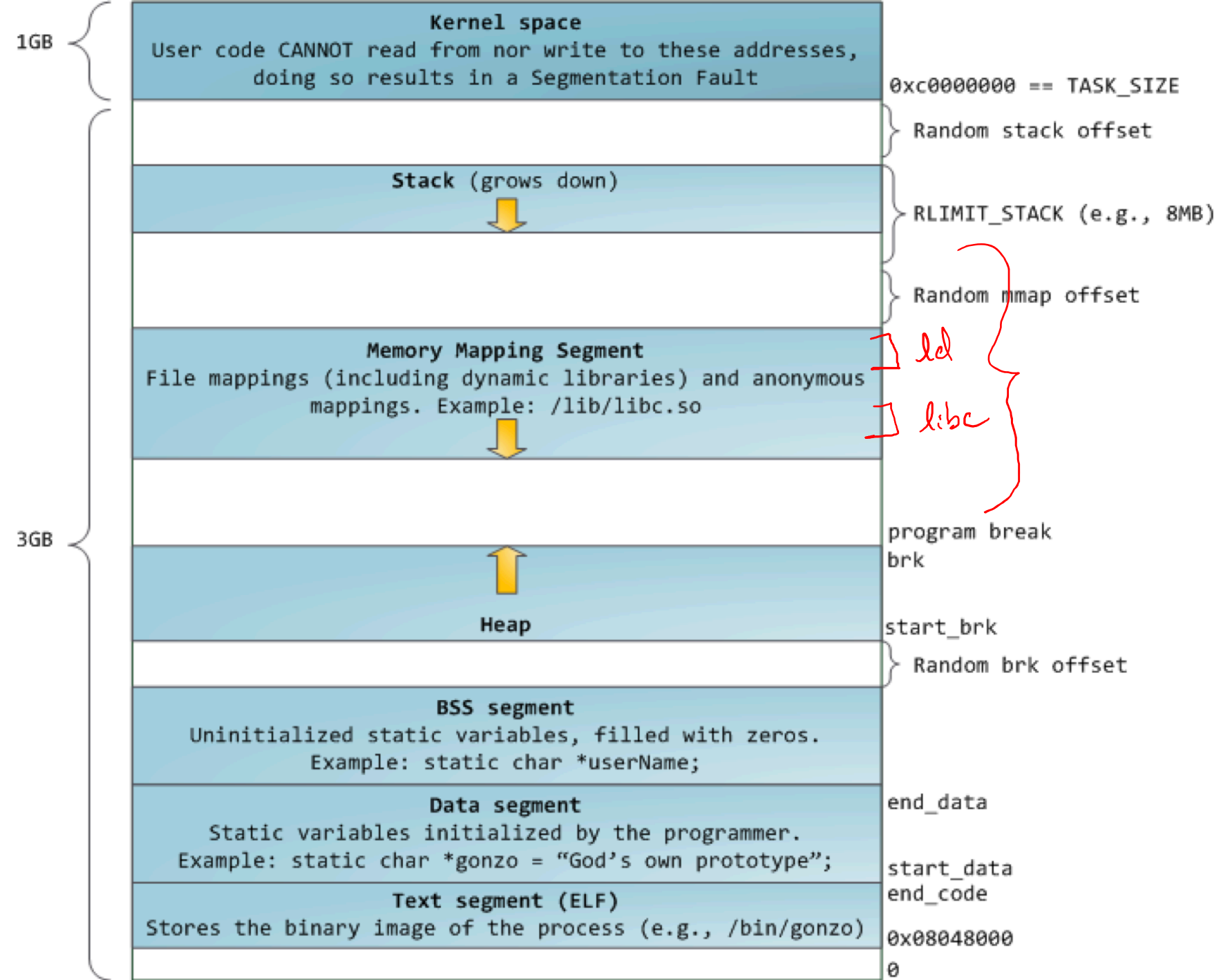


DEMO
DAY!



Recall...



Demo: probe.c

We'll walk through this demo in class. The code can be found here:

https://github.com/traviswpeters/cs476-code/tree/master/00_intro/probe

You Try!

