THE UNIVERSITY OF ARIZONA
UA South

CYBV 471 Assembly Programming for Security Professionals
Week 12

System Calls and I/O:
Build Your I/O Assembly functions

Agenda

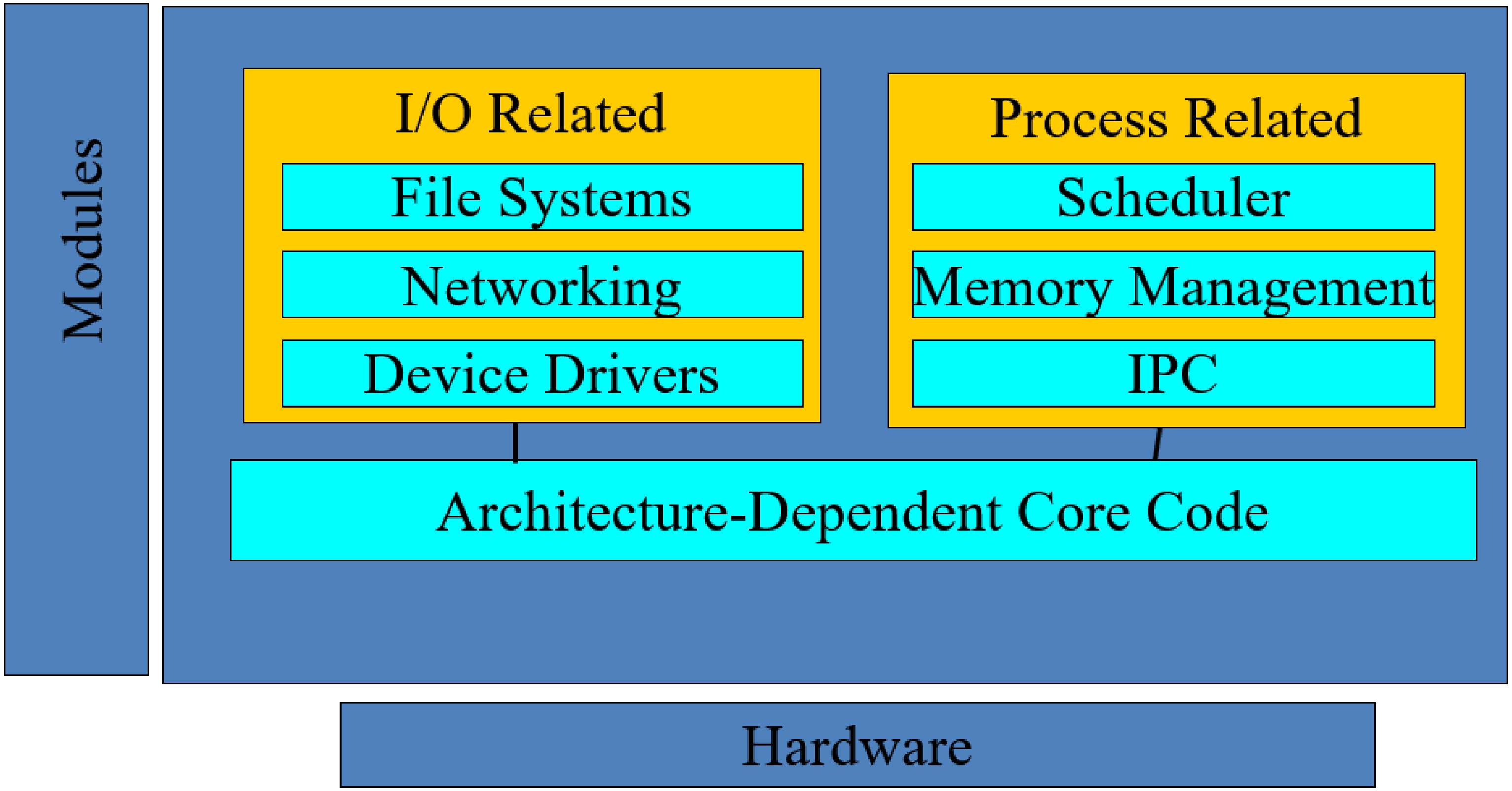
- **Linux kernel structure**
- **Kernel and Operating System**
- **User mode vs kernel mode**
- **What is a system call?**
- **Difference between function and system calls**
- **How do System Calls use Registers?**
- **Use system calls to**
  - Print string
  - Print character
  - Print integer value
  - Provide input to the program

