POLITECNICO DI MILANO





Advanced Operating Systems

Linux Application Development Toolkit

Giuseppe Massari

giuseppe.massari@polimi.it

Outline

From source to binary code

- Building process
- Binary format file
- Linking process

Build system

- Makefile

Debugging

- GNU debugger
- Tracing tools

Building process

“Hello World!” C language sample

```
#include <stdio.h>
#define NR_CYCLES 10

int main() {
    int i;
    for (i = 0; i < NR_CYCLES; ++i)
        fprintf(stderr, “Hello, World!\n”);
    return 0;
}
```

Building command

- GNU C Compiler (**gcc**) (...**g++** in case of C++ code)

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

Source files Executable binary output file name

Execution

- From the directory containing the `hello` executable

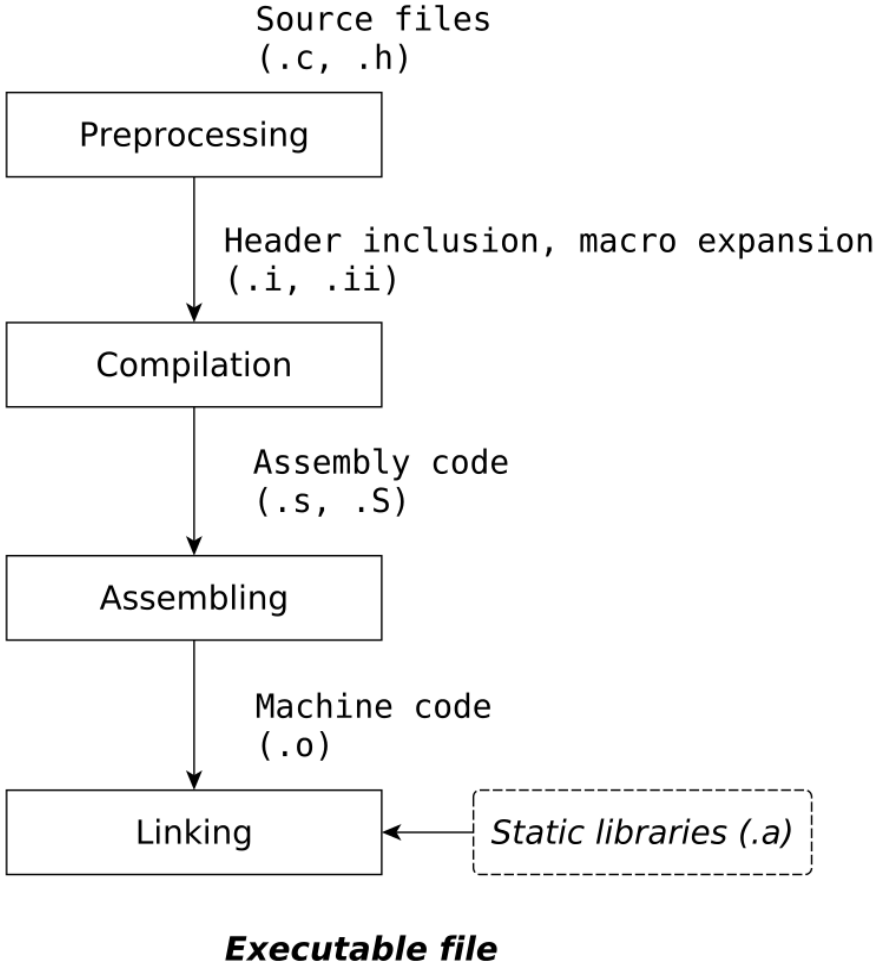
```
$ ./hello
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Building process

Step-by-step view

- A building process includes several intermediate steps
- The compiler (`gcc`) calls the suitable tools for each
  - Preprocessing: **`cpp`**
  - Compilation: **`gcc`**
  - Assembling: **`as`**
  - Linking: **`ld`**
- We can stop `gcc` to analyse the output of each step by specifying a given command line option



Building process

Preprocessing

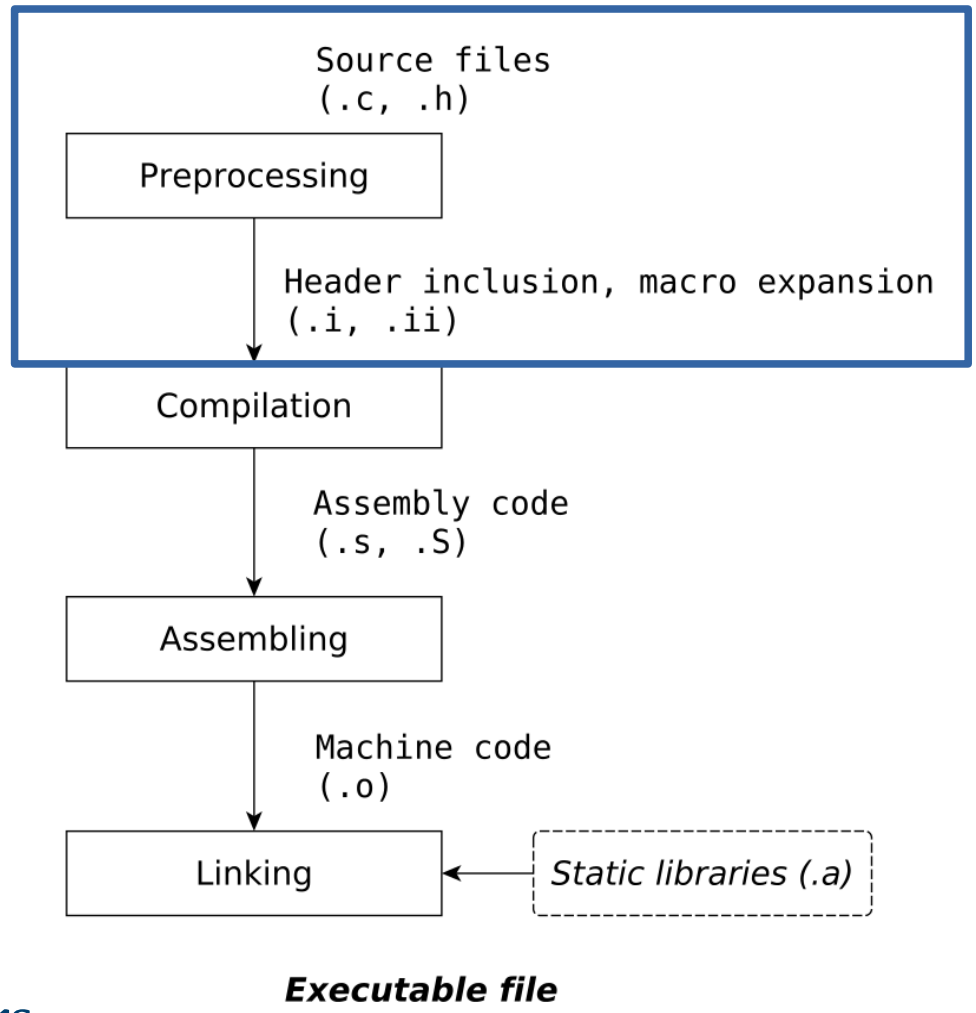
- Header file inclusion and macro expansion

