COSC 458-647
Application Software Security

Today

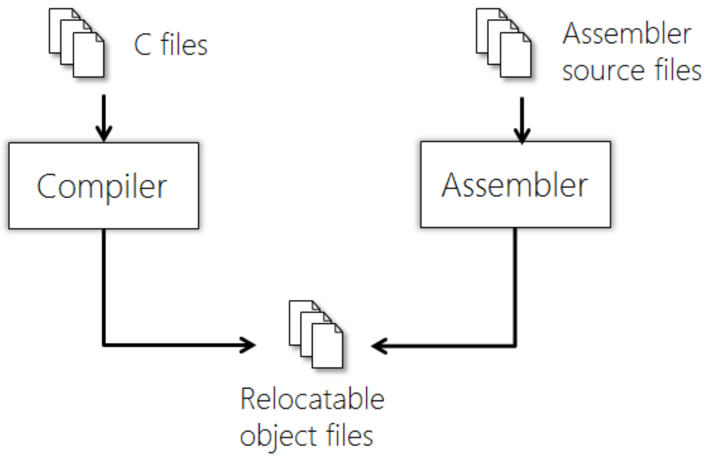
- Memory layout of C program
  - Code, Data, BSS, Stack and Heap Segments
- Assembly language basis
  - Introduction
  - First program on Linux

Memory layout of C program

First look

- *Compiler driver*
  - Invokes the language preprocessor, compiler, assembler, and linker, as needed on behalf of the user
  - Can generate *three types of object files* depending upon the options supplied to the compiler driver.
- Technically an object file is *a sequence of bytes stored* on disk in a file

Object files



- Relocatable object files
  - Are static library files
  - Static linkers (such as ld program) take collection of relocatable object files, command line arguments & generate a fully linked executable object file that can be loaded into memory and run.
  - Contain binary code and data in a form that can be combined with other relocatable object files at compile time to create an executable object file
- Executable object files
  - Contain binary code and data in a form that can be copied directly into memory and executed
- Shared object files
  - Special type of relocatable object files
  - Are loaded into memory and linked dynamically, at either load time or run time.

Object files (cont'd)

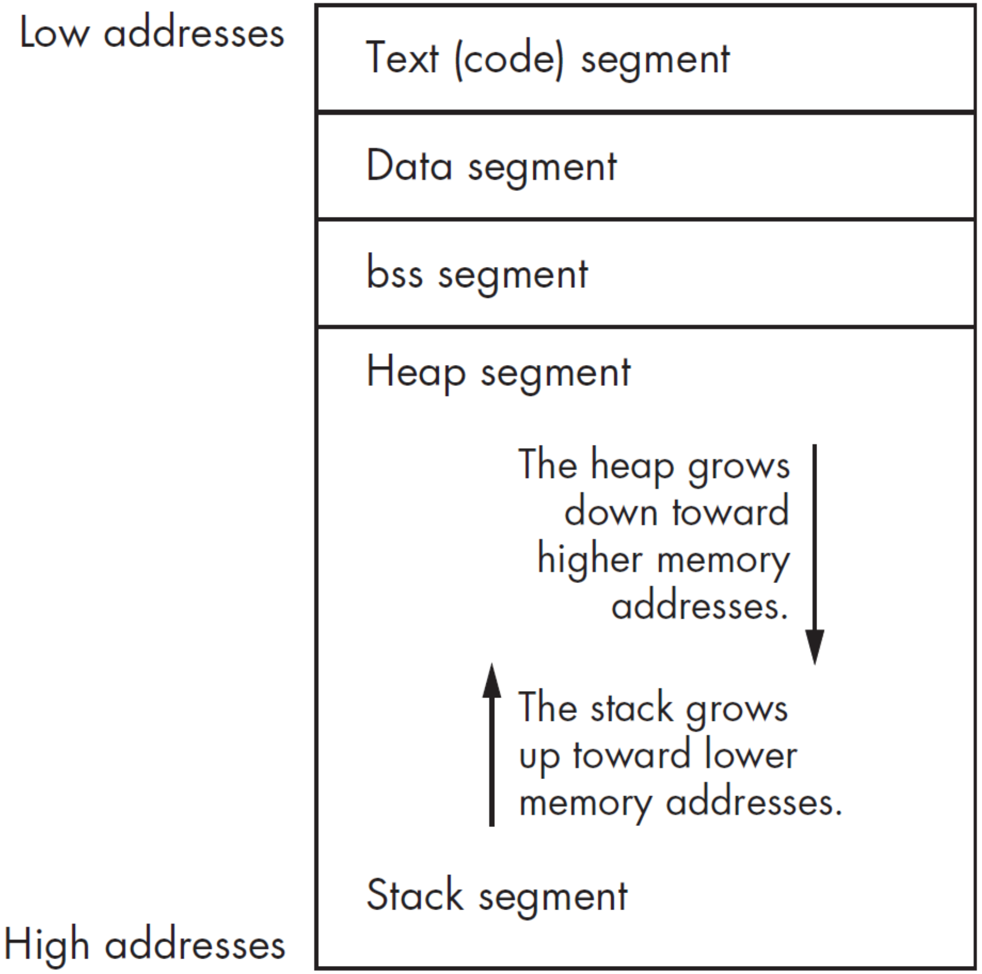
- Object files usually have a specific format
- This format may vary from system to system.
- Some most prevalent formats are
  - **.coff** (Common Object File Format),
  - **.pe** (Portable Executable),
  - **elf** (Executable and Linkable Format).

