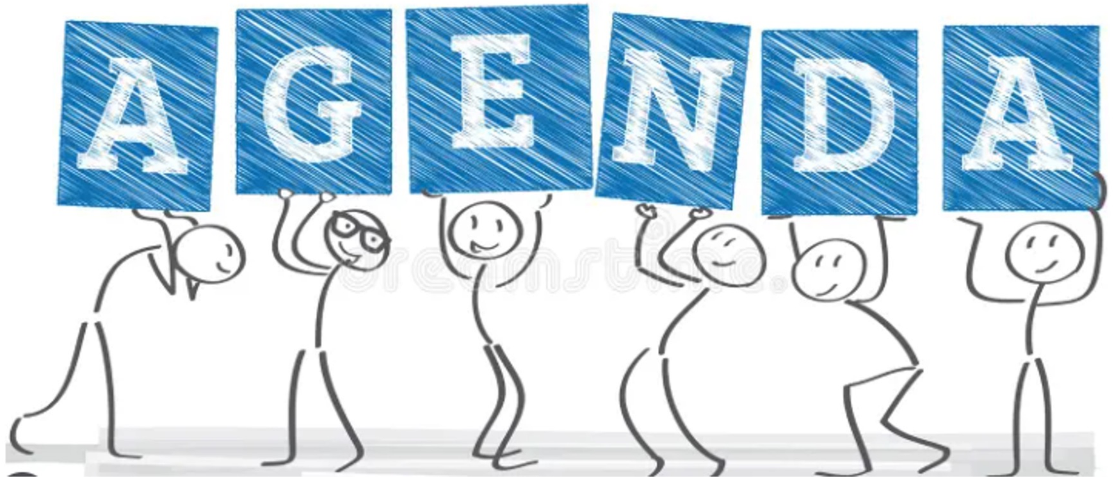
Operating Systems

Lecture 1.2

Recap of **`x86-64`** Assembly

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda



- System Programming Tool Chain
- AMD `x86-64` Processor Architecture
- `x86-64` Assembly Instructions
- Structure of `x86-64` Assembly Program
- What is a System Call?
- Why are System Calls Necessary?
- How to make a System Call in Assembly?
- What is a Library Call?
- Why use Library Calls?
- How to make a Library Call in Assembly?
- GNU Project Debugger

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`

Basic Necessary Installations

- **Update Package List:**

```
$ sudo apt update
```

- **Common Shell utilities (**`readelf, objdump, nm, strings, objcopy, strip`**):**

```
$ sudo apt install binutils file hexedit
```

- **Core compilation tools + debugging + docs:**

```
$ sudo apt install build-essential gdb git manpages-dev
```

- **Autotools, CMake, pkg-config, assembler:**

```
$ sudo apt install nasm autoconf automake libtool cmake pkg-config
```

- **Common Development Libraries:**

```
$ sudo apt install libssl-dev zlib1g-dev libncurses5-dev libncursesw5-dev
```

Checkout the versions:

```
$ uname -a         [Linux kali 6.12.20-amd64]
$ gcc --version    [14.2]
$ nasm --version   [2.16]
$ gdb --version    [16.3]
$ git --version    [2.47.2]
```

Checkout the versions:

```
$ ldd --version        [GLIBC 2.41]
$ make --version       [4.4.1]
$ cmake --version      [3.31.6]
$ autoconf --version   [2.72]
$ openssl --version    [3.5.1]
```

x86-64 Assembly Programming

AMD x86-64 Processor Architecture

General Purpose Registers

64-bit register	Lowest 32-bits	Lowest 16-bits	Lowest 8-bits
r0/rax	eax	ax	al
r1/rbx	ebx	bx	bl
r2/rcx	ecx	cx	cl
r3/rdx	edx	dx	dl
r4/rsi	esi	si	sil
r5/rdi	edi	di	dil
r6/rbp	ebp	bp	bpl
r7/rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r8d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

SSE Media Registers

511 255 127 0

zmm0	ymm0	xmm0
zmm1	ymm1	xmm1
zmm2	ymm2	xmm2
zmm3	ymm3	xmm3
zmm14	ymm14	xmm14
zmm15	ymm15	xmm15

Segment Registers

15 0

CS
DS
SS
ES
FS
GS

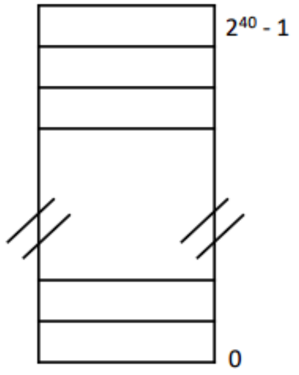
Misc Registers

Control Registers
Memory Management Registers
Debug Registers
Machine Specific Registers

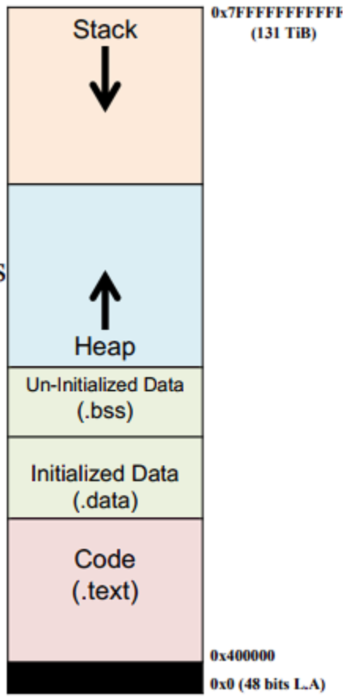
	63	31 0
RIP		EIP

	63		21	20	19	18	17	16		14	13	12	11	10	9	8	7	6		4		2		0
RFLAGS	-		ID	VIP	VIF	AC	VM	RF	-	NT	IOP1	IOP0	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF

Memory



Process Image



Instructor: Muhammad Arif Butt, PhD

Categories of x86-64 Assembly Instructions

Category	Description	Examples
Data Transfer	Move from source to destination	`mov, movzx, movsx, lea, lds, lss, xchg, push, pop, pusha, popa, pushf, popf`
Arithmetic	Arithmetic on integer	`add, addc, sub, subb, mul, imul, div, idiv, neg, inc, dec, cmp`
Bit Manipulation	Logical & bit shifting operations	`and, or, not, xor, test, shl/sal, shr, sar, ror, rol, rcr, rcl`

Control Transfer	Conditional and unconditional jumps, and procedure calls	`jmp`
		`jcc(jz,jnz,jg,jge,jl,jle,jc,jnc,...)`
		`call, ret`

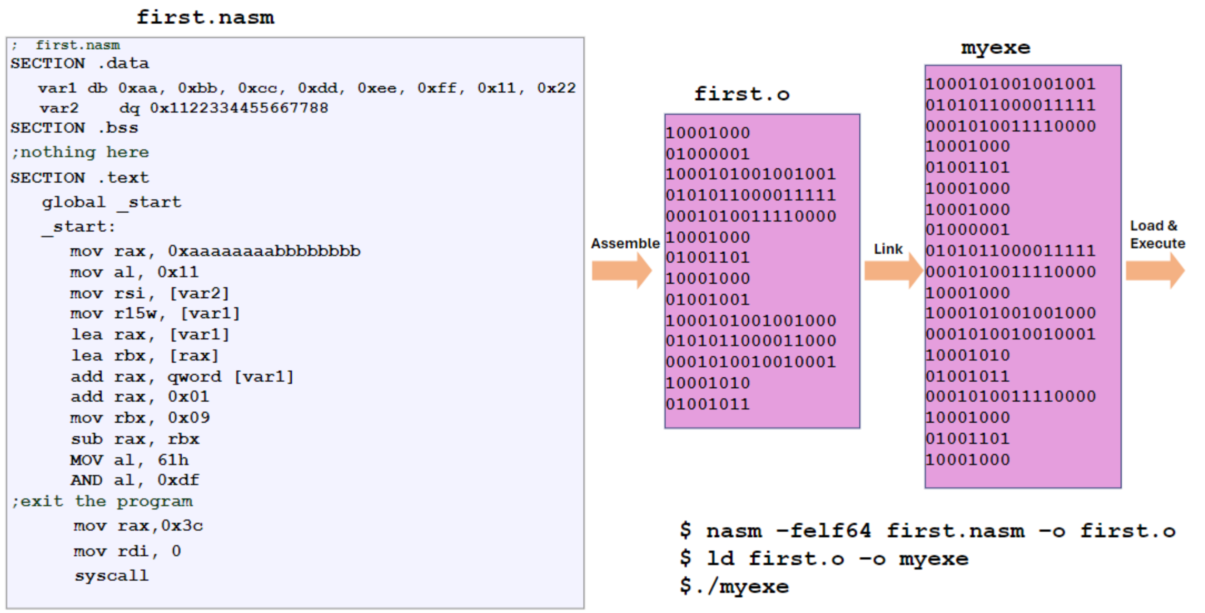
String	Move, compare, input and output	`movsb, movsw, lodsb, lodsw, stosb, stosw, rep, repz, repe, repnz, repne`
Floating Point	Arithmetic	`fld, fst, fstp, fadd, fsub, fmul, fdiv`
Conversion	Data type conversions	`cbw, cwd, cdq, xlat`
Input Output	For input and output	`in, out`
Miscellaneous	Manipulate individual flags	`clc, stc, cld, std, sti`

Note: A discussion on the working of all of the assembly instructions is beyond the scope of this session. Interested students are advised to go through related Video Lectures (26 – 46) from the x86-64 Assembly Programming course at the following link:

Video URL: https://www.youtube.com/playlist?list=PL7B2bn3G_wfCC2HDSXtMFsskasZ5fdLXz

Structure of x86-64 Assembly Program

The figure describes the structure of an `x86-64` assembly program in a text file named `first.nasm`. We can assemble it using `nasm`, to get an object file name `first.o`. Finally, we need to link the object file with the standard C to make an executable named `myexe`, ready to be loaded inside the memory and executed.



first.nasm

```
;  first.nasm
SECTION .data
   var1 db 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff, 0x11, 0x22
   var2    dq 0x1122334455667788
SECTION .bss
;nothing here
SECTION .text
   global _start
   _start:
      mov rax, 0xaaaaaaaabbbbbbbb
      mov al, 0x11
      mov rsi, [var2]
      mov r15w, [var1]
      lea rax, [var1]
      lea rbx, [rax]
      add rax, qword [var1]
      add rax, 0x01
      mov rbx, 0x09
      sub rax, rbx
      MOV al, 61h
      AND al, 0xdf
;exit the program
      mov rax,0x3c
      mov rdi, 0
      syscall
```

first.o

```
10001000
01000001
1000101001001001
0101011000011111
0001010011110000
10001000
01001101
10001000
01001001
1000101001001000
0101011000011000
0001010010010001
10001010
01001011
```

myexe

```
1000101001001001
0101011000011111
0001010011110000
10001000
01001101
10001000
10001000
01000001
0101011000011111
0001010011110000
10001000
1000101001001000
0001010010010001
10001010
01001011
0001010011110000
10001000
01001101
10001000
```

```
$ nasm -felf64 first.nasm -o first.o
$ ld first.o -o myexe
$./myexe
```

An assembly program is normally divided into three sections:

- **SECTION `.data`:** All initialized data like variables and constants are placed in the `.data` section
- **SECTION `.bss`:** All uninitialized data is declared in the `.bss` section (Block Storage Start)
- **SECTION `.text`:** This is actually the code section, and it will always include at least one label named `_start` or `main`, that defines the initial program entry point. The Linux linker `ld(1)`, expect the program entry point label with the name of `_start`, while `gcc(1)` expect the program entry point label with the name of `main`. The `global` directive is used to define a symbol, which is expected to be used by another module using the `extern` directive. The `extern` directive is used to declare a symbol which is not defined anywhere in the module being assembled, but is assumed to be defined in some other module.