```
#include <stdio.h>
#define NR_CYCLES  10
```

- `gcc` invokes the preprocessor (`cpp`)
- We can stop the building process here as follows:

```
$ gcc -E hello.c -o hello.i
```

- The output is still a text file including variables and functions defined in the headers file included (`hello.i`)



Preprocessing

- Input: *hello.c*

```
#include <stdio.h>
#define NR_CYCLES 10
int main() {
    for (i = 0; i < NR_CYCLES; ++i)
        fprintf(stderr, “Hello, World!\n”);
    return 0;
}
```

- Output: *hello.i*

```
#…
extern int fprintf (FILE *__restrict __stream,
       const char *__restrict __format, ...);
…
int main() {
    for (i = 0; i < 10; ++i)
        fprintf(stderr, “Hello, World!\n”);
    return 0;
}
```

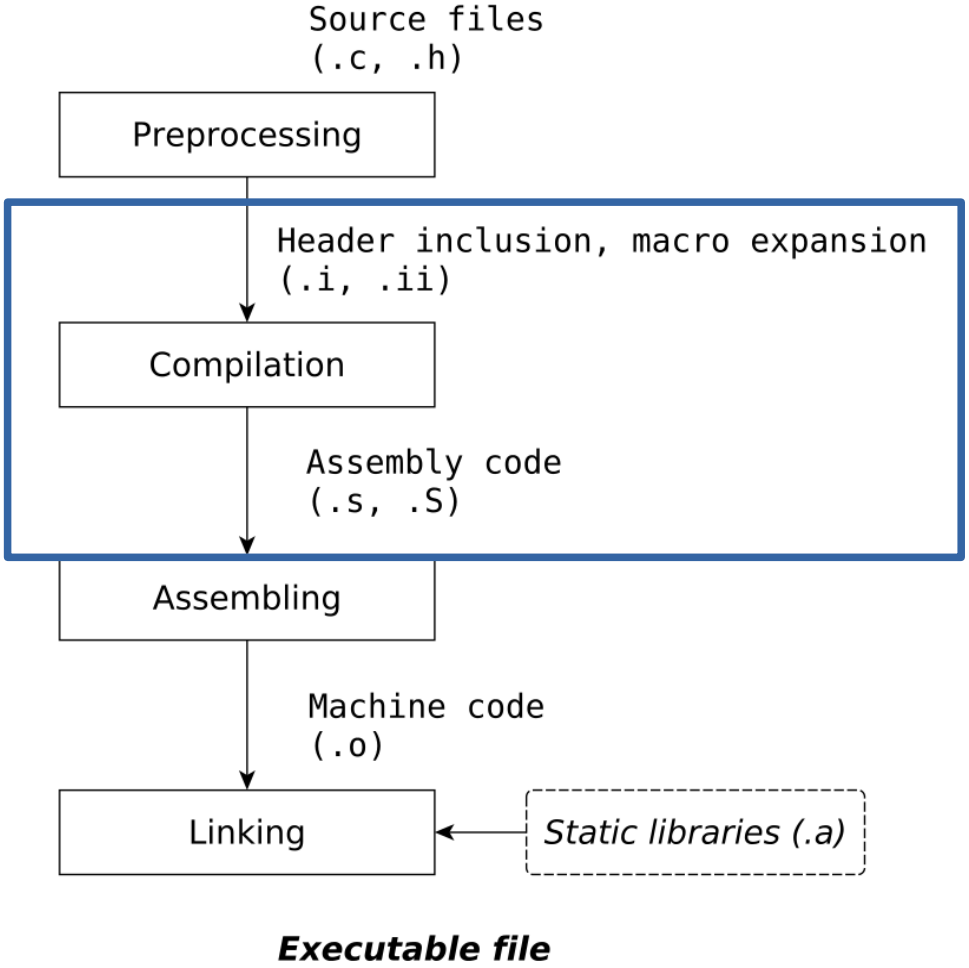
Building process

Compilation

- The C code is translated into the assembler code by `gcc`
- We can stop the building process here as follows:

```
$ gcc -S hello.c
```

- The output is still a text file containing data and source code in the assembler form (`hello.s`)



Building process

Compilation (before assembling)

- Output: *hello.s*

```
    .file   "hello.c"
    .section .rodata
.LC0:
    .string "Hello World!\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq%rsp, %rbp
    .cfi_def_cfa_register 6
    subq$32, %rsp
    movl%edi, -20(%rbp)
    movq%rsi, -32(%rbp)
    movl$0, -4(%rbp)
    jmp .L2
```

```
.L3:
    movqstderr(%rip), %rax
    movq%rax, %rcx
    movl$13, %edx
    movl$1, %esi
    movl$.LC0, %edi
    callfwrite
    addl$1, -4(%rbp)
.L2:
    cmpl$9, -4(%rbp)
    jle .L3
    movl$0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    …
```

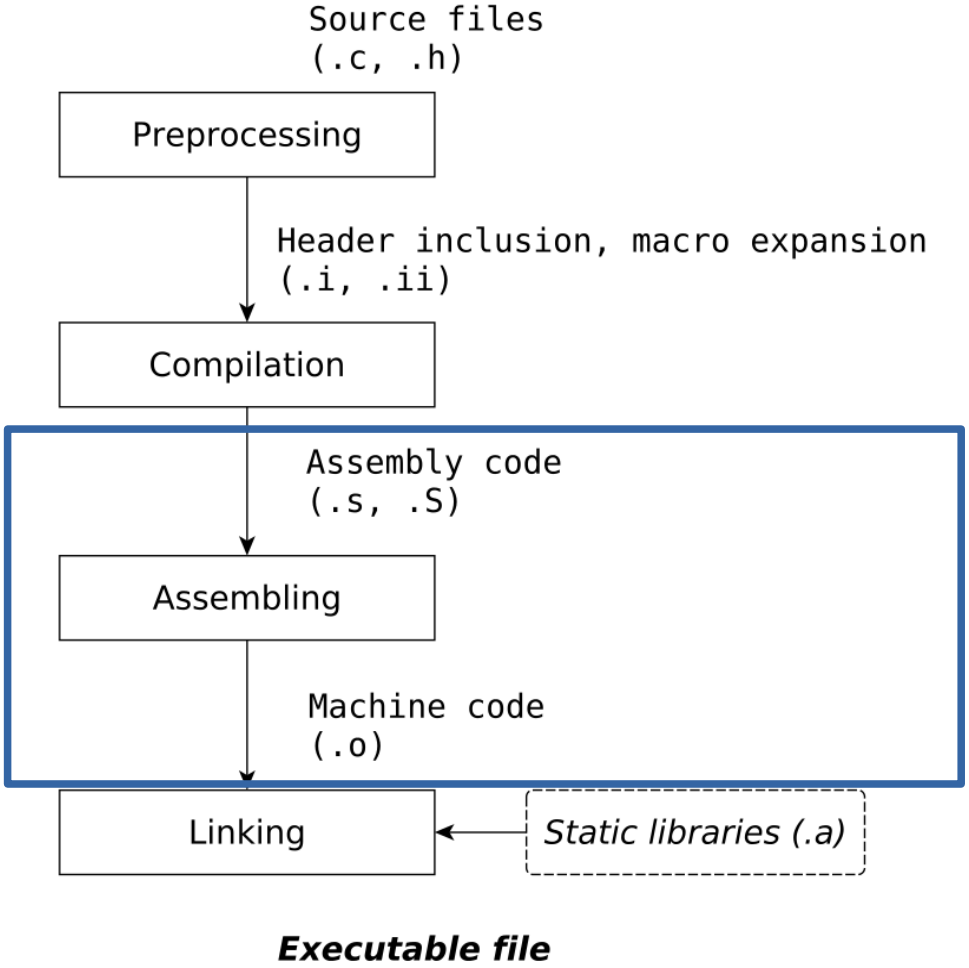
Building process

Assembling

- Compile the C source code and assemble it (but not link)
- `gcc` invokes `as`
- We can stop the building process here as follows:

