Computers
Compilation Environment & System Libraries

Grau en Ciència i Enginyeria de Dades

Xavier Martorell, Xavier Verdú

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC)

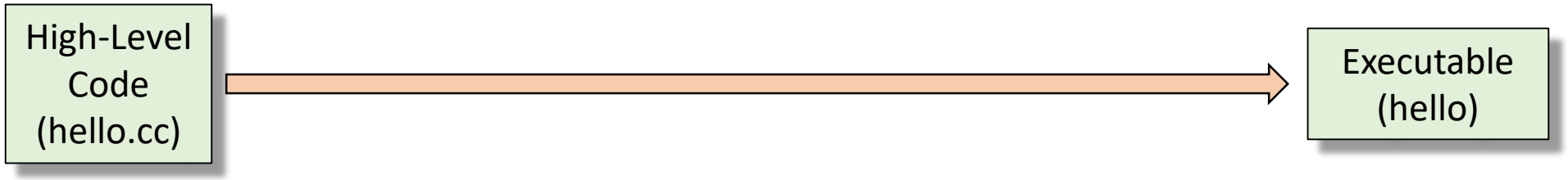
2019-2020 Q2

Creative Commons License

Table of Contents

From High-Level code to program execution

- **How can we transform the high-level source code to an executable file?**



- **Executable file:** program that can be executed on a computer.

Example: C/C++ code (similar to Lab S3 example)

- From high-level code to program execution

High-Level Code (hello.cc)

```
#include <unistd.h>
#include <stdio.h>

int global = 4;

int main(int argc, char **argv){
        char buf[512];
        int num;

        num = sprintf(buf, "Hello World %d!\n", global);
        write(1, buf, num);

        return 0;
}
```

High Level Language Code specified in a file that cannot be understood by the CPU to be executed right away

Example: C/C++ code (similar to Lab S3 example)

```
#include <unistd.h>
#include <stdio.h>

int global = 4;

int main(int argc, char **argv){
        char buf[512];
        int num;

        num = sprintf(buf, "Hello World %d!\n", global);
        write(1, buf, num);

        return 0;
}
```

Including Header Files

Global Variable

Local Variables

Calling to external functions

Return the exit status of the program,
Indicating success(0) or some failure(1)

Header files contain required information for the compiler about external components used in this code, such as “sprintf” and “write” functions

Example of Header Files provided by Libraries

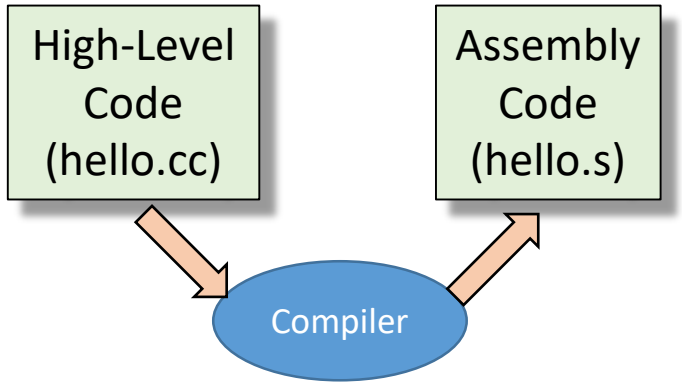
- Many services are reused by [all] applications
  - Operating system interface
  - Basic Math operations
  - Language support
  - Input/output
  - Memory management
  - Character & string manipulation
  - Filesystem
  - ...
  - Parallelism support
  - Message passing
  - ...

C header	/	C++ header
stdio.h, unistd.h, stdlib.h, fcntl.h, sys/types.h, sys/stat.h…		
math.h	/	cmath
stdio.h	/	iostream
	/	stdc++
malloc.h	/	new
string.h	/	string
pthread.h	/	thread
omp.h	/	omp.h
mpi.h	/	mpi.h

Basic Concepts: Compiler

- **Compiler:** software that translates a given code into another language
  - It usually translates high-level code into lower-level code

```
g++   -s   hello.cc
```



- **Assembly Language:** symbolic language that can be translated into binary machine language

Assembly Code (intermediate code)

- cat hello.s

```
    .text
    .globl   main
    .type    main,@function
main:
.LFB12:
    .cfi_startproc
    subq    $520, %rsp                  // buf,num
    .cfi_def_cfa_offset 528
    movl    global(%rip), %edx    // global
    movl    $.LC0, %esi                 // string
    movq    %rsp, %rdi                 // buf
    movb    $0, %al     // num. floating point args
    call      sprintf
    movslq   %eax, %rdx      // num
    movq     %rsp, %rsi       // buf
    movl     $1, %edi          // 1
    call       write
    movl     $0, %eax          // 0
    addq     $520, %rsp
    .cfi_def_cfa_offset 8
    ret
```

Assembly Code (intermediate code)

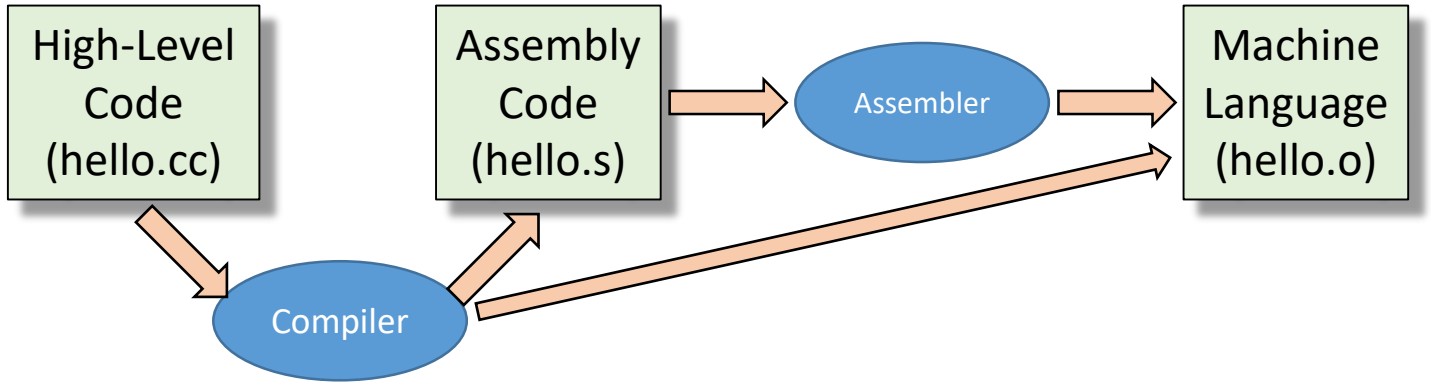
- cat hello.s (cont)

```
        .section        .rodata.str1.1,"aMS",@progbits,1
    .LC0:
        .string "Hello World %d!\n“
        .globl   global
        .data
        .align    4
        .type     global,@object
        .size     global, 4
    global:
         .long    4
```

From High-Level code to program execution

- **Assembler:** software that translates assembly code into machine language
  - The output is also known as **object file** (with **.o** extension)

```
g++   -c   hello.cc
```



- **Object File:** is a combination of machine language instructions, data, and information needed to place instructions properly in memory