Think. Pair (Break Out Rooms). Share.

- Get the code:

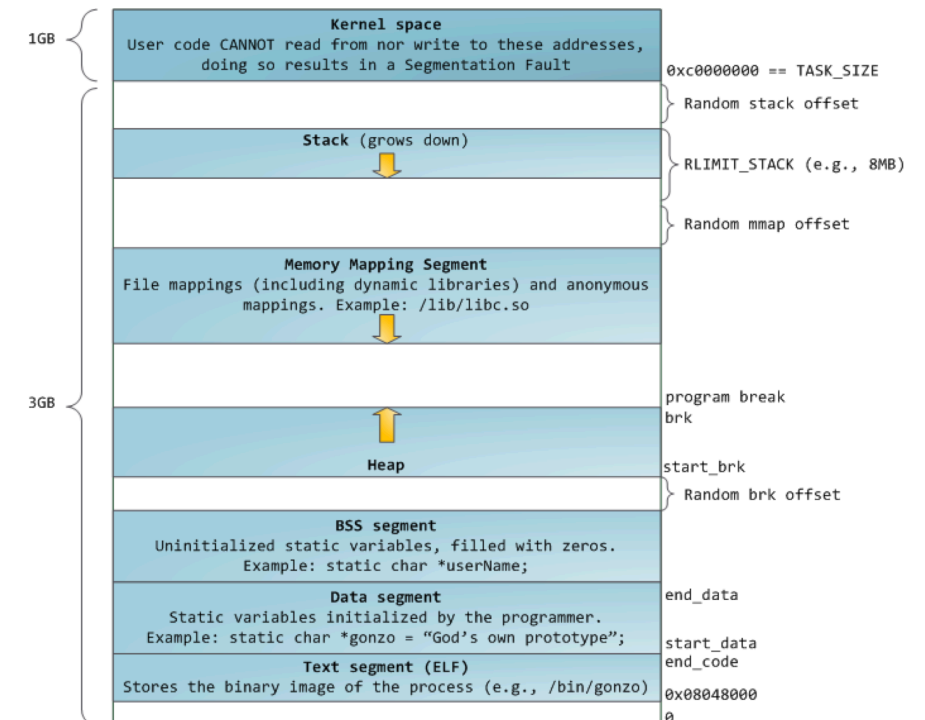
```
[seed@VM][~]$ git clone https://github.com/traviswpeters/cs476-code.git class-code
[seed@VM][~]$ cd class-code/00_intro/probe
[seed@VM][~]$ make
[seed@VM][~]$ ./probe
```

- Explore:

The "probe" program prints addresses for where various things are in memory.

According to the output from "probe":

- Where is "main" located in memory?
- Where is "printf" located in memory?
- Where is "argv" located in memory?
- Where is "environ" located in memory?
- Did you observe anything interesting/unexpected?