Kernel and Operating System

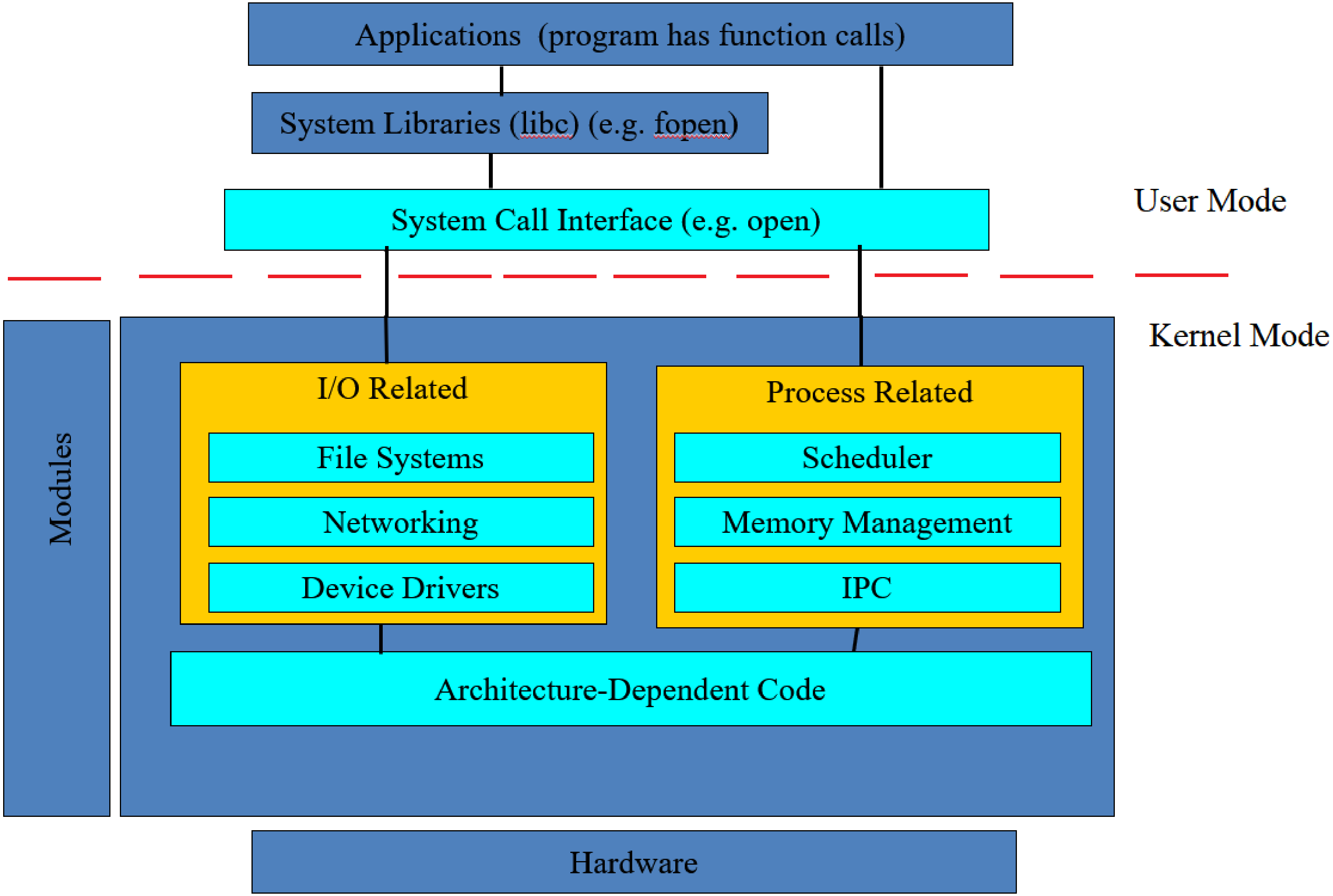
- A *kernel* is the heart of the operating system that control and mange access to system resources.
- It's responsible for enabling multiple applications to effectively share the hardware by controlling access to CPU, memory, disk I/O, and networking
- An *operating system* is the kernel plus applications that enable users to accomplish some tasks (e.g. text editor, file system utilities, etc)

Linux kernel structure

- Include core and dynamically loadable modules
- Modules include: device drivers, file systems, network protocols, etc
- Modules were originally developed to support the device drivers
- Modules can be dynamically loaded and unloaded



User Mode vs kernel Mode

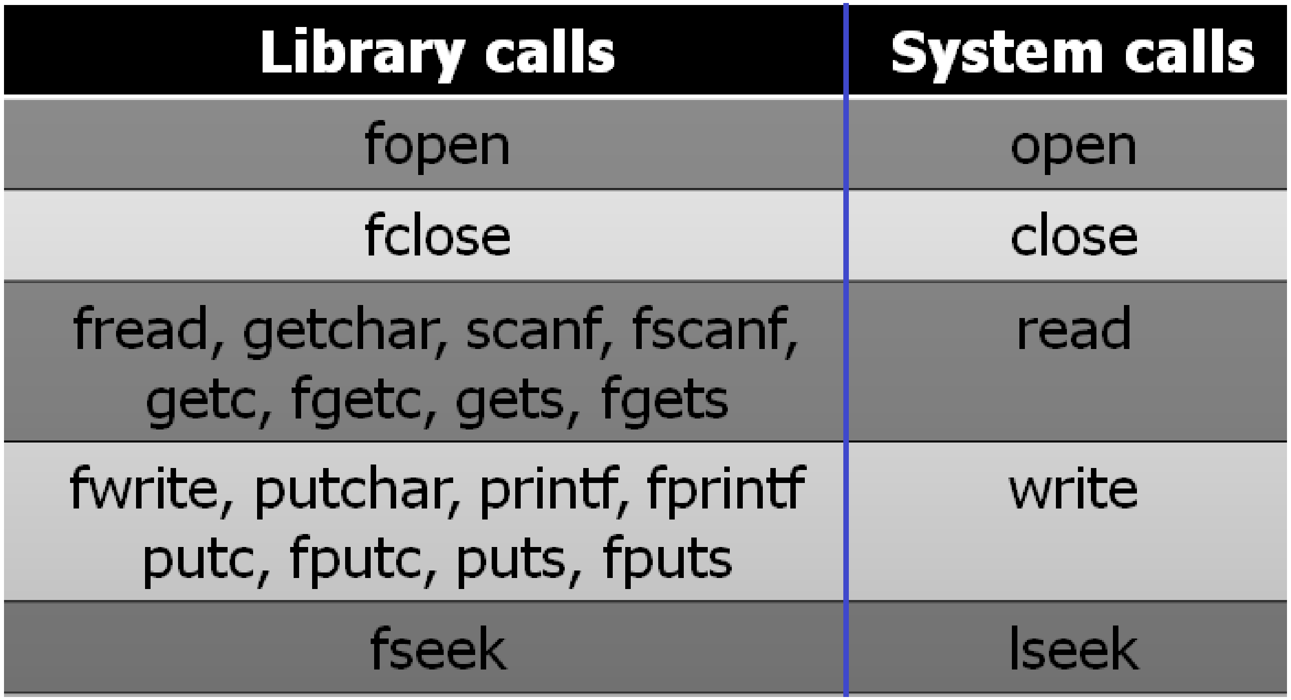

Applications (program has function calls)
System Libraries (libc) (e.g. fopen)
System Call Interface (e.g. open)
User Mode
Kernel Mode
Modules
I/O Related
File Systems
Networking
Device Drivers
Process Related
Scheduler
Memory Management
IPC
Architecture-Dependent Code
Hardware

What is a System Call?

