Александр Лавриненко
PSA R&D
Июль 2013

Разработка приложений в Linux. v.0.1.

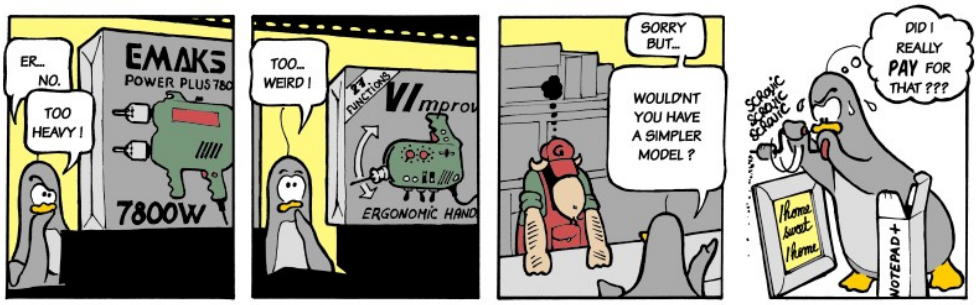
Содержание:

1. Введение

Инструментарий:

IDE:

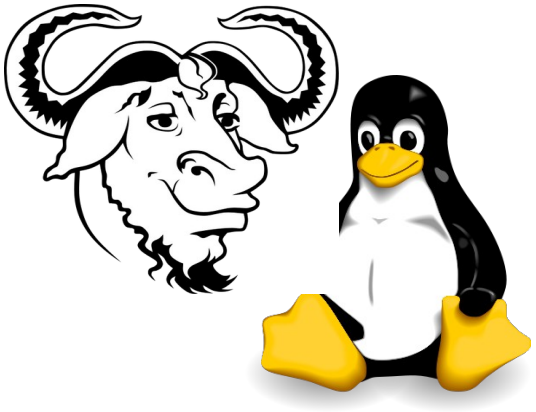
- vim
- emacs
- Eclipse
- Kdevelop
- CodeBloks
- . . .
- mcedit
- notepad



GNU toolchain — набор созданных в рамках проекта GNU пакетов программ, необходимых для компиляции и генерации выполняемого кода из исходных текстов. Являются стандартным средством разработки программ и ядра ОС Linux.

Состав GNU Toolchain:

- заголовочные файлы ядра Linux;
- binutils (компоновщик ld, ассемблер as и другие программы);
- GNU Compiler Collection — набор компиляторов;
- стандартная библиотека языка Си — GNU libc или другая, например uClibc или dietlibc;
- GNU make;
- autotools (autoconf и др.).



Установка:

```
# yum install gcc eclipse-cdt svn git
```

Информация:

```
$ man gcc
$ man 2 системные_вызовы
$ man 3 стандартные_библиотечные_функции
```

пример:

```
$ man sleep
$ man 3 sleep
```

http://ru.wikipedia.org/wiki/GNU_toolchain
http://ru.wikipedia.org/wiki/Binutils

2. Компилляция

Компилятор GCC- это свободно доступный оптимизирующий компилятор для различных языков программирования в том числе для языков C, C++, Java, Fortran, Ada 95, а также Objective C. Его версии позволяют генерировать код для множества процессоров и различных операционных систем.

И является ключевым компонентом GNU toolchain.

GCC используется для компиляции программ в объектные модули и для компоновки полученных модулей в единую исполняемую программу. Компилятор способен анализировать имена файлов, передаваемые ему в качестве аргументов, и определять, какие действия необходимо выполнить.