Compiler Optimizations

- -O**x** where x is a numerical digit indicating the optimization type
- Assuming we are focused on gcc/g++ compiler
  - -O0: no optimizations. Thus, it reduces compilation time and preserves debugging behaviour
  - -O1: tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time
  - -O2: performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to –O1, this option increases both compilation time and the performance of the generated code
  - -O3: turns on all above optimizations and also turns on advanced optimizations: function inlining, register renaming and loop unrolling among others

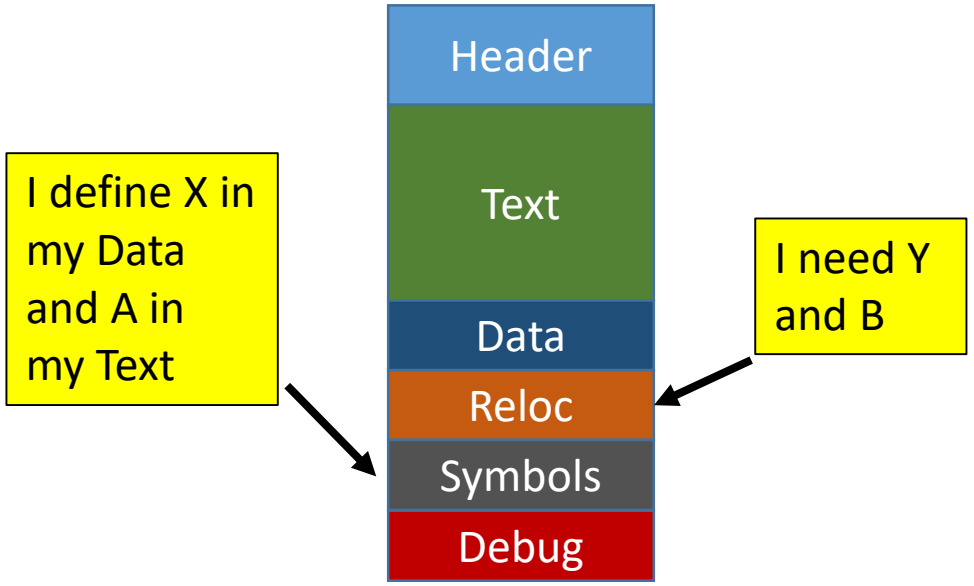
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Object file (machine code)

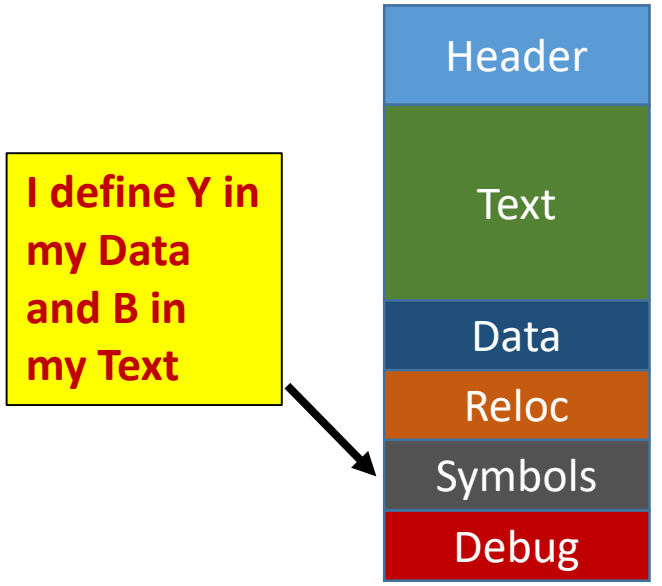
- Object file for UNIX-like OSes includes:
  1) **Object file header:** size and position of other pieces of the object file
  2) **Text segment:** the machine language code
  3) **Static data segment:** data allocated for the whole life of the program (e.g. global variables)
  4) **Relocation information:** identify instructions and data that depend on memory addresses allocated when the program is running
  5) **Symbol Table:** a table that matches label names to the addresses of the memory words that instructions and data occupy
  6) **Debugging information:** additional data for a software (called debugger) to associate machine instructions to high-level language code

- The assembler keeps track of labels used in branches and data transfer instructions to generate the executable

Object file



File1.o



File2.o

Object file (machine code)

- Contents of hello.o (od –t x1c hello.o)

```
0000000  7f  45  4c  46  02  01  01  00  00  00  00  00  00  00  00  00
       177   E   L   F 002 001 001  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020  01  00  3e  00  01  00  00  00  00  00  00  00  00  00  00  00
       001  \0   >  \0 001  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040  00  00  00  00  00  00  00  00  20  03  00  00  00  00  00  00
        \0  \0  \0  \0  \0  \0  \0  \0      003  \0  \0  \0  \0  \0  \0
0000060  00  00  00  00  40  00  00  00  00  00  40  00  0d  00  0c  00
        \0  \0  \0  \0   @  \0  \0  \0  \0  \0   @  \0  \r  \0  \f  \0
0000100  ...
0000200  48  65  6c  6c  6f  20  57  6f  72  6c  64  20  25  64  21  0a
         H   e   l   l   o       W   o   r   l   d       %   d   !  \n
0000220  00  00  47  43  43  3a  20  28  53  55  53  45  20  4c  69  6e
        \0  \0   G   C   C   :      (   S   U   S   E      L   i   n
0000240  75  78  29  20  38  2e  31  2e  30  00  00  00  00  00  00  00
         u   x   )      8   .   1   .   0  \0  \0  \0  \0  \0  \0  \0
0000260  14  00  00  00  00  00  00  00  01  7a  52  00  01  78  10  01
       024  \0  \0  \0  \0  \0  \0  \0 001   z   R  \0 001   x 020 001
```

Object file (machine code)

- objdump -xs hello.o

```
Contents of section .text:
 0000 4881ec08 0200008b 15000000 00be0000  H...............
 0010 00004889 e7b000e8 00000000 4863d048  ..H.........Hc.H
 0020 89e6bf01 000000e8 00000000 b8000000  ................
 0030 004881c4 08020000 c3                 .H.......
Contents of section .data:
 0000 04000000                             ....
Contents of section .rodata.str1.1:
 0000 48656c6c 6f20576f 726c6420 2564210a  Hello World %d!.
 0010 00                                   .
Contents of section .comment:
 0000 00474343 3a202853 55534520 4c696e75  .GCC: (SUSE Linu
 0010 78292038 2e312e30 00                 x) 8.1.0.
Contents of section .eh_frame:   // exception support
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
```

Object file (machine code)

- objdump -xs hello.o

```
Contents of section .text:
 0000 4881ec08 0200008b 15000000 00be0000  H...............
 0010 00004889 e7b000e8 00000000 4863d048  ..H.........Hc.H
 0020 89e6bf01 000000e8 00000000 b8000000  ................
 0030 004881c4 08020000 c3                 .H.......
Contents of section .data:
 0000 04000000                             ....
Contents of section .rodata.str1.1:
 0000 48656c6c 6f20576f 726c6420 2564210a  Hello World %d!.
 0010 00                                   .
Contents of section .comment:
 0000 00474343 3a202853 55534520 4c696e75  .GCC: (SUSE Linu
 0010 78292038 2e312e30 00                 x) 8.1.0.
Contents of section .eh_frame:   // exception support
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
```

