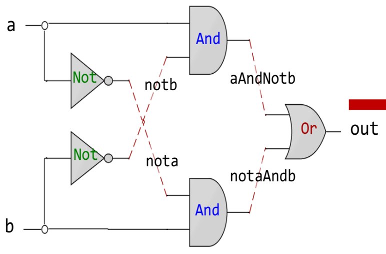
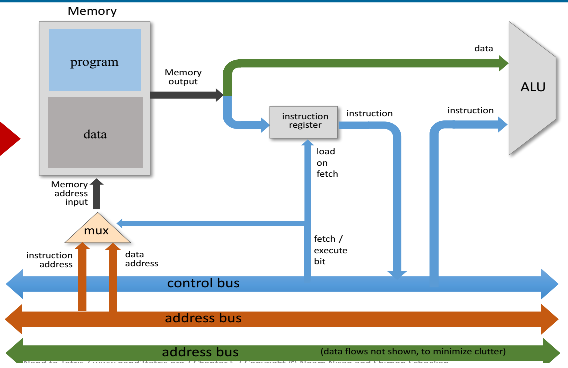
Computer Organization & Assembly Language Programming



```
CHIP Xor {
   IN a, b;
   OUT out;
   PARTS:
   Not(in=a, out=nota);
   Not(in=b, out=notb);
   And(a=nota, b=b, out=w1);
   And(a=a, b=notb, out=w2);
   Or(a=w1, b=w2, out=out);
}
```



```
@R1
D=M
@temp
M=D
```

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
```

Lecture # 46

Getting User Input via read and Command Line

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```

```
0:  b8 01 00 00 00
5:  bf 01 00 00 00
a:  48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```

For resources visit my personal website:
https://www.arifbutt.me
and course bitbucket repository:
https://bitbucket.org/arifpucit/coal-repo

Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

- Getting and validating user input
- Getting user input via system call **`read()`**
  - Demo (***`prog1.nasm`***)
- Getting user input via library call **`gets()`**
  - Demo (***`prog2.nasm`***)
- What are command line arguments
  - Demo (***`prog3.nasm`***)
- Getting user input via command line arguments
  - Demo (***`prog4.nasm`***)

Getting Input from User

- Writing a computer program in any programming language, a programmer may need to get input from user via keyboard (**`stdin`**) and later displays the result on screen (**`stdout`**)
- All programming languages like C, C++, Java, Python, and so on, provide library functions that perform all sort of such I/O functionality
- Internally, all the functions in these libraries make use of some operating system service to perform the task
- Today we are going to discuss three different ways to get input from user via keyboard in x86-64 assembly language programming:
    - By making a system call during the program execution
    - By making a library call during the program execution
    - Using Command Line Arguments before the program executes
- The input given by the user must be properly tested/validated to ensure that it is as per the expectation of the program. Malicious input can include code, scripts and commands, which if not validated correctly can be used to exploit vulnerabilities like Buffer Overflow, XSS, SQL injection etc.

Getting User Input via System Call

The **read** System Call

Making a System Call on x86-64 running Linux Operating System

- First of all depending on your architecture, you need to place the system call ID in an **`rax`** register
- Next step is to place the system call arguments inside registers: **`rdi, rsi, rdx, rcx, r8, r9`**. In case of more than six arguments push them on stack
- After the system call returns, the return value can be found inside **`rax`** register

List of available System Calls

- Every operating system has its own set of system calls and every system call has an associated ID
- On my Intel Core i7 CPU, running Kali Linux 5.3, there are a total of 433 system calls, whose IDs can be seen from the file `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`
- Some important system calls and their IDs are mentioned in the table

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

```
int read(int fd, void* buff, int count);
```

