Краткий конспект лекции за 14.09.2016 :

1. Что будем проходить:

Язык программирования Си (C programming language).
Здесь рассмотрим:
- Синтаксис языка,
- Основные конструкции,
- Принципы построения программ.

Вспомогательные утилиты
gcc – компилятор (правильнее коллекция компиляторов)
make – Система управления программными проектами
binutils – Утилиты для работы с двоичными файлами (as, ar, objdump и прочее)
gdb – Отладчик.
Autotools (по возможности) – Комплексная система конфигурирования, сборки и установки программного обеспечения.

Системное программирование:
Обзорно:
- Что предоставляется разработчику?
- Как это можно использовать?
- Как это работает?
Углублённо:
- Попытаться реализовать собственный вариант системных утилит (ls, cat, find)

Драйвера:
Обзорно:
- Модель представления аппаратного обеспечения в ядре.
- Взаимодействие с ядром
Углубленно:
- Практическая реализация модуля ядра.

Ядро Linux:
Обзорно:
- Процесс загрузки
- Основные подсистемы
Углублённо
- Конфигурация и сборка, запуск.

2. Инструмены, которыми будем пользоваться.

Операционная система Linux (дистрибутив значения не имеет).

Интерпретатор командной строки bash (или любой другой)

Vim – открытая, расширенная версия редактора vi, который есть практически на абсолютно всех unix-подобных системах. (Emacs – для желающих).

gcc – откытый компилятор си.

3. Языки высокого и низкого уровня. Появление языка Си.

Самый низкоуровневый язык программирования – машинный код процессора.
Располагается в памяти, считывается процессором и может сразу же им исполняться.
Главный недостаток – нечитаем для человека.

Более удобный для восприятия (а значит и для разработки) – язык ассемблера. Представляет собой более человекочитаемое представление машинных команд (а также некоторое их расширение).
Пример:

```
0: 55                    push   %rbp
1: 48 89 e5              mov    %rsp,%rbp
4: b8 01 00 00 00        mov    $0x1,%eax
9: 5d                    pop    %rbp
a: c3                    retq
```

Первый столбец – адрес начала команды.
Второй столбец – машинный код команды
Третий – команда.
Четвертый – аргументы для команды.

Код, написанный на языке низкого уровня является максимально эффективным, поскольку привязан к конкретной архитектуре. Побочным эффектом данного преимущества является непереносимость кода между процессорами разных архитектур.
Так же при программировании на языке ассемблера достаточно затруднительно осуществлять повторное использование кода, что негативно сказывается на скорости и удобстве разработки.
Поскольку часть кода ядра линукс написана на ассемблере, к рассмотрению самого языка и его возможностей мы ещё вернемся.

Необходимость в более универсальном и удобном средстве привела к созданию языков высокого уровня.
Языки высокого уровня можно условно разделить на компилируемые и интерпретируемые.

Компилируемые языки – языки исходный код программ которых преобразуется в машинный код для конкретной архитектуры. После этого программа может использоваться отдельно, вне зависимости от программы компилятора.

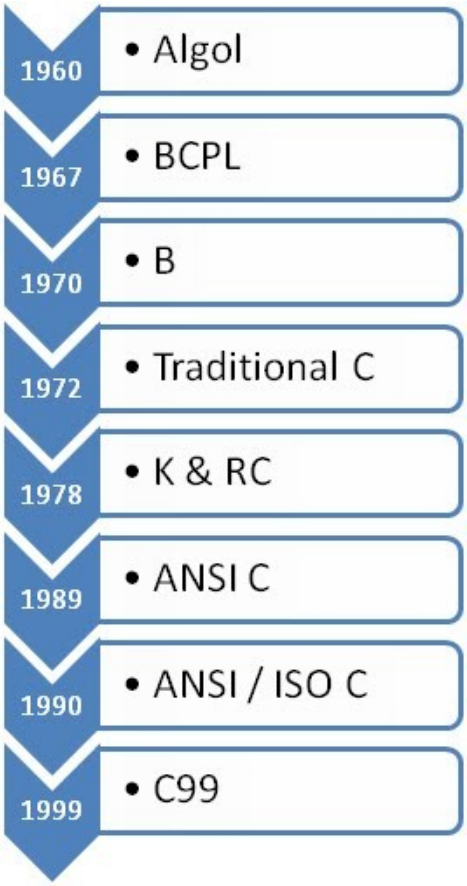
Интерпретируемые языки – языки исходный код которых не преоразуется в машинный код, а исполняется отдельной программой интерпретатором.
Здесь можно выделить простые и интерпретаторы промежуточного типа.
Простые интерпретаторы исполняют исходный код строчка за строчкой. Примером такого интерпретатора может служить bash.

