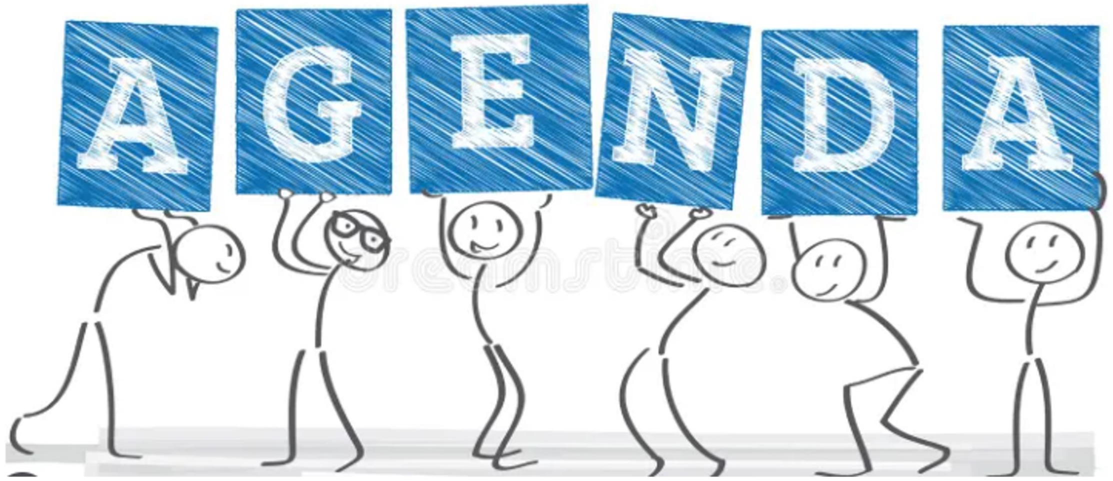
Operating Systems

Lecture 1.3

C-Compilation Toolchain: Static and Dynamic Libraries

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda



- C Compilation Process & its Tool Chain
- Static vs Dynamic Linking
- Multi-File C Compilation
- Loading and Executing a Program
- Creating & using your own Static Libraries
- Creating & using your own Dynamic Libraries
- PLT and GOT
- GDB with PEDA/GEF

System Programming Tools and Environment

The **set of programming tools** used to create a program is referred to as the **Toolchain**.

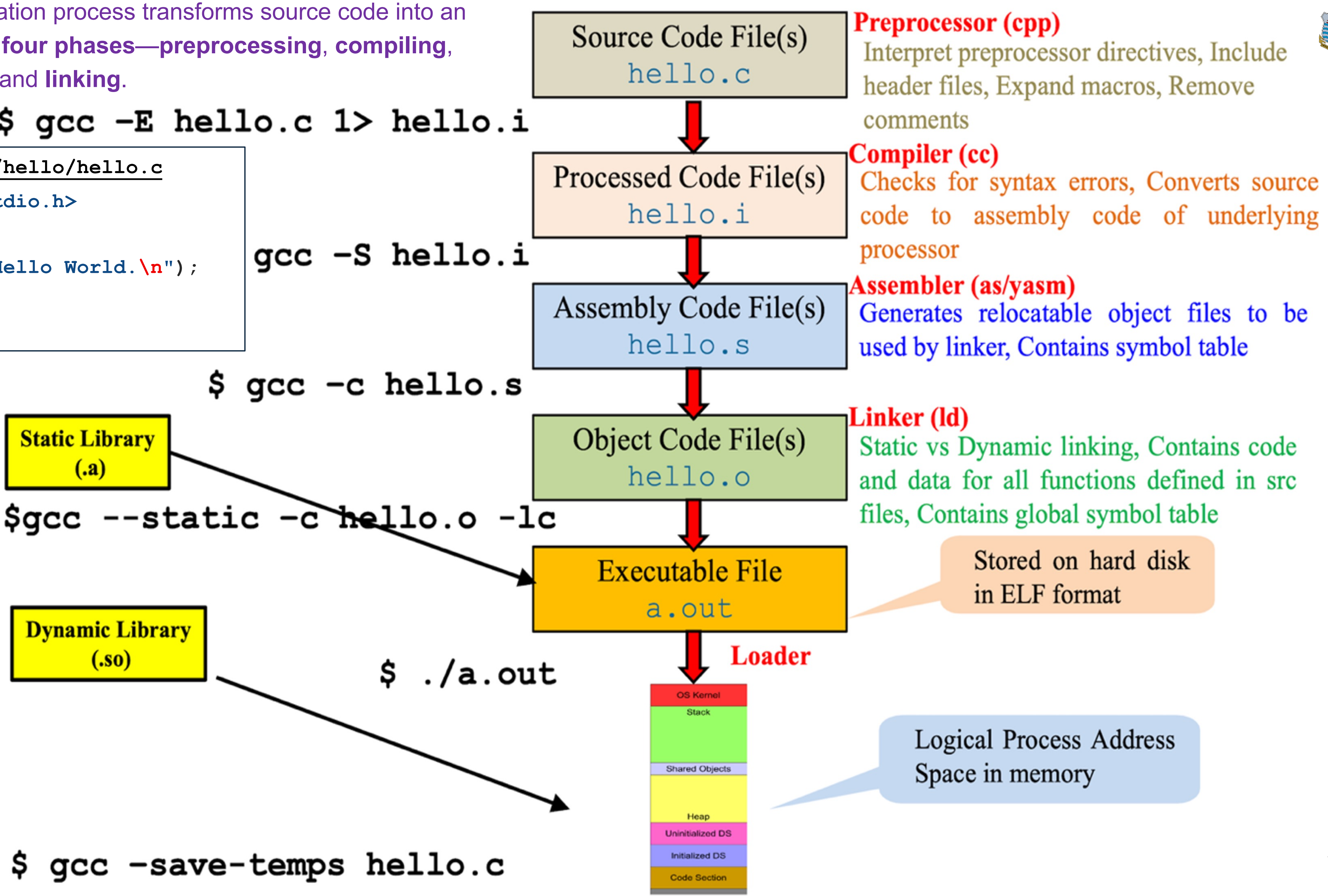
- **Processor:** Intel IA-32, Intel IA-64, AMD x86-64, Microprocessor without Interlocked Pipeline Stages (MIPS), Advanced RISC Machine (ARM), Sun Scalable Processor ARChitecture (Sun SPARC)
- **Operating System:** Windows, UNIX, Linux, MacOS
- **Editor/IDE:**
  - **Text Editors:** `gedit, vim, notepad`
  - **Code Editors:** `Atom, Sublime, Brackets,` **`Cursor, VS Code + GitHub Copilot`**
  - **IDEs:** `Visual Studio, Code::Blocks, PyCharm, Spider, Eclipse, Xcode,` **`Trae`**
- **Assembler:** `nasm, yasm, gas, masm`
- **Linker:** `ld` a GNU linker
- **Loader:** Default OS
- **Debugging/RE:** `readelf, objdump, nm, strings, file, hexedit, objcopy, strip, gdb (PEDA/GEF), valgrind, strace, ltrace, ftrace, bftrace, IDA Pro, ghidra, radare2, cutter, binaryninja`

C Compilation Process & its Tool Chain

Video Lecture: https://youtu.be/a7GhFL0Gh6Y?si=63ZfsCCN_9jiBrG3

The C compilation process transforms source code into an executable in **four phases—preprocessing**, **compiling**, **assembling**, and **linking**.

```
module1/1.2/hello/hello.c
#include <stdio.h>
int main() {
   printf("Hello World.\n");
   return 0;
}
```



```
$ gcc -E hello.c 1> hello.i
gcc -S hello.i
$ gcc -c hello.s
$gcc --static -c hello.o -lc
$ ./a.out
$ gcc -save-temps hello.c
```

Preprocessor (cpp)
Interpret preprocessor directives, Include header files, Expand macros, Remove comments

Compiler (cc)
Checks for syntax errors, Converts source code to assembly code of underlying processor

Assembler (as/yasm)
Generates relocatable object files to be used by linker, Contains symbol table

Linker (ld)
Static vs Dynamic linking, Contains code and data for all functions defined in src files, Contains global symbol table

Stored on hard disk in ELF format

Logical Process Address Space in memory

Demonstration

C Compilation Step-by-Step

`Lec1.3/hello/hello.c`

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Preprocessing

- The preprocessing step handles pre-processor directives, and remove comments.
- The pre-processor directives are not part of the C language but are used to perform operations before the actual compilation begins.
- The result of preprocessing is a preprocessed source file, which is usually saved with .i or .ii extension.
- The pre-processor directives perform following tasks:

  a. **File Inclusion**: `#include` directives are replaced with the contents of the specified files. This is typically used to include header files, whose default location is `/usr/include/`.

  b. **Macro Expansion**: `#define` macros are expanded to their defined values or code snippets.

  c. **Conditional Compilation**: `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` are used to include or exclude parts of the code based on certain conditions.

```
$ gcc -E  hello.c  -I ./   1>  hello.i
```

