Computer Architecture and System Programming Laboratory

TA Session 5

Listing file

LOOP, EQU

Local labels

System calls

Producing a listing file: > nasm -f elf sample.s -l sample.lst

```
 1
 2                                  section .data
 3 00000000 78563412                        numeric:    DD 0x12345678
 4 00000004 616263                          string:         DB 'abc'
 5 00000007 00000000                        answer:         DD      0
 6
 7                                  section .text
 8                                          global  _start
 9
10                                  _start:
11
12 00000000 60                             pushad
13 00000001 6A02                           push dword 2
14 00000003 6A01                           push dword 1
15 00000005 E815000000                     CALL myFunc
16                                  returnAddress:
17 0000000A A3[07000000]                   mov [answer], eax
18 0000000F 83C408                         add esp, 8
19 00000012 61                             popad
20 00000013 BB00000000                     mov ebx,0
21 00000018 B801000000                     mov eax,1
22 0000001D CD80                           int 0x80
23
24                                  myFunc:
25 0000001F 55                             push    ebp
26 00000020 89E5                           mov     ebp, esp
27 00000022 8B4508                         mov eax, dword [ebp+8]
28 00000025 8B5D0C                         mov ebx, dword [ebp+12]
29                                  myFunc_code:
30 00000028 01D8                           add eax, ebx
31                                  returnFrom_myFunc:
32 0000002A 89EC                           mov     esp, ebp
33 0000002C 5D                             pop     dword ebp
34 0000002D C3                             ret
```

first column is **line number** in the listing file

second column is **relative address** of where code will be placed in memory
- each section starts at relative address 0

third column is **compiled code**

forth column is **original code**

'**CALL myFunc**' is compiled to **opcode E8** followed by a **4-byte target address**, relative to the next instruction after the call.
→ address of myFunc label = 0x1F
→ address of the next instruction after the call (i.e. 'mov [answer], eax') is 0xA
→ 0x1F-0xA=0x15, and we get exactly the binary code written here '**E815000000**'

0x15 is how many bytes EIP should jump forward

```
Breakpoint 1, 0x08048080 in _start ()
(gdb) x /1xg $eip
0x8048080 <_start>:     0x0015e8016a026a60
(gdb)
0x8048088 <_start+8>:   0x83080490b7a30000
(gdb)
0x8048090 <returnAddress+6>:    0x00000000bb6108c4
(gdb)
0x8048098 <returnAddress+14>:   0x5580cd00000001b8
(gdb)
0x80480a0 <myFunc+1>:   0x0c5d8b08458be589
(gdb)
0x80480a8 <myFunc_code>:        0x0000c35dec89d801
```

executable

Advanced Instructions - LOOP

LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ – loop with counter (CX or ECX)

```
mov ecx, 3
myLoop:
        …
        loop myLoop, ecx        ; decrement ecx
                                ; if ecx != 0, jump to myLoop
```

LOOPE ≡ LOOPZ: jumps if the counter ≠ 0 **and** ZF = 1

LOOPNE ≡ LOOPNZ: jumps if the counter ≠ 0 **and** ZF = 0

Note: LOOP instruction does not set any flags

Note: if a counter is not specified explicitly, BITS setting dictates which is used

BITS directive specifies whether NASM should generate code designed to run on a processor operating in 16-bit mode, 32-bit mode, or 64-bit mode. The syntax is BITS 16, BITS 32, or BITS 64.

Local Labels Definition

A label beginning with a single period (.) is treated as a **local label**, which means that it is associated with the previous non-local label.

Example:

```
func1:
   …
 .loop:
     dec eax
     jne .loop

func2:
   …
 .loop:
     dec eax
     jne .loop
```

(this is indeed func1.loop)

(this is indeed func2.loop)

first character can be: letter, _, ?, and .

valid characters in labels are: letters, numbers, _, $, #, @, ~, ., and ?

Any local label can be accessed from anywhere, using the full, qualified path through the relevant non-local label

Each JNE instruction jumps to the closest .loop, because the two definitions of .loop are kept separate.

Linux System Calls

A system call is explicit request to the kernel, made via a software interrupt.

- System calls are low level functions, which represent the *interface* the kernel presents to user applications
- In Linux all low-level I/O is done by reading and writing files, regardless of what particular peripheral **device** is being accessed - a tape, a socket, even your terminal, they **are all *files***.
- File is referenced by an integer file descriptor

Linux System Call format

- Put the system call number in `EAX` register
- Set up the arguments to the system call
  - The first argument goes in EBX, the second in ECX, then EDX, ESI, EDI, and finally EBP. If more then 6 arguments needed (not likely), the EBX register must contain the memory location where the list of arguments is stored.
- Call the relevant interrupt (for Linux it is 0x80)
- The result is usually returned in `EAX`

sys_read – read from a file

- system call number (in EAX): **3**
- arguments:
  - EBX: file descriptor (to read from it)
  - ECX: pointer to input buffer (to keep a read data into it)
  - EDX: maximal number of bytes to receive (maximal buffer size)
- return value (in EAX):
  - number of bytes received
  - On errors: negative number