Compiling Code

Compiling C Programs

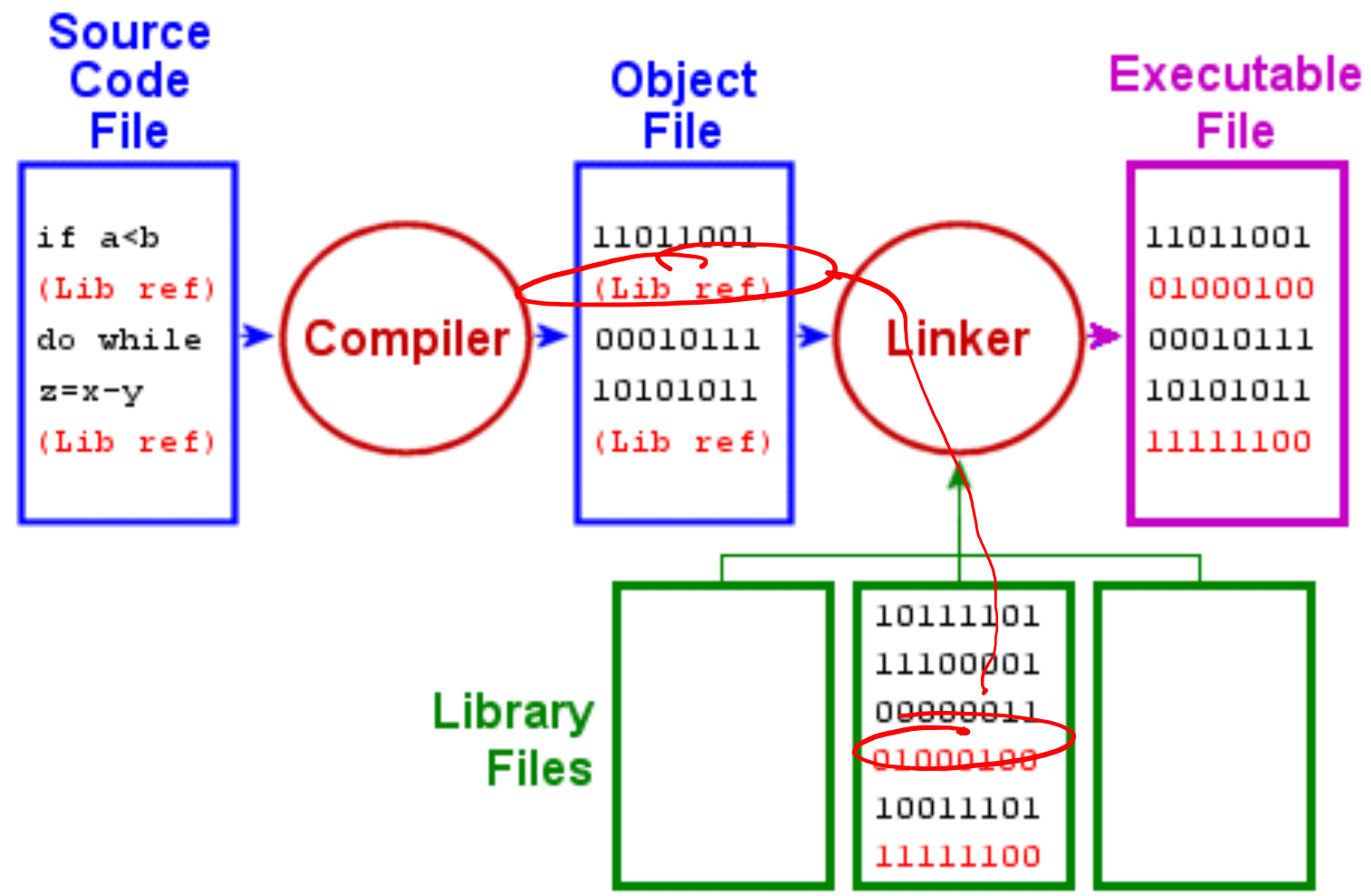
Source Code File

```
if a<b  
(Lib ref)  
do while  
z=x-y  
(Lib ref)
```

Executable File

```
11011001  
01000100  
00010111  
10101011  
11111100
```

Compiling C Programs (cont.)



Compiling C Programs (cont.)

0. write some C/C++ code (.c, .h, .cpp files)

\$ gcc ...

1. Invoke preprocessor to resolve directives
(e.g., #define, #include, #if, etc.)

preprocessor
(cpp)

.i files

2. invoke compiler to
produce assembly code

compiler
(cc, gcc, g++)

.s files

3. invoke assembler to
produce machine code
("object code")