Segments of a compiled C program

1. Code or Text Segment (.text)

2. Data Segment

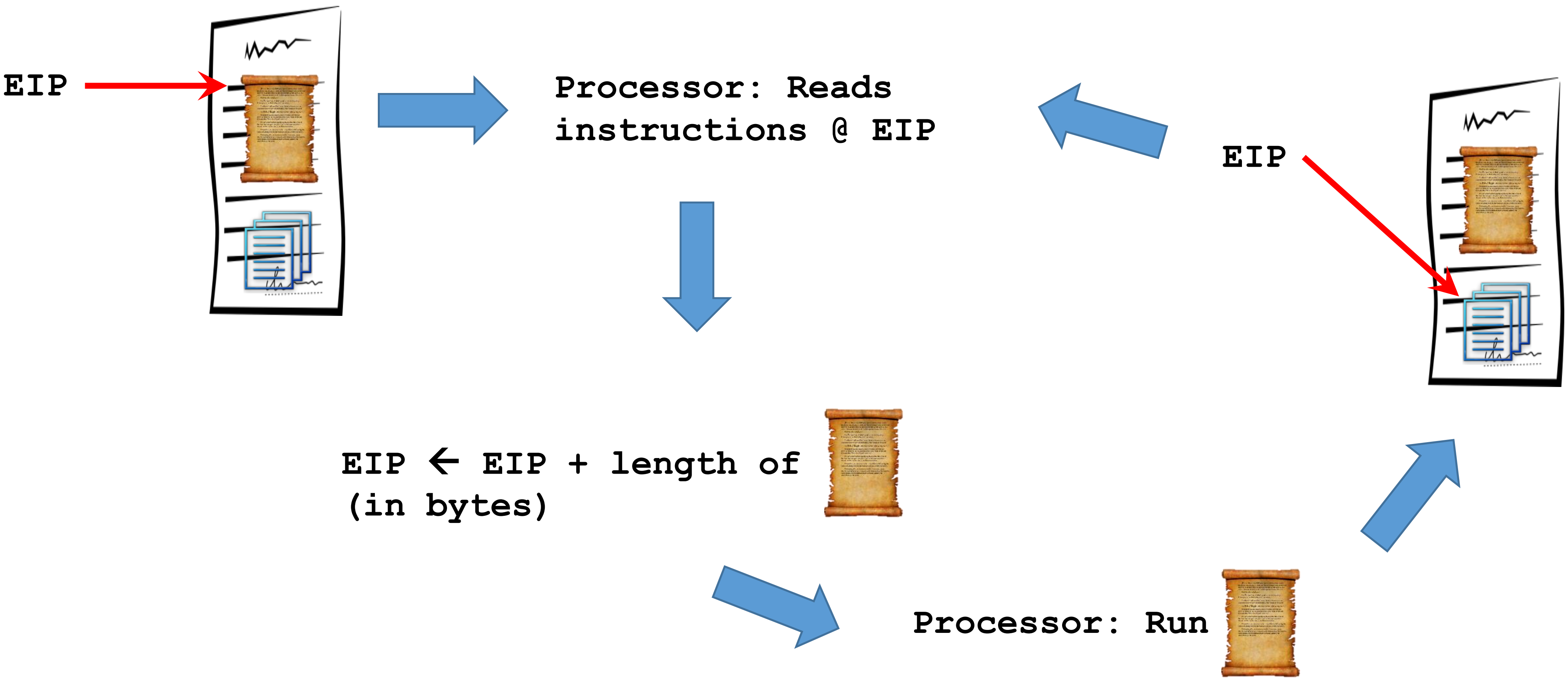
   - Initialized Data Segments (.data)
   - Uninitialized Data Segments (.bss)
   - Stack Segment
   - Heap Segment