Интерпретаторы промежуточного типа – дополнительно транслируют исходный код в некоторое промежуточное представление, которое потом будет выполнено. Важно, что для выполнения программы на интерпретируемом языке всегда необходима программа-интерпретатор. К таким интерпретируемым языкам можно отнести Python и Java.

История развития языков высокого уровня к 1972 году привела к созданию языка программирования Си, автором которого является Деннис Ритчи.
За основу языка Си был взят язык программирования Би, написанный Кеном Томпсоном.

Язык Си достаточно быстро обрел популярность среди программистов. Впоследствии на нём было переписано ядро Unix, что стало первым примером реализации ядра операционной системы на языке высокого уровня. До этого всё писалось на ассемблере.

Языки, оказавшие влияние на Си и история развития языка.



В настоящее время Си используется для написания наиболее эффективных по времени исполнения и минимальных по объму, занимаемой памяти программ.

Наиболее часто применяется при разработке:
- Ядер операционных систем (Linux, Minix, QNX и т.д)
- Драйверов устройств.
- Высоконагруженные системы (веб-сервер Apache, nginx)
- Мультимедиа приложения (OpenGL, библиотека x264)

Язык Си так же был взят Бьярном Страуструпом за основу для создания языка C++, поддерживающего объектно-ориентированную подход к разработке приложений.

4. Пример Hello world:

Рассмотрим пример программы на Си.

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

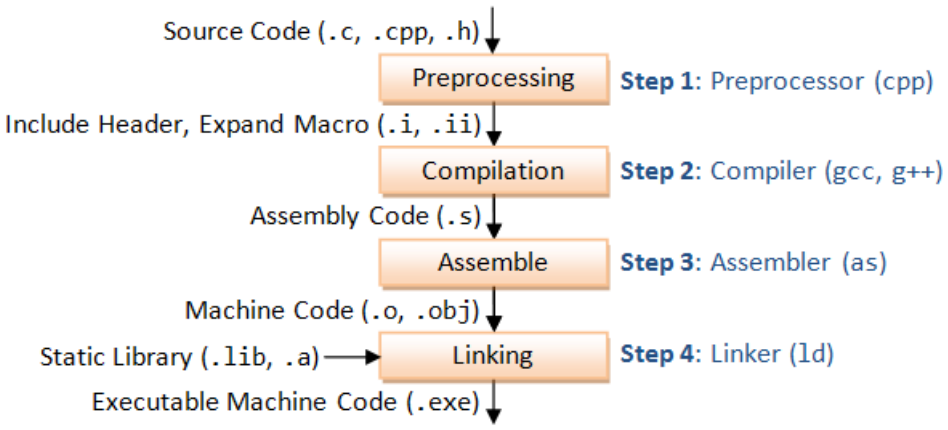
Для запуска компиляции необходимо выполнить:
(Прим. Знак “$” обозначает исполнение последущей команды в bash от имени пользователя, знак “#” будет обозначет исполнение команды от имени суперпользователя).

```
$ gcc hello.c -o hello_world
```

Для выполнения

```
$ ./hello_world
```

Описание процессов, происходящих при компиляции:
Общая схема.
Preprocessing – препроцессинг. Подготовка общего исходного текста. (директивы #include разворачиваются в полный текст добавляемого файлов)
Compilation – компиляция. Преобразование исходного кода на языке Си в язык ассемблера.
Assemble – ассемблирование. Транслирование программы на языке ассемблера в машинный код конкретной ахитектуры - подготовка объектного файла.
Linking – линковка. Сборка единого исполняемого файла из набора объектных файлов и разделяемых библиотек.