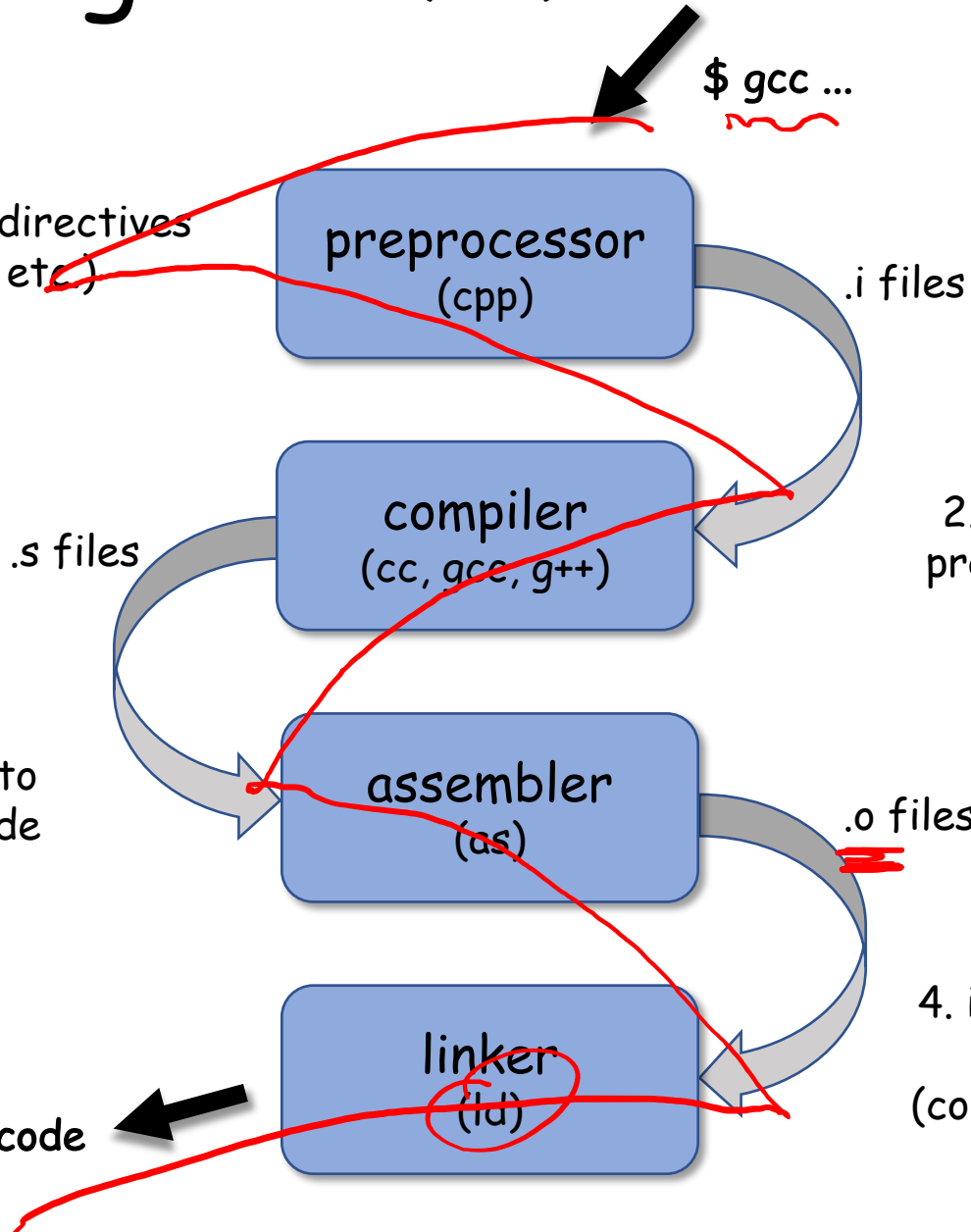
assembler
(as)

.o files

4. invoke the linker to produce
an executable file
(combines .o files and libraries:
.a, .lib, etc.)

linker
(ld)

executable machine code
(a.out, .exe, ...)