- A System Call
  - A request to the operating system to perform some task
  - Allow a user to control the operating system
- Example:
  - getuid() //get the user ID
  - fork() //create a child process
  - exec() //execute a program

Function and Standard Library calls vs System Calls

- Function and standard library calls perform in user mode
- Function and standard library calls use system calls to perform some OS activities
- A System Call
  - A request to the operating system to perform some activity
  - Allow a user to control the operating system



Library calls	System calls
fopen	open
fclose	close
fread, getchar, scanf, fscanf, getc, fgetc, gets, fgets	read
fwrite, putchar, printf, fprintf putc, fputc, puts, fputs	write
fseek	lseek

Function and Standard Library calls vs System Calls

- Examples

```
- fopen (char* filename, char* mode);               // Standard Library call
- open (char* filename, int flags [char* mode]);    // General System call

- fread (int fd, , char *buf, int size);        // Standard Library call
- read (int fd, char *buf, int size);          // General System call

- fwrite (int fd, , char *buf, int size);       // Standard Library call
- write (int fd, char *buf, int size);         // General System call

- fseek (int fd, int offset, int whence);      // Standard Library call
- lseek (int fd, int offset, int whence);      // General System call

- fclose (FILE *fp);                           // Standard Library call
- close (int fd);                              // General System call
```

Standard Input, Output and Error

- Standard file descriptors:
  - File descriptor 0 is standard input (keyboard)
  - File descriptor 1 is standard output (screen)
  - File descriptor 2 is standard error.
- You can read from standard input, using **read(0, ...).**
  - **eax= 03**
- You can write to standard output using **write(1, ...)**
  - **eax= 04**

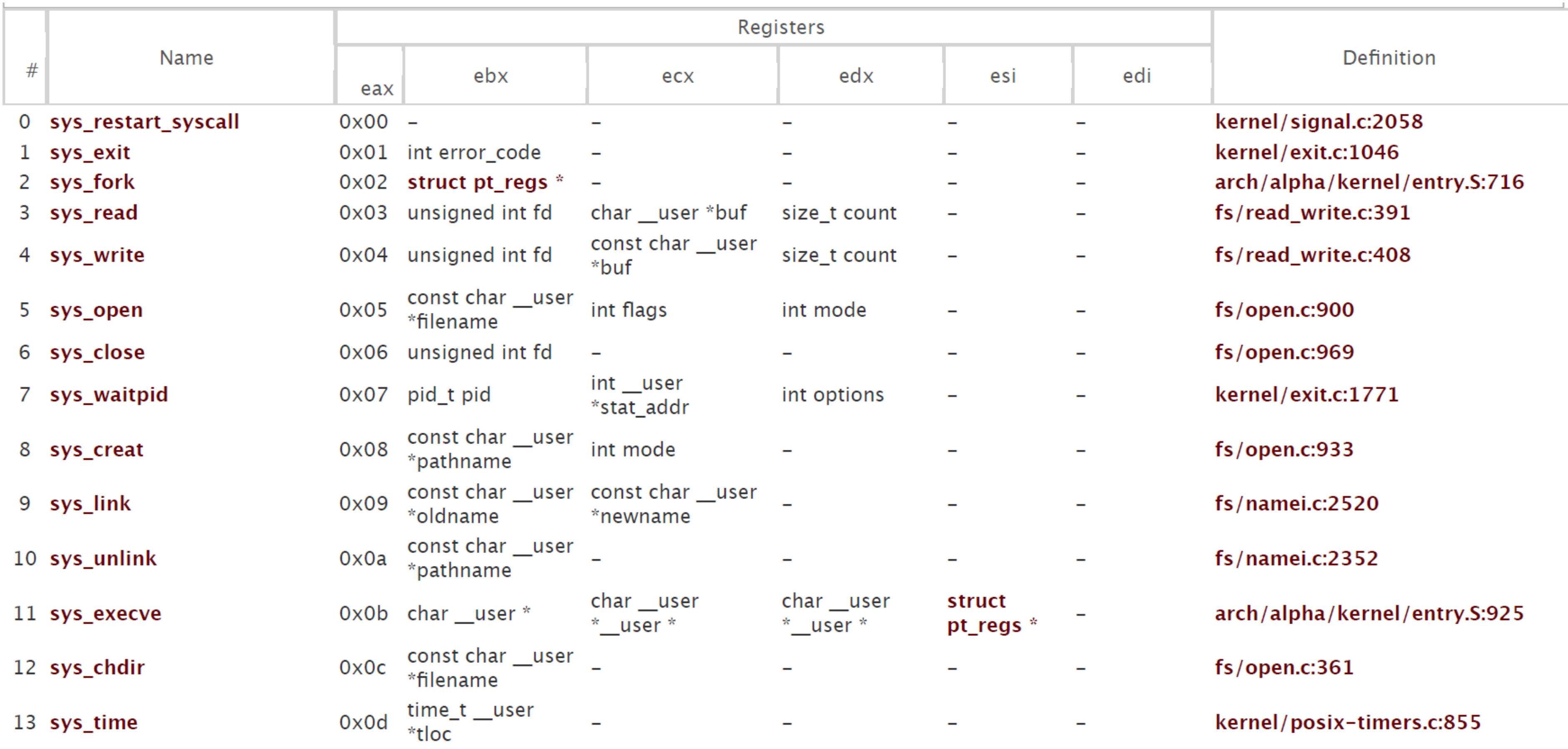
How do System Calls use Registers?

- Every system call uses some fields
- These fields must be saved in certain registers before calling the system call
- Therefore, to use a system call in assembly language, you must prepare the registers used by that system call
- Examples: To use

```
   read (int fd, char *buf, int size);          //  System call
         -  save 0x03 in eax
         -  save fd in ebx register             // 3 is file descriptor for standard input
         -  save *buf in ecx register
         - save size in edx register
         - call INT 80                          //  to execute the read system call

  write (int fd, char *buf, int size);           //  System call
         -  save 0x04 in eax                     // 4 is file descriptor for standard output
         -  save fd in ebx register
         -  save *buf in ecx register
          - save size in edx register
          - call INT 80                         //  to execute the write system call
```