Unfortunately object files are not self-contained

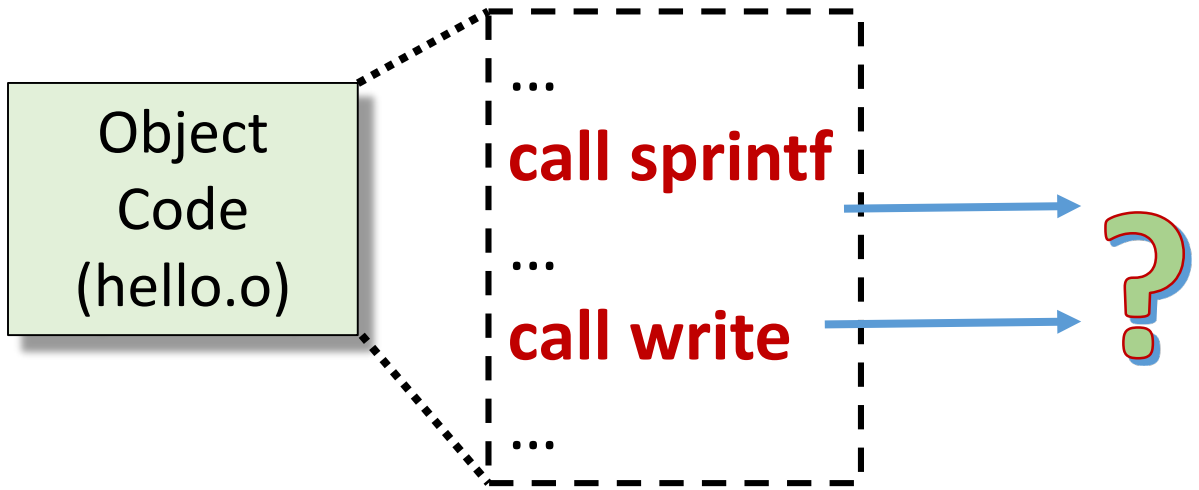
For example, there are calls to functions not defined in our object file:

- Sprintf
- Write

We need their code!!!

From High-Level code to program execution

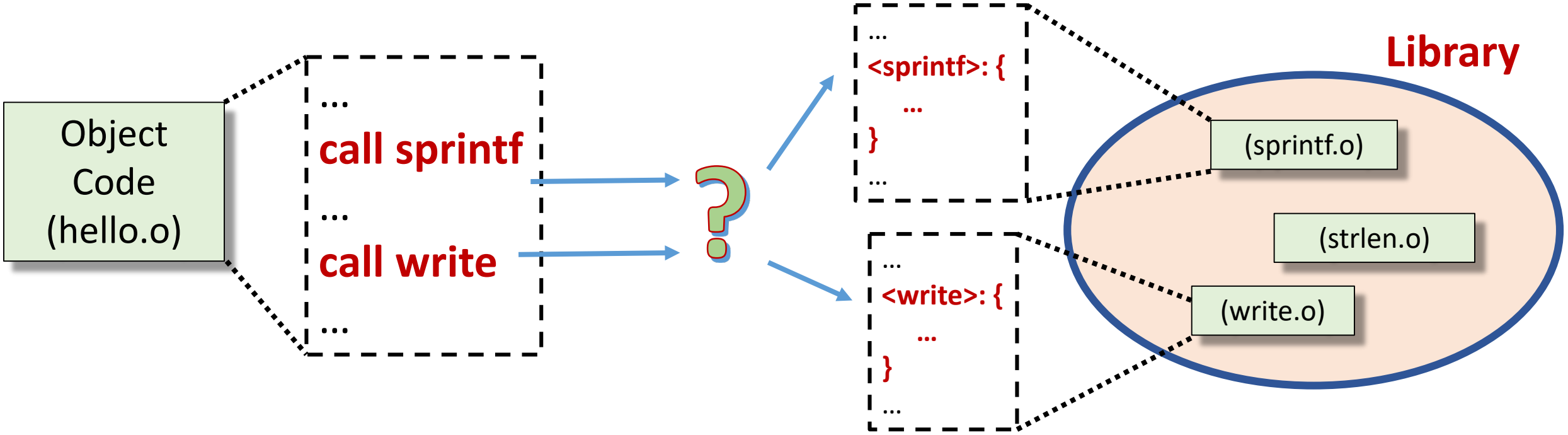
- The same problem happens when the code calls to functions implemented in other object files, although they are not in a library



Where is the implementation of such functions?
The current symbol table has no enough information

From High-Level code to program execution

- **Library:** a file that encapsulates a set of object files
  - Libraries grant **independency**
    - A high-level code is independent of **OS** and **Hw architecture**, but …
    - … executables are created for a particular **OS** and **Hw architecture**



From High-Level code to program execution

- Language Libraries
  - Bind a language to a particular OS, but independently of the hardware architecture
  - Provide functions of the programming language
  - Sometimes they are self-contained (e.g. **math**): independent of OS
  - Sometimes they request services to the OS (e.g. **sprintf**, cin, cout…)

- System Libraries
  - Bind an OS to a particular Hw architecture (independent of the high-level code)
  - Include functions that invoke services/routines of the OS (e.g. **write**, open, …)
  - Depends on the type of Operating System and the architecture of the CPU

- Examples of incompatibility:
  - Linux 32 vs 64 bits, CPU Intel vs ARM, Ubuntu vs OpenSUSE, …

Linux i386 (32bits) vs Linux x64 (64bits)