```
$ gcc -c hello.c
```

- The output is a binary file (object file) for each source file
- The object file contains data and machine code (`hello.o`)



Building process

Assembling

- The output is a binary file. We can check the SYMBOL TABLE

```
$ objdump -t hello.o
```

```
hello.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*    0000000000000000 hello.c
0000000000000000 l    d  .text    0000000000000000 .text
0000000000000000 l    d  .data    0000000000000000 .data
0000000000000000 l    d  .bss 0000000000000000 .bss
0000000000000000 l    d  .rodata  0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack  0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame    0000000000000000 .eh_frame
0000000000000000 l    d  .comment 0000000000000000 .comment
0000000000000000 g     F .text    0000000000000047 main
0000000000000000         *UND*    0000000000000000 stderr
0000000000000000         *UND*    0000000000000000 fwrite
```

stderr and **fwrite** (included in `fprintf`) are actually undefined symbols

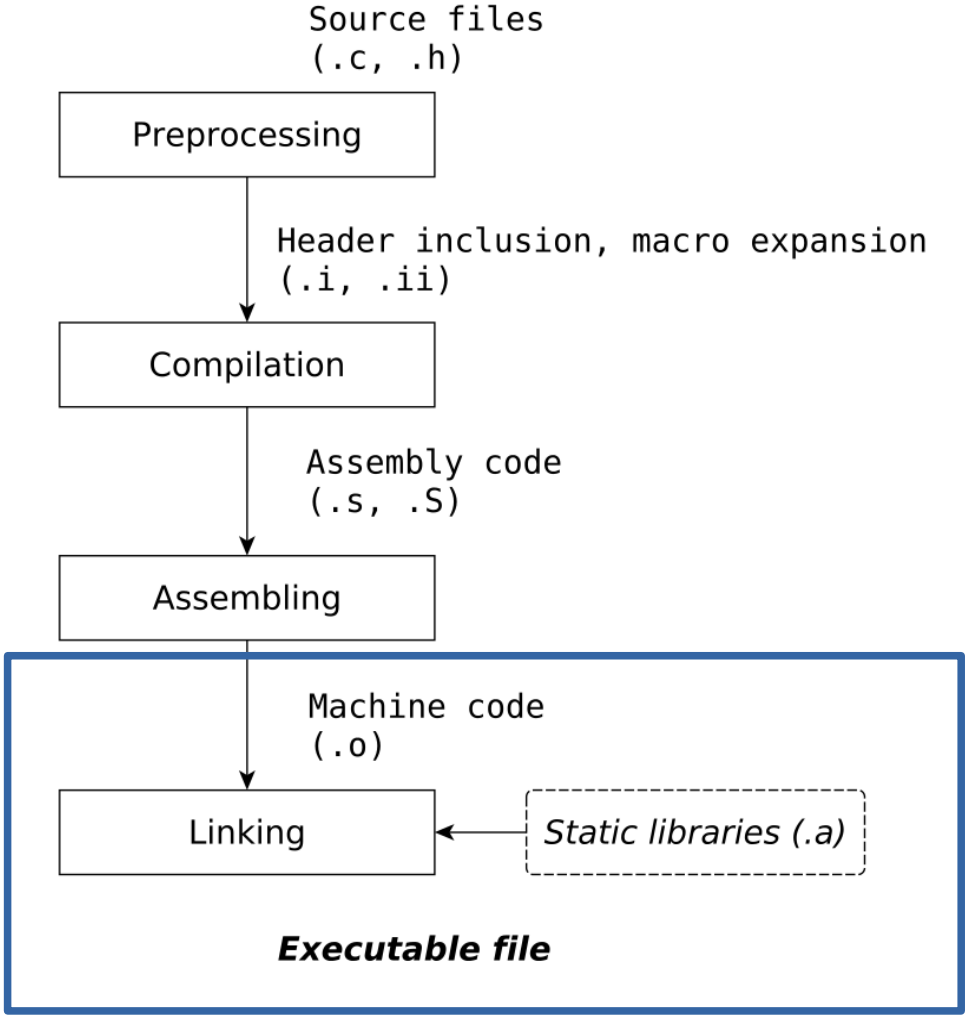
Building process

Linking

- `gcc` invokes the linker (`ld`)
- Object files are linked together
- (Static) libraries (`.a`) are linked (if any)
- Executable file is generated

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

- The output is an *executable* binary file containing code and data (`hello`)



Building process

Linking

- The output is the *executable* binary file

```
$ objdump -t hello
```

```
hello:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000  l    df *ABS*   0000000000000000   hello.c
0000000000000000  l    d  .text   0000000000000000   .text
0000000000000000  l    d  .data   0000000000000000   .data
0000000000000000  l    d  .bss    0000000000000000   .bss
0000000000000000  l    d  .rodata 0000000000000000   .rodata
0000000000000000  l    d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000  l    d  .eh_frame   0000000000000000 .eh_frame
0000000000000000  l    d  .comment    0000000000000000   .comment
0000000000000000  g    F .text    0000000000000047   main
0000000000000000         F *UND*    0000000000000000   fwrite@@GLIBC_2.2.5
0000000000601040  g    O .bss     0000000000000008   stderr@@GLIBC_2.2.5
```

stderr has been resolved, while **fwrite** will be resolved at run-time

Common `gcc` options

`-O<n>`	Optimization options
`-g`	Export debug symbols
`-Wall`	Emit all generally useful warnings that gcc can provide
`-v`	Verbose execution, showing the sequence of tools invoked
`-c`	Compilation without linking (generate the object file)
`-S`	Stop after compilation (generate assembly code)
`-E`	Stop after preprocessor execution