There are three types of statements in assembly language programming:

- **x86-64 Assembly Instructions:** are converted into machine code. (`mov, add, sub, syscall`)
- **Pseudo Instruction:** are not real x86 machine instructions. Some NASM specific pseudo instructions are `DB, DW, RESB, RESW`
- **Assembler Directives:** are statements that direct the assembler to do something. Some NASM specific directives are `SECTION, EXTERN, GLOBAL, BITS`

What is a System Call

What is a System Call?

A **programmatic way** for a computer **program** to **request a service** from operating system (OS) **kernel** is to make a System Call. It's a controlled entry point that allows user programs to interact with the kernel and access resources or perform actions that require special privileges.

Why are System Calls Necessary?

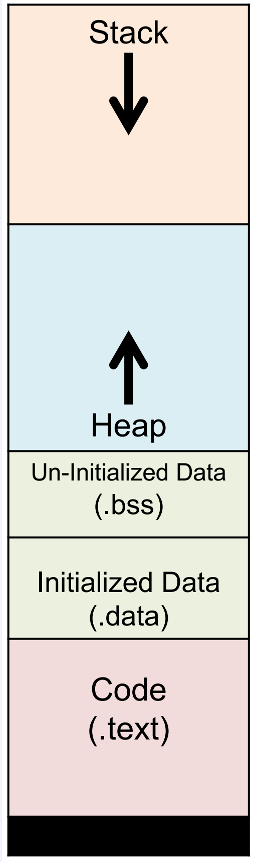
- The two methods using which a program can request the operating system to perform a service like printing on screen or reading from keyboard and so on are:
  - By making a system call
  - By making a library call
- A system call is a controlled entry point into the OS code, allowing a process to request OS to perform a privileged operation

List of available System Calls

- Every OS has its own set of system calls and every system call has an associated ID
- On my Intel Core i7 CPU, running Kali Linux 6.12, there are a total of 462 system calls, whose IDs can be seen from the file `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`

System Calls	ID
read()	0
write()	1
open()	2
close()	3
getpid()	39
shutdown()	48
fork()	47
exit()	60

```
; comment
SECTION .data
    msg db "Learning is fun with Arif", 0xA
    EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
    global _start
    _start:
;display a message on screen
        mov rax,1
        mov rdi,1
        mov rsi,msg
        mov rdx,26
        syscall
;exit the program
        mov rax,60
        mov rdi, EXIT_STATUS
        syscall
```



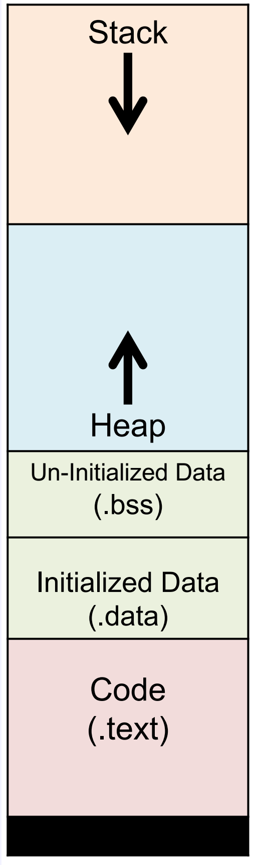
How to make a System Call?

- Depending on your processor architecture, you need to place the system call ID and arguments inside appropriate registers. Then you need to call the specific instruction to make the system call. Once the system call returns the return value can also be found inside a specific register as shown in the table below:

Architecture	System call ID	Instruction	Return Value
X86_64	rax	syscall	rax
80386	eax	int 0x80	eax
ARM	r7	svc 0	r7
ARM-64	x8	svc 0	r8

- For Linux running on `x86_64` processor, first six integer arguments are passed via `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9` registers and remaining (if any) are pushed on the stack.
- For MS Windows running on `x86-64` processor, first four integer arguments are passed via `rcx`, `rdx`, `r8`, `r9` registers and remaining (if any) are pushed on the stack.
- The Linux kernel interface avoids passing floating-point types to system calls as they are designed to work with integer types, pointers, and flags.

```
; comment

SECTION .data
   msg db "Learning is fun with Arif", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```



How to make a System Call? (cont…)

How to Display a Messsage on Screen

```
int write(int fd, void *buf, int count);
```

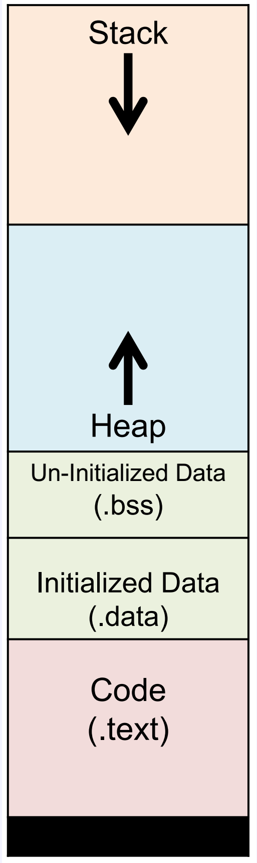
ID of write() system call	rax	1
arg1 (file descriptor)	rdi	1
arg2 (address of string)	rsi	msg
arg3 (length of string)	rdx	26

How to Terminate the Program

```
void exit(int status);
```

ID of exit() system call	rax	60
arg1 (exit status)	rdi	54