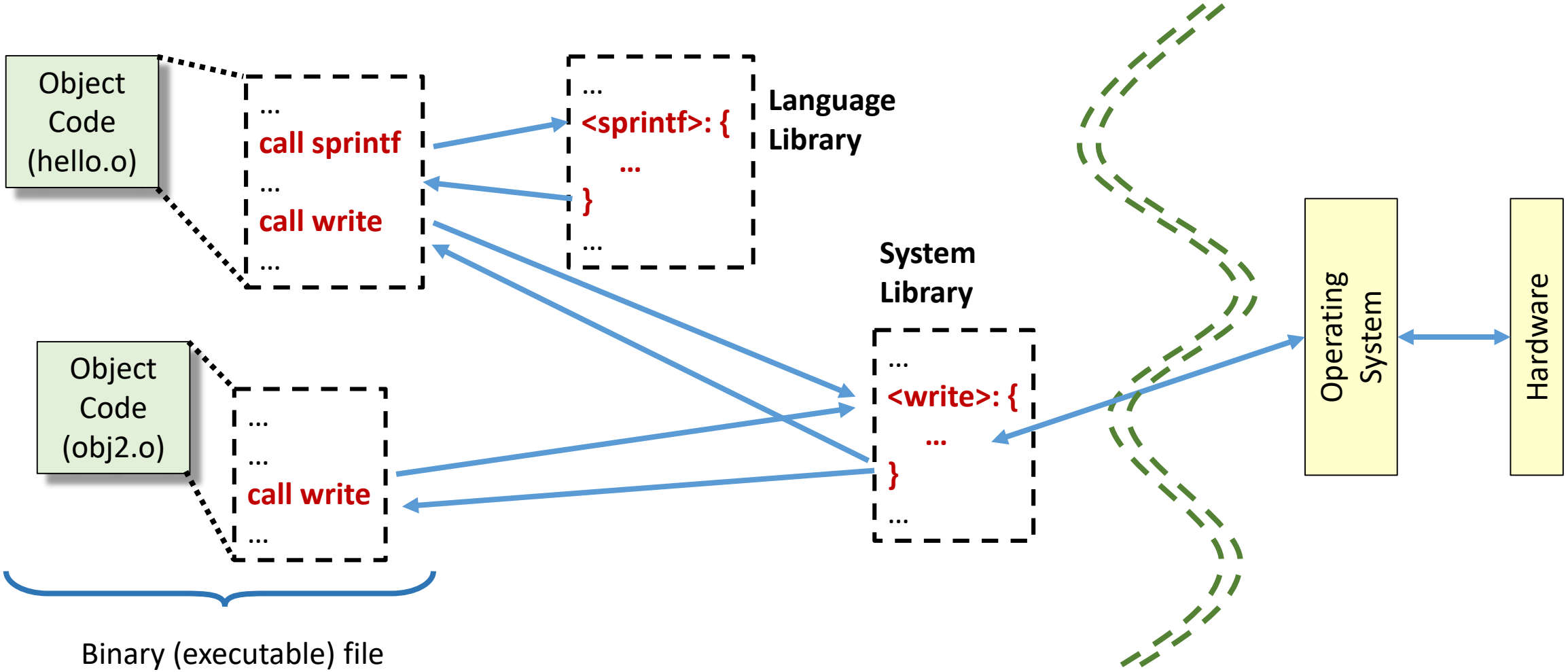
Linux i386

```
...
movl   $4, %eax   ; use the write syscall
movl   $1, %ebx   ; write to stdout
movl   $msg, %ecx ; use string "Hello ..."
movl   $14, %edx  ; write 14 characters
int    $0x80      ; make syscall
...
```

Linux x64

```
...
movq   $1, %rax   ; use the write syscall
movq   $1, %rdi   ; write to stdout
movq   $msg, %rsi ; use string "Hello ..."
movq   $14, %rdx  ; write 14 characters
syscall           ; make syscall
...
```

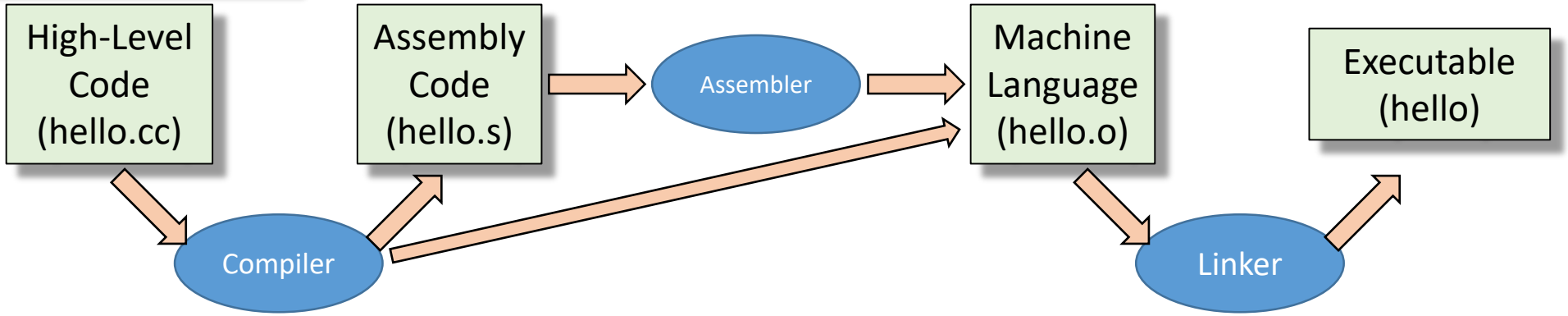
From High-Level code to program execution



From High-Level code to program execution

- **Linker:** a software that combines independent codes (object files) and resolves all undefined labels into an executable

g++ -o hello hello.o



- **Executable file:** program that can be executed on a computer. It has the format of an object file, but all references are resolved. There are no undefined labels
  - It is posible to have partially linked files (unresolved addresses of some library routines)

Static versus Dinamically linking

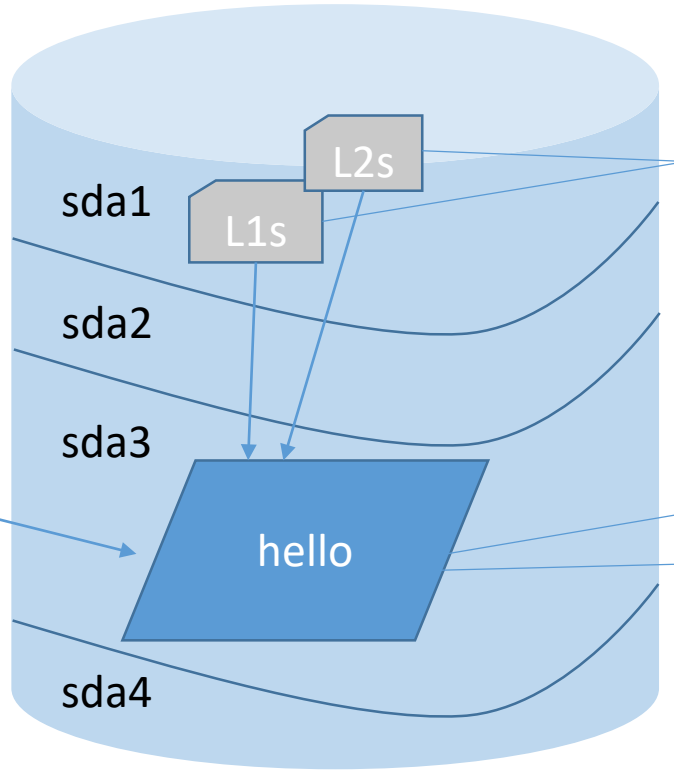
- Static linking
  - The codes of libraries are included INSIDE the executable
  - Waste of space (in disk when stored, and in memory when running)
    - Different programs may use the same libraries
  - Static libraries: "**.a**" in UNIX-like OS; "**.lib**" in Windows-like OS
- Dynamic linking
  - The linking stage is delayed to run-time. Thus, the code is not included in the program
  - Save space in both disk and memory
    - Different programs can access to a shared library loaded in memory
  - Dynamic libraries: "**.so**" in UNIX-like OS; "**.dll**" in Windows-like OS

Static linking

- Static compilation and linking phases

```
#include <unistd.h>
#include <stdio.h>

int global = 4;

int main(int argc, char **argv){
        char buf[512];
        int num;

        num = sprintf(buf, "Hello World %d!", global);
        write(1, buf, num);

        return 0;
}
```

g++ **-static** -o hello hello.o
(compile & link)



Static libraries
Archives
(lib*.a)