```
$ man gcc
```

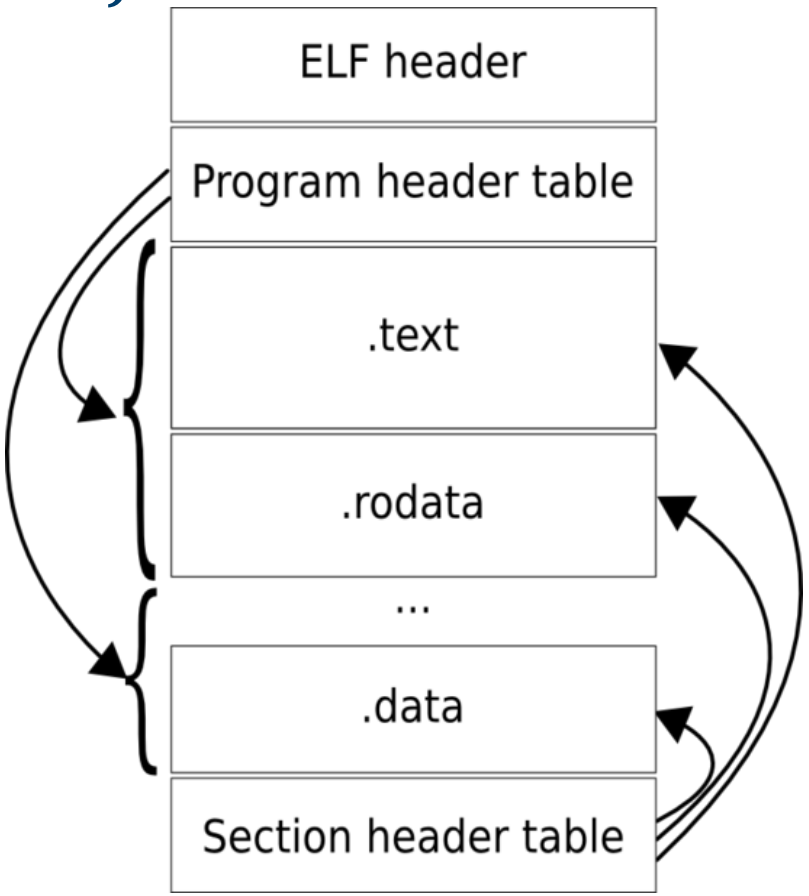
Binary format

Executable and Linkable Format (*ELF*)

- Linux standard binary format
- Define the memory layout

  Code and *data* placement
- Provide support information for the *linking* stage
- Provide support information for *loading* the program
- Several binary utilities (binutils) available to analyse an ELF file

  `objdump`, **`readelf`**, **`nm`**, **`file`**, **`ldd`**



Binary format

Executable and Linkable Format (*ELF*)

- ELF is flexible and is used to generate different file types

  Relocatable object file (.o files)

  - Code and data suitable for linking with other object file to create an *executable* or a *shared object* file

  Executable file

  - A program suitable for execution; the file including information that the exec() system call uses to create a process image

  Shared object file (.so files)

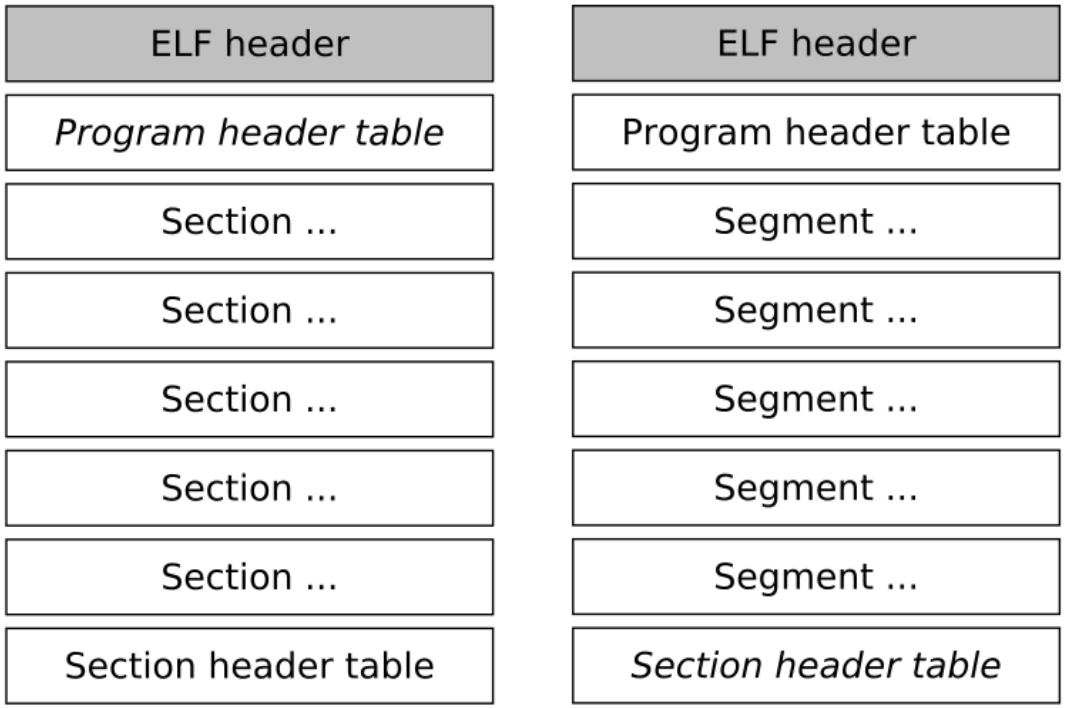
  - Code and data to...
    - »Process at compile time with other *relocatable* and o*bject files* to create another object file
    - »Combines at run-time with an *executable* file and other *shared objects* to create a process image

Binary format

Executable and Linkable Format (*ELF*)

- ELF files participate in *program building* and *program execution*

  We can distinguish two parallel views of an ELF file



Binary format

Executable and Linkable Format (*ELF*)

- *ELF header*

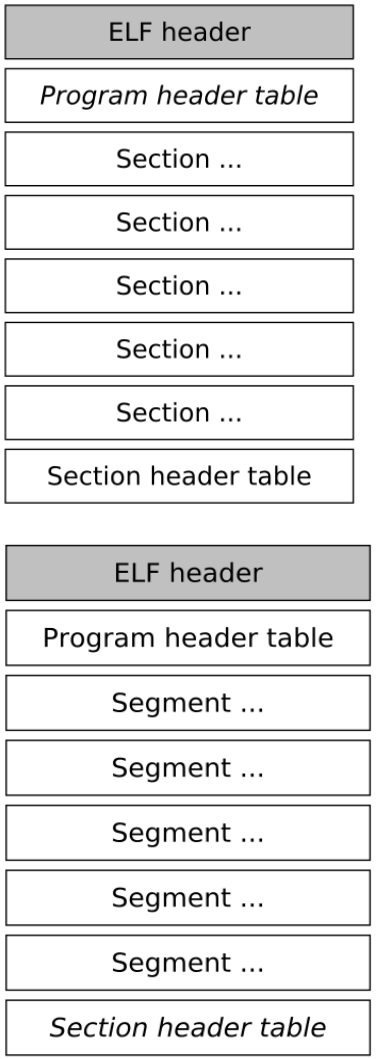
  Identify the binary type, information about the target architecture, and provides references to locate the other parts of the file

- *Sections*

  Located by the *Section header table*.
  Code, data, and other program information (e.g. the symbol table)

- *Segments*

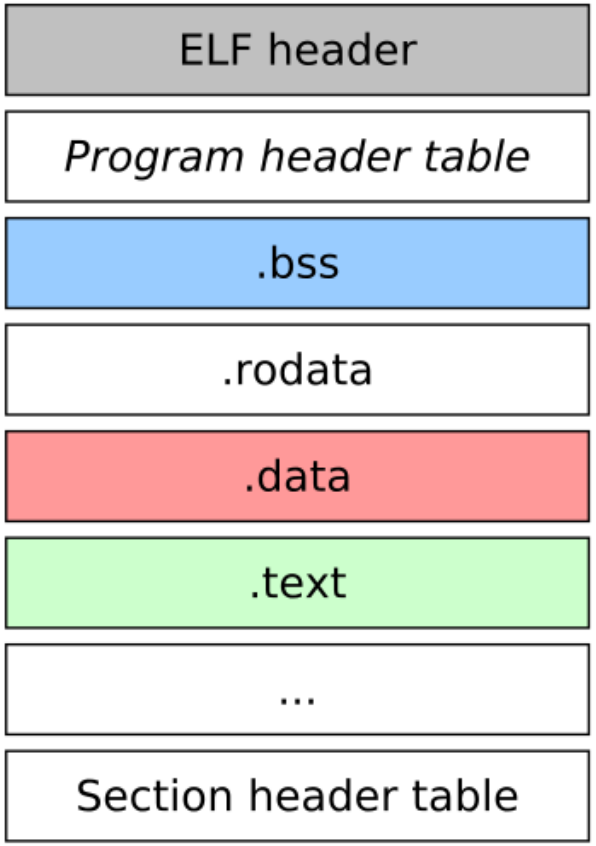
  Located by the *Program header table*.
  Sections in executable or shared object files are grouped into segments to specify loading support information



Binary format

Executable and Linkable Format (ELF)

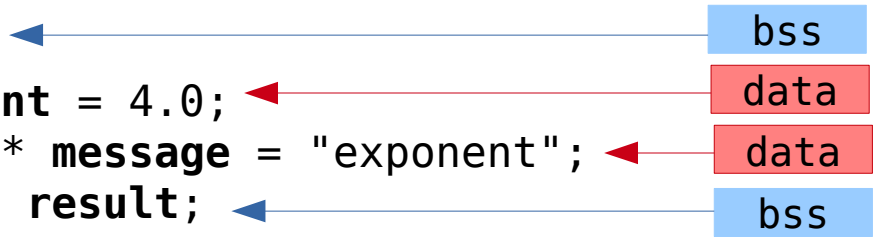
- The most common sections of a binary are related to the placement of data and code
  - **.bss** (Block Started by Symbols)
    - Zero initialized data
    - Zero initialized static variables
    - Uninitialized static variables
  - **.data**
    - Initialized data
    - Initialized (non-zero) static variables
  - **.text**
    - Executable code



Executable and Linkable Format (ELF)

- Example of data placement in the binary sections (*hello_pow.c*)

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 float i = 0;
 5 float exponent = 4.0;
 6 static char * message = "exponent";
 7 static float result;
 8
 9 int main(int argc, const char *argv[])
10 {
11     printf("The %s is: %2.2f\n", message, exponent);
12     int nr_cycles = 10;
13     for (; i < nr_cycles; ++i) {
14         result = pow(i, exponent);
15         printf("%2.0f^%2.2f = %2.2f\n", i, exponent, result);
16     }
17     return result;
18 }
```