```
; comment

SECTION .data
   msg db "Learning is fun with Arif", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global _start
   _start:
;display a message on screen
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,26
      syscall
;exit the program
      mov rax,60
      mov rdi, EXIT_STATUS
      syscall
```



Demonstration

System Call

Lec1.2/syscalls.nasm

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

What is a Library Call

What is a Library Call?

A request made by a program to use a specific function or set of functions that are stored in a pre-compiled library.

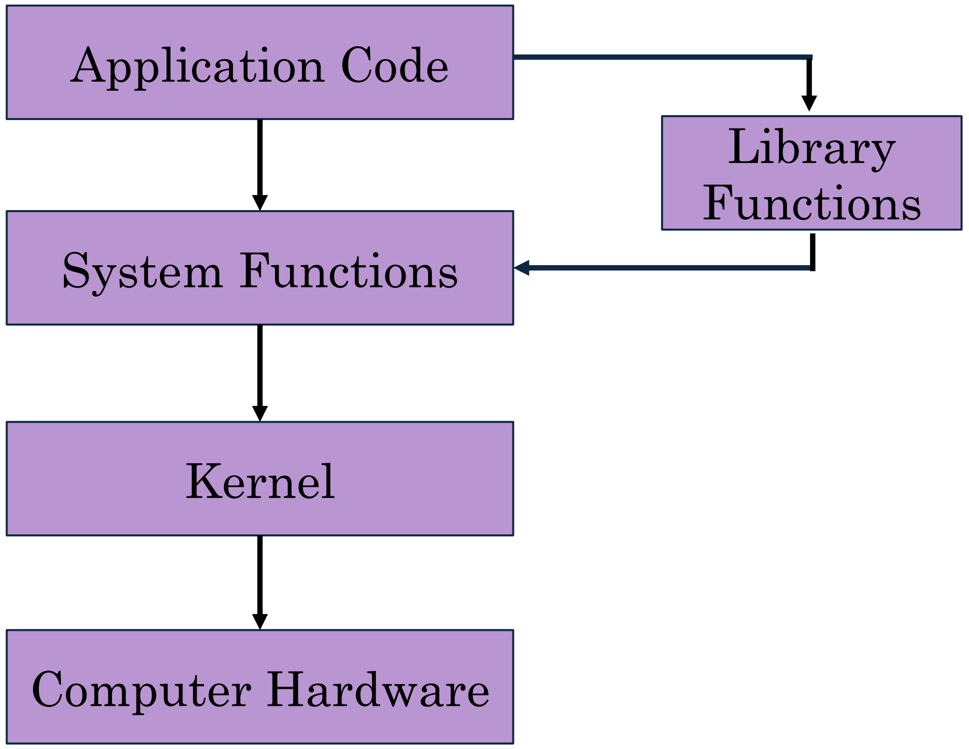
(It acts as a wrapper around system calls to provide user-friendly interfaces)

Why Use Library Calls?

- **Convenience:** Easier syntax, handles formatting and buffering.
- **Portability:** Same library code may work across platforms.
- **Functionality:** Combines multiple syscalls for complex tasks.
- **Examples:**
  - `printf()` - to display formatted output
  - `fopen()` - to open a file associating with a stream
  - `malloc()` - for dynamic memory allocation
  - `read()` - to read data from a file or input stream

How Library Call Work?

- Program calls a library function (e.g., `printf()`)
- The library processes arguments (e.g., formats string).
- It may eventually uses a system call like `write()` to display text.
- Control never leaves user mode until the system call is invoked.



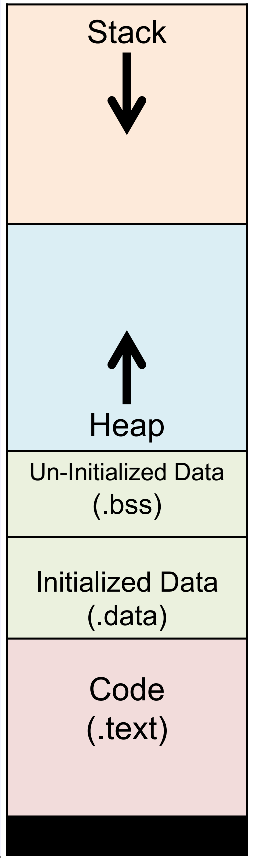
How to make a Library Call?

- An **x86-64** machine running **Linux** make a library call (user space function) using the **System V AMD64 ABI**. The first six integer arguments are placed inside rdi, rsi, rdx, rcx, r8, r9 registers and remaining (if any) are pushed on the stack, finally you make the **call** instruction to shift the control of execution to the library function. In case of floating-point arguments, the first eight are passed via xmm0 – xmm7 registers, rest are passed via stack. In case of integer return value it is saved in rax register, and the floating-point return value is stored in xmm0 register.

Note: The fourth argument in case of library call is stored in rcx, while in case of system call it is stored in r10. This is because, the syscall instruction clobbers the rcx register (it stores the return address there), so the system call convention had to use r10 to avoid this conflict.

- An **x86-64** machine running **Windows** makes a library call using the **Microsoft x64 calling convention**. The first four integer arguments are placed inside rcx, rdx, r8, r9 registers and remaining (if any) are pushed on the stack. In case of floating-point arguments, the first four are passed via xmm0-xmm3 registers, rest are passed via stack. In case of integer return value it is saved in rax register, and the floating-point return value is stored in xmm0 register.

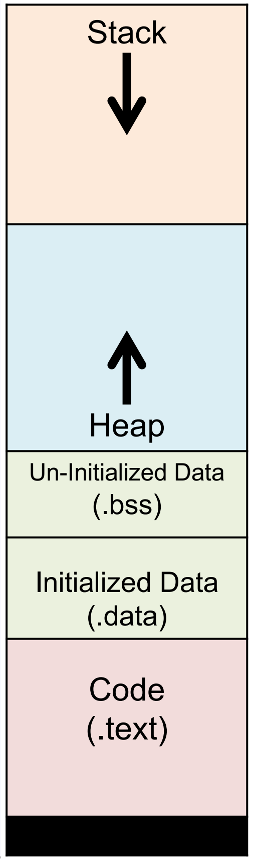
```
; comment