Static
Executable
Binary file

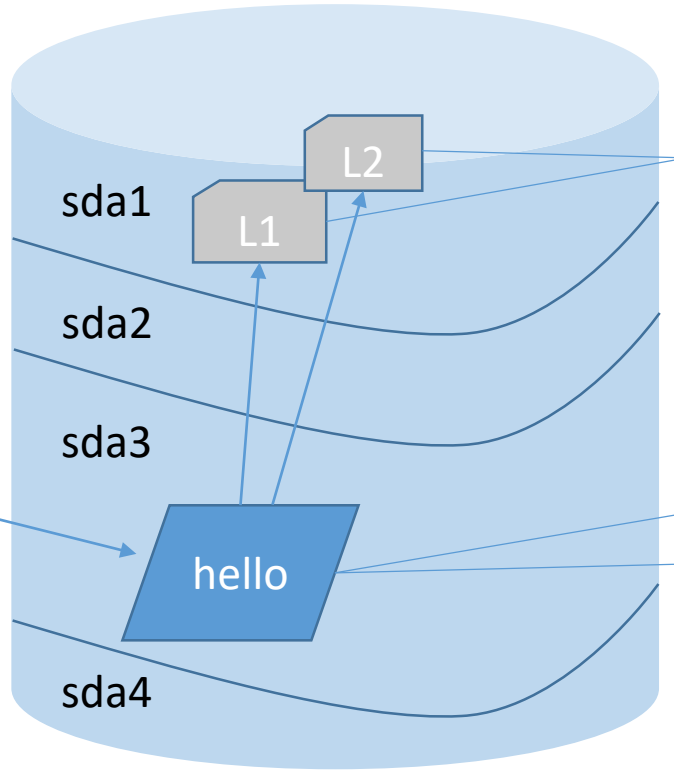
Files in the disk

Dinamically linking

- Dynamic compilation and linking phases

```
#include <unistd.h>
#include <stdio.h>

int global = 0;

int main(int argc, char **argv){
        char buf[512];
        int num;

        num = sprintf(buf, "Hello World %d!", global);
        write(1, buf, num);

        return 0;
}
```

g++ -o hello hello.o
(compile & link)



Executable / binary files

- Files that contain the program code, data, symbols, debug info…
- Generated by the process of compiling and linking
- Specific structure depends on the Operating System
  - Windows – PE32 Portable Executable, PE32+ for 64 bits
  - Linux – ELF Executable and Linkable Format
    - x86_64 64-bits data and 64-bits addresses
    - x32 64-bits data and 32-bits addresses
    - i386 32-bits data and 32-bits addresses (compatible with 32-bits systems)

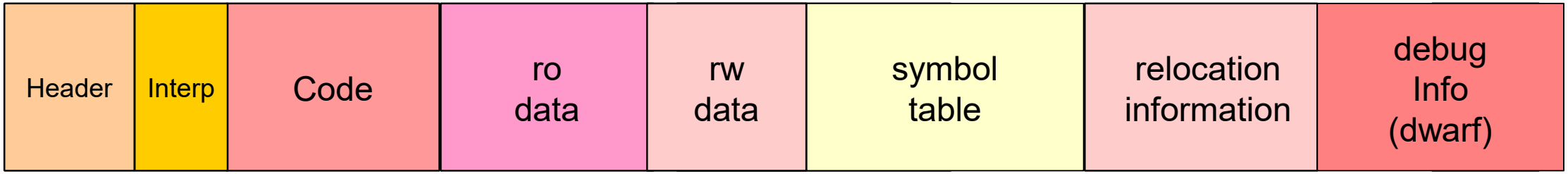
https://en.wikipedia.org/wiki/Portable_Executable

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Table of Contents

Executable File Format

- Contains an arbitrary number of sections:
  - Code: machine language code
  - Data (read-only, read-write):
  - Symbol table: labels and their already resolved addresses
  - Relocations: values that depend on the address where the file is loaded into mem
  - Debug information: associate machine data/instructions to high-level source code
    - File names (*.c, *.cxx, *.h...) and line numbers
    - Data types, classes, structures, unions, variables...



Executable File Format

- File header
  - Contains information about the executable
  - Used by...
    - The ELF utilities to read/understand/write this type of files
    - The Operating System to load **binary** files and **libraries** for execution