Linux System Call Reference

- Linux system call reference shows the necessary registers for each system call



#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	**sys_restart_syscall**	0x00	–	–	–	–	–	**kernel/signal.c:2058**
1	**sys_exit**	0x01	int error_code	–	–	–	–	**kernel/exit.c:1046**
2	**sys_fork**	0x02	**struct pt_regs** *	–	–	–	–	**arch/alpha/kernel/entry.S:716**
3	**sys_read**	0x03	unsigned int fd	char __user *buf	size_t count	–	–	**fs/read_write.c:391**
4	**sys_write**	0x04	unsigned int fd	const char __user *buf	size_t count	–	–	**fs/read_write.c:408**
5	**sys_open**	0x05	const char __user *filename	int flags	int mode	–	–	**fs/open.c:900**
6	**sys_close**	0x06	unsigned int fd	–	–	–	–	**fs/open.c:969**
7	**sys_waitpid**	0x07	pid_t pid	int __user *stat_addr	int options	–	–	**kernel/exit.c:1771**
8	**sys_creat**	0x08	const char __user *pathname	int mode	–	–	–	**fs/open.c:933**
9	**sys_link**	0x09	const char __user *oldname	const char __user *newname	–	–	–	**fs/namei.c:2520**
10	**sys_unlink**	0x0a	const char __user *pathname	–	–	–	–	**fs/namei.c:2352**
11	**sys_execve**	0x0b	char __user *	char __user *__user *	char __user *__user *	**struct pt_regs** *	–	**arch/alpha/kernel/entry.S:925**
12	**sys_chdir**	0x0c	const char __user *filename	–	–	–	–	**fs/open.c:361**
13	**sys_time**	0x0d	time_t __user *tloc	–	–	–	–	**kernel/posix-timers.c:855**

Recall Hello World Program from Lecture-1

- We used system calls to print message “Hello world” in the first program in Lecture #1

```
SECTION .DATA
        msg:     db 'Hello world!’,10         ;  message to be displayed
        msgLen:  equ $-msg                    ;  Length of the message to be displayed

; Code goes in the text section
SECTION .TEXT
        GLOBAL _start

_start:
        mov eax,4                ;  use 'write' system call = 4;
        mov ebx,1                ; file descriptor 1 = STDOUT
        mov ecx, msg             ; string to write
        mov edx, msgLen          ; length of string to write
        int 80h                  ; call the kernel

        ; Terminate program
        mov eax,1                ; 'exit' system call
        mov ebx,0                ; exit with error code 0
        int 80h                  ; call the kernel
```

write (int fd, char *buf, int size);

- save 0x04 in eax
- save fd in ebx register
- save *buf in ecx register
- save size in edx register
- call INT 80

Recall Hello World Program from Lecture-10

- We used “printf” to print message “Hello world” in Lecture #10

```
 section .data   (segment .data)
 msg:     db ‘Display Hello world with printf !’,10, 0

section .text
extern printf

 global main
 main:
        push ebp
        mov ebp, esp

         ; Code goes in the text section
       push msg       ; push the memory address of the message to the stack
       call printf      ; printf will display the contents of that memory address

      mov esp, ebp
     pop epb
     ret
```

Use system calls to print string

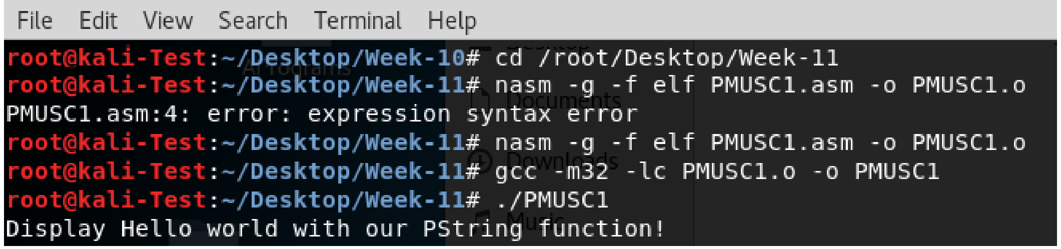
- In lecture#10, we used “printf” to print a message (string)
- In Lecture#1, we used system calls (registers) to print a message (string)
- In lecture#11, we need to use system calls (registers) to build our PString function
- Before using “PString” function, we have to prepare the required registers with prober values related to the message to be displayed
  - mov ecx, msg ; move the memory address of the first byte of the message in eax
  - mov edx, msgL ; move the message length in edx
  - call Pstring ; call Pstring from main function

- Then we need to define Pstring using system calls (e.g. registers)

```
PString:
        ; save register values of the called function
         pusha

        mov eax,4                    ; use 'write' system call = 4
        mov ebx,1                    ; file descriptor 1 = STDOUT
        int 80h                      ; call the kernel

         ; restore the old register values of the called function
         popa
         ret
```

Use system calls to print string

```
; PMUSC1.am     Print message using system calls

SECTION .data                    ; Data section
   msg:    db "Display Hello world with our PString function!",10, 0     ; message to be displayed
   msgL:  equ $-msg                    ;  Length of the message to be displayed

 SECTION .text
  global main
  main:
          push ebp
          mov ebp, esp

         ; Before calling PString function
         ; Assign the required registers with the proper values to display the  message

          mov ecx,msg             ; string to write  (address of the first byte)
          mov edx,msgL           ; length of the message to be displayed
          call PString

          ; exit the program and cleaning
          mov esp, ebp
          pop ebp
          ret

 PString:
                    ; save register values of the called fuction
                    pusha

                    mov eax,4                     ; use 'write' system call = 4
                    mov ebx,1                     ; file descriptor 1 = STDOUT
                    int 80h                       ; call the kernel

                   ; restore the old register values of the called function
                    popa
                    ret
```