SECTION .data
   msg db "A hello to C library functions", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global main
   extern printf, exit
   main:
;display a message on screen
      lea rdi, msg
      xor rax, rax
      call printf
;exit the program
      mov rdi, EXIT_STATUS
      call exit
```



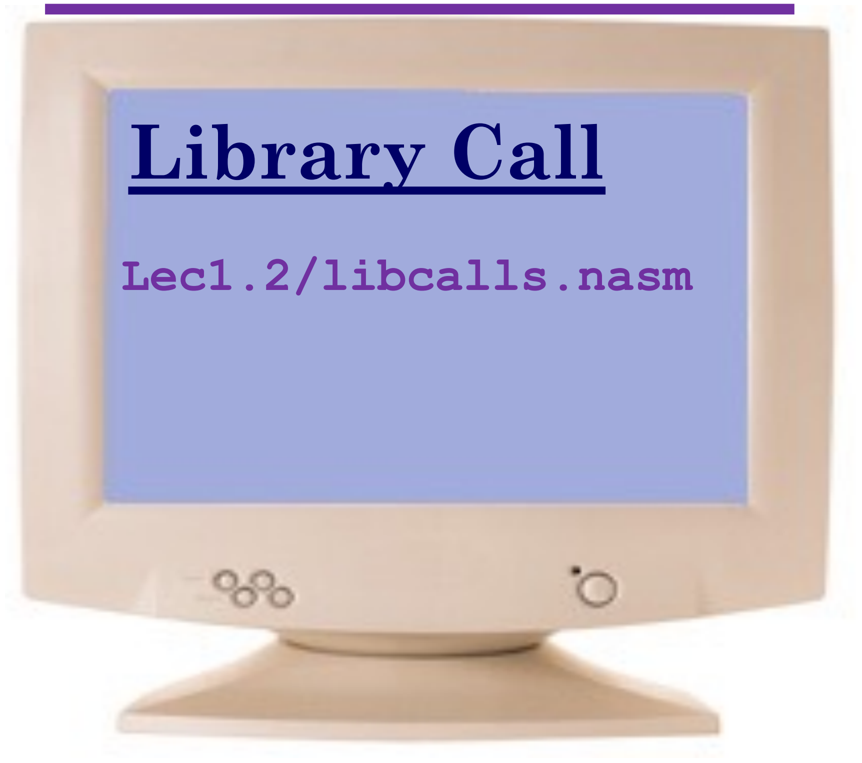
How to make a Library Call? (cont…)

- The **global** directive is used to **define** a symbol, which is expected to be used by another module using the extern directive. The **extern** directive is used to **declare** a symbol which is not defined anywhere in the module being assembled, but is assumed to be defined in some other module.
- Before calling the printf function, we need to place the first argument to printf inside the rdi register. Similarly, before calling the exit function, we need to place its first argument inside the rdi register.
- Since, printf is a variadic function, and it can be passed integer arguments as well as floating point arguments. In the x86-64 System-V calling convention, the rax register is used to specify the number of floating-point arguments passed to a function via vector registers (xmm0, xmm1, etc.). Since in the sample code, no floating-point arguments are passed to printf function, so rax must be set to 0. This ensures that the printf function knows that it doesn't need to fetch any values from the xmm registers.

```
; comment

SECTION .data
   msg db "A hello to C library functions", 0xA
   EXIT_STATUS equ 54

SECTION .bss
;nothing here yet

SECTION .text
   global main
   extern printf, exit
   main:
;display a message on screen
      lea rdi, [msg]
      xor rax, rax
      call printf
;exit the program
      mov rdi, EXIT_STATUS
      call exit
```



Demonstration



Library Call

Lec1.2/libcalls.nasm

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

System Call vs Library Call

Aspect	System Call	Library Call
Definition	A privileged operation that requires a transition from user mode to kernel mode to access operating system services	A function provided by programming libraries that executes in user space (may or may not call a system call)
Execution Mode	Executes in kernel mode with full system privileges and hardware access	Executes entirely in user mode with restricted access to system resources
Mode Transition	Requires special instructions to switch from user mode to kernel mode and back	No mode transition required and remains in user mode throughout execution
Performance	Slower execution due to context switching overhead and security validations like Spectre protection	Faster execution as there is no mode switching or context switching overhead
Implementation	Implemented in the kernel and provides direct interface to operating system services	Often implemented as wrappers around system calls or as standalone user-space functions
Security & Privileges	Runs with kernel privileges and includes permission/security validations	Runs with the calling process's privileges and typically has no additional security considerations
Portability	System-specific and closely tied to the underlying operating system	Generally more portable across different systems and platforms
Use Cases	Essential for operations requiring kernel services: file I/O, process management, memory allocation, network operations	Utility functions, data processing, mathematical operations, string manipulation
Examples	`open(), read(), write(), fork(), exec()`	`fopen(), printf(), malloc(), strcpy()`

User Defined Function

```
$ nasm -f elf64 -g funccalling.nasm
$ gcc -no-pie -g funccalling.o
$ ./a.out
OS course is fun!
$ ./a.out
5
```

```
;1.2/funccalling.nasm
SECTION .data
  msg db “OS course is fun!”, 0
SECTION .text
  global main
  extern printf, exit
  main:
    call printmsg
    mov rdi, 0
    call exit
printmsg:       ; display msg2 on screen
      lea rdi, [msg]
      xor rax, rax
      call printf
      ret
```

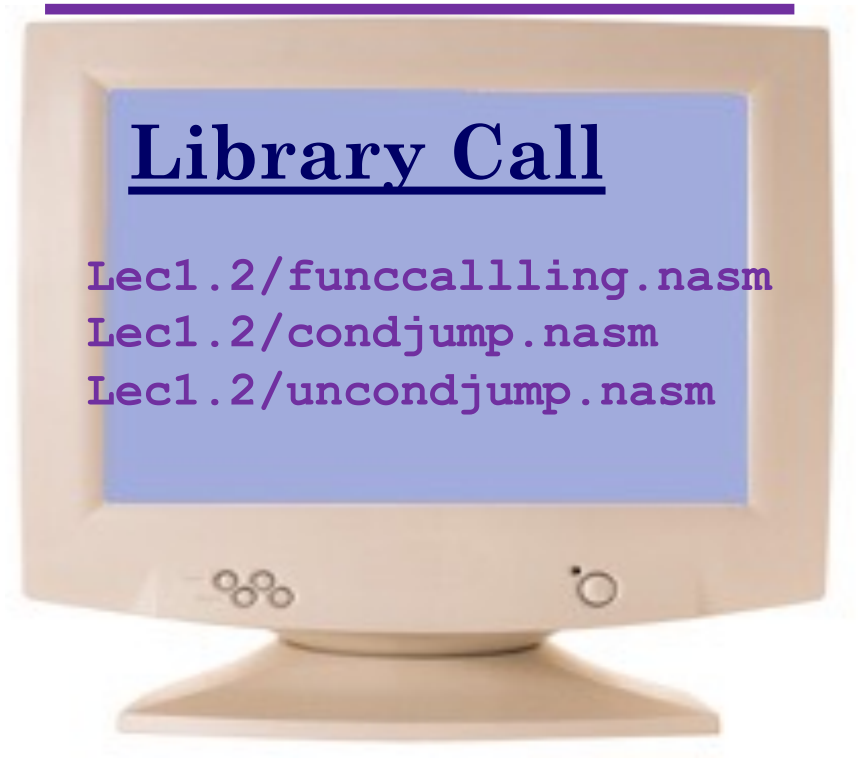
Instructor: Muhammad Arif Butt, PhD

Conditional Jump