Demo: compilation_example.c

We'll walk through this demo in class. The code can be found here:

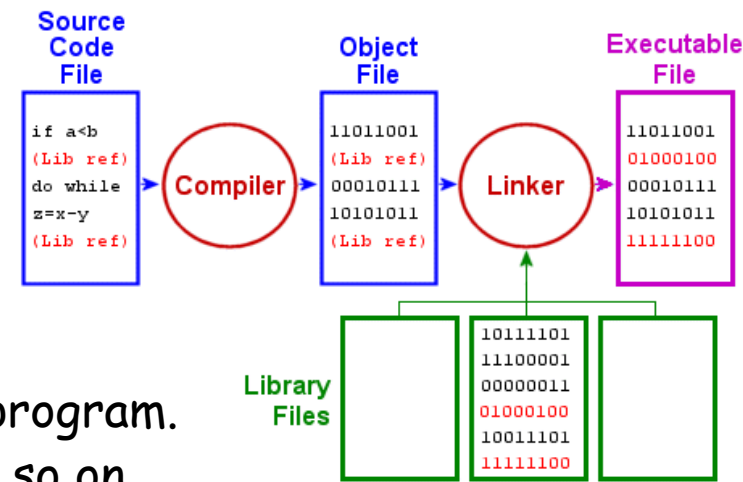
https://github.com/traviswpeters/cs476-code/tree/master/00_intro/pba

Common gcc Flags

- `gcc -o OUTFILE` *set the name of the resulting executable (default = a.out)*
- `gcc -E` *Stop after the preprocessing stage; do not run the compiler proper.*
- `gcc -S` *stop after the stage of compilation proper; do not assemble (produces .s files)*
- `gcc -C` *compile but do not link (produces .o files)*
- `gcc -DVAR` *acts like `#define` in the source code; it sets the value of a symbol (default is 1)*
- `gcc -I../headers` *specify include file in a non-standard directory.*
- `gcc -lname` *link a library*
 - *NOTE: library "name" is linked (system search e.g., /usr/lib/libname.a)*
 - *NOTE: this is a lowercase L ("ell") not an uppercase I ("eye")*
 - *NOTE: you can tell gcc to look in a non-standard location first with -L../libs/lib*
 - *NOTE: must be run at the end*

Also check out: `-Wall`, `-ggdb`, `-ON`, `-m32` `-fno-builtin`, `-masm=intel...`

Libraries



- A library is...
 - an assortment of pre-compiled pieces of code that can be reused in a program.
 - they provide reusable functions, routines, classes, data structures and so on (written by a another programmer), which they can use in their programs.
 - Examples: libc (/lib64/libc.so.6), glibc, libcurl, libcrypt, pthreads,
 - so = "shared object"
- Linux supports two classes of libraries, namely:
 - Static Libraries - are bound to a program statically at compile time.
 - Dynamic/Shared Libraries - are loaded when a program is launched and loaded into memory and binding occurs at run time. These can further be categorized into:
 - Dynamically Linked Libraries - here a program is linked with the shared library and the kernel loads the library (in case it's not in memory) upon execution.
 - Dynamically Loaded Libraries - the program takes full control by calling functions with the library.
 - Shared libraries are loaded by ld.so (or ld.so.x) and ld-linux.so (or ld-linux.so.x) programs