Code segment

- Contains *machine code of the compiled program*
- Is often read-only
    - Prevents the program from being accidentally modified
        - Alert user and terminate the program if there is a Write attempt
    - Can be shared among different copies of the program
        - Allow multiple executions of the program at the same time
- Execution of instructions in the Code Segment is NON-LINEAR
    - Due to the high level of control structures & functions
    - Due to branching, i.e. jump, call, etc
- Has a fixed size after the program is compiled

Code segment (cont’d)



Initialized data segment

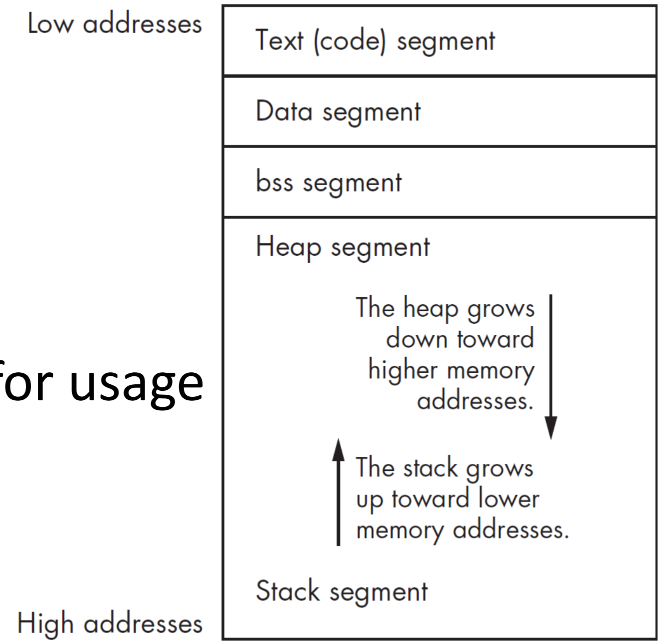
- `.data` segment
- Stores all global, static, constant, and external variables (declared with extern keyword) that are initialized beforehand.
  - `int x = 2; double y = 3; extern z = 10;`
- Is ***writable***
- Has a fixed size

Uninitialized data segment

- `.bss` segment
  - “Block Storage Start" instruction from the IBM 704 assembly language (circa 1957)
- Stores all uninitialized global, static, and external variables which by default initialized to zero.
  - `int x; double z, extern z;`
- Is Writable
- Has a fixed size
- This section occupies no actual space in the object file; it is merely a place holder
  - Object file formats distinguish between initialized and uninitialized variables for space efficiency;
  - Uninitialized variables do not have to occupy any actual disk space in the object file.

Heap Segment

- A segment of memory a programmer can directly control
- Blocks of memory in this segment can be used/allocated for *whatever the programmer needs*
- Does not have a fixed size (Why?)
  - Memory allocation/deallocation
  - Who manage memory allocation/deallocation?
- Is dynamic
  - Grows and shrinks depending on how much memory is reserved for usage