```
$ nasm -f elf64 -g condjump.nasm
$ gcc -no-pie -g condjump.o -o myexe
$ ./a.out
Negative Number!
```

```
;1.2/condjump.nasm
SECTION .data
msg1 db “Negative Number!”, 0
msg2 db “Positive Number!”, 0
SECTION .text
global main
extern printf, exit
main:
   mov ax, -5d
   cmp ax, 0
   jge _positive
   lea rdi, [msg1] ; display msg1 on screen
   xor rax, rax
   call printf
   jmp _end
_positive: ; display msg2 on screen
      lea rdi, [msg2]
      xor rax, rax
      call printf
_end:    ; exit the program gracefully
   mov rdi, 0
   call exit
```

Demonstration



Library Call

```
Lec1.2/funccallling.nasm
Lec1.2/condjump.nasm
Lec1.2/uncondjump.nasm
```

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

Running Assembly with gdb (GEF)

GNU Project Debugger (GDB)

A debugger is a program running another program allowing you to see what is going on inside another program while it executes, or what another program was doing at the moment it crashed. There exist different types of debuggers like `GNU gdb (PEDA/GEF)`, `IDA Pro`, `radare2`, `cutter`, `ghidra`, `OllyDbg`, `binaryninja`, `ptrace`, `strace`, `ltrace`, `ftrace`, `bpftrace` and so on. Using a debugger, a programmer can:

- Start a program, specifying anything that might affect its behavior.
- Make a program stop on specified conditions.
- Examine what has happened, when a program has stopped/crashed.
- Change things in a program, so you can experiment with correcting the effects of one bug and go on to learn about another.
- Last but not the least, can be used for run time analysis of binaries, disassembly, reverse engineering and cracking binaries.
- We will be using GDB, the GNU Project debugger that can debug a program running on the same machine as GDB (native), or may be another machine (remote), or may be on a simulator.
- GDB is a portable debugger that can run on the most popular UNIX and Microsoft Windows variants, as well as on Mac OS X. The target processors include IA-32, x86-64, arm, mips, powerpc, sparc, alpha and many others. GDB works for many programming languages including Assembly, C/C++, Objective C, OpenCL, Go, Modula-2, Fortran, Pascal and Ada.

Summary of GDB Commands

Commands	Description
`$ nasm -g -felf64 prog1.nasm`	To load and properly analyze a program in `gdb` you need to compile it with `-g` option, to instruct the compiler to keep debugging symbols, source file names and line numbers in the object files

`$ gdb`
`(gdb)file myexe`
`$ gdb myexe`
	There are two ways to load a binary inside `gdb` by either running `gdb` command and then specifying the binary name with the file command. Or by specifying the binary name as an argument to `gdb`.

`(gdb)quit`	Exits the current session of `gdb`.

`(gdb)help`
`(gdb)help <classname>`
`(gdb)help <command>`
	The `help` command of `gdb` is used to display the listing of twelve different classes in which `gdb` commands are categorized. You can also specify the classname (breakpoints, running, stack, …) or the command to get help about it.

`(gdb)run [arg1 arg2 …]`
`(gdb)set args arg1 arg2 …`
`(gdb)run`
	Once the program is loaded and `gdb` is running, you can pass command line arguments to the binary using the `run` command of `gdb`. Or can use the `set` command instead and later use the `run` command.

`(gdb)info sources/functions`
`(gdb)info registers/all`
`(gdb)info sharedlibrary`
`(gdb)info address <function name>`
	Once a program is loaded inside `gdb`, you can use the `info` command to display the name of all the source files from which symbols have been read in, name of functions, global variables, name of local variables inside a FSF, and the CPU registers.

`(gdb)disassemble`
`(gdb)disassemble <function name>`
`(gdb)set disassembly-flavor intel`