Compiling

- The compiler takes the preprocessed source code (`.i`) and translates it into assembly code for the underlying architecture and generate the output file(s) with `.s` or `.asm` extension.
- This step involves several key activities like **syntax analysis**, **semantic analysis**, **optimization**, and **code generation**.
- During compilation, we can mention the following (default settings depends on the `gcc` version):
    - **C-Standard to use:** The C programming language has several standard versions, that are aimed to unify various implementations of C and ensure code portability across different platforms. The most commonly used ones are `K&R C, c89/90, c95, c99, c11, c18, c23`.
    - **Optimization Level:** Use `-O0, -O1, -O2, -O3, -Ofast, -Os, -Og` flags, which instruct the compiler to optimize the generated machine code for performance, size, or a balance of both.
    - **Generate Code for Specific Architecture:** Use `-m32` or `-m64` flags, to instruct `gcc` to generate code for specific architecture, which affects the size of pointers, integer types, and function calling conventions used in the generated binary. By default, gcc generates 64-bit binaries on a 64-bit Linux system. If you want to generate 32-bit binary, you need to have the 32-bit libraries and development tools installed on your system. On Debian-based systems like Ubuntu, you can install the necessary packages by installing `gcc-multilib` package, and then need to mention the `-m32` option during compilation, assembling and linking phases.

```
$ gcc -S hello.i -std=c18 -Ofast -m64
```

Assembling

- The assembler converts the assembly code into machine code, which is an object file, typically with `.o` or `.obj` extension. Moreover, the assembler resolves symbolic names (e.g., variable names) to actual memory addresses and generates a symbol table. If you want to include debugging symbols, so as to load this file inside a debugger, use `-ggdb` option.
- In Linux, object files can be classified based on their formats and usage:
  - **Relocatable object file (*.o file*)** is a file generated by a compiler or assembler that contains machine code, data, and metadata, but is not yet a complete executable or library. Object files are intermediate files that are linked together to produce final executables or shared libraries. They are crucial in the software build process, allowing modular development and incremental compilation. Each .o file is produced from exactly one .c file.
  - **Executable object file (*a.out file*)** Contains binary code and data in a form that can be copied directly into memory and executed. Linkers generate executable object files.
  - **Shared object file (*.so file*)** A special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run time. Called dynamic link libraries (`.dll`) in Windows. Compilers and assemblers generate shared object files.
  - **Core file:** A disk file that contains the memory image of the process at the time of its termination. This is generated by system in case of abnormal process termination.

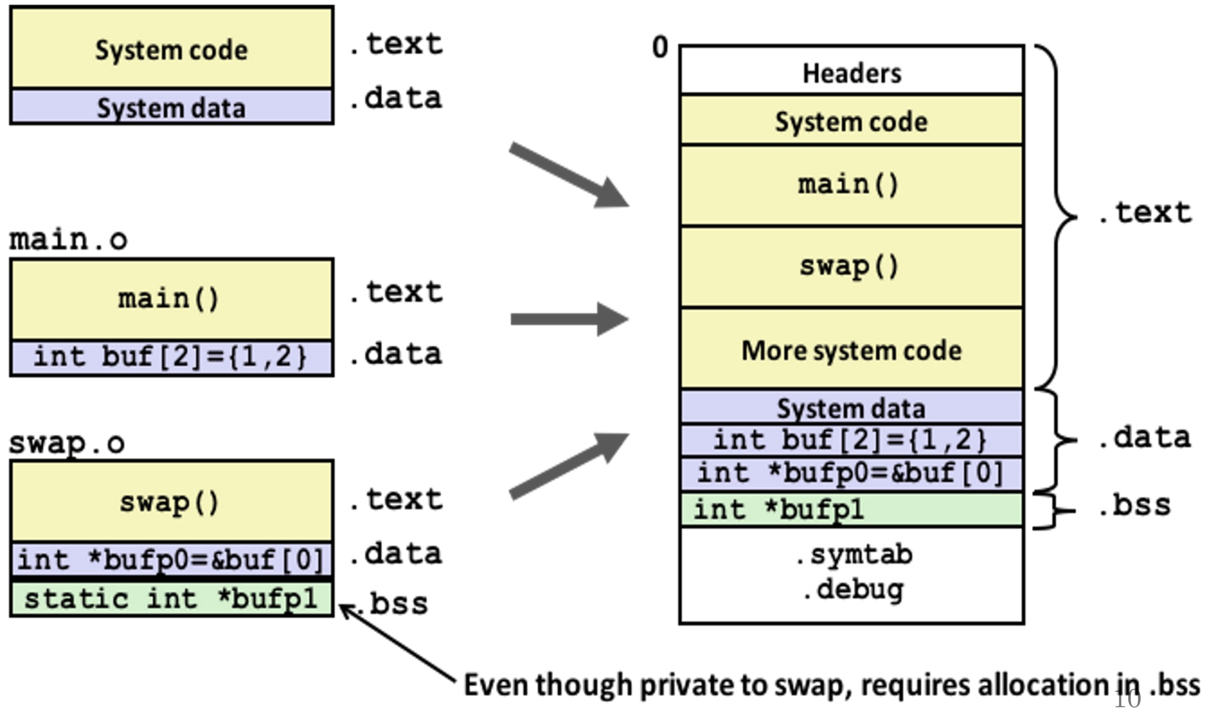
```
$ gcc -c -ggdb hello.s
```

Linking

- The linking phase combines object files and libraries into a single executable file with a specific format (e.g., ELF on Linux, PE on Windows). This format includes headers (containing information about how to load the executable into memory) and sections for code, data, and metadata. The .text sections from multiple object files are combined into a single .text section in the executable. Similarly, the .data and .bss sections from multiple object files are merged, with .data holding initialized data and .bss holding uninitialized data. Moreover, since every .o file has its own symbol table, so if the same symbol (e.g., a function or variable) is defined in multiple object files, the linker must resolve which definition to use.

- For x86_64 architecture running Linux, the standard C library along with other libraries resides in /usr/lib/x86_64-linux-gnu/ directory. These libraries come into two flavors:
  - **Static Linking** (/usr/lib/x86_64-linux-gnu/libc.a)
  - **Dynamic Linking** (/usr/lib/x86_64-linux-gnu/libc.so)

```
$ gcc hello.o -o dynamic-exe -lc
$ gcc --static hello.o -o static-exe -lc
```



Static Linking vs Dynamic Linking

Aspect	Static Linking	Dynamic Linking
Definition	Library code is copied into your executable file at compile time	Library code is loaded into memory at run time
Linking Time	Occurs during compilation/build process	Occurs at program startup (load time) and potentially during execution
File Structure	Single monolithic executable containing all dependencies	Executable with references to external shared libraries (.so/.dll files)
Executable Size	Significantly larger (includes complete library code)	Smaller (contains only library references and linking metadata)
Memory Usage (Single Process)	Uses less runtime RAM because only functions you actually need are loaded	Higher per-process memory usage as entire shared objects are loaded
Memory Usage (System-wide)	Higher, since each process contains duplicate library code	More efficient as system libraries are loaded into memory only once and shared by all processes
Startup Performance	Faster execution because we don't have to run the query for unresolved symbols at runtime	Slightly slower due to dynamic symbol resolution and library loading overhead
Update & Maintenance	For updated library versions, you need to re-compile and re-link	For updated library versions, no need to re-compile, just re-link
Portability	Runs on any binary-compatible system	Shared library need to be present on target system
Version Management	No runtime version conflicts (libraries frozen at compile time)	Potential for version conflicts and dependency issues ("DLL Hell")

Loading and Executing a Program

Program Loading is a process of copying a program from disk to main memory in order to make it a process

- The `./` before the program name actually specifies that the loader should look for the program file in the current working directory. Otherwise, the loader will look for the program inside the directories mentioned inside the `PATH` variable, which is an environment variable that contains colon separated absolute path names of the directories where the shell look for the executables.
  - A process is created using `fork()` / `clone()` system call that create a nearly exact copy of the parent process.
  - The program binary (`a.out`) is loaded inside the child process usually using the `execve()` system call.
  - The binary is initialized, using constructors/functions that are there in every ELF, e.g., libc initialize memory regions for dynamic allocations when the program is initialized.
  - A normal ELF automatically calls `__libc_start_main()` in `libc`, which in turn calls the program's `main()` function and your code starts running.
  - The binary reads its input from the outside world, from command-line arguments and environment variables.
  - The binary code executes and does what the developer has coded it for.
  - The binary terminates by either receiving an unhandled signal or by calling the `exit()` system call. After termination, the process will remain in a zombie state until they are `wait()`ed on by their parent. When this happens, their exit code will be returned to the parent, and the process will be freed. If their parent dies without `wait()`ing on them, they are re-parented to the `init/systemd` process and will stay there until they're cleaned up.
- The program's return value (often called the exit status or exit code) indicates whether the program completed successfully or if an error occurred. This value is returned to the parent process (shell in our case) when the program finishes execution. By convention, a return value of 0 typically indicates that the program was executed successfully. Any non-zero return value usually indicates that an error occurred, specifying the type of error. In Linux Bash shell, you can check the return value of the last executed program inside the environment variable `$?`. In Linux and other Unix-like operating systems, the exit status of a process is represented as an 8-bit integer. This means that the exit status is effectively limited to values between 0 and 255. However, the convention is that exit statuses above 127 are reserved for special purposes related to process termination via signals. More on this later…. ☺