Example: *prog1.nasm*

```
; COAL Video Lecture: 46
;  Programmer: Arif Butt
;  prog1.nasm
SECTION .data
   text1       db  "What is your name? "
   len_text1   equ  $ - text1
   text2       db   "Hello Mr. "
   len_text2   equ  $ - text2

SECTION .bss
   name resb 10

SECTION .text
   global main
   main:
;prompt the user to input his/her name
      mov rax,1
      mov rdi,1
      mov rsi,text1
      mov rdx,len_text1
      syscall      ;write(1,text1,len)
;get name from user
      mov rax,0
      mov rdi,0
      mov rsi,name
      mov rdx,10
      syscall      ;read(0,name,10)
```

```
; cont…
;display hello
      mov rax,1
      mov rdi,1
      mov rsi,text2
      mov rdx,len_text2
      syscall ;write(1,text2,len)
;display name
      mov rax,1
      mov rdi,1
      mov rsi,name
      mov rdx,10
      syscall ;write(1,name,10)
;exit program
      mov rax,60
      mov rdi,0
      syscall ;exit(0)
```

```
$ nasm –felf64 prog1.nasm
$ gcc prog1.o –o myexe
$ ./myexe
What is your name? Arif Butt
Hello Mr. Arif Butt
```

Example: Getting User input via read System Call

Getting User Input via Library Call

Example: *prog2.nasm*

```
; COAL Video Lecture: 46
;  Programmer: Arif Butt
;  prog2.nasm
SECTION .data
   formatStr1 db "What is your name?", 0
   formatStr2 db "Hello Mr. %s", 0xA
SECTION .bss
   name resb 10
SECTION .text
   global main
   extern printf
   extern gets
   extern exit
   main:
;prompt the user to input his/her name
   lea rdi, formatStr1
   call printf
;get name from user
   lea rdi, name
   call gets
;display hello message with name
   lea rdi, formatStr2
   lea rsi, name
   call printf
;make the exit library call
   mov rdi, 0
   call exit
```

```
$ nasm –felf64 prog1.nasm
$ gcc prog1.o –o myexe
$ ./myexe
What is your name? Arif Butt
Hello Mr. Arif Butt
```

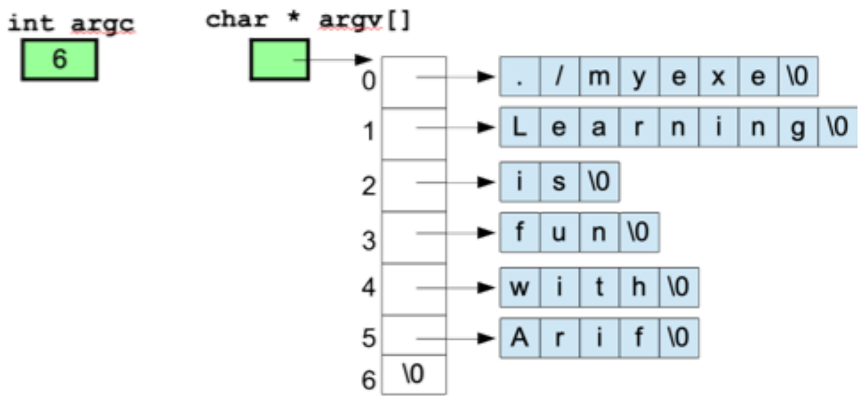
Example: Getting User input via Library Call

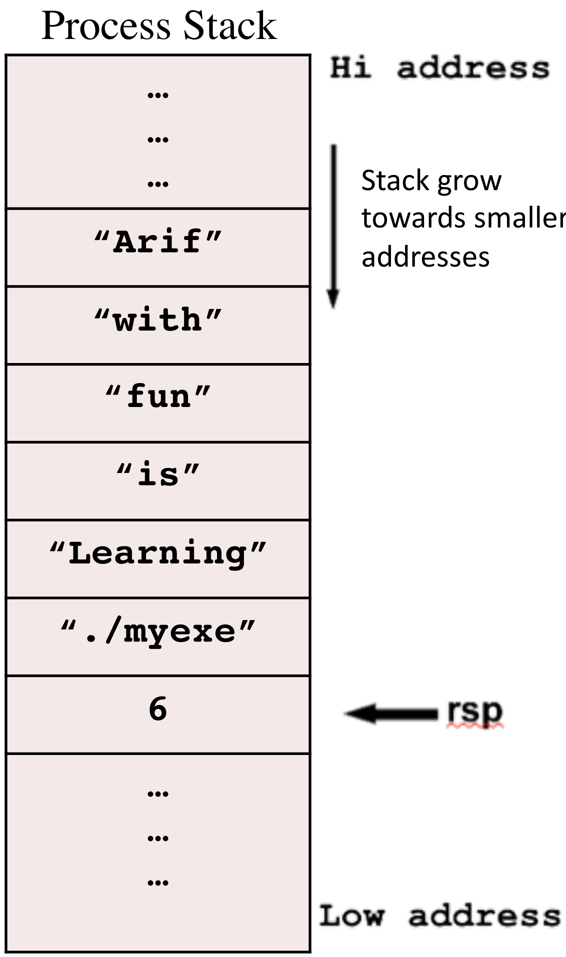
Getting User Input via Command Line Arguments