Executable File Format

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400470
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6672 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 28
```

Executable File Format

- Interpreter: refers to a support library for loading the executable file onto memory (e.g. /lib64/ld-linux-x86-64.so.2)
- Code: contains the executable instructions
- Data: contains the global variables
  - Read-only data: constant values pre-initialized in the program
    - const int val = 10;
  - Read-write data: variables pre-initialized in the program
    - double degrees = -273.15;
- BSS: Block Started by Symbol. It holds non initialized global variables
  - int x, y, z;

Executable File Format

- Symbol table: local and exported symbols to find/resolve every “label”

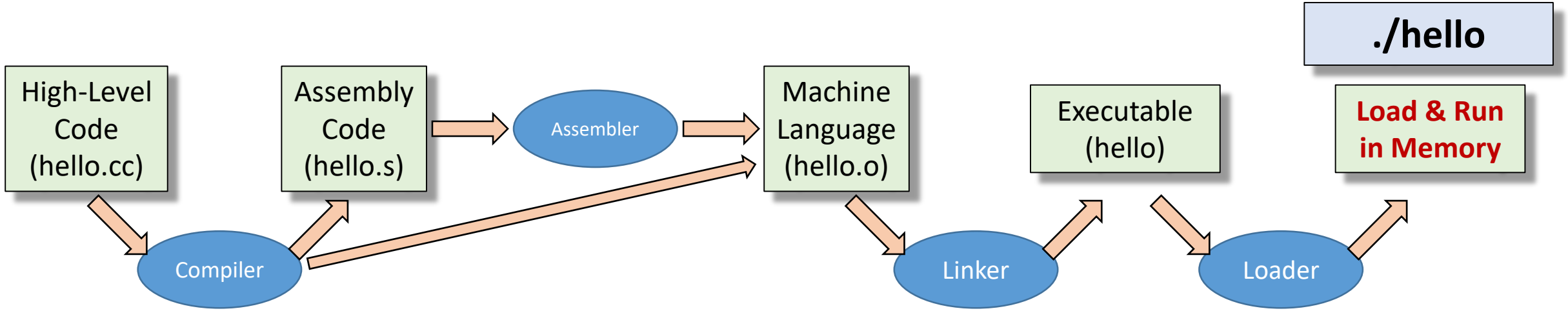
  - <address> <size> function1
  - <address> <size> x
  - <address> <size> fahrenheit_degrees
  - …

Executable File Format

- Debugging information
  - DWARF (http://dwarfstd.org): to keep fantasy from ELF
    - Debugging With Attributed Record Formats
  - Generated by the –g option to gcc/g++
  - Multiple subsections
    - Source code line numbers
    - Macro information
    - Call frame information
    - Data types
    - Enumerations
    - Strings
    - Structures, unions, classes
    - Functions
    - ...

From High-Level code to program execution

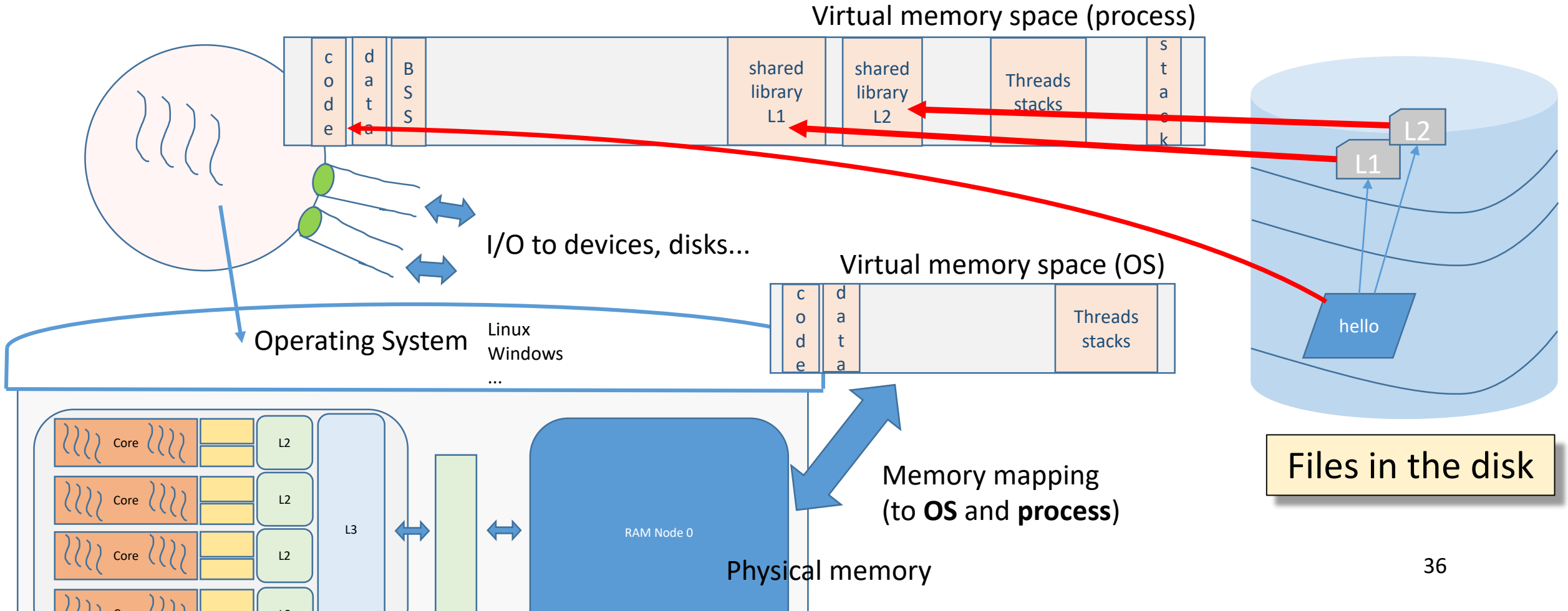
- **Loader:** a system program that reads a program from a storage device and place it in main memory (e.g. RAM) so that it is ready to execute



- The details about how to start executing the program in memory will be studied in the next lesson (focused on the OS)

Sharing services among applications

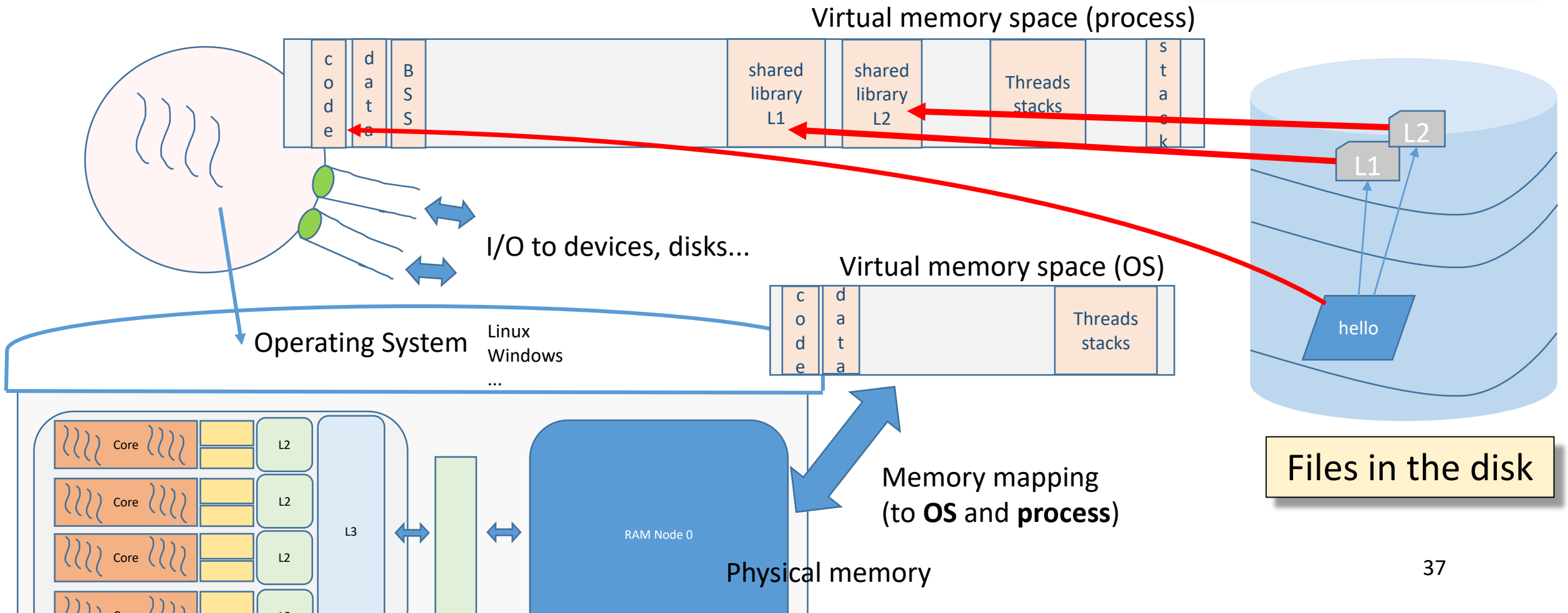
- Execution phase (with shared libraries)



Sharing services among applications

Virtual memory space is the location used when the program is loaded by the OS

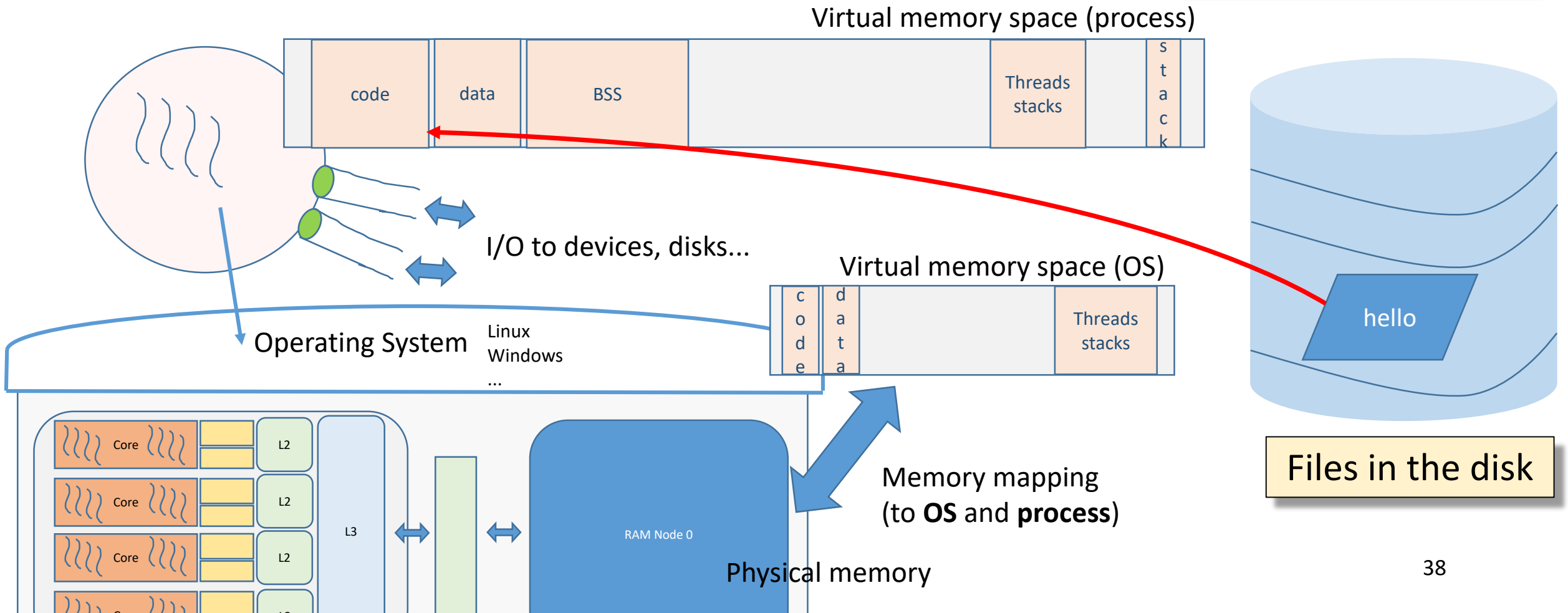
- Execution phase (with shared libraries)



Sharing services among applications

Virtual memory space is the location used when the program is loaded by the OS

- Execution phase (with static libraries)



Binary format utilities (GNU Linux)

- We will play with them in Lab Sessions
  - ld – linker... gets ELF **Object** files and generates ELF **Binary** files or **Shared Libraries**
  - ar – archive... gets ELF **Object** files and generates **Static Libraries**
  - nm – namelist... lists file symbols and some debugging information
  - size – reports various code/data/bss sizes
  - strings – shows the strings present in the ELF files
  - strip – deletes the debugging information from ELF files
  - addr2line – translates addresses to source code line numbers

Binary format utilities (GNU Linux)

- c++filt – demangle C++ names