Making System Calls Directly & via wrappers

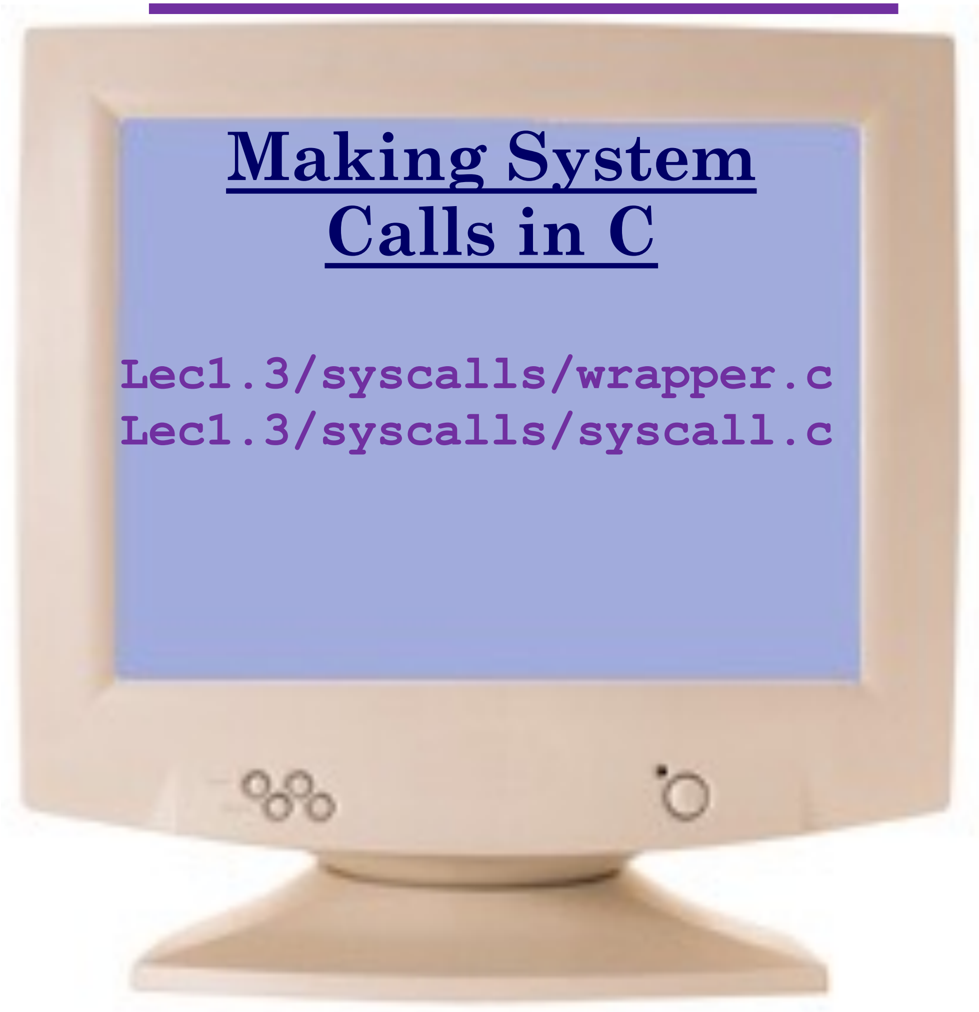
- *System calls* are the primary interface for user programs to request kernel services, which can be made:
  - **Using system call wrappers:**
    - ➢ The `write()` function is a C library wrapper in the `glibc` library.
    - ➢ It performs argument validation and type checking.
    - ➢ May do additional processing (buffering, signal handling).
    - ➢ Eventually makes the actual system call via assembly (`syscall` instruction).
    - ➢ Handles return values and sets `errno` appropriately.
  - **Using direct `syscall()` function:**
    - ➢ The `syscall()` system call is a direct interface to the kernel's system call mechanism.
    - ➢ Minimal overhead (almost direct kernel call)
    - ➢ No additional validation or processing by glibc
    - ➢ Raw Kernel interface
- Use system call wrapper for normal application development, cross-platform code and when you want library benefits.
- Use direct `syscall()` when, wrapper doesn't exist, writing performance critical code, for educational purposes.

```
//Lec-1.3/syscalls/wrapper.c
#include <unistd.h>
int main(){
   char str[] = "Welcome students!\n";
   int rv = write(1, str, sizeof(str)));
   return rv;
}                       int write(int fd, void *buf, int count);
```

```
//Lec-1.3/syscalls/syscall.c
#include <unistd.h>
int main(){
   char str[] = "Welcome students!\n";
   int rv = syscall(1, 1, str, sizeof(str));
   return rv;
}                                 long syscall(long ID,…);
```