Compiled Code

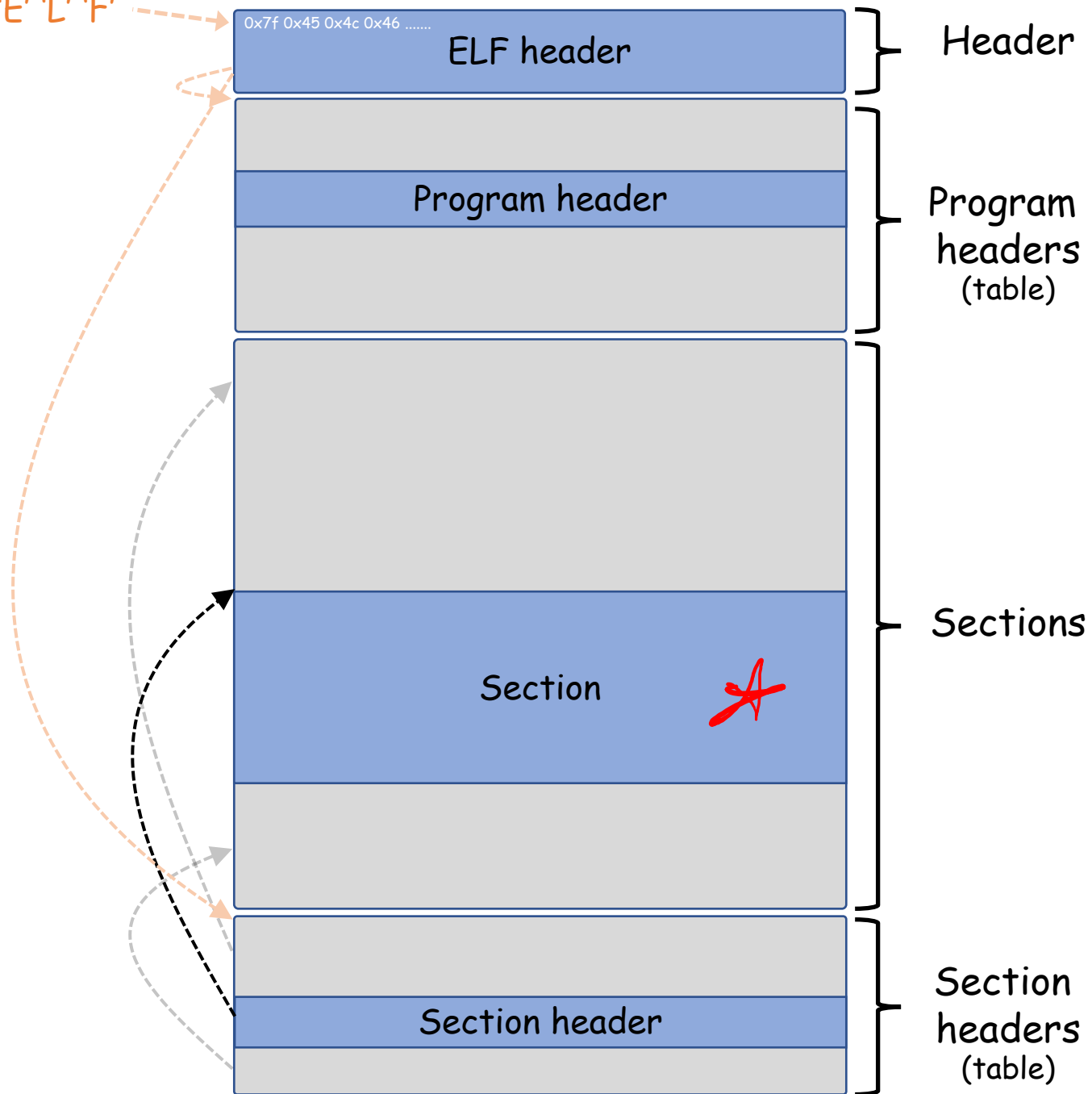
ELF

Executable and Linkable Format

```
[seed@VM] [~]$ readelf ...  
[seed@VM] [~]$ objdump ...
```



0x7f 'E' 'L' 'F'



ANGE ALBERTINI
CORKAMI.COM

```
~$ uname -m
armv7l
~$ ./simple.ARM
Hello World!
```

SIMPLE.ARM

www.simplearm.com

SECTIONS

HEADER²⁷

00 00 00 00	ELF HEADER	0	..ELF
00 00 00 00	IDENTIFY AS AN ELF TYPE	0	..ELF
00 00 00 00	SPECIFY THE ARCHITECTURE	0	..ELF
PROGRAM HEADER TABLE			