Executable and Linkable Format (*ELF*)

- After the compilation we can inspect the symbols table

```
$ objdump -t hello_pow
```

```
…
0000000000601050 l     O .data  0000000000000008              message
0000000000601068 l     O .bss   0000000000000004              result
…
0000000000601064 g     O .bss   0000000000000004              i
000000000040067a g     F .text  00000000000000fd              main
…
0000000000601048 g     O .data  0000000000000004              exponent
…
```

- Where is the *nr_cycles* variable gone?

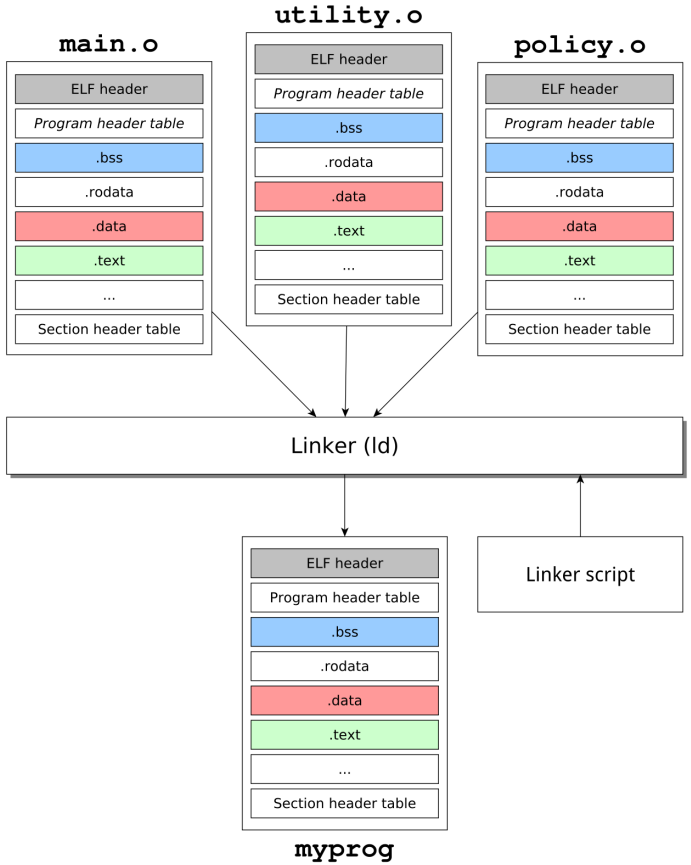
Linking process

Example

```
$ gcc -Wall -O3 main.c utility.c policy.c -o myprog
```

- **Input**: Before the linking step we have three source files compiled into object files
- **Output**: The executable `myprog`