`/usr/include/x86_64-linux-gnu/asm/unistd_64.h`

Demonstration

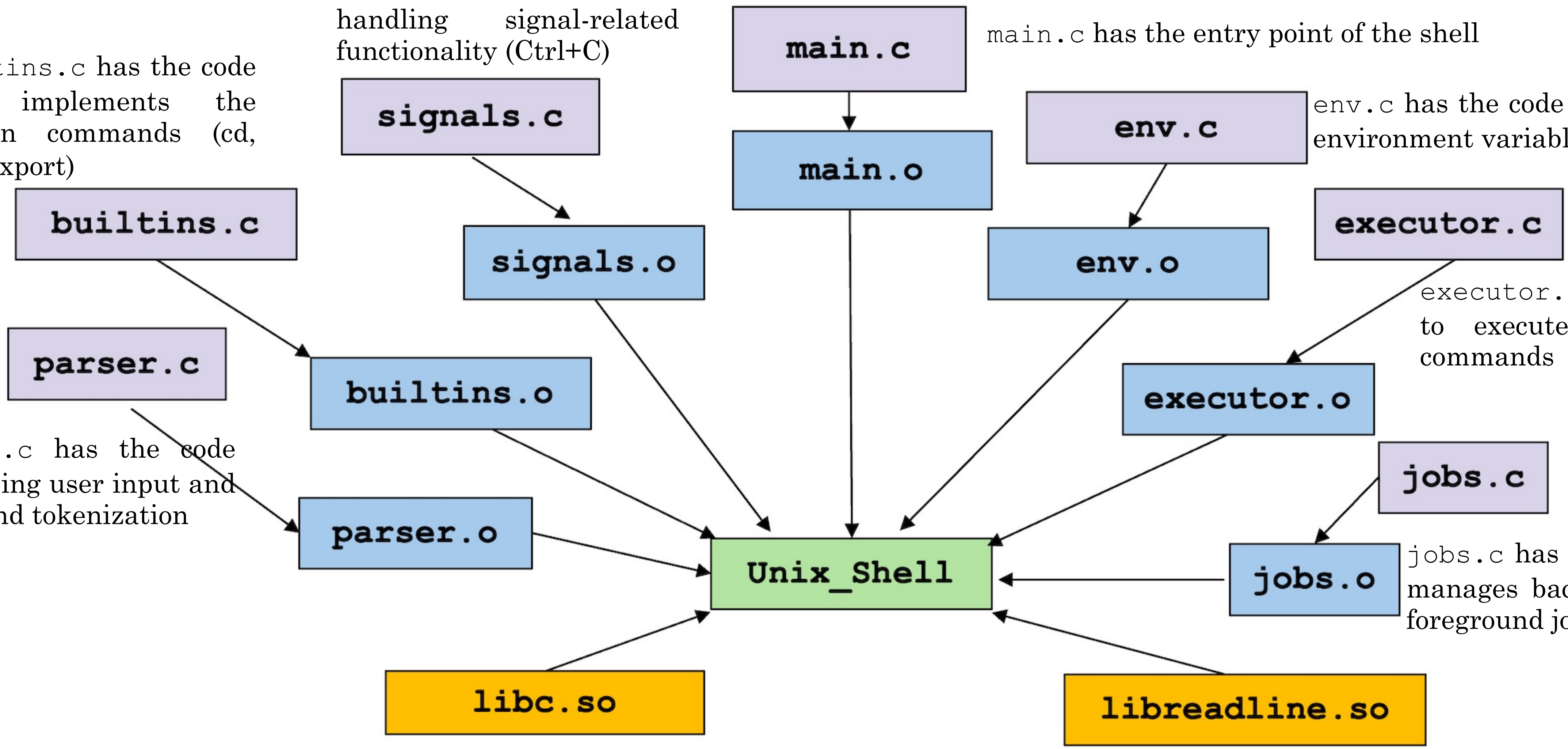


Making System Calls in C

```
Lec1.3/syscalls/wrapper.c
Lec1.3/syscalls/syscall.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Compiling Multi-File C Program



`signal.c` has the code for handling signal-related functionality (Ctrl+C)

`main.c` has the entry point of the shell

`builtins.c` has the code that implements the built-in commands (cd, exit, export)

`env.c` has the code for managing environment variables

`executor.c` has the code to execute the parsed commands

`parser.c` has the code for parsing user input and command tokenization

`jobs.c` has the code that manages background and foreground jobs

Demonstration

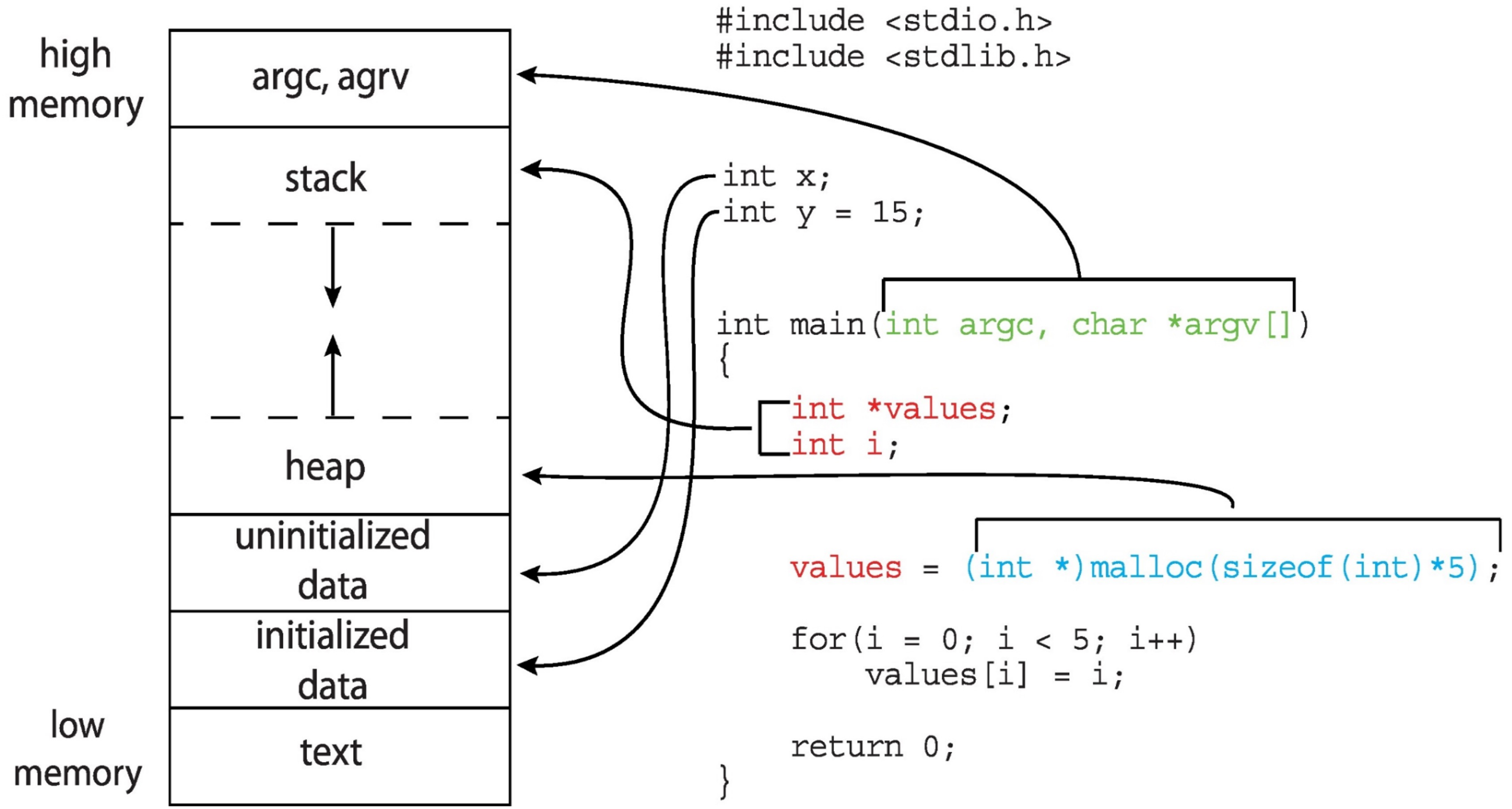
C Compilation Multi-Files

`Lec1.3/multifile/*.c`

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Program vs Process

Mapping of C source into Process address space

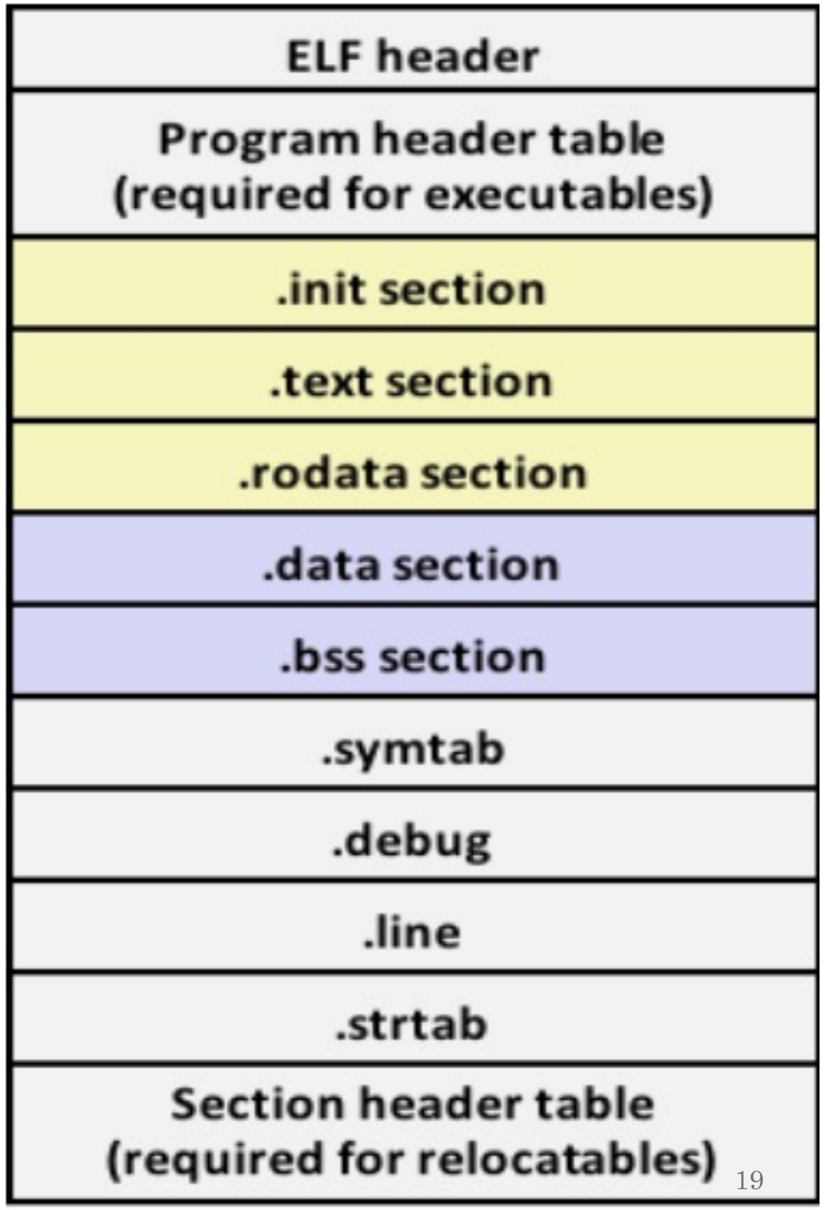


```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Instructor: Muhammad Arif Butt, PhD

A Program on Disk

- A program file contains all the necessary data to create a process when loaded into memory.
- Different operating systems use different formats for executable files, e.g., `a.out`, `COFF`, `PE`, and `ELF`.
- On most modern UNIX and Linux systems, the ELF (Executable and Linkable Format) is the standard format for executables, object code, shared libraries, and core dumps.
- An ELF file is divided into logical parts that serve different roles in building and running a program.
- The major components of an ELF file include:
  - **ELF Header:** Contains overall information about the binary, like file type, target architecture, and entry point address.
  - **Program Headers / Segments:** Breakdown the structure of an ELF binary into suitable chunks to prepare the executable to be loaded into memory. Needed at run time.
  - **Section Headers / Sections:** Comprise all the information needed for linking the object file into working executable describes sections like .text (code), .data (variables), .symtab (symbol table), and more.



Reading and Viewing Contents of Object Files

We have covered all four phases of the C Compilation process, and generated all the intermediate files (preprocessed file, assembly file, object file) and the final executable. We have already read the contents of preprocessed file with `.i` extension and the assembly file with `.s` extension, however, we haven't checked the contents of relocatable object files and final executable. You can't view the contents of these files using normal programs like `cat` and `less`. To deal with the object and executable files in Linux, you can use the following utilities:

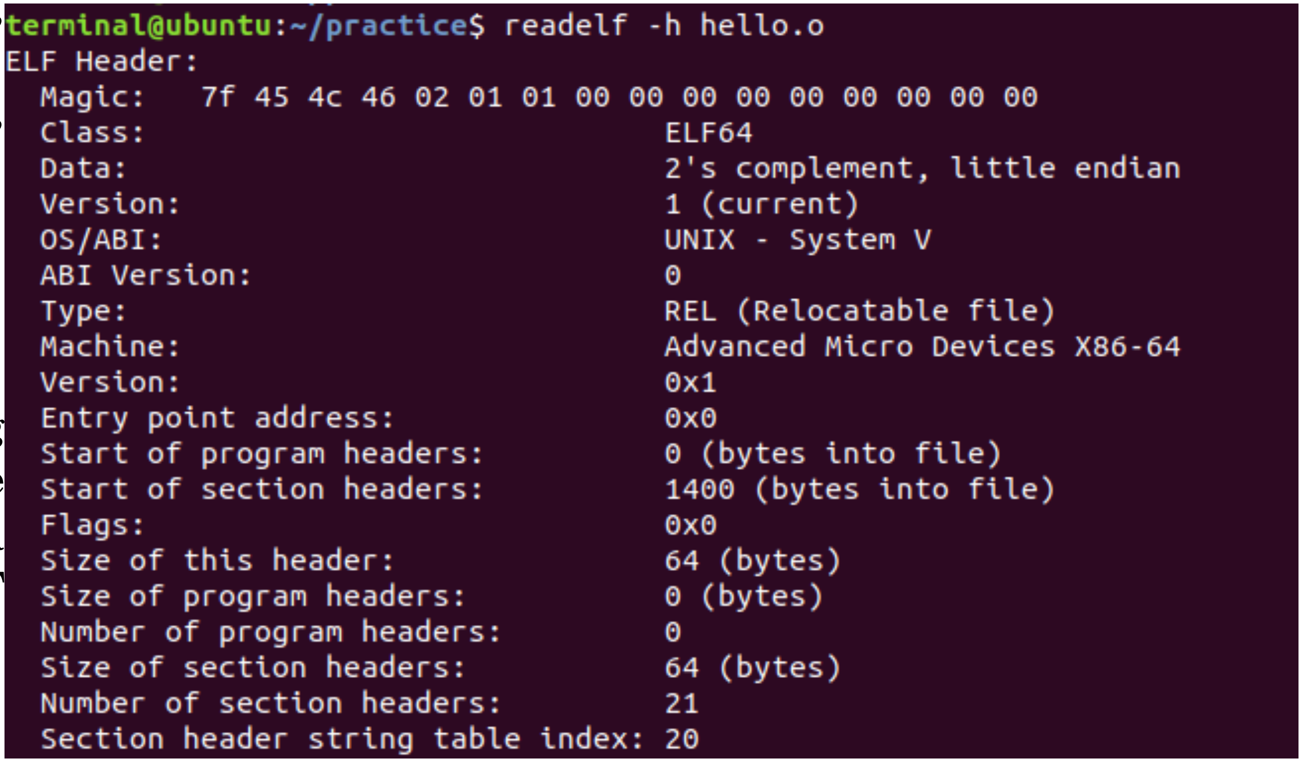
- **`readelf`** utility is used to display information about ELF files.
- **`objdump`** utility is used to disassemble and inspect object files, executables and libraries.
- **`nm`** utility is used to display the symbol table of object files, executables and libraries.
- **`strings`** utility is used to extract/display the ASCII/Unicode text embedded in binary files.
- **`file`** utility is used to determine the type of a file by inspecting the file's header.
- **`ldd`** utility is used to display the shared libraries with which the final executable is linked.
- **`strip`** utility is used to discard/remove symbols and debugging info from binaries.
- **`objcopy`** utility allows you to modify object files by copying them with alterations, such as stripping sections, changing formats, and extracting specific parts of the file.
- **`checksec`** utility is used to analyze security features of binaries and shows which exploit mitigation features are enabled/disabled in a binary (`NX`, `PIE`, `Canary`, `RELRO`, `FORTIFY`)