```
$ c++filt   __Z9reductionRSt6vectorIfSaIfEE
reduction(std::vector<float, std::allocator<float> >&)
```

- objdump – full access to the ELF files contents
  - ELF header
  - File sections
  - Local and dynamic symbols
  - Relocations
  - List code instructions
  - Debugging information
  - Can support several different architectures and specific file formats

Objdump examples

- objdump -h vectors # list sections

```
vectors:     file format pei-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0008d620  0000000000401000  0000000000401000  00000600  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data         0000be10  000000000048f000  000000000048f000  0008de00  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  2 .rdata        000039e0  000000000049b000  000000000049b000  00099e00  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .bss          00000a00  000000000049f000  000000000049f000  00000000  2**5
                  ALLOC
  4 .idata        0000134c  00000000004a0000  00000000004a0000  0009d800  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .CRT          00000068  00000000004a2000  00000000004a2000  0009ec00  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  6 .tls          00000048  00000000004a3000  00000000004a3000  0009ee00  2**5
                  CONTENTS, ALLOC, LOAD, DATA
```

Objdump examples

- objdump -t vectors # local symbol table

```
SYMBOL TABLE:
[  0](sec -2)(fl 0x00)(ty   0)(scl 103) (nx 1) 0x0000000000000031 crtexe.c
File
[  2](sec  1)(fl 0x00)(ty  20)(scl   3) (nx 1) 0x0000000000000000
                                                ___mingw_invalidParameterHandler
AUX tagndx 0 ttlsiz 0x0 lnnos 0 next 0
[  4](sec  1)(fl 0x00)(ty  20)(scl   3) (nx 0) 0x0000000000000010 _pre_cpp_init
[  5](sec  4)(fl 0x00)(ty   0)(scl   3) (nx 0) 0x0000000000000020 _envp
[  6](sec  4)(fl 0x00)(ty   0)(scl   3) (nx 0) 0x0000000000000018 _argv
[  7](sec  4)(fl 0x00)(ty   0)(scl   3) (nx 0) 0x0000000000000014 _argc
[  8](sec  4)(fl 0x00)(ty   0)(scl   3) (nx 0) 0x0000000000000030 _startinfo
[  9](sec  4)(fl 0x00)(ty   0)(scl   3) (nx 0) 0x0000000000000028 _argret
[ 10](sec  1)(fl 0x00)(ty  20)(scl   3) (nx 0) 0x0000000000000060 _pre_c_init
[ 11](sec  4)(fl 0x00)(ty   0)(scl   3) (nx 0) 0x000000000000002c _managedapp
[ 12](sec  1)(fl 0x00)(ty  20)(scl   3) (nx 0) 0x00000000000001b0 ___tmainCRTStartup
...
```

Objdump examples

- objdump -d vectors # disassemble instructions

```
0000000000401730 <__Z11reduction_iRSt6vectorIfSaIfEE>:
  401730:       48 83 ec 08             sub    $0x8,%rsp
  401734:       0f 57 c0                xorps  %xmm0,%xmm0
  401737:       48 8b 11                mov    (%rcx),%rdx
  40173a:       31 c0                   xor    %eax,%eax
  40173c:       0f 1f 40 00             nopl   0x0(%rax)
  401740:       f3 0f 58 04 02          addss  (%rdx,%rax,1),%xmm0
  401745:       48 83 c0 04             add    $0x4,%rax
  401749:       48 3d 40 9c 00 00       cmp    $0x9c40,%rax
  40174f:       75 ef                   jne    401740 <__Z11reduction_iRSt6v...+0x10>
  401751:       f3 0f 2c c0             cvttss2si %xmm0,%eax
  401755:       48 83 c4 08             add    $0x8,%rsp
  401759:       c3                      retq
  40175a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)