Рассмотрим эти этапы на примере компиляции программы hello_world.
Красным цветом выделены основные программы выполняющие операции по созданию программы.
Синим цветом выделены входные параметры, представляющие интерес.
Фиолетовым цветом выделены результирующие файлы.
Зеленым цветом обозначены комментарии.
Серым цветом скрыт не представляющий на данный момент интереса вывод утилит.

```
$ gcc -v hello.c -o hello_world
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 5.2.1-22ubuntu2'
--with-bugurl=file:///usr/share/doc/gcc-5/README.Bugs --enable-
languages=c,ada,c++,java,go,d,fortran,objc,obj-c++ --prefix=/usr --program-
suffix=-5 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib
--without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls
--with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-
```

```
libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object
--disable-vtable-verify --enable-libmpx --enable-plugin --with-system-zlib
--disable-browser-plugin --enable-java-awt=gtk --enable-gtk-cairo --with-java-
home=/usr/lib/jvm/java-1.5.0-gcj-5-amd64/jre --enable-java-home --with-jvm-root-
dir=/usr/lib/jvm/java-1.5.0-gcj-5-amd64 --with-jvm-jar-dir=/usr/lib/jvm-
exports/java-1.5.0-gcj-5-amd64 --with-arch-directory=amd64 --with-ecj-
jar=/usr/share/java/eclipse-ecj.jar --enable-objc-gc --enable-multiarch
--disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-
list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-
checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu
--target=x86_64-linux-gnu
Thread model: posix
gcc version 5.2.1 20151010 (Ubuntu 5.2.1-22ubuntu2)
COLLECT_GCC_OPTIONS='-v' '-o' 'hello_world' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/5/cc1 -quiet -v -imultiarch x86_64-linux-gnu
hello.c -quiet -dumpbase hello.c -mtune=generic -march=x86-64 -auxbase hello
-version -fstack-protector-strong -Wformat -Wformat-security -o /tmp/ccqKcASC.s
/* Преобразование исходного файла hello.c во временный ассемблерный файл
/tmp/ccqKcASC.s */
GNU C11 (Ubuntu 5.2.1-22ubuntu2) version 5.2.1 20151010 (x86_64-linux-gnu)
      compiled by GNU C version 5.2.1 20151010, GMP version 6.0.0, MPFR version
3.1.3, MPC version 1.0.3
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
ignoring nonexistent directory "/usr/local/include/x86_64-linux-gnu"
ignoring nonexistent directory "/usr/lib/gcc/x86_64-linux-
gnu/5/../../../../x86_64-linux-gnu/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/lib/gcc/x86_64-linux-gnu/5/include
 /usr/local/include
 /usr/lib/gcc/x86_64-linux-gnu/5/include-fixed
 /usr/include/x86_64-linux-gnu
 /usr/include
End of search list.
GNU C11 (Ubuntu 5.2.1-22ubuntu2) version 5.2.1 20151010 (x86_64-linux-gnu)
      compiled by GNU C version 5.2.1 20151010, GMP version 6.0.0, MPFR version
3.1.3, MPC version 1.0.3
GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
Compiler executable checksum: ae1f57641df2bca5e5adf4e90874d7ef
COLLECT_GCC_OPTIONS='-v' '-o' 'hello_world' '-mtune=generic' '-march=x86-64'
 as -v --64 -o /tmp/ccdODZz1.o /tmp/ccqKcASC.s
GNU assembler version 2.25.1 (x86_64-linux-gnu) using BFD version (GNU Binutils
for Ubuntu) 2.25.1
COMPILER_PATH=/usr/lib/gcc/x86_64-linux-gnu/5/:/usr/lib/gcc/x86_64-linux-
gnu/5/:/usr/lib/gcc/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-
gnu/5/:/usr/lib/gcc/x86_64-linux-gnu/
LIBRARY_PATH=/usr/lib/gcc/x86_64-linux-gnu/5/:/usr/lib/gcc/x86_64-linux-
gnu/5/../../../x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-
gnu/5/../../../../lib/:/lib/x86_64-linux-gnu/:/lib/../lib/:/usr/lib/x86_64-
linux-gnu/:/usr/lib/../lib/:/usr/lib/gcc/x86_64-linux-
gnu/5/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-v' '-o' 'hello_world' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/5/collect2 -plugin /usr/lib/gcc/x86_64-linux-
gnu/5/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper
-plugin-opt=-fresolution=/tmp/cc8B1fiq.res -plugin-opt=-pass-through=-lgcc
-plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -plugin-opt=-
pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --sysroot=/ --build-id
--eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker
/lib64/ld-linux-x86-64.so.2 -z relro -o hello_world /usr/lib/gcc/x86_64-linux-
gnu/5/../../../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-
gnu/5/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-
gnu/5/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/5 -L/usr/lib/gcc/x86_64-linux-
gnu/5/../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-
gnu/5/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib -L/usr/lib/x86_64-
```

```
linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-gnu/5/../../..
/tmp/ccdODZz1.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed
-lgcc_s --no-as-needed /usr/lib/gcc/x86_64-linux-gnu/5/crtend.o
/usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crtn.o
```