Процесс компиляции

Не нужно указывать каждый шаг явно, т.к. компиллятор gcc самостоятельно проходит все шаги

Проходят следующие шаги:
- Препроцессор — реализуются макро-расширения;
- Компиляция - создается ассемблерный код;
- Ассемблирование - создается машинный код;
- Компоновка (линковка, связывание) - создается исполняемый файл;

Для сохранения промежуточных файлов для каждого шага можно указать опцию –save-temps, тогда будут доступны файлы с расширениями:

.i для кода на языке C
.ii для кода на языке C++
.s код на языке ассемблера

Маленькая программка на C, hello.c:

```
#include <stdio.h>
int main (int argc, char* argv[]) {
    printf("Hello,world!\n");
    return 0;
}
```

Команда компилирования файла:

```
$ gcc hello.c -o hello
$ ls -l
$ ./hello
```

Компилляция нескольких файлов:

```
$ gcc -c hello_fn.c
$ gcc -c main.c

// Убедимся что объектные файлы сделаны
# ls *.o
hello_fn.o main.o

// Собираем исполняемый файл
# gcc main.o hello_fn.o -o newhello

# ./newhello
```

Полезные флаги:
-v или -### вывод опций по умолчанию во время компиляции;
"-m32" 32 битный режим
-g добавляет глобальные объявления для gdb ;
-OX уровень оптимизации -O0, -O1, -O2, -O3;
-W управление варнингами;

`-Dname=value`

Определить имя `name` в компилируемой программе как значение `value`.
Эффект такой же, как наличие строки `#define name value` в начале программы. Часть `` `=value' `` может быть опущена, в этом случае значение по умолчанию равно 1.

Пример приложения:

```
int main(void) {
#ifdef DEBUG
printf ("Отладка включена.\n");
#endif
printf ("Нормальный запуск.\n");
return 0;
}
```

Пример запуска:

```
$ gcc prep.c
Нормальный запуск.
$ ./a.out

$ gcc -DDEBUG prep.c
Отладка включена.
Нормальный запуск.
```

```
$ ./a.out
```

Использование заголовочных файлов

Опции компиллятора для указания пути к заголовочным файлам:
-I (заглавная i)

```
$ gcc -I/home/user/includes hello.c -o hello
```

gcc ищет заголовочные файлы в следующем порядке:
1. Опции командной строки -I -L слева направо;
2. Переменные окружения:
   C_INCLUDE_PATH;
   CPLUS_INCLUDE_PATH;
3. Расположение по умолчанию в системе:

/usr/include;
/usr/local/include.

Использование библиотек

статические библиотеки (архивы)	**динамические библиотеки**
.a (аналог .lib)	.so (аналог .dll)
gcc calc.c /usr/lib/libm.a -o calc	gcc myfunc.c -L/home/user/libs -lmylibs -o myfunc

Опции компиллятора для указания пути к библиотекам:
-L(каталог с библиотеками)
-l(имя_библиотеки)

Компилятор будет искать библиотеку libmylibs.a или libmylibs.so в каталоге /home/user/libs.

В большинстве систем библиотеки находятся в /lib, /lib64, /usr/lib, /usr/lib64.
По умолчанию компилятор просматривает /usr/lib и /usr/lib/local.

Поиск и установка библиотеки и заголовочных файлов, SDK.

```
//Поиск библиотек в репозитории
# yum list > rpmlist.txt
# grep имя_библиотеки-libs  rpmlist.txt
# grep имя_библиотеки-devel rpmlist.txt


# yum search  имя_библиотеки
# yum install  rpm-пакет
```

```
# rpm -ql rpm-пакет            // список файлов в пакете
```

Пример:

```
# yum list > rpmlist.txt
# grep SDL rpmlist.txt

# yum install SDL-devel.x86_64

# rpm -ql SDL-devel.x86_64            // список файлов в пакете
```

Получить список библиотек из репозитория:

```
# yum list > rpmlist.txt
# grep devel rpmlist.txt
```

3. Создание статических и динамических библиотек

Библиотека - это набор скомпонованных особым образом объектных файлов. Библиотеки подключаются к основной программе во время линковки. По способу компоновки библиотеки подразделяют на *архивы* (статические библиотеки, static libraries, lib*.a) и *совместно используемые* (динамические библиотеки, shared libraries, lib*.so).