Inspecting Object Files using `readelf`

The **`readelf`** is a command-line utility used to display detailed information of Executable and Linkable Format (ELF) files on Linux and other Unix-like operating systems. ELF is the standard file format for executables, object code, shared libraries, and core dumps in Linux. You can view the man page of `readelf` to get more information.

```
$ readelf -[option] hello.o
```

- The **`-a`** option displays all available info about the elf file, including headers, sections, segments and symbols
- The **`-S`** option displays different section headers (.text, .plt, .got, .data, .rodata, .bss etc) of the ELF file.
- The **`-s`** option displays the symbol table (function/variable names with addresses)
- The **`-h`** option displays information about ELF header, that contains metadata about the ELF file, such as:
  - Magic Number starts with 0x7F 45 4C 46 specifying that it is an ELF file. The 02 after that specifies the class of binary (64-bit). The 01 after that specifies data encoding (little-endian). The last 01 specifies the ELF version.
  - File type (Executable, Shared Library, or Object File)
  - Target architecture (e.g., x86-64, ARM)
  - Entry point address (where execution starts)
  - Offset locations for different sections



```
terminal@ubuntu:~/practice$ readelf -h hello.o
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:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          1400 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         21
  Section header string table index: 20
```

Inspecting Object Files using `objdump`

The **`objdump`** is more general-purpose tool as compared to `readelf`, that is used for disassembling and inspecting binary files and works on multiple object formats like ELF, PE, COFF etc. It is particularly useful for debugging, analyzing binaries, and understanding how code is translated into machine instructions. By default, it displays the disassembly in AT&T format.

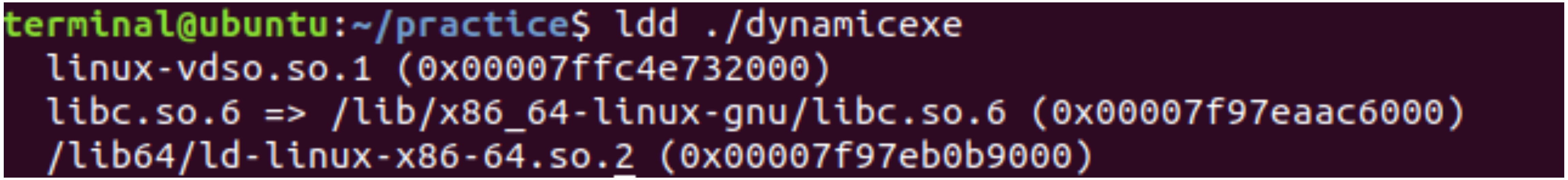
```
$ objdump -[option] -M intel hello.o
```

- The **`-f`** option displays the **file header** information (architecture, format, entry point, etc)
- The **`-h`** option displays information about the **section headers**, such as their sizes and offsets.
- The **`-t`** option displays the **symbol table** (function/variable names with addresses).
- The **`-d [-M intel]`** option **disassembles** only the executable sections (e.g., .text)
- The **`-D [-M intel]`** option **disassembles** all sections, including the non-executable ones.

Inspecting Dynamically Linked Files using `ldd`

The **`ldd`** is a command-line utility used on Linux and other Unix-like operating systems to display the shared library dependencies of an executable or shared library. It shows which dynamic libraries an executable or shared library relies on, helping you understand the runtime requirements and ensuring that all necessary libraries are available on the system. To show the shared libraries required by an executable or shared library:

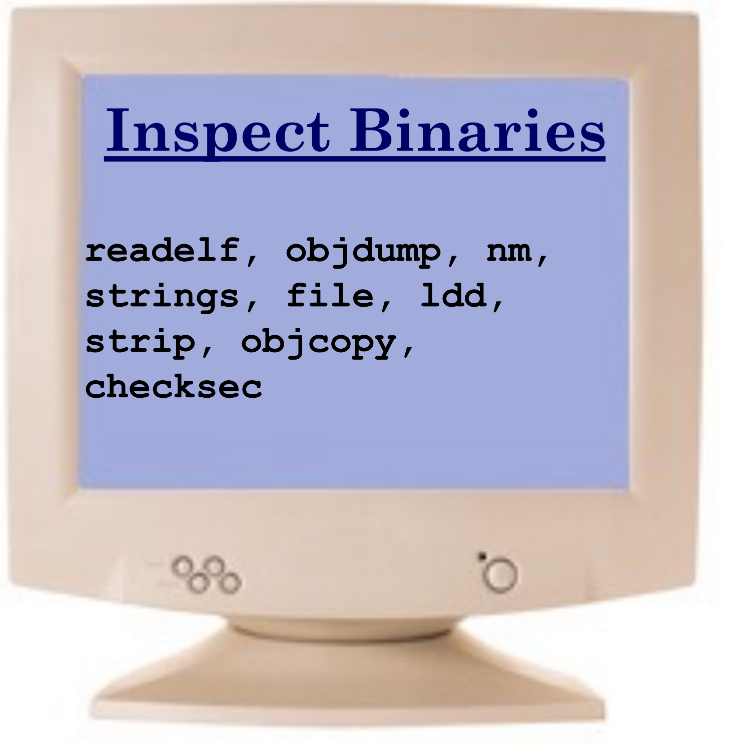
```
$ ldd ./dynamicexe
```



```
terminal@ubuntu:~/practice$ ldd ./dynamicexe
	linux-vdso.so.1 (0x00007ffc4e732000)
	libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f97eaac6000)
	/lib64/ld-linux-x86-64.so.2 (0x00007f97eb0b9000)
```

- `linux-vdso.so.1` is a virtual dynamic shared object.
- `libc.so.6` is the C standard library.
- `/lib64/ld-linux-x86-64.so.2` is the dynamic linker/loader.

Demonstration



Inspect Binaries

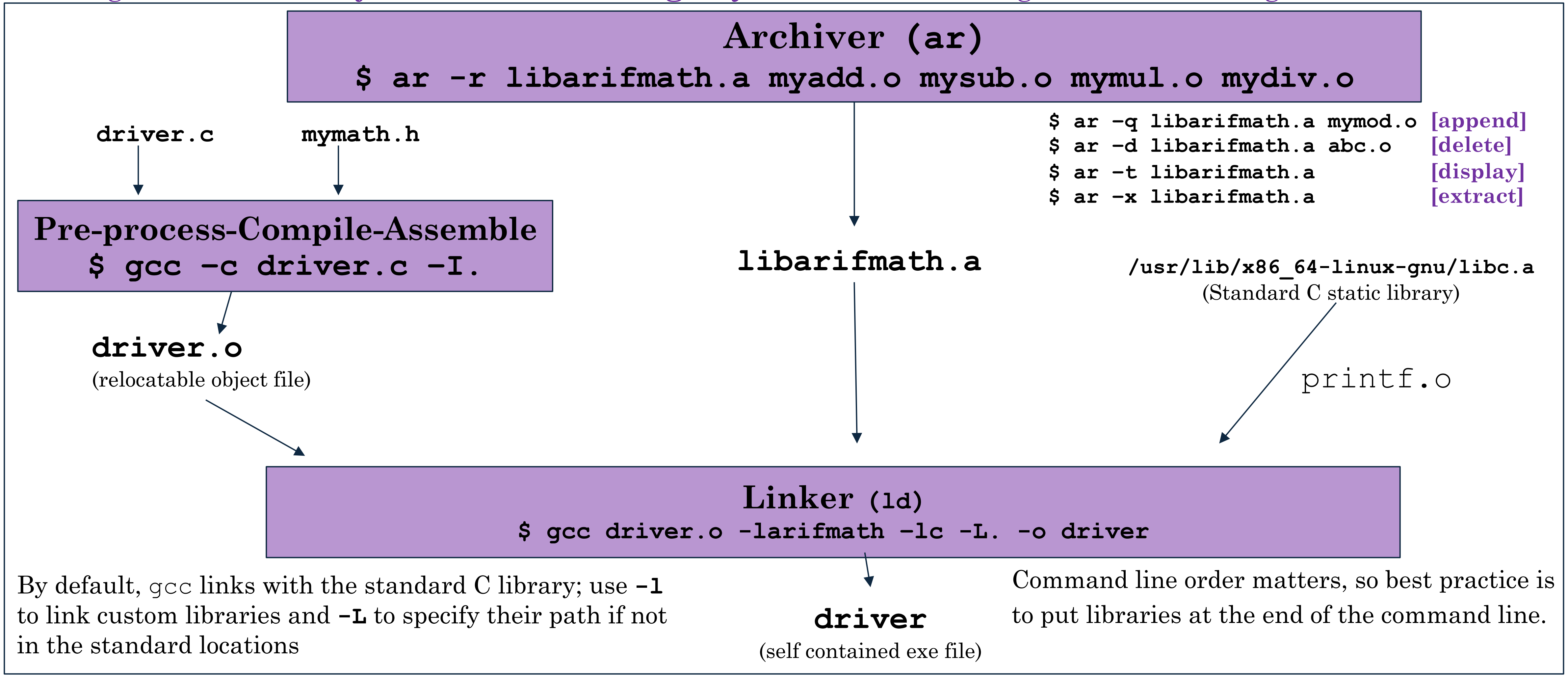
```
readelf, objdump, nm,
strings, file, ldd,
strip, objcopy,
checksec
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Creating Your Own Static Library

Creating Your Own Static Library

Creating a **static library** involves **combining object files** into a single **archive** using the **ar tool** in Linux.



Archiver (ar)