	Disassembles the current function or code segment. By default, `gdb` disassembles in `AT&T` format, to change the format to `intel`, use the `set disassembly-flavor` command.

`(gdb)break <filename>:<line#>`
`(gdb)break <filename>:<func-name>`
`(gdb)break <filename>:*0x2xfff0500`
	Breakpoint is the LOC in your program where you want to stop the execution. You can set as many break points as you feel like using the `break` command of `gdb` by mentioning the line#, function name, or by virtual address

Summary of GDB Commands (cont…)

Commands	Description

`(gdb)info break`
`(gdb)delete <breakpoint#>`
`(gdb)clear <breakpoint#>`
	To get the information about the existing breakpoints already set in your program, you can use the `info` command. Moreover, you can `disable`, `enable`, `delete`, and `clear` breakpoints.

`(gdb)watch <variable name>`
`(gdb)info watch`
`(gdb)clear <watchpoint#>`
	Like breakpoints, we can set watchpoints on variables. Whenever the value of that variable will change, `gdb` will interrupt the program and print out the old and the new value. Moreover, you can disable, enable, delete and clear watchpoints.

`(gdb)continue / c / ci`
`(gdb)next / n / ni`
`(gdb)step / s / si`
`(gdb)finish`
	Once a breakpoint is hit, you can do the following:
	o `c`: Continue till the next breakpoint or end of program.
	o `n`: Execute and move to next instruction, but don't dive into functions.
	o `s`: Execute and move to next instruction, by diving into functions.
	o `finish`: Continue until the current function returns.

`(gdb)print /format-char <var-name>`	Once a breakpoint is hit during execution of a program, you can inspect contents of variables using the `print` command in the specified format (`/d` is for signed decimal, `/x` for printing as hex, `/o` for printing as octal, `/t` for printing as binary, `/f` for floating point number, `/s` for C-string, `/a` for address). You can also use the `display` command that displays the value of variable, each time the program stops.

`(gdb)set variable var1 = <value>`
`(gdb)set $rdi = 0x7fff12345678`
	Once a breakpoint is hit during execution of a program, you can use the `set` command to modify the value of a variable or register.

`(gdb)x/12cb <address>`
`(gdb)x/12db &var1`
`(gdb)x/4xb *0x601000`
`(gdb)x/32b $rsp`
	Once a breakpoint is hit during execution of a program, the `examine` command or its alias `x` is passed a memory address to display its contents. It is optionally followed by a forward slash (/) and then a **count** field, which is a number in decimal, a **format** field, which is a single letter with `d` for decimal, `o` for octal, `x` for hex, `t` for binary, `c` for ASCII, and `s` for string a **size** field, which is single letter with `b` for byte, `h` for 16-bit word, `w` for 32-bit word and `g` for 64-bit word.

`(gdb)! clear`	To run the OS shell commands inside gdb, you can precede the command with a ! symbol.

GDB with GEF Plugin

- GNU GDB is too good a debugger, however, it lacks intuitive interface, do not have a smart context display, do not have commands for exploit development, and has weak scripting support. So, to enhance the fire power of gdb for analyzing, exploiting and doing reverse engineering on executables, one can use:
  - a gdb plug-in called **PEDA** (Python Exploit Development Assistance)
  - a gdb plug-in called **GEF** (GDB Enhanced Features)

Installation of PEDA: https://github.com/longld/peda

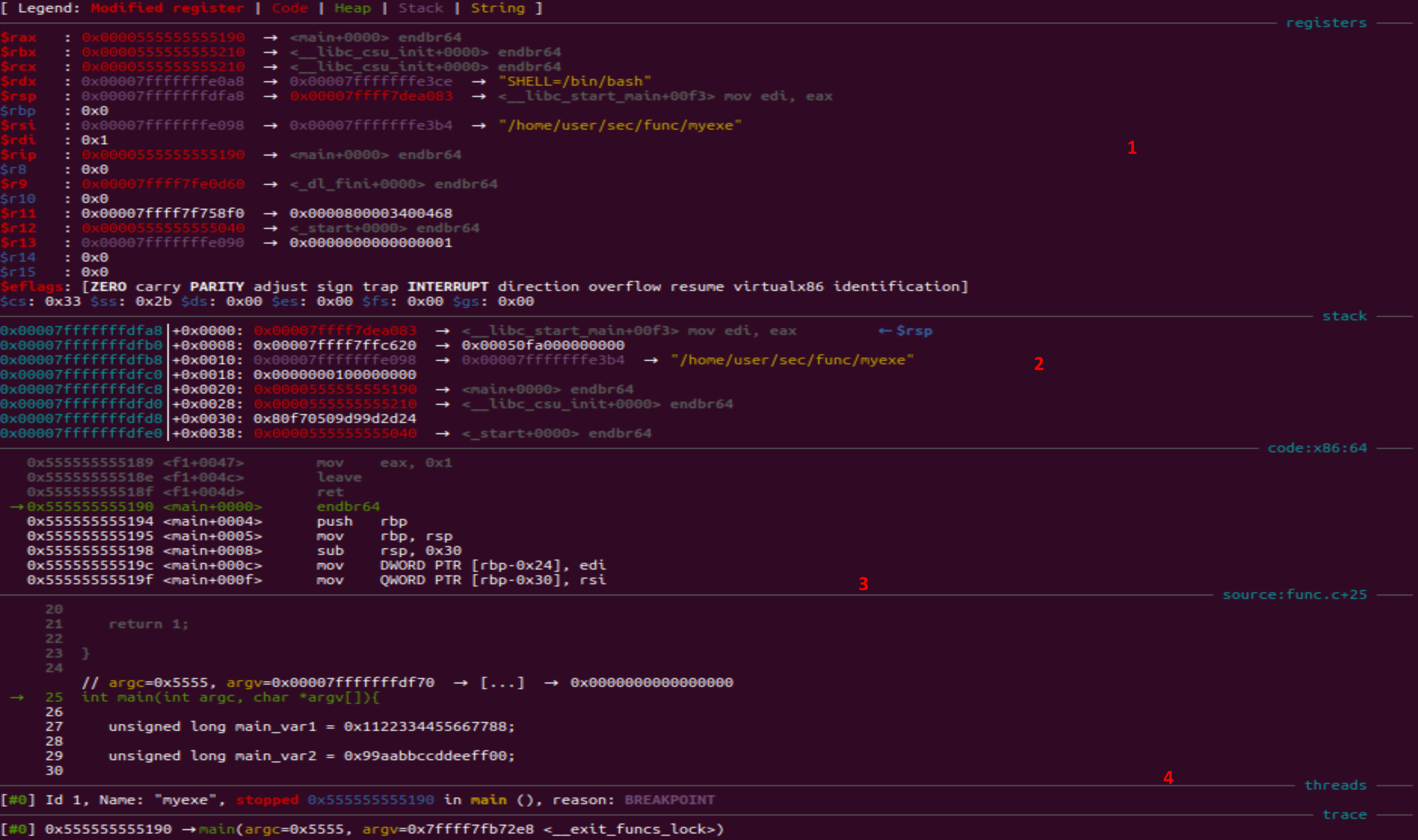
PEDA is available only on Linux and supported by `gdb 7.x` and `Python 2.6` onwards. In order to install PEDA plugin for `gdb`, you simply have to download or clone its repository and then update the `.gdbinit` file in your home directory:

```
$ git  clone https://github.com/longld/peda.git     ~/peda
$ echo  "source ~/peda/peda.py"    >>     ~/.gdbinit
```

Installation of GEF: https://github.com/hugsy/gef.git

On the same grounds, if you want to install GEF plugin for `gdb`, you simply have to download it and then update the `.gdbinit` file in your home directory as shown below:

```
$ git clone https://github.com/hugsy/gef.git  ~/gef
$ echo "source ~/gef/gef.py" >> ~/.gdbinit
```



```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x0000555555555190  →  <main+0000> endbr64
$rbx   : 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
$rcx   : 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
$rdx   : 0x00007fffffffe0a8  →  0x00007fffffffe3ce  →  "SHELL=/bin/bash"
$rsp   : 0x00007fffffffdfa8  →  0x00007ffff7dea083  →  <__libc_start_main+00f3> mov edi, eax
$rbp   : 0x0
$rsi   : 0x00007fffffffe098  →  0x00007fffffffe3b4  →  "/home/user/sec/func/myexe"
$rdi   : 0x1                                                                   1
$rip   : 0x0000555555555190  →  <main+0000> endbr64
$r8    : 0x0
$r9    : 0x00007ffff7fe0d60  →  <_dl_fini+0000> endbr64
$r10   : 0x0
$r11   : 0x00007ffff7f758f0  →  0x0000800003400468
$r12   : 0x0000555555555040  →  <_start+0000> endbr64
$r13   : 0x00007fffffffe090  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffdfa8│+0x0000: 0x00007ffff7dea083  →  <__libc_start_main+00f3> mov edi, eax      ← $rsp
0x00007fffffffdfb0│+0x0008: 0x00007ffff7ffc620  →  0x00050fa000000000
0x00007fffffffdfb8│+0x0010: 0x00007fffffffe098  →  0x00007fffffffe3b4  →  "/home/user/sec/func/myexe"      2
0x00007fffffffdfc0│+0x0018: 0x0000000100000000
0x00007fffffffdfc8│+0x0020: 0x0000555555555190  →  <main+0000> endbr64
0x00007fffffffdfd0│+0x0028: 0x0000555555555210  →  <__libc_csu_init+0000> endbr64
0x00007fffffffdfd8│+0x0030: 0x80f70509d99d2d24
0x00007fffffffdfe0│+0x0038: 0x0000555555555040  →  <_start+0000> endbr64
──────────────────────────────────────────────────────────────────────── code:x86:64 ────
   0x555555555189 <f1+0047>        mov    eax, 0x1
   0x55555555518e <f1+004c>        leave
   0x55555555518f <f1+004d>        ret
 →0x555555555190 <main+0000>       endbr64
   0x555555555194 <main+0004>      push   rbp
   0x555555555195 <main+0005>      mov    rbp, rsp
   0x555555555198 <main+0008>      sub    rsp, 0x30
   0x55555555519c <main+000c>      mov    DWORD PTR [rbp-0x24], edi
   0x55555555519f <main+000f>      mov    QWORD PTR [rbp-0x30], rsi                3
──────────────────────────────────────────────────────────────────── source:func.c+25 ────
     20
     21     return 1;
     22
     23  }
     24
        // argc=0x5555, argv=0x00007fffffffdf70  →  [...]  →  0x0000000000000000
 →   25  int main(int argc, char *argv[]){
     26
     27     unsigned long main_var1 = 0x1122334455667788;
     28
     29     unsigned long main_var2 = 0x99aabbccddeeff00;
     30                                                                          4
──────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "myexe", stopped 0x555555555190 in main (), reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x555555555190 →main(argc=0x5555, argv=0x7ffff7fb72e8 <__exit_funcs_lock>)
```

GDB with GEF Plugin (cont…)

1. **Registers Panel:** The Registers Panel in GEF displays the current values of the CPU registers/flags, providing an organized and easily readable view. It helps in analyzing the state of the CPU, tracking changes in register values, and debugging at a lower level. It does not show the floating-point registers, however, you can view the contents of all registers, use the info all command of gdb.
2. **Stack Panel:** The Stack Panel displays top of the call stack, which includes a list of function calls that are currently active. This is really beneficial to understand the current Function Stack Frame of a function. Remember, the top of the stack is displayed at the top of this panel, where the rsp register is pointing.
3. **Code Panel:** The Code Panel displays the assembly code along with the virtual addresses. The line currently being executed or where the breakpoint is set is typically highlighted or marked to provide a clear point of focus.
4. **Source Panel:** This panel displays the corresponding high level language code, with the current LOC highlighted. This way you can corelate the high-level code with its corresponding assembly.
5. **Threads & Trace Panels:** This provides information about the threads in a multithreaded program, including their states and stack traces.

Note: To configure the panels to be displayed, you can use the following command inside `gef`:

```
gef> gef config context.layout “regs stack code source”
```

Demonstration

Running Assembly inside GDB

```
Lec1.2/gdb/movingdata.nasm
Lec1.2/gdb/arithmetic.nasm
Lec1.2/gdb/logical.nasm
Lec1.2/gdb/bitshift.nasm
Lec1.2/gdb/bitrotate.nasm
Lec1.2/gdb/stack.nasm
```

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 COAL videos on x86-64 Assembly:
  https://www.youtube.com/watch?v=sg3GIXvS36w&list=PL7B2bn3G_wfCC2HDSXtMFsskasZ5fdLXz&index=29&t=1s
  https://www.youtube.com/watch?v=aed7-FDu0qg&list=PL7B2bn3G_wfCC2HDSXtMFsskasZ5fdLXz&index=30&t=2s
  https://www.youtube.com/watch?v=9vkRSkS26_k&list=PL7B2bn3G_wfCC2HDSXtMFsskasZ5fdLXz&index=31&t=3s
  https://www.youtube.com/watch?v=2x-pkzSmsD8&list=PL7B2bn3G_wfCC2HDSXtMFsskasZ5fdLXz&index=32&t=467s

- Watch GDB video for Assembly:
  https://www.youtube.com/watch?v=WgGogyMcM7s&list=PL7B2bn3G_wfCC2HDSXtMFsskasZ5fdLXz&index=41

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