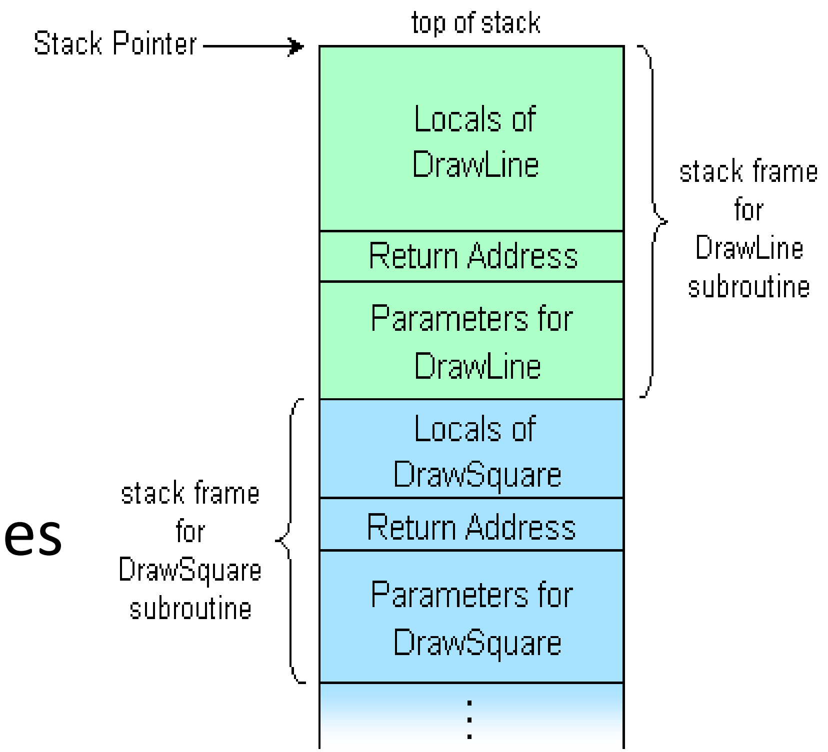


Stack Segment

- Is used as a temporary scratch pad to store local function variables and context during function call
  - This is where the debugger's trace back commands look at
- Does not have a fixed size
- When the program calls a functions, that function will have its own set of passed variables
  - The function code is, still, at a different location in the code segment
- Stack is used to remember
  - All passed variables
  - The location EIP should return to (return address)
  - All local variables used by the function

Stack segment

- Stack is **FILO**
  - Push & Pop
- Stack grows from **HIGHER → lower** memory addresses
- Is a stack data structure containing stack frames
- When a function is called, several things are put into a stack frame
  - EBP register (or frame pointer FP, or local base LB pointer)
  - Parameters to the function
  - Its local variables
  - Saved Frame Pointer (to restore EBP)
  - Return address pointer (to restore EIP)