```
$ ar -r libarifmath.a myadd.o mysub.o mymul.o mydiv.o
```

```
$ ar -q libarifmath.a mymod.o   [append]
$ ar -d libarifmath.a abc.o     [delete]
$ ar -t libarifmath.a           [display]
$ ar -x libarifmath.a           [extract]
```

`driver.c` `mymath.h`

Pre-process-Compile-Assemble

```
$ gcc -c driver.c -I.
```

`driver.o`
(relocatable object file)

`libarifmath.a`

`/usr/lib/x86_64-linux-gnu/libc.a`
(Standard C static library)

`printf.o`

Linker (ld)

```
$ gcc driver.o -larifmath -lc -L. -o driver
```

`driver`
(self contained exe file)

By default, `gcc` links with the standard C library; use **`-l`** to link custom libraries and **`-L`** to specify their path if not in the standard locations

Command line order matters, so best practice is to put libraries at the end of the command line.

Demonstration

Static Library

`Lec1.3/staticlib/`

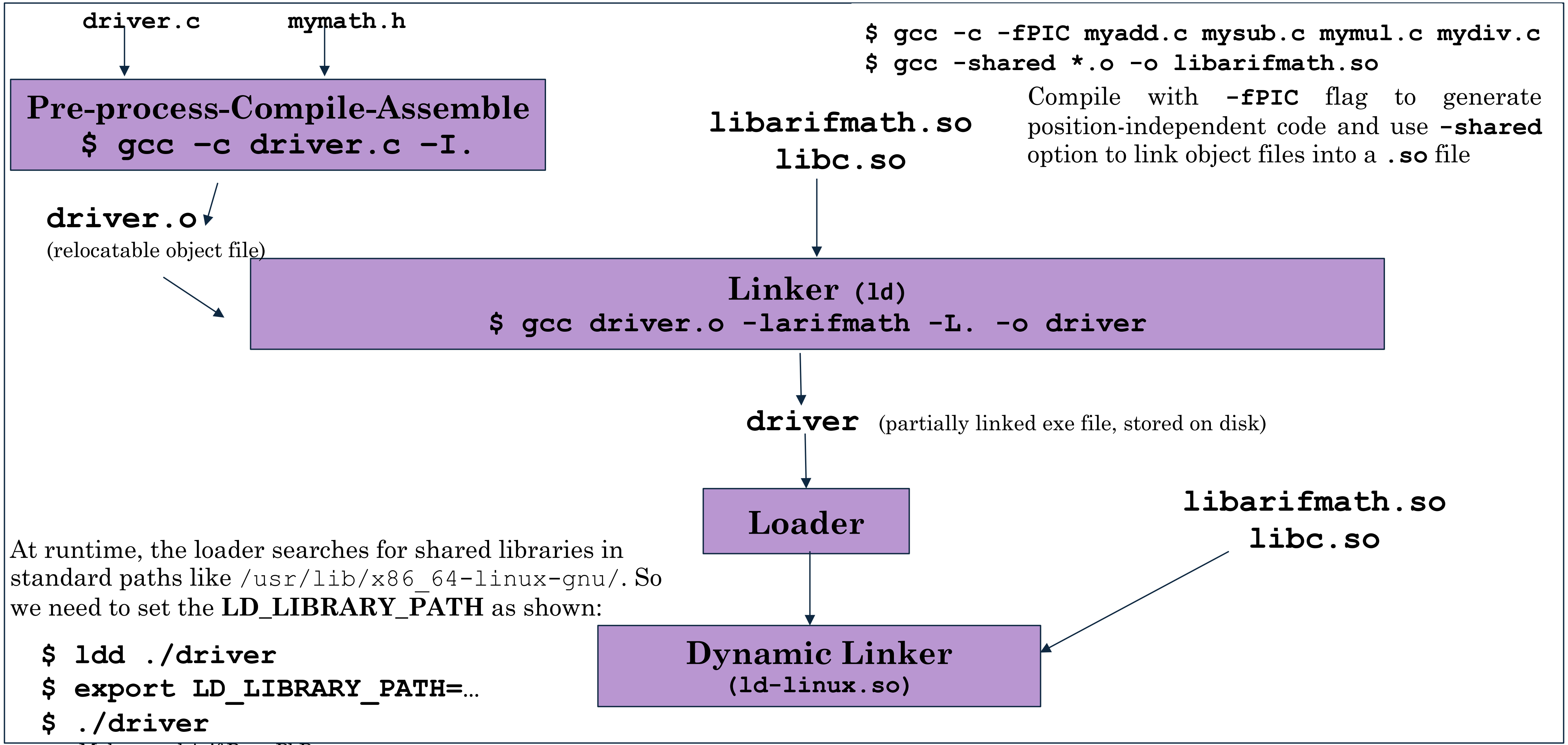
GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Video Lecture: https://youtu.be/A67t7X2LUsA?si=8Q_912JSy1bPC3li

Creating Your Own Dynamic Library

Creating Your Own Dynamic Library

Dynamic library is a **collection of object files**, similar to a static library, but **behaves differently** during **linking** and **loading**



```
$ gcc -c driver.c -I.
```

```
$ gcc -c -fPIC myadd.c mysub.c mymul.c mydiv.c
$ gcc -shared *.o -o libarifmath.so
```

Compile with `-fPIC` flag to generate position-independent code and use `-shared` option to link object files into a `.so` file

```
$ gcc driver.o -larifmath -L. -o driver
```

At runtime, the loader searches for shared libraries in standard paths like `/usr/lib/x86_64-linux-gnu/`. So we need to set the **LD_LIBRARY_PATH** as shown:

```
$ ldd ./driver
$ export LD_LIBRARY_PATH=...
$ ./driver
```

Demonstration

Dynamic Library

`Lec1.3/dynamiclib/`

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Video Lecture: https://youtu.be/A67t7X2LUsA?si=8Q_912JSy1bPC3li

PLT vs GOT

PLT vs GOT

When a C-program is dynamically linked with `libc.so`, for functions like `print`, `scanf` etc., it doesn't know the address of these functions in memory at compile time. So, in the final executable it places stubs for those functions and at load/run time, it uses two special tables to find and call these functions at runtime:

GOT (Global Offset Table): is a section inside a binary, that holds addresses (offsets) of functions that are dynamically linked. Initially, when a program gets loaded in the memory, the GOT doesn't know the addresses of these functions and it points to an entry in the PLT.

PLT (Procedure Linkage Table): contains stub (a small helper function) that look up the addresses in the `.got.plt` section. The first time you call a dynamically linked function, the PLT checks the GOT:

- If GOT has the address, jump there directly.
- If not, the PLT calls the dynamic linker (ld.so) to find the address of the function, update the GOT, and then jump to it.
- After the first call, future calls skip the lookup and go straight to the function (because GOT now has the right address).

GOT.PLT (A Special Part of GOT): This is the GOT for the PLT, containing the target addresses (after the functions have been looked up), or an address back in the PLT to trigger the lookup.

First call:

main() → PLT stub → dynamic linker → resolve symbol → update GOT → actual function

Subsequent calls:

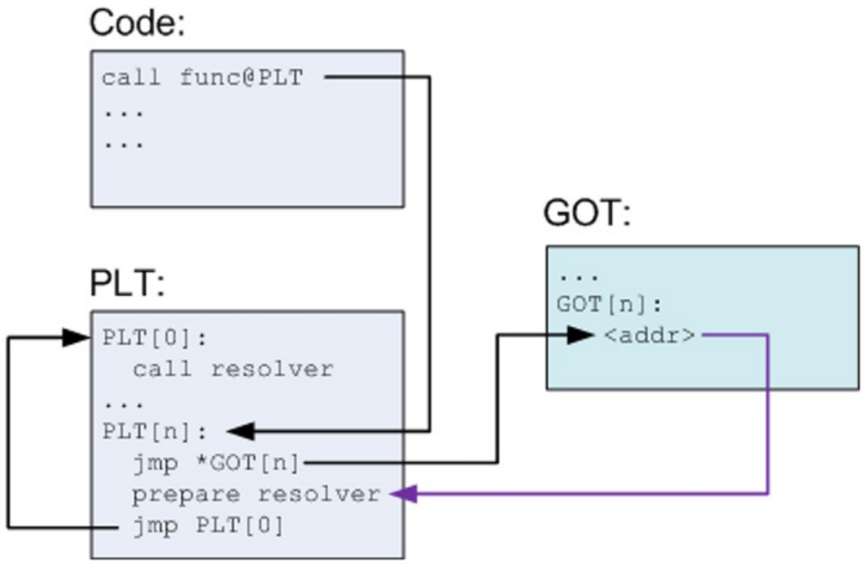
main() → PLT stub → GOT → actual function (direct jump)

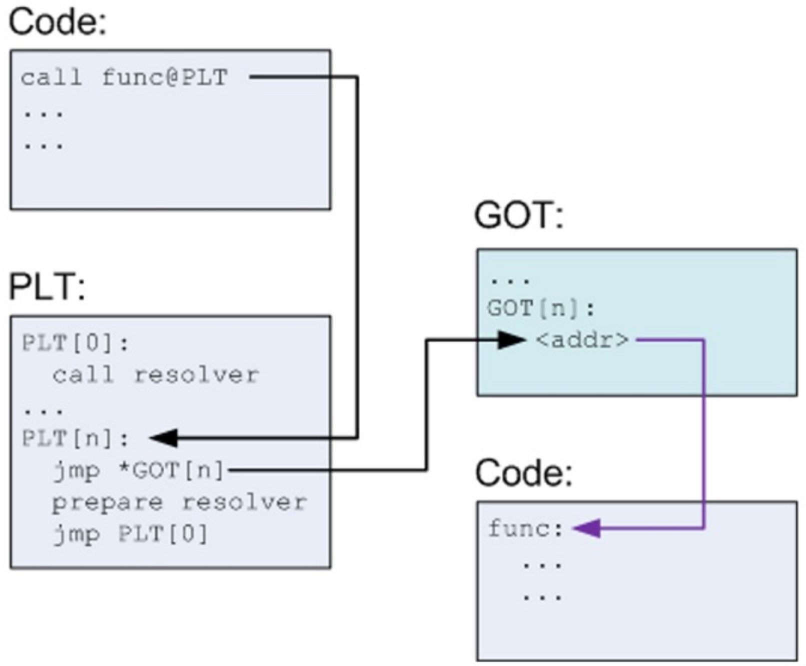
How a Dynamically Linked Function is Called?

a. The dynamically linked binary doesn't contain the actual address of `func()`, instead it contains a `call func@plt` instruction (which is a stub inside the `.plt` section).

b. This stub jumps through the GOT at runtime.

- For the first call, GOT points back to the PLT's resolver that invokes the dynamic linker (`ld.so`), which finds address of `func()` in `libc.so`, updates the GOT with the real address (shown in the left image), and then control flow of execution jumps to the actual code of `func()`.
- All future calls to `func@plt` will go to GOT, which now has the address of `func()`, and skipping the lookup process, the control flow of execution jumps to `func()` directly (shown in right image).





Understanding the working of PLT and GOT

- **Examine Sections Related to PLT/GOT:**

```
$ gcc lazy.c -o lazy
$ ldd lazy
$ file lazy
$ obdjump -h lazy | grep -E "(plt|got)"
$ obdjump -d -j .plt lazy
$ objdump -R lazy
$ nm -D lazy | grep puts
$ readelf -s lazy | grep puts
```

- **Static Analysis with GDB:**

```
$ gdb lazy
(gdb) info functions
(gdb) set disassembly-flavor intel
(gdb) disas main
```

```
//Lec-1.3/plt-got/lazy.c
#include <stdio.h>
int main(){
  puts("puts called 1st time\n");
  puts("puts called 2nd time.\n");
  return 0;
}
```

- **Dynamic Analysis with GDB:**

```
(gdb) break main
(gdb) break puts@plt
(gdb) run
(gdb) disas puts@plt
(gdb) x/a 0x<GOT_ADDRESS>
```

Step through the first puts call and see it will go through the PLT resolver mechanism. Check GOT after first call to puts

```
(gdb) next
(gdb) step
(gdb) continue
(gdb) x/a 0x[GOT_ADDRESS]
(gdb) continue
(gdb) stepi
```
[go directly through the GOT without resolver overhead]

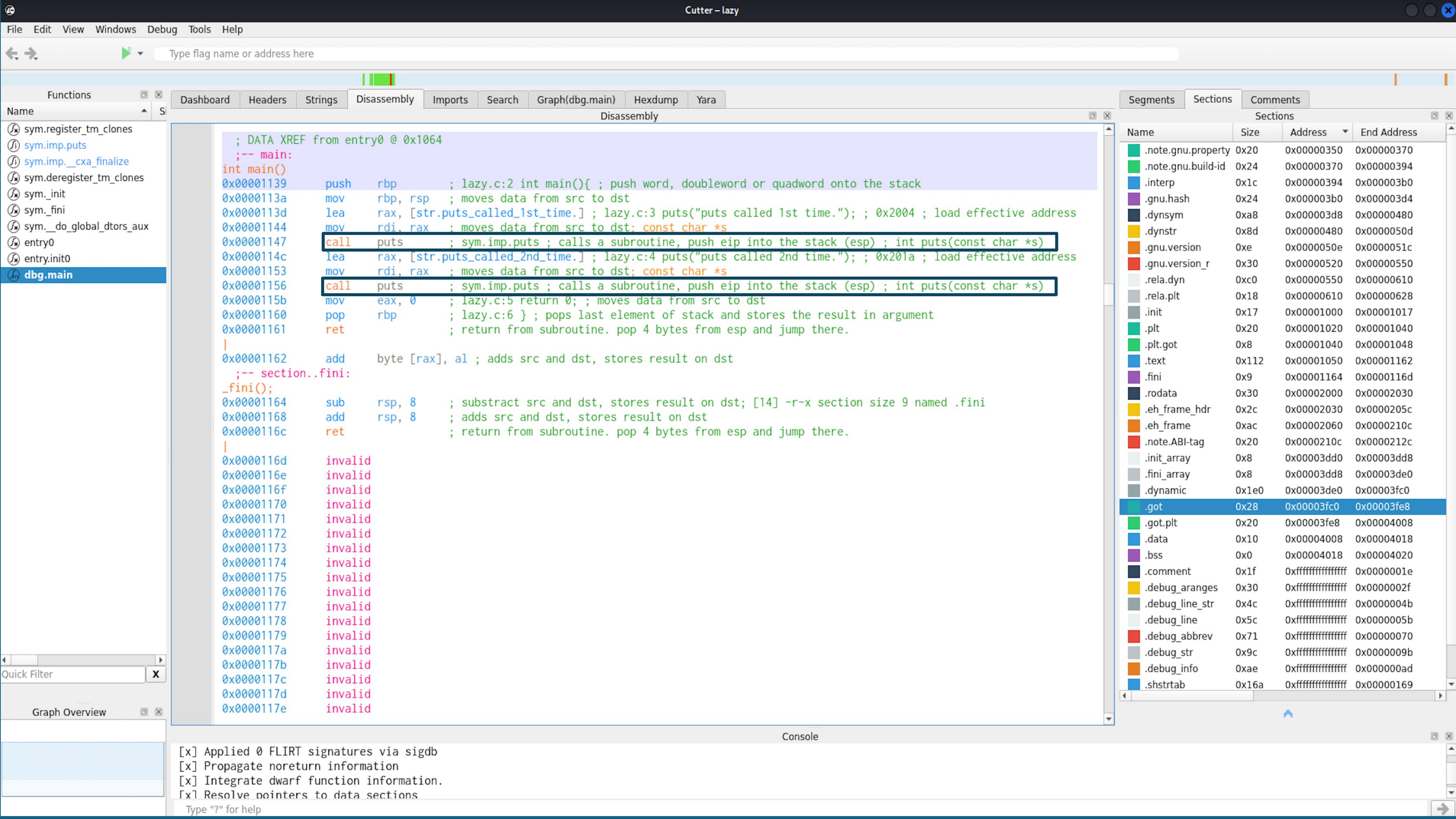
Understanding the working of PLT and GOT

- **Examine Sections Related to PLT/GOT:**

```
$ gcc lazy.c -o lazy
$ ldd lazy
$ file lazy
$ obdjump -h lazy | grep -E ”(plt|got)”
$ obdjump -d -j .plt lazy
$ objdump -R lazy
$ nm -D lazy | grep puts
$ readelf -s lazy | grep puts
```

```
//Lec-1.3/plt-got/lazy.c
#include <stdio.h>
int main(){
  puts(“puts called 1st time\n”);
  puts(“puts called 2nd time.\n”);
  return 0;
}
```

- **Download and Install Cutter:** (https://cutter.re/, https://github.com/rizinorg/cutter)
- Cutter is a free, open-source reverse engineering platform that provides a modern Qt-based graphical interface for the powerful Radare2 framework.
- It enables debugging and analysis of ELF and PE binaries across multiple architectures including x86, MIPS, and ARM processors.
- With Cutter’s intuitive visual interface, understanding the role of the PLT and GOT in dynamically linked C binaries becomes much easier compared to the manual, step-by-step inspection required in gdb.
- So install Cutter on your Linux Virtual machine, load and run the lazy binary to have a crystal clear understanding of the concept discussed on previous slides.


Cutter – lazy
File Edit View Windows Debug Tools Help
Type flag name or address here
Functions
Name
Dashboard Headers Strings Disassembly Imports Search Graph(dbg.main) Hexdump Yara
Disassembly
Segments Sections Comments
Sections
Graph Overview
Quick Filter
Console
Type "?" for help