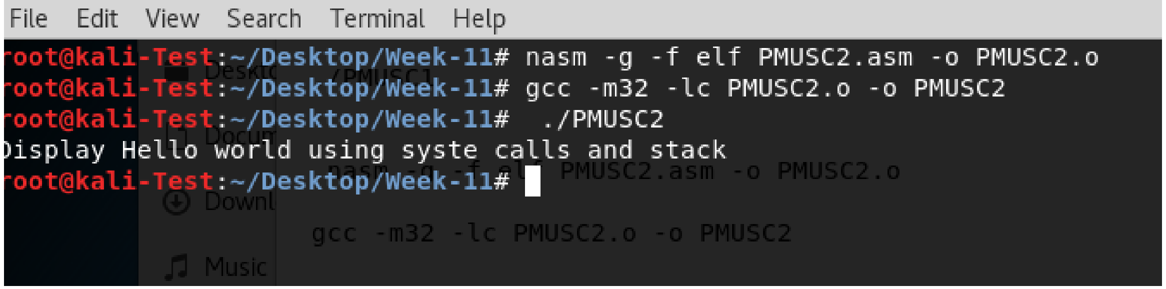


```
root@kali-Test:~/Desktop/Week-10# cd /root/Desktop/Week-11
root@kali-Test:~/Desktop/Week-11# nasm -g -f elf PMUSC1.asm -o PMUSC1.o
PMUSC1.asm:4: error: expression syntax error
root@kali-Test:~/Desktop/Week-11# nasm -g -f elf PMUSC1.asm -o PMUSC1.o
root@kali-Test:~/Desktop/Week-11# gcc -m32 -lc PMUSC1.o -o PMUSC1
root@kali-Test:~/Desktop/Week-11# ./PMUSC1
Display Hello world with our PString function!
```

Use the stack and system calls to print string

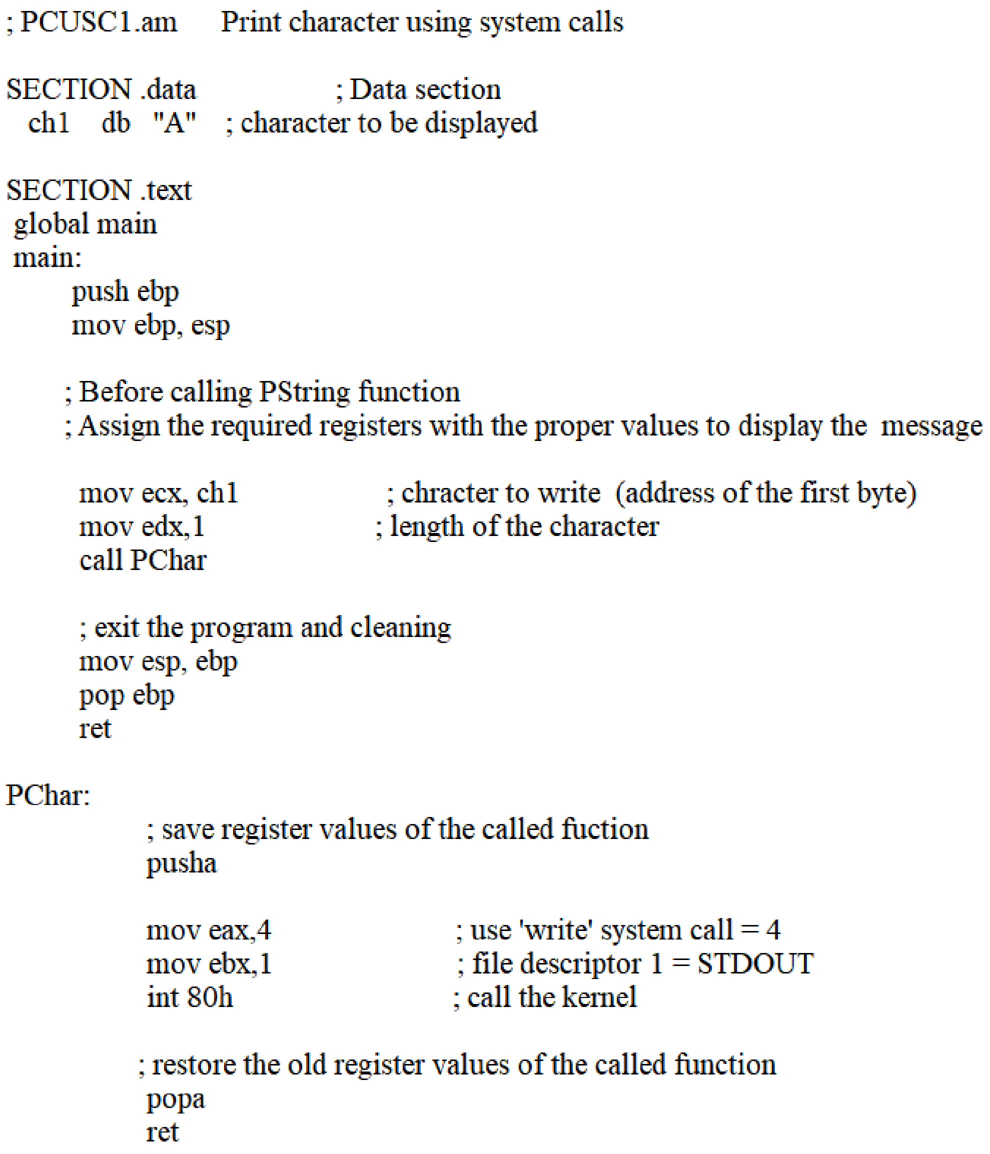
```
; PMUSC2.am      using satck and system calls to print message

SECTION .data                  ; Data section
   msg:     db "Display Hello world using syste calls and stack",10, 0

   msgL:  equ $-msg

SECTION .text

 global main
 main:
         push ebp
         mov ebp, esp

        ; Before calling PString function
        ; Assign the required registers with the proper values to display the  message


         push DWORD msgL
         push msg
         call PString2

        ; exit the program and cleaning
         mov esp, ebp
         pop ebp
         ret
```

```
PString2:
        push ebp
        mov ebp, esp

        mov eax,4
        mov ebx,1
        mov ecx, [ebp + 8]
        mov edx, [ebp + 12]
        int 80h

        mov esp, ebp
        pop ebp
        ret
```



```
oot@kali-Test:~/Desktop/Week-11# nasm -g -f elf PMUSC2.asm -o PMUSC2.o
oot@kali-Test:~/Desktop/Week-11# gcc -m32 -lc PMUSC2.o -o PMUSC2
oot@kali-Test:~/Desktop/Week-11#  ./PMUSC2
Display Hello world using syste calls and stack
oot@kali-Test:~/Desktop/Week-11# 
```