- The linker properly merging the sections of the input files into the output file (the *executable* binary)
- The *linker script* drives the linker during this process



Linker scripts

- Written in the linker command language
- Map sections from input files (e.g., object files) into output file (e.g., the executable)
- *In Linux the ld linker uses a default script if none is explicitly provided*

  To look at the default linker script:

```
$ ld --verbose
```

- In embedded systems with limited capacity memory units (RAM, Flash, etc...) it plays a key role
- Define the memory layout of the output file

  Code and data placement in memory

Linking process

Linker scripts

- Example 1

```
SECTIONS
{
  . = 0x10000;
  .text : { *(.text) }
  . = 0x8000000;
  .data : { *(.data) }
  .bss  : { *(.bss) }
}
```

"." = *Location counter*
The assignment means that the following section will start @0x10000

(...) *input* file section to map

"*" = Map the section from ALL the input files

.text, **.data**, **.bss** on the left are *output* file sections

Linking process

Linker scripts

- Example 2

```
ENTRY(Reset_Handler)

MEMORY
{
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 0x10000 /*64K*/
  RAM (xrw)  : ORIGIN = 0x20000000, LENGTH = 0x02000 /*8K*/
}

/* Stack and heap size */
stack_size = 1024;
heap_size  = 256;

/* Beginning and ending of stack */
_stack_start = ORIGIN(RAM)+LENGTH(RAM);
_stack_end   = _stack_start - stack_size;

SECTIONS
{
  /* Startup code goes first into FLASH */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH
```

```
  .text :  {
    . = ALIGN(4);
    *(.text)
  } >FLASH

  .data : AT ( _sidata ) {
    . = ALIGN(4);
    *(.data)
  } >RAM

  . = ALIGN(4);
  .bss : {
    _sbss = .;
    __bss_start__ = _sbss;
    *(.bss)
    _ebss = .;
    __bss_end__ = _ebss;
  } >RAM

  . = ALIGN(4);
  .heap :   {
      _heap_start = .;
      . = . + heap_size;
      _heap_end = .;
   } >RAM
}
```

Linking process

Using libraries

- The C library (e.g., *glibc*) provides a large set of APIs for developing applications

- Many other libraries are typically available on every Linux distribution, focusing specific application or functionality domains, like multimedia, networking, numerical computations, graphical processing, etc...

- Conventionally library packages are named as follows:

  `lib`*`foo`*: *foo* library itself, including binary files to link to our binaries

  `lib`*`foo``-dev`: Package including header and configuration files, required for building a new application using *functions* and *data structures* provided by foo

  `lib`*`foo``-dbg`: *foo* library debug symbols

Linking process

Using libraries

- Let's add to the Hello World sample a math function from the *libm* (math library) and let's try to compile

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 #define NR_CYCLES 10
 5
 6 int main(int argc, const char *argv[])
 7 {
 8     float i;
 9     for (i = 0; i < NR_CYCLES; ++i) {
10         fprintf(stderr, "Hello World!\n");
11         fprintf(stderr, "%2.0f^2 = %2.2f\n", i, pow(i, 2.0));
12     }
13     return 0;
14 }
```

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

Linking process

Using libraries

- Let's try to compile to sample now

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

```
/tmp/ccVYsLxP.o: In function `main':
hello.c:(.text+0x57): undefined reference to `pow'
collect2: error: ld returned 1 exit status
```

- The linker (`ld`) is not able to find a reference to function `pow()`
- We must tell to `gcc` (that forwards to `ld`) where to find `pow()`
- This is done by adding the "`-l`" option and the name of the library

  Given the name of the library package, remove "lib" prefix

  `libm` → ~~`lib`~~`m`

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

Linking process

Using libraries

- The linker searches the library starting from the filesystem paths specified in its configuration file

```
$cat /etc/ld.so.conf
```

```
include /etc/ld.so.conf.d/*.conf
```

- `ld.so.conf` can include further configuration files, usually added during the installation of the library packages

```
$ls -l /etc/ld.so.conf.d/
```

```
…
-rw-r--r-- 1 root root 108 apr 12  2014 i686-linux-gnu.conf
-rw-r--r-- 1 root root  44 ago  9  2009 libc.conf
-rw-r--r-- 1 root root  68 apr 12  2014 x86_64-linux-gnu.conf
…
```

Using libraries

- Example of a set of paths through which `ld` will look for libraries

```
$cat /etc/ld.so.conf.d/*.conf
```

```
# Multiarch support
/lib/i386-linux-gnu
/usr/lib/i386-linux-gnu
/lib/i686-linux-gnu
/usr/lib/i686-linux-gnu
# libc default configuration
/usr/local/lib
# Multiarch support
/lib/x86_64-linux-gnu
/usr/lib/x86_64-linux-gnu
/usr/lib/x86_64-linux-gnu/mesa-egl
/usr/lib/x86_64-linux-gnu/mesa
# Legacy biarch compatibility support
/lib32
/usr/lib32
```

Linking process

Using libraries

- The *libm* installation deploys the binaries (.a, .so) and the header files (.h) under standard path

```
/usr/include/math.h
/usr/lib/x86_64-linux-gnu/libm.a
/usr/lib/x86_64-linux-gnu/libm.so
```

- Can we specify headers and libraries locate under different paths?

```
$ gcc hello.c -o hello -Llibrary-dir -Iheader-dir
-llibrary-name
```

- We could also use `pkg-config` tool

```
$ gcc hello.c -o hello $(pkg-config --cflags --libs
library-package)
```

Linking process

Static libraries

- Binary available in form of “archive” (`.a` files)
- Simple format, generated by the “`ar`” utility
- Linked at *compile time*

```
$ gcc hello.c /usr/lib/x86_64-linux-gnu/libm.a -o hello
```

Dynamic libraries

- Binary available in form of “shared object” (`.so` files)
- *ELF binary format*, generated by `ld`
- More memory efficient
- Linked at *run-time*

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

Linking process

Dynamic libraries

- Run-time symbols resolution required
- A *dynamic linker* (**ld-linux.so**) is involved when the program is loaded
- In case of errors in the symbols resolution, it is possible to check the set of shared object required by the program with the `ldd` utility

```
$ ldd hello
```

```
    linux-vdso.so.1 =>  (0x00007fff88a8f000)
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f28875b1000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f28871ea000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f28878dc000)
```

- if `ldd` returns "**=> not found**" entries, there are missing references
- There are several approach to solve the problem...

Linking process

Dynamic libraries

Using the environmental variable LD_LIBRARY_PATH

The dynamic linker will search the library in the list of path specified in the variable

```
$ export LD_LIBRARY_PATH=/path/to/lib:$LD_LIBRARY_PATH
```

- Adding the library path to the dynamic linker library cache

```
$ ldconfig -n /path/to/lib
```

- Pre-loading the library

```
$ export LD_PRELOAD=/path/to/lib/libname.so
```

- Adding the library path to the linker configuration file