```
sym.register_tm_clones
sym.imp.puts
sym.imp.__cxa_finalize
sym.deregister_tm_clones
sym._init
sym._fini
sym.__do_global_dtors_aux
entry0
entry.init0
dbg.main
```

```
  ; DATA XREF from entry0 @ 0x1064
  ;-- main:
int main()
0x00001139      push   rbp         ; lazy.c:2 int main(){ ; push word, doubleword or quadword onto the stack
0x0000113a      mov    rbp, rsp    ; moves data from src to dst
0x0000113d      lea    rax, [str.puts_called_1st_time.] ; lazy.c:3 puts("puts called 1st time."); ; 0x2004 ; load effective address
0x00001144      mov    rdi, rax    ; moves data from src to dst; const char *s
0x00001147      call   puts        ; sym.imp.puts ; calls a subroutine, push eip into the stack (esp) ; int puts(const char *s)
0x0000114c      lea    rax, [str.puts_called_2nd_time.] ; lazy.c:4 puts("puts called 2nd time."); ; 0x201a ; load effective address
0x00001153      mov    rdi, rax    ; moves data from src to dst; const char *s
0x00001156      call   puts        ; sym.imp.puts ; calls a subroutine, push eip into the stack (esp) ; int puts(const char *s)
0x0000115b      mov    eax, 0      ; lazy.c:5 return 0; ; moves data from src to dst
0x00001160      pop    rbp         ; lazy.c:6 } ; pops last element of stack and stores the result in argument
0x00001161      ret                ; return from subroutine. pop 4 bytes from esp and jump there.
|
0x00001162      add    byte [rax], al ; adds src and dst, stores result on dst
  ;-- section..fini:
_fini();
0x00001164      sub    rsp, 8      ; substract src and dst, stores result on dst; [14] -r-x section size 9 named .fini
0x00001168      add    rsp, 8      ; adds src and dst, stores result on dst
0x0000116c      ret                ; return from subroutine. pop 4 bytes from esp and jump there.
|
0x0000116d      invalid
0x0000116e      invalid
0x0000116f      invalid
0x00001170      invalid
0x00001171      invalid
0x00001172      invalid
0x00001173      invalid
0x00001174      invalid
0x00001175      invalid
0x00001176      invalid
0x00001177      invalid
0x00001178      invalid
0x00001179      invalid
0x0000117a      invalid
0x0000117b      invalid
0x0000117c      invalid
0x0000117d      invalid
0x0000117e      invalid
```

Name	Size	Address	End Address
.note.gnu.property	0x20	0x00000350	0x00000370
.note.gnu.build-id	0x24	0x00000370	0x00000394
.interp	0x1c	0x00000394	0x000003b0
.gnu.hash	0x24	0x000003b0	0x000003d4
.dynsym	0xa8	0x000003d8	0x00000480
.dynstr	0x8d	0x00000480	0x0000050d
.gnu.version	0xe	0x0000050e	0x0000051c
.gnu.version_r	0x30	0x00000520	0x00000550
.rela.dyn	0xc0	0x00000550	0x00000610
.rela.plt	0x18	0x00000610	0x00000628
.init	0x17	0x00001000	0x00001017
.plt	0x20	0x00001020	0x00001040
.plt.got	0x8	0x00001040	0x00001048
.text	0x112	0x00001050	0x00001162
.fini	0x9	0x00001164	0x0000116d
.rodata	0x30	0x00002000	0x00002030
.eh_frame_hdr	0x2c	0x00002030	0x0000205c
.eh_frame	0xac	0x00002060	0x0000210c
.note.ABI-tag	0x20	0x0000210c	0x0000212c
.init_array	0x8	0x00003dd0	0x00003dd8
.fini_array	0x8	0x00003dd8	0x00003de0
.dynamic	0x1e0	0x00003de0	0x00003fc0
.got	0x28	0x00003fc0	0x00003fe8
.got.plt	0x20	0x00003fe8	0x00004008
.data	0x10	0x00004008	0x00004018
.bss	0x0	0x00004018	0x00004020
.comment	0x1f	0xffffffffffffffff	0x0000001e
.debug_aranges	0x30	0xffffffffffffffff	0x0000002f
.debug_line_str	0x4c	0xffffffffffffffff	0x0000004b
.debug_line	0x5c	0xffffffffffffffff	0x0000005b
.debug_abbrev	0x71	0xffffffffffffffff	0x00000070
.debug_str	0x9c	0xffffffffffffffff	0x0000009b
.debug_info	0xae	0xffffffffffffffff	0x000000ad
.shstrtab	0x16a	0xffffffffffffffff	0x00000169

```
[x] Applied 0 FLIRT signatures via sigdb
[x] Propagate noreturn information
[x] Integrate dwarf function information.
[x] Resolve pointers to data sections
```

Demonstration

Dynamic Library

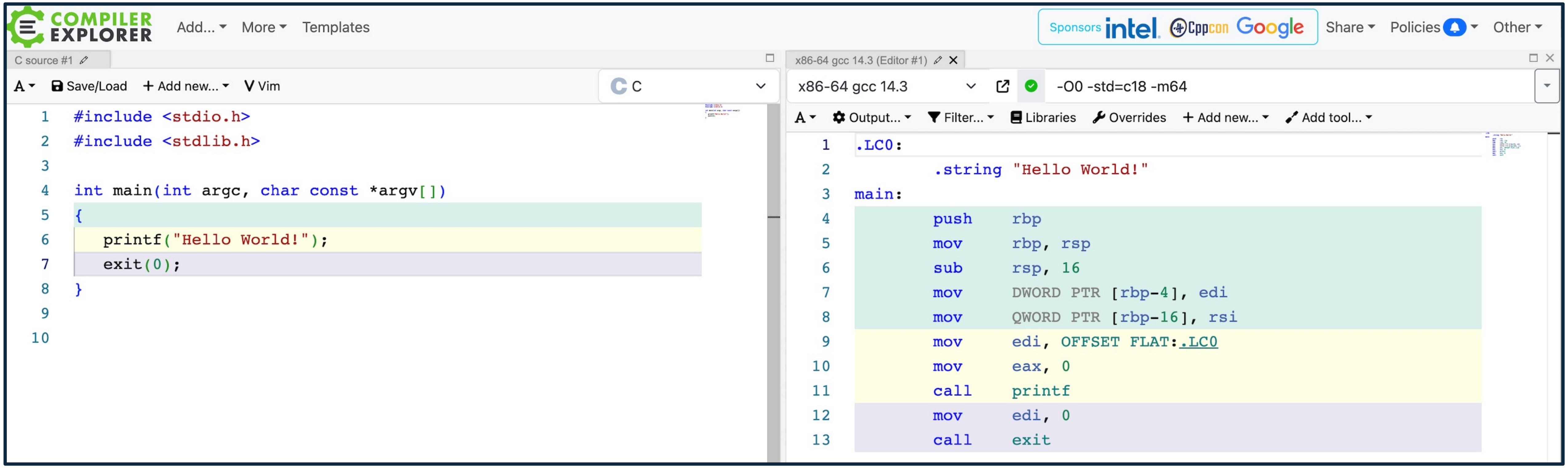
```
Lec1.3/plt-got/lazy.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Compiler Explorer

Compiler Explorer (https://godbolt.org/)

- An opensource, interactive, real-time compilation tool created by Matt Godbolt, that shows assembly output for C, C++, Rust, Go, Python, and 70+ other programming languages.
- Invaluable for understanding compiler optimizations, performance implications, and debugging systems-level code.



Compiler options: -O0 -std=c18 -m64

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char const *argv[])
{
   printf("Hello World!");
   exit(0);
}
```

```
.LC0:
        .string "Hello World!"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     QWORD PTR [rbp-16], rsi
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     edi, 0
        call    exit
```

Running C Binary with gdb (GEF)

GDB with GEF Plugin (cont…)

- Compile the Program: `$ gcc -g -O0 debug.c -o debugme`
- Launch GDB with Binary: `$ gdb ./debugme`
- Configure the panels: `gef> gef config context.layout “regs stack code source”`
- Show program info: `gef> info sources/functions/variables/args/registers/all/break`
- Setting breakpoint: `gef> break main`
- Run: `gef> run [arg1 arg2 . . . ]`
- Set disassembly flavor: `gef> set disassembly-flavor intel`
- Show disassembly: `gef> disassemble <function-name>`
- Step through code: `gef> continue/next/step/finish`
- Print memory content: `gef> print /format-char <var-name>`
- Examine memory content: `gef> x/12cb <addr>`
- Change register/variable values: `gef> set variable var1=<value>  |  set $rdi=<value>`
- GEF Specific Commands:
  - `gef> process-status` [show process information]
  - `gef> vmmap` [show memory mappings with colors]
  - `gef> dereference $rsp 10` [show stack with colors]
  - `gef> process-status`
  - `gef> process-status`

Demonstration

Running C Binary inside gdb (GEF)

Lec1.3/gdb-gef/debugme.c

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

To Do

- Install the required tool chain discussed in today's session on your Linux system.

- Watch OS video on understanding C Compilation process: https://youtu.be/2bYGoOTXrUg?si=oblFZJLEMJ2pDVdJ

- Watch SP C-Compilation video: https://www.youtube.com/watch?v=a7GhFL0Gh6Y&list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW&index=2&t=48s

- Watch SP Libraries video: https://www.youtube.com/watch?v=A67t7X2LUsA&list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW&index=3

- Watch GDB video for C: https://www.youtube.com/watch?v=2x-pkzSmsD8&list=PL7B2bn3G_wfCC2HDSXtMFsskasZ5fdLXz&index=32&t=467s

Coming to office hours does NOT mean that you are academically weak!