Use system calls to print character

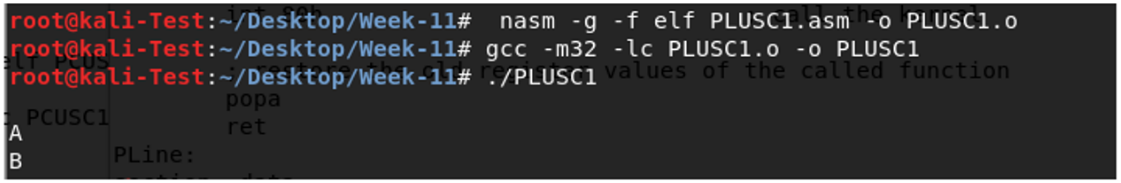


```
; PCUSC1.am     Print character using system calls

SECTION .data                    ; Data section
   ch1    db   "A"    ; character to be displayed

SECTION .text
 global main
 main:
         push ebp
         mov ebp, esp

        ; Before calling PString function
        ; Assign the required registers with the proper values to display the  message

          mov ecx, ch1                    ; chracter to write  (address of the first byte)
          mov edx,1                      ; length of the character
          call PChar

          ; exit the program and cleaning
          mov esp, ebp
          pop ebp
          ret

PChar:
                   ; save register values of the called fuction
                   pusha

                   mov eax,4                       ; use 'write' system call = 4
                   mov ebx,1                       ; file descriptor 1 = STDOUT
                   int 80h                        ; call the kernel

                  ; restore the old register values of the called function
                   popa
                   ret
```

Use system calls to Print New Line (local variable)

```
; PLUSC1.am     Print character using system calls
SECTION .data                   ; Data section
 ch1      db   "A"   ; character to be displayed
 ch2      db   "B"   ; character to be displayed

SECTION .text
 global main
 main:
         push ebp
         mov ebp, esp

         PLine                       ; print new Line
         mov ecx, ch1                ; print A
         mov edx,1                   ; length of the character
         call PChar

         PLine                        ; print new Line
         mov ecx, ch2                 ; print B
         mov edx,1                   ; length of the character
         call PChar

         ; exit the program and cleaning
         mov esp, ebp
         pop ebp
         ret
```

```
PChar:
              ; save register values of the called fuction
              pusha

              mov eax,4                       ; use 'write' system call = 4
              mov ebx,1                       ; file descriptor 1 = STDOUT
              int 80h                         ; call the kernel

             ; restore the old register values of the called function
              popa
              ret
PLine:
section .data
nl  db "", 10                ;  local variable
section .text
             ; save register values of the called fuction
              pusha
              mov ecx,nl
              mov edx,1
              mov eax,4                       ; use 'write' system call = 4
              mov ebx,1                       ; file descriptor 1 = STDOUT
              int 80h                         ; call the kernel

             ; restore the old register values of the called function
              popa
              ret
```

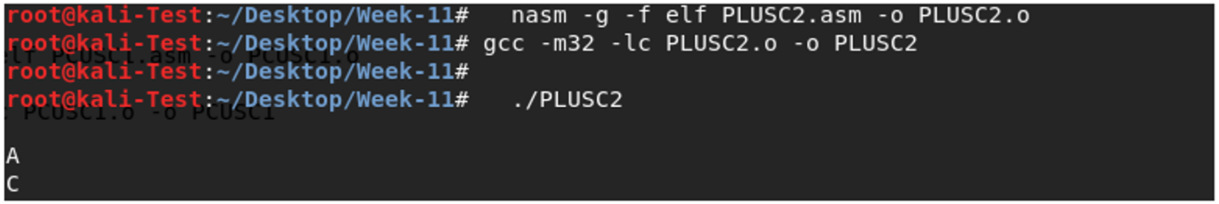


```
root@kali-Test:~/Desktop/Week-11#  nasm -g -f elf PLUSC1.asm -o PLUSC1.o
root@kali-Test:~/Desktop/Week-11# gcc -m32 -lc PLUSC1.o -o PLUSC1
root@kali-Test:~/Desktop/Week-11# ./PLUSC1

A
B
```

Use system calls to Print New Line (global variable)

```
; PMUSC2.am     Print message using system calls

SECTION .data                  ; Data section
 ch1     db  "A"   ; character to be displayed
 ch2     db  "C"   ; character to be displayed
 nl      db "", 10

SECTION .text
 global main
 main:
        push ebp
        mov ebp, esp

        PLine                    ; print new Line
        mov ecx, ch1             ; print A
        mov edx,1                ; length of the character
        call PChar

        PLine                     ; print new Line
        mov ecx, ch1              ; print B
        mov edx,1                ; length of the character
        call PChar

        ; exit the program and cleaning
        mov esp, ebp
        pop ebp
        ret
```

```
PChar:
              ; save register values of the called fuction
              pusha

              mov eax,4                    ; use 'write' system call = 4
              mov ebx,1                    ; file descriptor 1 = STDOU
              int 80h                      ; call the kernel

             ; restore the old register values of the called function
              popa
              ret
PLine:

           ; save register values of the called fuction
            pusha
            mov ecx,nl
            mov edx,1
            mov eax,4                    ; use 'write' system call = 4
            mov ebx,1                    ; file descriptor 1 = STDOUT
            int 80h                      ; call the kernel

           ; restore the old register values of the called function
            popa
            ret
```



```
root@kali-Test:~/Desktop/Week-11#   nasm -g -f elf PLUSC2.asm -o PLUSC2.o
root@kali-Test:~/Desktop/Week-11# gcc -m32 -lc PLUSC2.o -o PLUSC2
root@kali-Test:~/Desktop/Week-11#
root@kali-Test:~/Desktop/Week-11#   ./PLUSC2

A
C
```

Get Input From a User using System Calls

```
SECTION .data                    ; Data section
   msg1:     db "What is your name!",10, 0      ; message to be displayed
   msgL1:  equ $-msg1                              ;  Length of the message to be displayed
```

To print (write) the previous message using system call, we can use the following code

```
          ; print msg1
            mov eax,4                    ; use 'write' system call = 4
            mov ebx,1                    ; file descriptor 1 = STDOUT
            mov ecx,msg1                 ; string to write  (address of the first byte)
            mov edx,msgL1                ; length of the message to be displayed
            int 80h                      ; call the kernel
```

