OS Lab
Session 5: Programming in Linux
&
IPC

AUT – CEIT

Instructor: @MerajNouredini

Programming in Linux

Programming in Linux

- In previous session, We executed our code using CLion IDE
- In this session we examine the **execution** and the **programming process** in detail

Executables in Linux

Executables in Linux

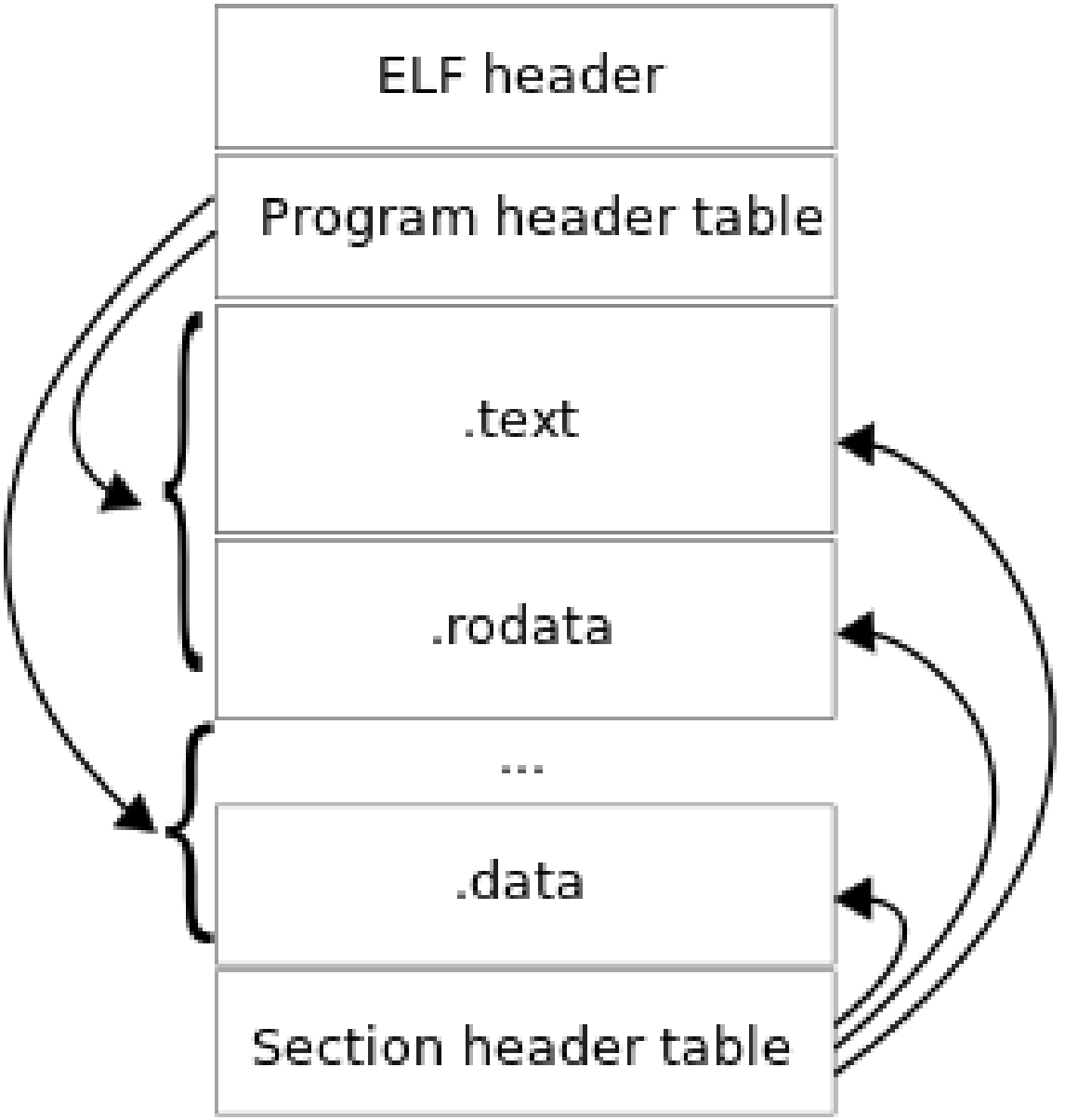
- We know **exe** files from Windows
- But in Linux, nearly any file can be executable!
- We saw that before! the `x` file attribute!
- So, What was the format of the files we executed in previous session?

ELF

ELF

- **Executable and Linkable Format** (ELF, formerly named Extensible Linking Format), is a common **standard file format** for *executable files, object code, shared libraries, and core dumps*.
- First published in the specification for the application binary interface (ABI) of the Unix operating system version named System V Release 4 (SVR4)

ELF File layout



How to generate ELF?

GNU Compiler Collection

GCC

- The original GNU C Compiler (GCC) is developed by **Richard Stallman**
- GCC, formerly for "GNU C Compiler", has grown over times to support many languages such as C++, Objective-C, Java, Fortran and Ada. It is now referred to as "GNU Compiler Collection"