Статическая библиотека - это архив объектных файлов, который подключается к программе во время линковки.

```
/* world.h */
void h_world (void);
void g_world (void);
```

```
/* h_world.c */
#include <stdio.h>
#include "world.h"

void h_world (void) {
        printf ("Hello World\n");
}
```

```
/* g_world.c */
#include <stdio.h>
#include "world.h"

void g_world (void) {
        printf ("Goodbye World\n");
}
```

```
/* main.c */
#include "world.h"

int main (int argc, char* argv[]) {
        h_world ();
        g_world ();
        return 0;
}
```

Сценарий для make

```
# Makefile for World project
binary: main.o libworld.a
```

```
	gcc -o binary main.o -L. -lworld

main.o: main.c
	gcc -c main.c

libworld.a: h_world.o g_world.o
	ar cr libworld.a h_world.o g_world.o

h_world.o: h_world.c
	gcc -c h_world.c

g_world.o: g_world.c
	gcc -c g_world.c

clean:
	rm -f *.o *.a binary
```

```
$ make
gcc -c main.c
gcc -c h_world.c
gcc -c g_world.c
ar cr libworld.a h_world.o g_world.o
gcc -o binary main.o -L. -lworld
$ ./binary
Hello World
Goodbye World
$
```

Команда ar создает статическую библиотеку (архив). В данном случае два объектных файла объединяются в один файл libworld.a.

Компилятор gcc сам вызывает линковщик, когда это нужно. Опция -l, переданная компилятору, обрабатывается и посылается линковщику для того, чтобы тот подключил к бинарнику библиотеку. У имени библиотеки "обрублены" префикс и суффикс (lib-world-.a). Это делается для того, чтобы создать "видимое безразличие" между статическими и динамическими библиотеками.

Опция -L указывает линковщику, где ему искать библиотеку. По умолчанию /lib или /usr/lib и опция -L не требуется. В текущем случае библиотека находится в текущем каталоге, поэтому опция -L. (точка означает текущий каталог) необходима.

Создание динамической библиотеки:

```
# Makefile for World project

binary: main.o libworld.so
```

```
	gcc -o binary main.o -L. -lworld -Wl,-rpath,.

main.o: main.c
	gcc -c main.c

libworld.so: h_world.o g_world.o
	gcc -shared -o libworld.so h_world.o g_world.o

h_world.o: h_world.c
	gcc -c -fPIC h_world.c

g_world.o: g_world.c
	gcc -c -fPIC g_world.c

clean:
	rm -f *.o *.so binary
```

-Wl,-rpath,. - передать линковщику (-Wl) опцию option с аргументами optargs. В нашем случае мы передаем линковщику опцию -rpath с аргументом . (точка, текущий каталог).
-rpath, - указывает линковщику искать библиотеки в заданных каталогах
Или использовать переменную окружения LD_LIBRARY_PATH

-shared — указвывает компиллятору вызывать линковщик для создания динамической библиотеки.

-fPIC (-fpic) — указывает компилятору, что объектные файлы, полученные в результате компиляции должны содержать **позиционно-независимый код** (PIC - Position Independent Code), который используется в динамических библиотеках. Такой код используют не фиксированные позиции (адреса), а плавающие, благодаря чему код из библиотеки имеет возможность подключаться к программе в момент запуска.