```
# echo /path/to/lib/ >> /etc/ld.so.conf
# ldconfig
```

GNU binary utilities (binutils)

- Utility tools used by the compiler to create and manipulate binary files, or by the developer for debug purpose

add2line	Converts addresses into file names and line numbers
ar	A utility for creating, modifying and extracting from archive
nm	Lists symbols from object files
objdump	Displays information from object files
readelf	Displays information from any ELF format object file
strip	Discards symbols
strings	Lists printable strings from files
...	...

Build system

Problem

- When the project includes a “not negligible” amount of source files and library dependencies, writing gcc command-line strings becomes cumbersome
- To the sake of *portability* (e.g., support different processor architectures) we need to specify different compilation flags

```
$ gcc -Wall -O2 myfile1.c myfile.2.c … myfileN.c -o myprogram
-I./include -I/path/to/custom/lib1/include -L/path/to/custom/lib1/lib
-L/path/to/custom/lib2/lib -lcustom1 -lcustom2 …
```

Solution

- We need a more “automatic” system, a so called *build system*
- Actually, many build system are available but most of them rely on the generation of a `Makefile`

  Therefore, in this lecture we focus just on understanding how makefiles work!

Definition

- A text file (`Makefile` or `makefile`) including a set of *targets*, with *prerequisites* and *commands* to execute accordingly

```
<target>:<prerequisites>
    <commands>
```

- The `make` tool parses the file, following the target dependencies and executing the commands

```
$ make target-name
```

- A common set of target rules
  - **`all`**: Build sources in the source directories
  - **`install`**: Copy executable, configuration files under the installation path
  - **`uninstall`**: Remove the installation from the system
  - **`clean`**: Remove compilation product (without uninstall)

Build system

Example

```
my_project
├── dummy_functions.c
├── main.c
├── include
│   └── dummy_functions.h
└── Makefile
```

```c
#include <stdio.h>
#include "dummy_functions.h"

#define NR_CYCLES 20

int main(int argc, const char *argv[])
{
    int i;
    for (i = 0; i < NR_CYCLES; ++i) {
        if (is_even(i)) {
            printf("%d is even. ", i);
            printf("%d^2=%f.0\n", i, pow2(i));
        }
        else
            printf("%d is odd\n", i);
    }
    return 0;
}
```

main.c

```c
#ifndef DUMMY_FUNCS_
#define DUMMY_FUNCS_

int is_is_even(int num);
double pow2(double num);

#endif
```

dummy_functions.h

```c
#include "dummy_functions.h"
#include <math.h>

int is_even(int number) {
    return !(number%2);
}

double pow2(double number) {
    return pow(number, 2.0);
}
```

dummy_functions.c

Example

- Source tree

```
my_project
├── dummy_functions.c
├── main.c
├── include
│   └── dummy_functions.h
├── Makefile
```

- Build command-line

```
$ gcc -Wall -O3 main.c dummy_functions.c -I./include -lm
-o myprog
```

- `-Wall` option enables all the warning messages
- `-O3` option enables compile time optimization

Makefile

- Example (version 1)

```
NAME    = myprog
CC      = gcc
INCDIR  = ./include
CFLAGS  = -Wall -O3 -I $(INCDIR)
LDFLAGS = -lm
BINDIR  = /usr/local/bin

all:
    $(CC) main.c dummy_functions.c -o $(NAME) $(CFLAGS) $(LDFLAGS)

clean:
    rm -f $(NAME)
    rm -f main.o dummy_functions.o

install: $(NAME)
    cp $(NAME) $(BINDIR)

uninstall:
    rm -f $(BINDIR)/$(NAME)
```

Configurable stuff should be placed in suitable variables

Makefile

- Example (version 2)

```
NAME    = myprog
CC      = gcc
INCDIR  = ./include
CFLAGS  = -Wall -O3 -I $(INCDIR)
LDFLAGS = -lm
BINDIR  = /usr/local/bin

all: $(NAME)

$(NAME):
    $(CC) main.c dummy_functions.c -o $(NAME) $(CFLAGS) $(LDFLAGS)

clean:
    rm -f $(NAME)
    rm -f main.o dummy_functions.o

install: $(NAME)
    cp $(NAME) $(BINDIR)
…
```

To have a *program name* target is a common approach

Makefile

- Example (version 3): automatic variables exploitation

```
NAME    = myprog
CC      = gcc
INCDIR  = ./include
CFLAGS  = -Wall -O3 -I $(INCDIR)
LDFLAGS = -lm
BINDIR  = /usr/local/bin

all: $(NAME)

$(NAME):
    $(CC) main.c dummy_functions.c -o $@ $(CFLAGS) $(LDFLAGS)

clean:
    rm -f $(NAME)
    rm -f main.o dummy_functions.o

install: $(NAME)
    cp $(NAME) $(BINDIR)
…
```

Automatic variable with value set to the target name, in this case "$(NAME)"

Makefile

- Example (version 4): automatic variables exploitation

```
NAME    = myprog
CC      = gcc
INCDIR  = ./include
CFLAGS  = -Wall -O3 -I $(INCDIR)
LDFLAGS = -lm
BINDIR  = /usr/local/bin

all: $(NAME)

$(NAME): main.o dummy_functions.o
    $(CC) $^ -o $@ $(CFLAGS) $(LDFLAGS)

main.o:
    $(CC) -c main.c $(CFLAGS) $(LDFLAGS)

dummy_functions.o:
    $(CC) -c dummy_functions.c $(CFLAGS) $(LDFLAGS)
…
```

Automatic variable set to the list of the target prerequisites (`main.o dummy_functions.o`)

Makefile

- Example (version 5): automatic variables exploitation

```
NAME    = myprog
CC      = gcc
INCDIR  = ./include
CFLAGS  = -Wall -O3 -I $(INCDIR)
LDFLAGS = -lm
BINDIR  = /usr/local/bin

all: $(NAME)

$(NAME): main.o dummy_functions.o
    $(CC) $^ -o $@ $(CFLAGS) $(LDFLAGS)

%.o: %.c
    $(CC) -c $< $(CFLAGS) $(LDFLAGS)

clean:
    rm -f $(NAME)
    rm -f *.o
…
```

Automatic variable set to the first prerequisite (the .c file)

Makefile

- Project building

  If no target is specified, "`all`" is provided by default

```
$ make
```

```
gcc -c main.c -O3 -Wall -I ./include  -lm
gcc -c dummy_functions.c -O3 -Wall -I ./include  -lm
gcc main.o dummy_functions.o -o myprog -O3 -Wall -I ./include  -lm
```

- Program execution

```
0 is even. 0^2=0
1 is odd
2 is even. 2^2=4
3 is odd
4 is even. 4^2=16
5 is odd
6 is even. 6^2=36
7 is odd
8 is even. 8^2=64
9 is odd
```

```
10 is even. 10^2=100
11 is odd
12 is even. 12^2=144
13 is odd
14 is even. 14^2=196
15 is odd
16 is even. 16^2=256
17 is odd
18 is even. 18^2=324
19 is odd
```

Useful `make` options

`-j<n>`	Number of jobs to run simultaneously
`-f`	Specify the makefile to process
`-d`	Print debug information during the process
`-e`	Take variable values from the environment
`-v`	Version of the make tool
`-n`	Print the command to execute, without executing
…	…

```
$ man make
```

Debug version

- To properly debug an application we need to “export symbols”
  This is achieved by adding the **-g** option to **gcc**
- Given our example Makefile we can “optionally” add the option

```
NAME   = myprog
CFLAGS = -Wall -I $(INCDIR)
ifdef (DEBUG)
CGFLAGS += -g -DDEBUG
else
CFLAGS  += -O3
endif
…
```

Perform compiler optimization only if this is not the DEBUG build version!