Command Line Arguments

```
// A C-program that receives command line arguments
  int main(int argc, char *argv[]){
    printf("No of arguments passed are: %d\n",argc);
    printf("Parameters are:\n");
    for(int i = 0; argv[i] != NULL ; i++)
      printf("argv[%d]:%s \n", i, argv[i]);
    return 0;
}
```

```
$ gcc prog1.c –o myexe
$ ./myexe Learning is fun with Arif
No of arguments passed are: 6
Parameters are:
argv[0]: ./myexe
argv[1]: Learning
argv[2]: is
argv[3]: fun
argv[4]: with
argv[5]: Arif
```

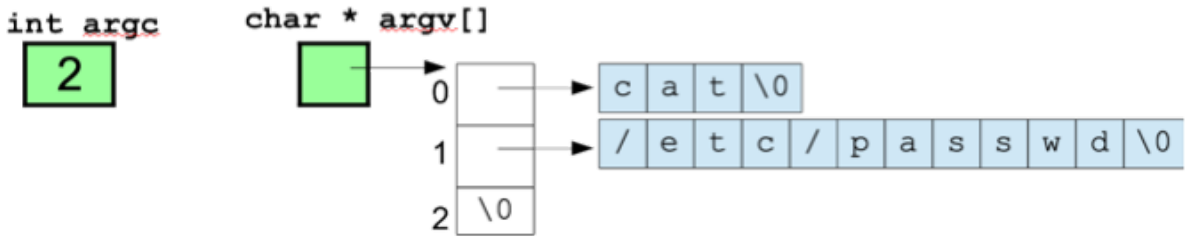




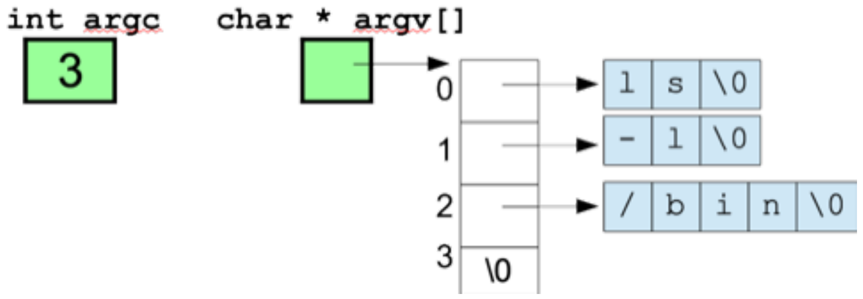
Use of Command Line Arguments

Command line arguments allow a user to input data into a program without the program requiring a user interface

```
$ cat /etc/passwd
```



```
$ ls -l /bin
```



Example: *prog3.nasm*

```
SECTION .text
   global  main
   extern  puts
main:
; need to save registers rdi and rsi that puts uses
   push    rdi
   push    rsi
; need to align stack before call
   sub     rsp, 8
; need to place argument string to display
   mov     rdi, [rsi]
   call    puts
; need to restore rsp, rdi and rsi
   add     rsp, 8
   pop     rsi
   pop     rdi
; let rsi point to next argument and decrement argument count
   add     rsi, 8
   dec     rdi
   jnz     main
   ret
```

Example: *prog3.nasm*

```
SECTION .text
   global  main
   extern  puts
main:
; need to save registers rdi and rsi that puts uses
   push    rdi
   push    rsi
; need to align stack before call
   sub     rsp, 8
; need to place argument string to display
   mov     rdi, [rsi]
   call    puts
; need to restore rsp, rdi and rsi
   add     rsp, 8
   pop     rsi
   pop     rdi
; let rsi point to next argument and decrement argument count
   add     rsi, 8
   dec     rdi
   jnz     main
   ret
```

Example: Getting User input via Command Line

Demo

```
46/prog3.nasm
46/prog4.nasm
```

Things To Do

Coming to office hours does NOT mean you are academically week!