http://www.opennet.ru/docs/RUS/zlp/index.html

Полезные команды:
ldd ; перечислить динамические объектные зависимости
nm ; информацию о бинарных файлах (объектных файлах, библиотеках, исполняемых файлах и т. д.), таблицу имён.

```
[lvr@lvr exercises]$ ldd hello
        linux-vdso.so.1 =>  (0x00007fff121d8000)
        libc.so.6 => /lib64/libc.so.6 (0x000000389c600000)
        /lib64/ld-linux-x86-64.so.2 (0x000000389c200000)



[lvr@lvr exercises]$ nm hello
00000000006006c8 d _DYNAMIC
0000000000600870 d _GLOBAL_OFFSET_TABLE_
00000000004005e8 R _IO_stdin_used
                 w _Jv_RegisterClasses
00000000006006a8 d __CTOR_END__
00000000006006a0 d __CTOR_LIST__
00000000006006b8 D __DTOR_END__
00000000006006b0 d __DTOR_LIST__
0000000000400698 r __FRAME_END__
00000000006006c0 d __JCR_END__
00000000006006c0 d __JCR_LIST__
00000000006008a8 A __bss_start
0000000000600898 D __data_start
00000000004005a0 t __do_global_ctors_aux
0000000000400450 t __do_global_dtors_aux
00000000006008a0 D __dso_handle
                 w __gmon_start__
000000000060069c d __init_array_end
000000000060069c d __init_array_start
0000000000400500 T __libc_csu_fini
0000000000400510 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
00000000006008a8 A _edata
00000000006008b8 A _end
00000000004005d8 T _fini
00000000004003b8 T _init
0000000000400400 T _start
000000000040042c t call_gmon_start
00000000006008a8 b completed.6294
```

```
0000000000600898 W data_start
00000000006008b0 b dtor_idx.6296
00000000004004c0 t frame_dummy
00000000004004e4 T main
                 U puts@@GLIBC_2.2.5
```

4. Утилита make

http://www.linuxlib.ru/prog/make_379_manual.html

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Утилита использует специальные make-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла `make` определяет и запускает необходимые программы.

```
make [ -j <число потоков> ] [ -f make-файл ] [ цель ] ...
```

Стандартные цели для сборки дистрибутивов GNU:

- all — выполнить сборку пакета
- install — установить пакет из дистрибутива (производит копирование исполняемых файлов, библиотек и документации в системные директории)
- uninstall — удалить пакет (производит удаление исполняемых файлов и библиотек из системных директорий)
- clean — очистить дистрибутив (удалить из дистрибутива объектные и исполняемые файлы созданные в процессе компиляции)
- distclean — очистить все созданные при компиляции файлы и все вспомогательные файлы созданные утилитой ./configure в процессе настройки параметров компиляции дистрибутива

Процесс сборки

Компилятор берет файлы с исходным кодом и получает из них объектные файлы. Затем линковщик берет объектные файлы и получает из них исполняемый файл. Сборка = компиляция + линковка.

Компиляция руками

```
# g++ main.cpp hello.cpp factorial.cpp -o hello
```

Простой Makefile:

```
цель: зависимости
[tab] команда
```

Пример:

```
all:hello
    g++ main.cpp hello.cpp factorial.cpp -o hello
```

```
clean:
    rm -rf *.o hello
```

Использование переменных и комментариев

```
# комментарий
CC=g++
CFLAGS=-c -Wall

all: hello

hello: main.o factorial.o hello.o
    $(CC) main.o factorial.o hello.o -o hello

main.o: main.cpp
    $(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
    $(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
    $(CC) $(CFLAGS) hello.cpp

clean:
    rm -rf *.o hello
```

Использование автоматических переменных

```
.SUFFIXES: .cpp .o

.cpp.o:
    $(CC) $(CFLAGS) -c -o $@ $<
```

$@ — имя .o-файла
$< — имя .cpp-файла

Использование «include»

```
#file default.mk

SVNDEV := -D'SVN_REV="$(shell svnversion -n .)"'
```

```
ifeq ($(BOARD),X86)
   CC       = gcc -g
   CXX      = g++ -g
   LINK     = ar cr.
   CFLAGS   = -Wall -O1  -DCENTOS $(SVNDEV)
   CXXFLAGS = -c -Wall -O1 $(SVNDEV)
else
   CC       = nios2-linux-gcc
   CXX      = nios2-linux-g++
   LINK     = nios2-linux-ar cr.
   CFLAGS   = -Wall -O1 $(SVNDEV)
   CXXFLAGS = -c -Wall -O1 $(SVNDEV)
endif
```

```
#file Makefile

include ../default.mk


all: hello

hello: main.o factorial.o hello.o
    $(CC) main.o factorial.o hello.o -o hello

main.o: main.cpp
    $(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
    $(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
    $(CC) $(CFLAGS) hello.cpp

clean:
    rm -rf *.o hello
```

5. Утилита cmake

http://www.devexp.ru/2010/01/sborka-proektov-s-cmake-vvedenie/

CMake (от англ. cross platform make) — это кроссплатформенная система автоматизации сборки программного обеспечения из исходного кода.