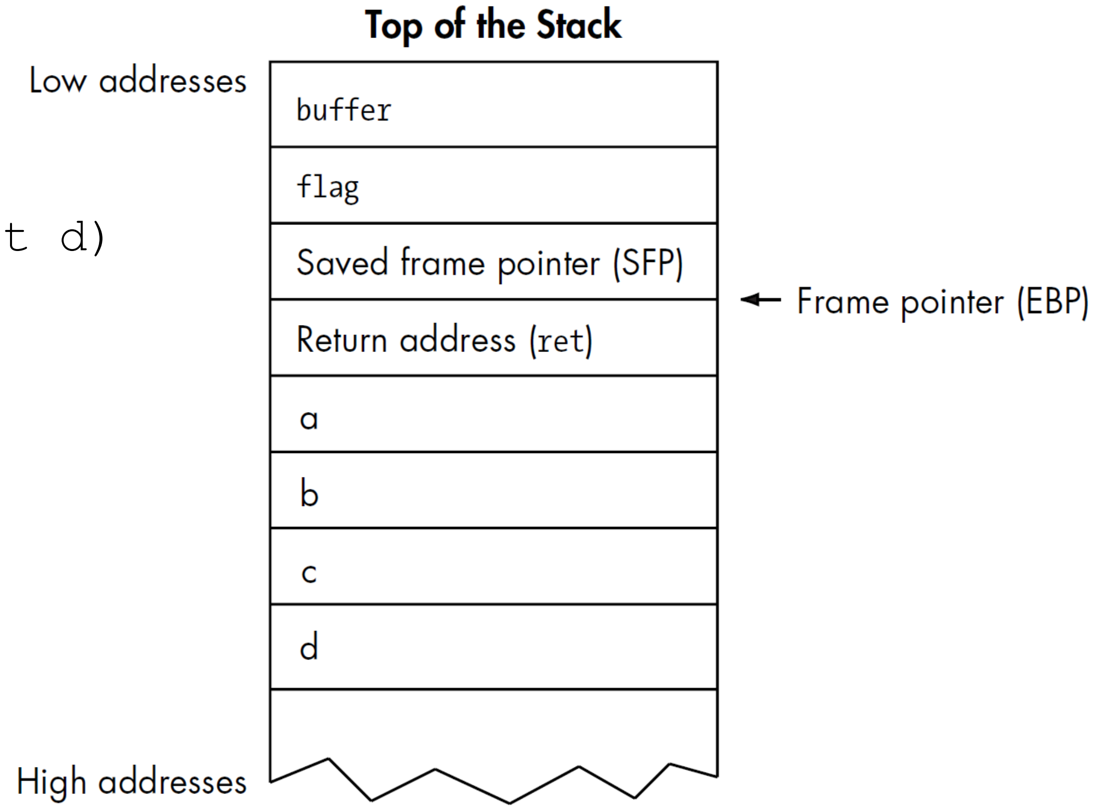


Stack example

stack_example.c

```
void test_function(int a, int b, int c, int d)
{
     int flag;
     char buffer[10];
     flag = 31337;
     buffer[0] = 'A';
}

int main() {
     test_function(1, 2, 3, 4);
}
```



Basic Assembly Programming

We will be using the nasm assembler
other assemblers: masm, as, gas, etc

Writing a useful program with **nasm**

- References
- Platform: Linux
- What do we need
  - GCC
  - NASM
  - Text editor
- Write the first program
  - helloWorld_C_ver1.asm
  - Improve the above program: helloWorld_C_ver2.asm

At a glance

- There are in general three types of usages for assembly language
  - In use with a borrowed C function
  - In use with NO borrowed C functions (pure Assembly Language)
  - In use as a subfunction of a C program (for speed).

In use with a C function: helloWorld_C_ver1.asm

```
;; These externals are all from the standard C library:
extern printf ; Notify linker that we're calling printf from C

section .data ; Section containing initialised data
  msg   db   "Hello, world!", 0xa, 0 ; our dear string

section .text
  global     main   ; Required so linker can find entry point for C

main:
      push   msg            ; push the message on top of the stack
      call   printf         ; call printf function from C
      add    esp, byte 4    ; clean up stack after printf call
                            ; what if we add less/more than 4 bytes?

return:
      mov    eax, 0
      ret              ; Return control to Linux
```

Compiling …

1. Build helloWorld.o from helloWorld.asm

```
nasm -f elf helloWorld.asm
```

-f Option: Specifying the Output File Format

2. Build binary file helloWorld from helloWorld.o

```
gcc -o helloWorld_C_ver1 helloWorld.o
```

Name of the executable file

3. Run the file

```
./helloWorld_C_ver1
```

In use with a C function: helloWorld_C_ver2.asm

```
1. nasm -f elf helloWorld_C_ver2.asm
2. gcc -o helloWorld_C_ver2 helloWorld_C_ver2.o
```

```
extern printf
section .data
            msg db "Hello, world!", 0xa, 0    ;our dear string
section .text
            global main
main:
  1.        push   ebp            ; save ebp by pushing  it on top of the stack
  2.        mov    ebp, esp       ; ebp and esp are pointing to the same place
  3.        push   dword msg      ; push msg on top of the stack
  4.        call   printf         ; call printf function
  5.        add    esp, byte 4    ; move esp down by 4 bytes.
                                  ; Is line 5 necessary in our program?
return:
  6.        mov esp, ebp          ; destroy stack frame before returning
  7.        pop ebp               ; restore saved ebp
  8.        mov eax, 0            ; call return function
  9.        ret
```

Notes

- In msg db “Hello World”, 0xa, 0: 0xa (or 0x10, or 10) stands for a new line, 0 stand for NULL (string termination)
- Assembly language is not CaSe SeNsitive
- The global label `main` is used in conjunction with C function only
- Initialized and uninitialized data and code are defined within sections .data, .bss and .text

Pure Assembly Language: helloWorld_ASM

- Assembly language usually uses interrupts to execute a system call
- Use libc wrappers
  - Ex: read, write etc;
  - Works indirectly with assembly code to execute system calls;
- Directly use assembly code
  - System call via software interrupts, for example `int 0x80`;

In Linux, a shell code uses **`int 0x80`** to raise system calls.

Executing a system call

- The *specific system call* is loaded into EAX.
- Arguments to the system call function are placed in other registers.