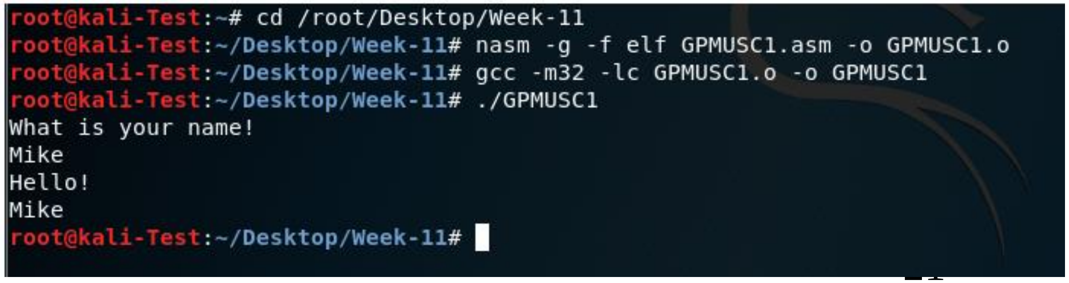
To get input (read) from a user using system call, we can use the following code

```
SECTION .bss
    msg2  resb 16                        ; reserve 16 bytes to hold the input message

          ; Get  msg2
            mov eax,3                    ; use ‘read' system call = 3
            mov ebx,0                    ; file descriptor 0 = STDINPUT  (Keyboard)
            mov ecx, msg2                ; string to write  (address of the first byte)
            mov edx,16                   ;  assume length of the input message = 16 bytes
            int 80h                      ; call the kernel
```

Example: Get and print user name using System Calls

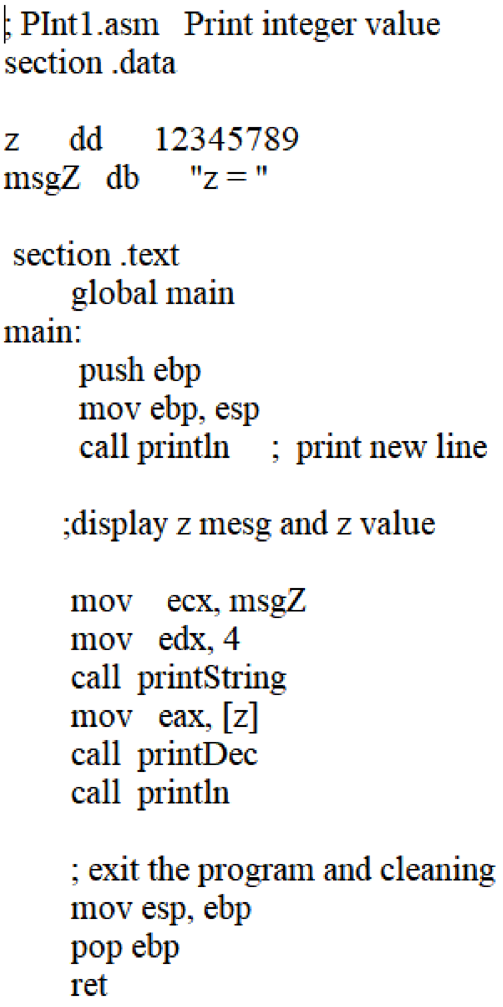
```
; GPMUSC1.am     Get input from from a user and print message using system calls

SECTION .data                  ; Data section
   msg1:    db "What is your name!",10, 0     ; message to be displayed
   msgL1:  equ $-msg1                      ;  Length of the message to be displayed

   msg2:    db "Hello!",10, 0     ; message to be displayed
   msgL2:  equ $-msg2                      ;  Length of the message to be displayed

SECTION .bss
    name resb 16     ; reserve 16 bytes to hold the input (user's name)

SECTION .text
 global main
 main:
        push ebp
        mov ebp, esp

        ; print msg1
         mov eax,4                         ; use 'write' system call = 4
         mov ebx,1                         ; file descriptor 1 = STDOUT
         mov ecx,msg1                      ; string to write  (address of the first byte)
         mov edx,msgL1                     ; length of the message to be displayed
         int 80h                           ; call the kernel
```

```
; Get name from user
  mov eax,3                     ; use 'read' system call = 3
  mov ebx,0                     ; file descriptor 0 = STDINPUT
  mov ecx, name                 ; string to read.   saved in memory (name)
  mov edx,16                    ; we reserve 16 bytes for the name
  int 80h                       ; call the kernel

  ; print msg2
   mov eax,4
   mov ebx,1
   mov ecx,msg2
   mov edx,msgL2
   int 80h

 ; print name
   mov eax,4
   mov ebx,1
   mov ecx,name
   mov edx,16
   int 80h


 ; exit the program and cleaning
    mov esp, ebp
    pop ebp
    ret
```