Таким образом наш исходный файл hello.c был скомпилирован в ассемблерный файл /tmp/ccqKcASC.s
Ассемблерный файл /tmp/ccqKcASC.s был преобразован в объектный файл /tmp/ccdODZz1.o, который был слинкован в исполняемый файл hello_world.

Отсылки к данной схеме в целом и отдельным её этапам, а также их более подробное рассмотрение, будет происходить неоднократно на протяжении всего курса.

5. Несколько слов о линковке.

Наиболее простое определение процесса линковки – это связывание нескольких объектных файлов в единый исполняемый.

Объектный файл представляет собой готовый, полученный в результате компиляции машинный код, который однако не может быть исполнен процессором, поскольку может содержать неопределённые ссылки на компоненты других объектных файлов или разделяемых библиотек.
Однако этот файл может быть использован для объединения с другими объектными файлами для создание единой исполняемой программы.

Простой пример.
Пусть есть два файла.

foo.c

```
int foo()
{
    return 1;
}
```

bar.c

```
#include <stdio.h>

int foo();

int main()
{
    printf("foo() : %d\n", foo());
    return 0;
}
```

Каждый из файлов может быть скомпилирован в объектный файл (при помощи опции компилятора -c)

```
$ gcc -c foo.c -o foo.o

$ gcc -c bar.c -o bar.o
```

Однако, если мы попробуем получить исполняемый файл из bar.o

```
$ gcc bar.o -o bar_exec
```

То получим сообщение, говорящее нам об ошибке линковки (в данном случае неопределенная ссылка на функцию foo).

```
bar.o: In function `main':
bar.c:(.text+0xa): undefined reference to `foo'
collect2: error: ld returned 1 exit status
```

Линковщик не в процессе подготовки исполняемого файла не обнаружил код для функции foo() , который содержится в объектном файле foo.o

Добавив foo.o в качестве входного параметра для линковщика.

```
$ gcc foo.o bar.o -o bar_exec
```

Ошибка исчезнет и в результате работы мы получим готовый исполняемый файл bar_exec, который можно выполнить:

```
./bar_exec
foo() : 1
```