  EBX, ECX, EDX
- The Instruction **`int 0x80`** is executed;
- The CPU switches to Kernel mode;
- The system call function is executed.

```
main(){
    exit(0);
}
```

eax	Name	ebx	ecx	edx	esx
1	sys_exit	int	-	-	-
2	sys_fork	struct pt_regs	-	-	-
3	sys_read	unsigned int	char *	size_t	-
4	sys_write	unsigned int	const char *	size_t	-
5	sys_open	const char *	int	int	-
6	sys_close	unsigned int	-	-	-

Write a shell code for exit()

- The shell code should do the following:
  - Store the value of 0 into EBX;
  - Store the value of 1 into EAX;
  - Execute int 0x80 instruction

- First, we write ASM codes (exit.asm) as follows:

```
Section .text
        global _start
_start:
        mov ebx, 0
        mov eax, 1
        int 0x80
```

exit(0) example

- nasm -f elf exit.asm
- ld -o exit_1 exit.o
- objdump -d exit_1

- Disassembly of section .text:

```
08048060 <_start>:
8048060:     bb 00 00 00 00      mov    ebx, 0x0
8048065:     b8 01 00 00 00      mov    eax, 0x1
804806a:     cd 80               int    0x80
```

Red words can be used as the shell code.

Pure Assembly Language: helloWorld_ASM

```
section .data
      msg db "hello world ASM", 0xa, 0
      len equ $-msg                          ; the length of the string

section .text
      global _start  ; *** pure ASM uses _start, not main ***

_start:
; "write" syscall = 4 (eax), stdout = 1 (ebx), string addr (ecx), length (edx)
       mov    eax, 0x04     ; move syscall ID #4 to eax
       mov    ebx, 0x01     ; move stdout ID #1 to ebx
       mov    ecx, msg      ; move the msg pointer to ecx
       mov    edx, len      ; move the length to edx
       int    0x80          ; call interrupt 80

; "exit" syscall = 0 (eax), success = 0 (ebx)
       mov    eax, 0x01
       mov    ebx, 0x00
       int    0x80
```

Notes

- This version of helloWorld program works by raising interrupt 0x80 in order to call a system function
- When an interrupt is raised, the system function will be identified by the ID in eax register, an consequently by ebx, ecx and edx, etc.
- Example

eax	Name	Source	ebx	ecx	edx	esx	edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-

In use as a subfunction of a C program

```c
/*
 * callmaxofthree.c
 *
 * Illustrates how to call the maxofthree function we wrote in assembly
 * language.
 */

#include <stdio.h>

int maxofthree(int, int, int);

int main() {
    printf("%d\n", maxofthree(1, -4, -7));
    printf("%d\n", maxofthree(2, -6, 1));
    printf("%d\n", maxofthree(2, 3, 1));
    printf("%d\n", maxofthree(-2, 4, 3));
    printf("%d\n", maxofthree(2, -6, 5));
    printf("%d\n", maxofthree(2, 4, 6));
    return 0;
}
```

```nasm
; ----------------------------------------------------------------------------
; maxofthree.asm
;
; NASM implementation of a function that returns the maximum value of its
; three integer parameters.  The function has prototype:
;
;   int maxofthree(int x, int y, int z)
;
; Note that only eax, ecx, and edx were used so no registers had to be saved
; and restored.
; ----------------------------------------------------------------------------

        global  maxofthree

        section .text
maxofthree:
        mov     eax, [esp+4]
        mov     ecx, [esp+8]
        mov     edx, [esp+12]

        cmp     eax, ecx
        cmovl   eax, ecx
        cmp     eax, edx
        cmovl   eax, edx
        ret
```

Assembly files: Five simple things

- Labels
  - Variables are declared as labels pointing to specific memory locations
  - Labels mark the start of subroutines or locations to jump to in your code
- Instructions – cause machine code to be generated
- Directives – affect the operation of the assembler
- Comments
- Data

Comments, comments, comments!

```
; Comments are denoted by semi-colons.
; Please comment your code thoroughly.
; It helps me figure out what you were doing
; It also helps you figure out what you were
;   doing when you look back at code you
;   wrote more than two minutes ago.

; everything from the semi-colon to the end
; of the line is ignored.
```

Labels

```
; Labels are local to your file/module
; unless you direct otherwise, the colon
; identifies a label (an address!)

MyLabel:

; to make it global we say

    global          MyLabel

; And now the linker will see it
```

Example with simple instructions

```
var1    dd      0FFh
str1    db      “my dog has fleas”,10
var2    dd      0

; Here are some simple instructions
        mov      eax, [var1] ; notice the brackets
        mov      edx, str1   ; notice the not brackets
        call     dspmsg
        jmp      done
        mov      ebx, [var2] ; this will never happen
        cmp      ecx, 0x8 ; this also will never happen

done:   nop
```

Directives

- A directive is an artifact of the assembler not the CPU.
- They are generally used to either instruct the assembler to do something or inform the assembler of something.
- *They are not translated into machine code.*
- Common uses of directives are:

  + Define constants
  + Define memory to store data into
  + Group memory into segments
  + Conditionally include source code
  + Include other files

- NASM's preprocessor directives start with a % instead of a # as in C.