```
$ DEBUG=1 make
```

```
gcc -c main.c -O3 -Wall -I ./include -g -DDEBUG -lm
gcc -c dummy_functions.c -O3 -Wall -I ./include -g -DDEBUG -lm
gcc main.o dummy_functions.o -o myprog -O3 -Wall -I ./include -g -DDEBUG -lm
```

GNU debugger

- **gdb** is the most common tool for code debugging in Linux
- Once we got the "debug version" of our program we can launch `gdb` and load the program

```
$ gdb myprog
(gdb)_
```

OR

```
$ gdb
(gdb)file myprog
(gdb)_
```

- `gdb` provides an interactive shell with history and auto-completion
- `help` [command] is available to get information about commands usage

GNU debugger

- To run the program

```
(gdb)run
```

- To set a breakpoint on a code line

```
(gdb)break dummy_functions.c:10
```

- To set a breakpoint on a function entry point

```
(gdb)break pow2
```

- To continue the execution until the program end or a new breakpoint has been reached

```
(gdb)continue
```

GNU debugger

- To continue step-by-step

```
(gdb)step
```

- To continue step-by-step, without entering the next function

```
(gdb)next
```

- To see the value of a variable

```
(gdb)print variable-name
```

- To control write accesses to a variable, and stop the execution whenever the variable value changes

```
(gdb)watch variable-name
```

GNU debugger

- In case of *segmentation faults* it becomes useful to observe the stack of function calls

```
(gdb)backtrace

#0  pow2 (number=2) at dummy_functions.c:10
#1  0x000000000040067e in main (argc=1, argv=0x7fffffffd8f8) at main.c:1
```

- To see the chunk of code source on which the execution is stopped

```
(gdb) list

5   int is_even(int number) {
6       return !(number%2);
7   }
8
9   double pow2(double number) {
10      return pow(number, 2.0);
11  }
```

GNU debugger

- To list the breakpoints

```
(gdb)info breakpoints

1       breakpoint     keep y   0x0000000000400674 in main at main.c:12
    breakpoint already hit 2 times
2       breakpoint     keep y   0x00007ffff7afa990 in __pow at w_pow.c:26
    breakpoint already hit 1 time
4       breakpoint     keep y   0x00000000004006ef in pow2 at
dummy_functions.c:10
    breakpoint already hit 1 time
```

- Then it is possible to delete previously added breakpoints (e.g., breakpoint at "`dummy_functions.c:10`")

```
(gdb)delete 4
```

strace

- Applications access the operating system services through *system calls* (syscalls)
- We can get a trace of the system calls invocations by exploiting the `strace` tool

```
$ strace ./myprog
```

```
execve("./myprog", ["./myprog"], [/* 91 vars */]) = 0
…
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 5), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff6ab749000
write(1, "4 is even. 4^2=16\n", 184 is even. 4^2=16)     = 18
write(1, "5 is odd\n", 95 is odd)                 = 9
write(1, "6 is even. 6^2=36\n", 186 is even. 6^2=36)     = 18
write(1, "7 is odd\n", 97 is odd)                 = 9
write(1, "8 is even. 8^2=64\n", 188 is even. 8^2=64)     = 18
write(1, "9 is odd\n", 99 is odd)                 = 9
write(1, "10 is even. 10^2=100\n", 2110 is even. 10^2=100
...
write(1, "19 is odd\n", 1019 is odd)               = 10
exit_group(0)
```

strace

- It is possible to generate a profiling report about system calls

```
$ strace -c ./myprog
```

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 38.58    0.000049           2        20           write
 19.69    0.000025           1        22        20 open
 11.81    0.000015           1        20        18 stat
  9.45    0.000012           2         8           mmap
  7.09    0.000009           2         4           mprotect
  3.94    0.000005           2         3         3 access
  3.15    0.000004           4         1           munmap
  3.15    0.000004           4         1           execve
  0.79    0.000001           1         1           read
  0.79    0.000001           0         3           fstat
  0.79    0.000001           1         1           brk
  0.79    0.000001           1         1           arch_prctl
  0.00    0.000000           0         2           close
------ ----------- ----------- --------- --------- ----------------
100.00    0.000127                    87        41 total
```

ltrace

- Similarly to `strace` this this tool allows us to trace all the calls to library functions

```
$ ltrace ./myprog
```

```
__libc_start_main(0x400470, 1, 0x7fff8a5fed18, 0x4005f0 <unfinished ...>
__printf_chk(1, 0x400674, 0, 0)                                 = 11
__printf_chk(1, 0x400681, 0, 10 is even. 0^2=0)                 = 6
__printf_chk(1, 0x40068c, 1, 51 is odd)                         = 9
__printf_chk(1, 0x400674, 2, 1)                                 = 11
__printf_chk(1, 0x400681, 2, 12 is even. 2^2=4                  = 6
__printf_chk(1, 0x40068c, 3, 53 is odd)                         = 9
__printf_chk(1, 0x400674, 4, 1)                                 = 11
__printf_chk(1, 0x400681, 4, 14 is even. 4^2=16)                = 7
__printf_chk(1, 0x40068c, 5, 65 is odd)                         = 9
…

__printf_chk(1, 0x40068c, 19, 819 is odd)                       = 10
+++ exited (status 0) +++
```

ltrace

- Also in this case we can profile the number of function calls the and time spent

```
$ ltrace -c ./myprog
```

```
0 is even. 0^2=0
1 is odd
2 is even. 2^2=4
3 is odd
…
17 is odd
18 is even. 18^2=324
19 is odd

% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
100.00    0.002146          71        30 __printf_chk
------ ----------- ----------- --------- --------------------
100.00    0.002146                    30 total
```

References

(1)https://sourceware.org/binutils/docs/ld/Scripts.html

(2) http://wiki.osdev.org/Linker_Scripts

(3)http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf

(4)http://www.gnu.org/software/gdb/

(5)http://www.gnu.org/software/make/

(6)https://gcc.gnu.org/

(7)http://www.cprogramming.com/