E3 14 10 8F	CODE
EF 01 00 80	EXECUTABLE INFORM

DATA
INFORMATION USED BY THE

68 73 7 SECTIONS' NAMES

SECTION HEADER TABLE

HEXADECIMAL DUMP

ASCII DUMP

	FIELDS	VALUES	EXPLANATION
1	<code>e_identry</code>	<code>0x7f</code> "ELF"	CONSTANT SIGNATURE
	<code>e_mh</code>	<code>1</code> <small>1 = 32 bits, 2 = 64 bits</small>	32 BITS LITTLE-ENDIAN
	<code>e_class</code> , <code>ei_data</code>	<code>1</code>	ALWAYS 1
	<code>ei_version</code>	<code>1</code>	EXECUTABLE
	<code>e_type</code>	<code>2</code>	ARM PROCESSOR
	<code>e_machine</code>	<code>28</code>	ARM PROCESSOR
	<code>e_version</code>	<code>1</code>	ALWAYS 1
2	<code>e_phoff</code>	<code>0x00000000</code>	ADDRESS WHERE EXECUTION STARTS
	<code>e_shoff</code>	<code>0x40</code>	PROGRAM HEADERS OFFSET
	<code>e_shsize</code>	<code>0x00000000</code>	SECTION HEADERS OFFSET
	<code>e_phentsize</code>	<code>0x10</code>	ELF HEADERS SIZE
	<code>e_phnum</code>	<code>1</code>	SIZE OF A SINGLE PROGRAM HEADER
	<code>e_shentsize</code>	<code>0x28</code>	SIZE OF A SINGLE SECTION HEADER
	<code>e_shnum</code>	<code>4</code>	COUNT OF SECTION HEADERS
	<code>e_shstrndx</code>	<code>3</code>	INDEX OF THE NAMES SECTION IN THE TABLE
3	<code>p_offset</code>	<code>0</code>	THE SEGMENT SHOULD BE LOADED IN MEMORY
	<code>p_vaddr</code>	<code>0x00000000</code>	OFFSET WHERE IT SHOULD BE READ
	<code>p_paddr</code>	<code>0x00000000</code>	VIRTUAL ADDRESS WHERE IT SHOULD BE LOADED
	<code>p_filesz</code>	<code>0x90</code>	PHYSICAL ADDRESS WHERE IT SHOULD BE LOADED
	<code>p_memsz</code>	<code>0x90</code>	SIZE IN MEMORY
	<code>p_flags</code>	<code>5</code> <small>0 = readable, 1 = writable, 2 = executable</small>	READABLE AND EXECUTABLE

ARM ASSEMBLY

```

3  mov r2, #13
   add r1, pc, #20
   mov r0, #1
   mov r7, #1
   svc 0
   =write(STDOUT_FILENO, "hello world!\n", len("Hello world!\n"))

   mov r0, #1
   mov r7, #1
   svc 0
   =write(1, "hello world!\n", len("Hello world!\n"))

```

ST

```

..      "Hello World!\n", 0
.....
SECTION NAMES
text    "" .shstrtab .text .rodata

```

INDEX	NAME	TYPE	FLAGS	ADDRESS	OFFSET	SIZE
0	<null>	0				
1	.text	1	6	0x80000060	0x60	0x20
2	.rodata	2	2	0x80000080	0x80	0x00
3	.shstrtab	3			0x90	0x19

THIS IS THE WHOLE FILE, HOWEVER, MOST ELF FILES CONTAIN MANY MORE ELEMENTS. EXPLANATIONS ARE SIMPLIFIED FOR CONCISENESS.

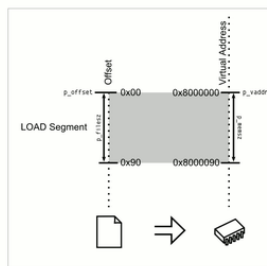
LOADING PROCESS

1 HEADER

```
THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
(SECTIONS ARE NOT USED)
```

2 MAPPING

THE FILE IS MAPPED IN MEMORY
ACCORDING TO ITS SEGMENT(S)



3 EXECUTION

ENTRY IS CALLED
SYSCALLS ARE ACCESSED VIA:

- SYSCALL NUMBER IN THE R7 REGISTER
- CALLING INSTRUCTION SVC

TRIVIA

THE ELF WAS FIRST SPECIFIED BY U.S. L. AND U.I.
FOR UNIX SYSTEM V, IN 1989

THE ELF IS USED, AMONG OTHERS, IN:

- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