CMake не осуществляет сборку проекта, а генерирует файлы управления сборкой из файлов **CMakeLists.txt**, создавая **Makefile** для конкретной платформы:

Unix makefile;
QT;
Microsoft Visual Studio;
XCode для MacOS;
Eclipse;
CodeBlocks.

```
$ cmake CMakeLists.txt
$ ls -l Makefile
```

Простой CMake-файл Hello, World!

CMakeLists.txt

```
# минимальная версия CMake необходимая для успешной интерпретации файла.
cmake_minimum_required (VERSION 2.6)

# определяется переменная PROJECT и ей задается значение hello_world – так будет называться наша программа
set (PROJECT hello_world)

# конструкция ${ИМЯ_ПЕРЕМЕННОЙ} возвращает значение переменной, таким образом
# проект будет называться hello_world.
project (${PROJECT})

# HEADERS и SOURCES переменные содержащие список файлов необходимых для сборки проекта.
set (HEADERS
     hello.h)
set (SOURCES
     hello.cpp
     main.cpp)
```

```
# команда собрать исполняемый файл с именем указанном в переменной PROJECT
# из файлов имена которых находятся в переменных HEADERS и SOURCES.
add_executable (${PROJECT} ${HEADERS} ${SOURCES})
```

hello.h

```
void hworld (void);
```

hello.cpp

```
#include <stdio.h>
#include "hello.h"

void hworld (void) {
        printf ("Hello World\n");
}
```

main.c:

```
#include "hello.h"

int main (int argc, char* argv[]) {
        hworld ();
        return 0;
}
```

```
[lvr@lvr hello]$ cmake CMakeLists.txt
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to:
/home/lvr/lvr/1.linux/2.developing/example/hello

[lvr@lvr hello]$ make
Scanning dependencies of target hello_world
```

```
[ 50%] Building CXX object CMakeFiles/hello_world.dir/hello.cpp.o
[100%] Building CXX object CMakeFiles/hello_world.dir/main.cpp.o
Linking CXX executable hello_world
[100%] Built target hello_world

[lvr@lvr hello]$ ./hello_world
Hello World
```

Сбока статической и динамической библиотеки

Точно также как и исполняемый файл, но в последней строке вместо команды add_executable, укажите команду add_library.

В этом случае будет собрана статическая библиотека, для сборки динамической библиотеки надо указать параметр SHARED после имени библиотеки:

```
...
add_library (${PROJECT} SHARED ${HEADERS} ${SOURCES})
```

Подключение библиотек

```
cmake_minimum_required (VERSION 2.6)

set (PROJECT ИМЯ_ПРОЕКТА)

project (${PROJECT})

# указать путь (в данном случае это корень проекта) по которому
компилятор будет искать # подключаемые заголовочные файлы.
include_directories (../)

set (LIBRARIES
          ИМЯ_БИБЛИОТЕКИ_1
          ИМЯ_БИБЛИОТЕКИ_2)


# указание CMake взять файл из директории build подпроекта,
выполнить его и результат
# работы положить в директорию ./bin/ИМЯ_БИБЛИОТЕКИ.
# для каждой библиотеки
foreach (LIBRARY ${LIBRARIES})
     add_subdirectory (../${LIBRARY}/build bin/${LIBRARY})
```

```
endforeach ()

set (HEADERS
          ../ЗАГОЛОВОЧНЫЕ_ФАЙЛЫ)
set (SOURCES
          ../ФАЙЛЫ_С_РЕАЛИЗАЦИЕЙ)

add_executable (${PROJECT} ${HEADERS} ${SOURCES})

# указывает, что проект надо линковать вместе с указанной
библиотекой
target_link_libraries (${PROJECT} ${LIBRARIES})
```