. Признак окончания значения константы — конец строки. Иначе говоря, любые символы, стоящие между знаком "равно" и символом переноса строки, будут являться значением константы.

Значение константы можно подставить в любую часть make-файла (кроме комментария). Если имя константы состоит из одного символа, то для подстановки достаточно добавить перед именем литеру $ (доллар). Когда имя состоит из нескольких символов, для подстановки применяется следующий шаблон:

```
$(NAME)
```

Утилита make поддерживает также целый ряд специализированных констант. Две из них используются в целевых связках и представляют особый интерес:

- $@ — содержит имя текущей цели;
- $^ — содержит список зависимостей в текущей связке.

Использование констант не только уменьшает в размере Makefile, но и делает его более гибким. При изменении целей или списков зависимостей не требуется параллельно изменять инструкции.

Можно усовершенствовать Makefile до такой степени, что для добавления в проект нового исходного файла достаточно будет дописать его имя в константу.

Рекурсивный вызов make

Иногда программные проекты разделяют на несколько независимых подпроектов. В этом случае каждый подпроект имеет свой make-файл. Но рано или поздно понадобится все соединить в один большой проект. Для этого используется концепция рекурсивного вызова make, предполагающая наличие главного make-файла, который инициирует автосборку каждого подпроекта, а затем объединяет полученные файлы.

Посредством опции **-C** можно передать автосборщику make имя каталога, в котором следует искать Makefile. Это позволяет вызвать make для каждого подпроекта из главного make-файла. Чтобы понять, как это осуществляется на практике, рассмотрим пример многокомпонентного программного проекта.

Любой программный проект можно разделить на подпроекты, используя концепцию *рекурсивного вызова* make. Подобным образом, например, разрабатывается ядро Linux: в настоящее время в его исходниках насчитывается более 1400 make-файлов!

Best of LUCK with it, and remember to HAVE FUN while you're learning :)
Sergey Stankewich

УПРАЖНЕНИЯ

Упражнение 1

Изучите и выполните требования, представленные в **лабораторной работе №2** «Компиляция и отладка простейшего приложения в Linux». Разработайте исходные коды приложений в соответствии с заданиями. Компиляцию приложений проведите с помощью консольных команд.

Попробуйте написать похожий скрипт:

```
#!/bin/sh
gcc -E test.c -o test.i
gcc -S test.i -o test.s
gcc -c test.s
gcc -o testprogram test.o
```

При разработке приложений не забудьте отразить **информацию о разработчиках**.

Файлы с содержанием лабораторной работы предоставлены.

Упражнение 2

Изучите и выполните требования, представленные в **лабораторной работе №3** «Многофайловые проекты и шаблон MVC». Разработайте исходные коды приложений в соответствии с заданиями.

Компиляцию приложений проведите с помощью консольных команд.

При разработке приложений не забудьте отразить **информацию о разработчиках**.

Файлы с содержанием лабораторной работы предоставлены.

Упражнение 3

Изучите и выполните требования, представленные в **лабораторной работе №4** «Автоматическая сборка многофайловых проектов».

При разработке приложений не забудьте отразить **информацию о разработчиках**.

Файлы с содержанием лабораторной работы предоставлены.

We hope you enjoy working with Linux!

ЗАДАНИЯ

Задание 1

Используя материалы (исходные файлы) **упражнения №1** проведите компиляцию однофайлового проекта с прохождением **всех стадий компиляции**. Для ускорения работы примените скрипты **bash**.

Исходные файлы программ обязательно должны содержать **коментарии**. Сборка проекта должна содержать файлы с результатами препроцессинга. Исследуйте фалы препроцессора, найдите в них код своей программы.

Определите **размеры** *исходных*, *препроцессорных, ассемблерных, объектных* и *исполняемых* файлов. С помощью соответствующий консольных команд определите **форматы этих файлов**. Результаты подтвердите скриншотами.

Задание 2

Используя материалы (исходные файлы) **упражнения №1** и **№2** создайте один многофайловый проект, руководствуясь принципом **Single Responsibility Principle**. Каждое задание должно быть представлено отдельным исходным файлом. Для связывания файлов проекта обязательно используйте заголовочные файлы. Все файлы могут располагаться в одной директории.

Все задания должны вызываться консольным меню. Для выхода из приложения предусмотрите отдельный параметр.

Проведите автосборку проект с прохождением всех стадий компиляции с использованием утилиты **make**.

Задание 3

Используя материалы (исходные файлы) **задания №2** создайте один многофайловый проект, руководствуясь принципом **Single Responsibility Principle**. Каждое задание должно быть представлено отдельным исходным файлом. Используйте архитектурный шаблон проектирования MVC. Здесь под шаблоном MVC понимается файлы или группа файлов в трех ***отдельных директориях*** (папках), а не просто три отдельных файла.

Все задания должны вызываться консольным меню. Для выхода из приложения предусмотрите отдельный параметр. Для связывания файлов проекта обязательно используйте заголовочные файлы.

Проведите **рекурсивную** автосборку проект с прохождением всех стадий компиляции с использованием утилиты **make**.

Задание «Бонус»

Проведите рекурсивную автосборку с использованием утилиты **cmake** проекта с исходными файлами **задания №3**.

«Easy things should be easy and hard things should be possible»
«Простые вещи должны быть простыми, а сложные вещи должны быть возможными»

Контрольные вопросы:

1) Перечислите основные инструментарии для разработки программ?
2) Что такое исходные файлы программы?
3) Что такое исполняемые файлы программы?
4) Какой командой можно определить тип файла? Определите типы файлов вашей сборки проекта?
5) Назовите основные стадии компиляции программы?
6) В чем различие компиляции и интерпритации программы?
7) Приведите известные вам примеры компилируемых и интерпритируемых языков программирования?
8) Что такое препроцессор и зачем он нужен?
9) В чем разница между понятиями «*аргумент функции*» и «*параметр функции*» ?
10) В чем разница использования двух форм директивы: **#include <filename>** или **#include “filename”**?
11) Что такое автосборка? Какие приемущества она дает?
12) Какой командой можно определить определить формат файла?
13) Что такое утилита **make**?
14) В чем разница автосборки скриптами **bash** и утилитой **make**?
15)

Дополнительная информация

Подробоно песочница придставлена к книге: Шоттс У. «Командная строка Linux. Полное руководство.» — СПб.: Питер, 2017. — 480 с.: ил. — (Серия «Для профессионалов»). на страницах 225-226.

Иванов Н.Н. И20 Программирование в Linux. Самоучитель. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2012. — 400 с.: ил. **Смотрите страницы 15 – 36, 58 – 65.**

Мэтью, Н. М97 Основы программирования в Linux: Пер. с англ. / Н. Мэтью, Р. Стоуне. — 4-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2009. — 896 е.: ил.

Робачевский А. М. Операционная система UNIX®. - СПб.: 2002. - 528 ил.

Харви Дейтел, Пол Дейтел «Как программировать на С», с.994. **Смотрите страницы 587 – 595**.

Интернет источники

https://librebay.blogspot.com/2016/12/how-to-compile-c-cpp-program-in-Ubuntu.html

http://cs.mipt.ru/cpp/labs/lab1.html#id28

http://rus-linux.net/lib.php?name=/MyLDP/algol/compilation/compilation-1.html

http://linux.yaroslavl.ru/docs/prog/gcc/gcc1-2.html (можно использовать как справочник)