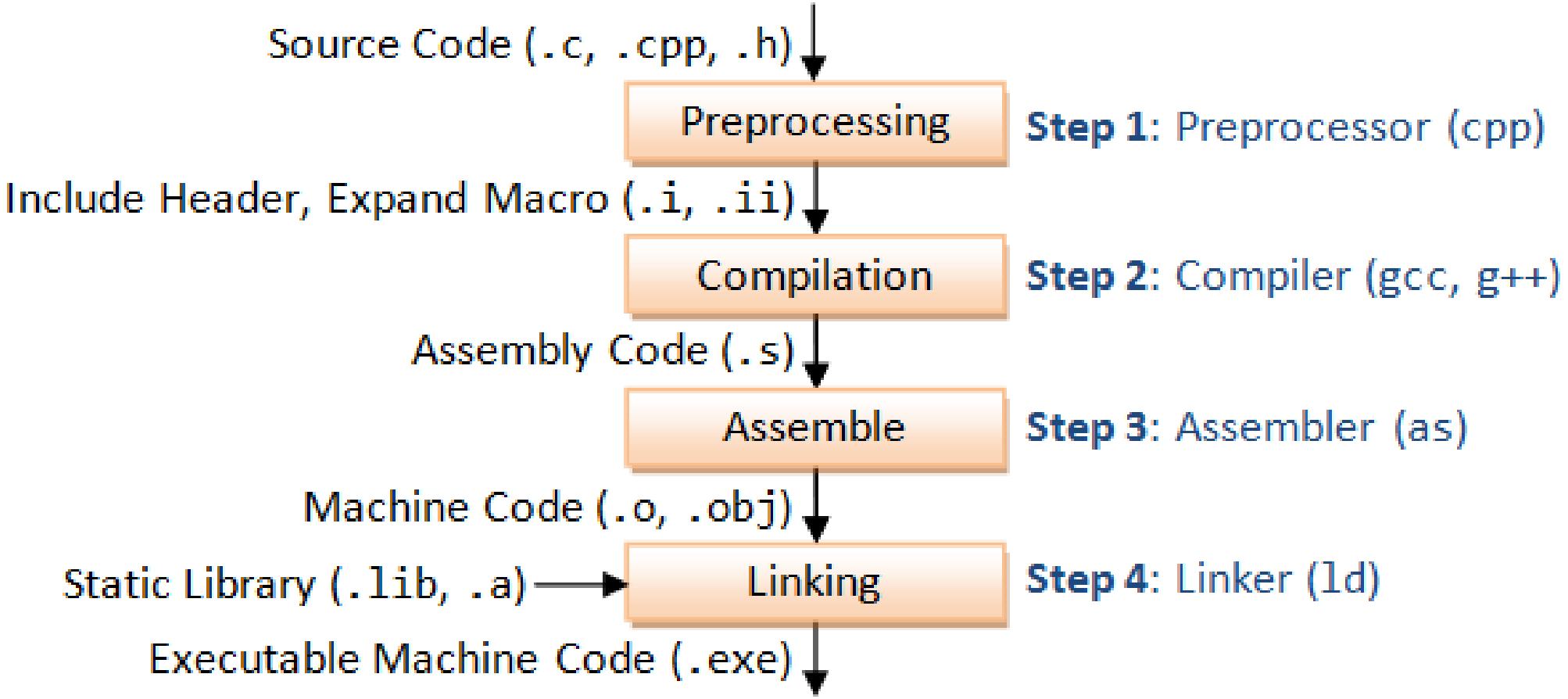
GCC

- GCC is a key component of "GNU Toolchain":
  - GNU Compiler Collection (GCC): a compiler suit that supports many languages, such as C/C++, Objective-C and Java.
  - GNU Make: an automation tool for compiling and building applications.
  - GNU Binutils: a suit of binary utility tools, including linker and assembler.
  - GNU Debugger (GDB).
  - GNU Autotools: A build system including Autoconf, Autoheader, Automake and Libtool.
  - GNU Bison: a parser generator (similar to lex and yacc).

Let’s see GCC in action

gcc in action

- gcc hello.c
  - Compile and link source file hello.c
- gcc -Wall -g -o hello.out hello.c
  - -o: specifies the output executable filename.
  - -Wall: prints "all" warning messages.
  - -g: generates additional symbolic debuggging information for use with gdb debugger.
- gcc -c -Wall -g hello.c
  - Compile-only with -c option
- gcc -o myprog.out file1.c file2.c


Source Code (.c, .cpp, .h)
Preprocessing
Step 1: Preprocessor (cpp)
Include Header, Expand Macro (.i, .ii)
Compilation
Step 2: Compiler (gcc, g++)
Assembly Code (.s)
Assemble
Step 3: Assembler (as)
Machine Code (.o, .obj)
Static Library (.lib, .a)
Linking
Step 4: Linker (ld)
Executable Machine Code (.exe)

gcc in action

- cpp hello.c > hello.i
- gcc -S hello.i
- as -o hello.o hello.s
- ld -o hello.exe hello.o ...libraries...

gcc in action

- Header Files and Libraries (-I, -L and -l)
- Default Include-paths:
  - cpp -v
  - -I<dir> or or environment variable CPATH, LIBRARY_PATH
- nm
- ldd

That’s very DIFFICULT!

Makefile

makefile

- target: pre-req-1 pre-req-2 ...

  <tab>command

- Phony Targets
  - A target that does not represent a file (e.g clean)
- Variables
  - A variable begins with a $ and is enclosed within parentheses (...) or braces {…}

makefile

- Automatic Variables
  - $@: the target filename.
  - $*: the target filename without the file extension.
  - $<: the first prerequisite filename.
  - $^: the filenames of all the prerequisites, separated by spaces, discard duplicates.
  - $+: similar to $^, but includes duplicates.
  - $?: the names of all prerequisites that are newer than the target, separated by spaces.

makefile

```
# $@ matches the target; $< matches the first dependent
hello.exe: hello.o
    gcc -o $@ $<
```

makefile

- Virtual Path - VPATH & vpath
  - VPATH = src include
  - vpath %.c src
  - vpath %.h include

makefile

- Pattern Rules
  - A pattern rule, which uses pattern matching character '%' as the filename, can be applied to create a target, if there is no explicit rule

Questions?