The **equ** directive

- The equ directive can be used to define a symbol. Symbols are named constants that can be used in the assembly program. The format is

```
symbol equ value
```

- Example:

```
msg db  “hello world ASM”, 0xa, 0
len equ $-msg
```

- Symbol can not be redefined later

The `%define` directive

- Similar to C’s #define directive. It is mostly used to define constant macros just in C

```
%define SIZE 100
move      eax, SIZE
```

- Macros are more flexible than symbols in two ways
  - They can be redefined
  - Can be more than simple constant numbers

Data directives

- Used in data segments to define room for memory.
- There are two ways memory can be reseved.
  - 1. Only defines room for data
  - 2. Defines room for data with an initial value

Unit	Letter
byte	B
word	W
double word	D
quad word	Q
ten bytes	T

Table 1.3: Letters for RESX and DX Directives

- Common structures

1. `data_label resX #of_units`

2. `data_label dX init_value`

Data Directive

```
L1      db      0          ; byte labeled L1 with initial value 0
L2      dw      1000       ; word labeled L2 with initial value 1000
L3      db      110101b    ; byte initialized to binary 110101 (53 in decimal)
L4      db      12h        ; byte initialized to hex 12 (18 in decimal)
L5      db      17o        ; byte initialized to octal 17 (15 in decimal)
L6      dd      1A92h      ; double word initialized to hex 1A92
L7      resb    1          ; 1 uninitialized byte
L8      db      "A"        ; byte initialized to ASCII code for A (65)
```

- Double and single quote are treated the same.
- Consecutive data definitions are stores sequentially in the memory
  - L2 is stored right after L1 in the memory

Data directives

- Sequences of memory may also be defined.

```
L9    db    0, 1, 2, 3                ; defines 4 bytes
L10   db    "w", "o", "r", 'd', 0     ; defines a C string = "word"
L11   db    'word', 0                 ; same as L10
```

- For large sequence, **times** is normaly used

```
L12   times 100 db 0                  ; equivalent to 100 (db 0)'s
L13   resw  100                       ; reserves room for 100 words
```

Declaring static variables

```
.DATA
var   DB   64      ; Declare a byte, referred to as location var, containing the value 64.

var2  DB   ?       ; Declare an uninitialized byte, referred to as location var2
      DB   10      ; Declare a byte with no label, containing the value 10. Its location is var2 + 1.

X     DW   ?       ; Declare a 2-byte uninitialized value, referred to as location X.

Y     DD   30000   ; Declare a 4-byte value, referred to as location Y, initialized to 30000.
```

Declaring array

- `bytes DB 10 DUP(?)` ; Declare 10 uninitialized bytes starting at the address “bytes”.
- `arr DD 100 DUP(0)` ; Declare 100 4 bytes words, all initialized to 0,
  ; starting at memory location “arr”.
- `str DB 'hello',0` ; Declare 5 bytes starting at the address “str”
  ; initialized to the ASCII character values for
  ; the characters ‘h’, ‘e’, ‘l’, ‘l’, ‘o’, and
  ; ‘\0’(NULL), respectively.

Labels usages

- Labels can be used to refer to data in code.
- There are two ways that a label can be used
  - If a plain label is used, it is interpreted as ***the address (or offset)*** of the data
  - If a label is placed inside square brackets ([ ]), it is interpreted as the ***data at the address***
- Label should be thought of as **pointers**.

```
mov     al, [L1]       ; copy byte at L1 into AL
mov     eax, L1        ; EAX = address of byte at L1
mov     [L1], ah       ; copy AH into byte at L1
mov     eax, [L6]      ; copy double word at L6 into EAX
add     eax, [L6]      ; EAX = EAX + double word at L6
add     [L6], eax      ; double word at L6 += EAX
mov     al, [L6]       ; copy first byte of double word at L6 into AL
```

Example