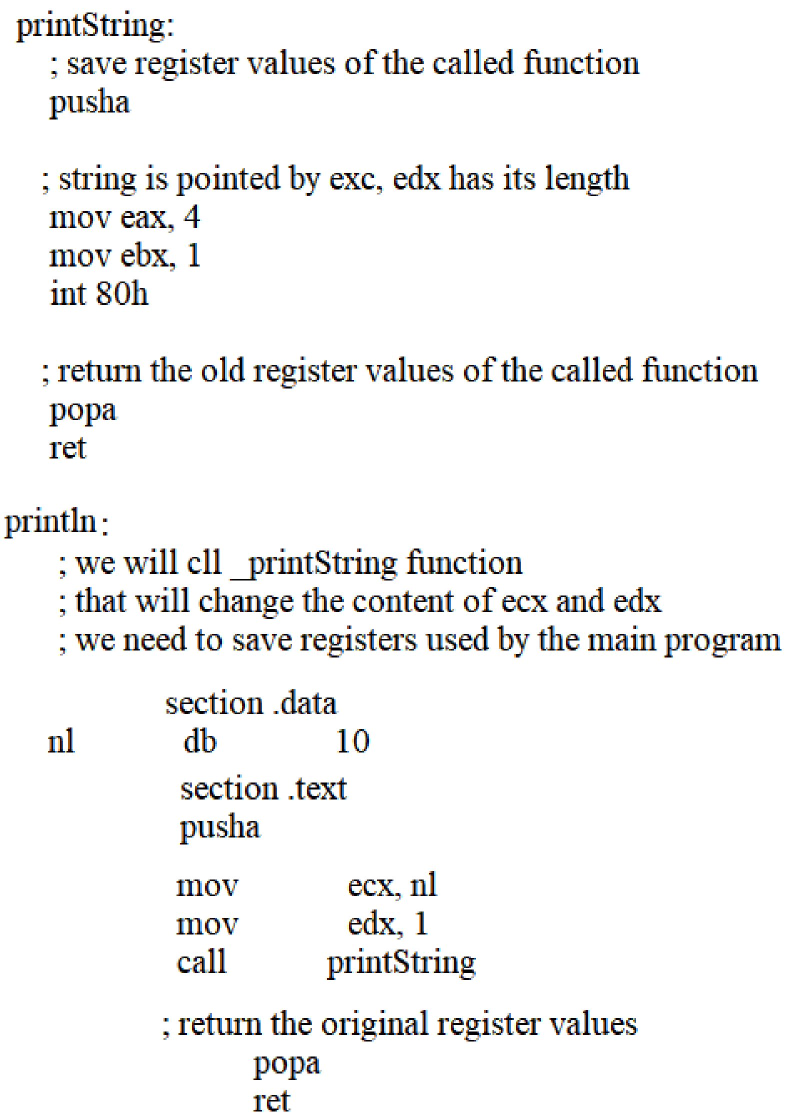


```
root@kali-Test:~# cd /root/Desktop/Week-11
root@kali-Test:~/Desktop/Week-11# nasm -g -f elf GPMUSC1.asm -o GPMUSC1.o
root@kali-Test:~/Desktop/Week-11# gcc -m32 -lc GPMUSC1.o -o GPMUSC1
root@kali-Test:~/Desktop/Week-11# ./GPMUSC1
What is your name!
Mike
Hello!
Mike
root@kali-Test:~/Desktop/Week-11# █
```

Print Integer value using system calls



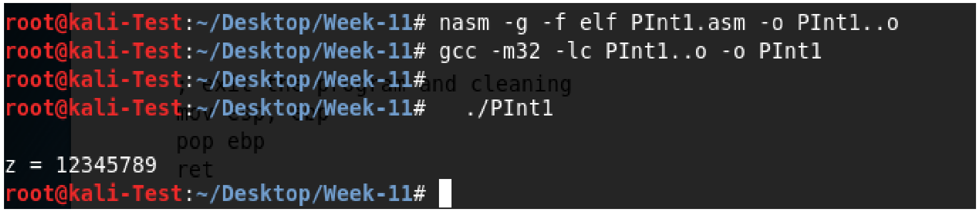
```
; PInt1.asm   Print integer value
section .data

z     dd      12345789
msgZ  db      "z = "

 section .text
        global main
main:
         push ebp
         mov ebp, esp
         call println     ;  print new line

       ;display z mesg and z value

        mov    ecx, msgZ
        mov   edx, 4
        call  printString
        mov   eax, [z]
        call  printDec
        call  println

        ; exit the program and cleaning
        mov esp, ebp
        pop ebp
        ret
```



```
printString:
    ; save register values of the called function
    pusha

   ; string is pointed by exc, edx has its length
    mov eax, 4
    mov ebx, 1
    int 80h

   ; return the old register values of the called function
    popa
    ret

println:
     ; we will cll _printString function
     ; that will change the content of ecx and edx
     ; we need to save registers used by the main program

               section .data
nl              db           10
                section .text
                pusha

                mov          ecx, nl
                mov          edx, 1
                call       printString

               ; return the original register values
                       popa
                       ret
```

Print Integer value using system calls

```
printDec:
;;; saves all the registers so that they are not changed by the function
;;; We build the function to handle the dword size (4 bytes)

          section      .bss
 decstr     resb        10     ; 10 digits number for 32 bits
 ct1        resd        1      ; to keep track of the size of the dec-string

          section .text
          pusha                      ; save all registers

          mov         dword[ct1],0    ; assume initially 0
          mov         edi,decstr      ; edi points to dec-string in memory
          add         edi,9           ; moved to the last element of string
          xor         edx,edx         ; clear edx for 64-bit division
whileNotZero:
          mov         ebx,10          ; get ready to divide by 10
          div         ebx             ; divide by 10
          add         edx,'0'         ; converts to ascii char
          mov         byte[edi],dl    ; put it in sring
          dec         edi             ; mov to next char in string
          inc         dword[ct1]      ; increment char counter
          xor         edx,edx         ; clear edx
          cmp         eax,0           ; is remainder of division 0?
          jne         whileNotZero    ; no, keep on looping

          inc         edi             ; conversion, finish, bring edi
          mov         ecx, edi        ; back to beg of string. make ecx
          mov         edx, [ct1]      ; point to it, and edx gets # chars
          mov         eax, 4          ; and print! to the studardout
          mov         ebx, 1
          int         0x80

          popa                       ; restore all registers
          ret
```



```
root@kali-Test:~/Desktop/Week-11# nasm -g -f elf PInt1.asm -o PInt1..o
root@kali-Test:~/Desktop/Week-11# gcc -m32 -lc PInt1..o -o PInt1
root@kali-Test:~/Desktop/Week-11#
root@kali-Test:~/Desktop/Week-11#   ./PInt1

z = 12345789
root@kali-Test:~/Desktop/Week-11#
```

HW-Q1: Draw the flow chart and explain how “printDec” work?

Assume

```
z1 dd 232

z2  dd 434

z3 dd 0

Pstring  (z1)
Pstring (z2)

 mov eax, z1
 add eax, z2
 mov z3, eax
```

PrintDEC (z3)	Pstring (z3)
???	???

Putting It All Together

You should know:

- **Linux kernel structure**
- **User mode vs kernel mode**
- **What is a system call?**
- **Difference between function and system calls**
- **How do System Calls use Registers?**
- **Use system calls to**
  - Print string
  - Print character
  - Print integer value
  - Provide input to the program

Questions?

Coming Next Week
Arras

Week 11 Assignments

- **Learning Materials**

  1- Week 11 Presentation

  2- Read Pages 452-462 & 487-493 (Ch.12: Duntermann, Jeff. Assembly Language Step by Step, Programming with Linux, PP: 201-211

- **Assignment**

  1- Complete "Lab 12" by coming Sunday 11:59 PM.