0000000000401760 <__Z9reductionRSt6vectorIfSaIfEE>:
  401760:       48 83 ec 08             sub    $0x8,%rsp
```

Objdump examples

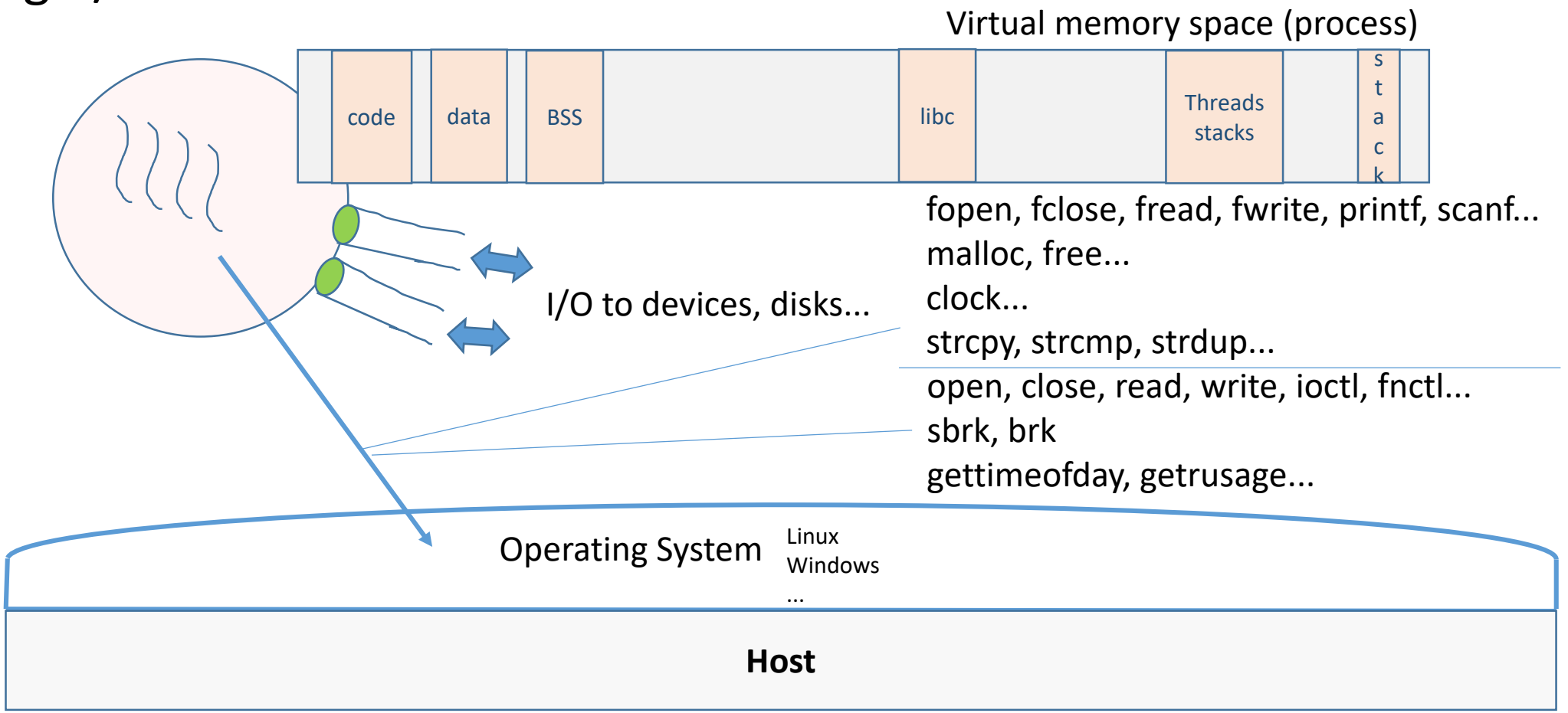
- objdump -WL vectors # debugging information: line numbers

```
File name                                Line number    Starting address
vectors.cpp                                      13           0x401730

vectors.cpp                                      13           0x401734
vectors.cpp                                      16           0x401740
vectors.cpp                                      18           0x401755
vectors.cpp                                      20           0x401760
vectors.cpp                                      52           0x401790

c:/xilinx/vivado/2016.2/tps/mingw/4.4.3/win64.o/nt64/bin/../lib/gcc/x86_64-w64-
mingw32/4.4.3/../../../../include/c++/4.4.3/iostream:
iostream                                         72           0x401790


./vectors.cpp:[++]
vectors.cpp                                      52           0x401797
...
```

Table of Contents

System libraries

- High / low levels



System libraries: C Library

- The basic library to Access Operating System services directly
- It follows the ISO C standard

  https://www.iso.org/obp/ui/#iso:std:iso-iec:9899:ed-3:v1:en
- Used by all applications
- Usually in **/lib/x86_64-linux-gnu/libc.so** (shared)
  /usr/lib/x86_64-linux-gnu/libc.a (static)

- Provides access to the low-level services offered by the Operating System
  - Process and memory management, files, directories, I/O…
- Provides higher level functions
  - Easier to use

Standard C++ library

- Dynamic memory management: new, delete
- Error handling, exceptions
- Strings, wide characters, Unicode
- Containers: array, vector, deque, list, stack, queue, set, map
- Iterators
- Common math functions
- Input/output: iostream
- Localization library

Standard C++ library

- Regular expressions
- Thread support, atomics, mutex, condition variables
- Filesystem (C++17)
- Experimental
- C compatibility headers

Sample functions from the C++ library

- Iostream
  - High-level data transfer and automatic conversion of datatypes to strings

```
std::cout << “Hello, world” << std::endl;

double degrees;
std::cout << “Temperature = “ << degrees << “C” << std::endl;
```

Hello, world\n
Temperature = 27.3C\n

  - Interfaces

```
template<typename _CharT, typename _Traits>
    class basic_ostream : ...
    {
     public:
       typedef basic_ostream<_CharT, _Traits>   __ostream_type;
       ...

       __ostream_type& operator<<(double __f);
       ...
```

Table of Contents

Interpreter vs Compiler

- Interpreter
  - It directly executes instructions from the source code (through different approaches)
  - Less time to analyze the source code, but slower execution
  - No object file is generated
  - Executes until the first error → easy debugging
  - E.g. Python, MATLAB
- Compiler
  - Scans the whole code and translates it into machine code
  - Long time to analyze the source code, but faster execution
  - Generates object file
  - Error messages after scanning the whole program
  - E.g. C, C++

Combining Interpreter & Compiler

- AOT: Ahead-of-Time
  - Compile the program before it is running: at compile time
- JIT: Just-in-Time
  - Compile the program while it is running: at runtime
- Steps to run a program

  Source code →AOT→ **Bytecode** (intermediate code) →JIT→ **Native code**
- Use cases
  - Java
    - Java code -> Compiled to -> Java Bytecode -> Interpreted by: Java Virtual Machine
  - Python
    - Python -> Compiled to -> CPython -> Interpreted by: Python Interpreter
  - Microsoft .NET
    - Any .NET language -> Compiled to -> CIL (Common Intermediate Language) -> Interpreted by: CLR (Common Language Runtime)

E.g. Standard Library for interpreted language

- Distributed with the Python execution environment
  - Strings
  - Numbers and math
  - File and directory access
  - File formats and data compression
  - Operating System services
  - Communications & protocols
  - Internationalization
  - …

https://docs.python.org/2/library/

https://docs.python.org/3/library/

Bibliography

- GCC Compiler Tutorial
  - http://gcc.gnu.org/
- Makefile Tutorial
  - http://www.gnu.org/software/make/manual/make.html

Next steps

- Support to the programming environment
  - Operating System
  - Programming Foundations