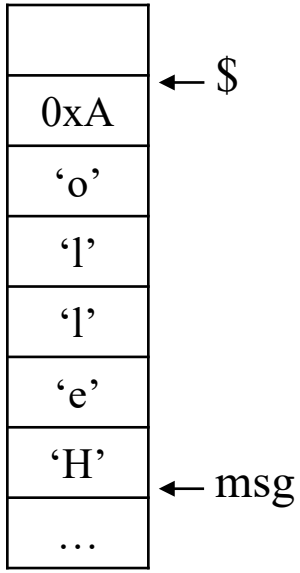
```
section     .bss
   buffer:   resb    1

section     .text
  global _start
_start:

   mov eax, 3                ; system call number (sys_read)
   mov ebx, 0                ; file descriptor (stdin)
   mov ecx, buffer           ; buffer to keep the read data
   mov edx, 1                ; bytes to read
   int 0x80                   ;call kernel

   mov eax,1                 ;system call number (sys_exit)
   mov ebx, 0                ;exit status
   int 0x80                  ;call kernel
```

sys_write – write into a file

- system call number (in EAX): **4**
- arguments:
  - EBX: file descriptor (to write to it)
  - ECX: pointer to the first byte to write (beginning of the string)
  - EDX: number of bytes (characters) to write
- return value (in EAX):
  - number of bytes written
  - On errors: negative number

*$ is a special label which exists **only at compile time**. $ always points to the next free byte in the current section.*

By using $ we can easily calculate the length of the message.




```
section     .data
   msg: db 'Hello',0xA          ; string to print
   len:  equ $ - msg            ; length of string

section     .text
   global _start
_start:

   mov eax,4                    ;system call number (sys_write)
   mov ebx,1                    ;file descriptor (stdout)
   mov ecx, msg                 ;message to write
   mov edx, len                 ;message length
   int 0x80                     ;call kernel

   mov eax,1                    ;system call number (sys_exit)
   mov ebx, 0                   ;exit status
   int 0x80                     ;call kernel
```

sys_open - open a file

- system call number (in EAX): **5**

- arguments:
  - EBX: pathname of the file to open/create
  - ECX: set file access bits (can be bitwise OR'ed together)
    - O_RDONLY (0x000) open for reading only
    - O_WRONLY (0x001) open for writing only
    - O_RDRW (0x002) open for both reading and writing
    - O_APPEND (0x008) open for appending to the end of file
    - **O_CREAT (0x100) create the file if it doesn't exist**
  - EDX: set file permissions (in a case **O_CREAT** is set; can be bitwise OR'ed together)
    - S_I**RWX**U 0x700 ; **RWX** mask for owner
    - S_I**R**USR 0x400 ; **R**(read) for owner USR(user)
    - S_I**W**USR 0x200 ; **W**(write) for owner
    - S_I**X**USR 0x100 ; **X**(execute) for owner

- return value (in EAX):
  - file descriptor
  - On errors: negative number

must choose (at least) one of these

may add some of these

note that file name must be NULL terminated string

```
section         .data
   fileName:  db  "file.txt", 0
   handle:  dd 0

section         .text
  global _start
_start:

file_open:
  mov eax, 5                ; system call number (sys_open)
  mov ebx, fileName         ; set file name
  mov ecx, 0x101            ; set file access bits
                            ; (O_WRONLY | O_CREAT)
  mov edx, S_IRWXU          ; set file permissions
  int 0x80                  ; call kernel
  mov [handle], eax         ; move file handle to memory

   mov eax, 1                   ;system call number (sys_exit)
   mov ebx, 0                   ;exit status
   int 0x80                     ;call kernel
```

sys_lseek – change a file pointer

- system call number (in EAX): **19**
- arguments:
  - EBX: file descriptor
  - ECX: offset (number of bytes to move)
  - EDX: where to move from
    - SEEK_SET (0) ; beginning of the file
    - SEEK_CUR (1) ; current position of the file pointer
    - SEEK_END (2) ; end of file
- return value (in EAX):
  - Current position of the file pointer
  - On errors: SEEK_SET

```
section         .data
   fileName:  db  “file.txt", 0
   handle:  dd 0

section         .text
  global _start
_start:

file_open:
  mov eax, 5                ; system call number (sys_open)
  mov ecx, 0                ; set file access bits (O_RDONLY )
  mov ebx, fileName         ; set file name
  int 0x80                  ;call kernel
  mov [handle], eax         ; move file handle to memory

  mov eax,19                ; system call number (lseek)   mov
  ebx, [handle]             ; file descriptor
  mov ecx,15                ; number of byte to move
  mov edx,0                 ; move from beginning of the file
  int 0x80                  ;call kernel

   mov eax, 1                   ;system call number (sys_exit)
   mov ebx, 0                   ;exit status
   int 0x80                     ;call kernel
```

sys_close – close file

- system call number (in EAX): **6**
- arguments:
  - EBX: file descriptor
- return value (in EAX):
  - nothing meaningful
  - On errors: negative number

```
section         .data
   fileName:  db  “file.txt", 0
   handle :  dd 0

section         .text
  global _start
_start:

file_open:
  mov eax, 5                ; system call number (sys_open)
  mov ecx, 0                ; set file access bits (O_RDONLY)
  mov ebx, fileName         ; set file name
  int 0x80                  ; call kernel
  mov [handle], eax         ; move file handle to memory

  mov eax, 6
  mov ebx, [handle]
  int 0x80

   mov eax, 1                   ;system call number (sys_exit)
   mov ebx, 0                   ;exit status
   int 0x80                     ;call kernel
```

Linux System Calls - Example

self reading

```
section         .data
   fileName:  db  “file.txt", 0
   handle:  dd 0

section .bss
   buffer:   resb    1

section         .text
  global _start

_exit:
   mov ebx, [handle]
   mov eax, 6                ; system call (sys_close)
   int 0x80                  ; call kernel

   mov eax,    1             ; system call (sys_exit)
   mov ebx,    0             ; exit status
   int 0x80                  ; call kernel
```

```
_start:

   mov eax, 5                ; system call (sys_open)
   mov ebx, fileName         ; set file name
   mov ecx, O_RDONLY         ; set file access bits (O_RDONLY)
   int 0x80                  ; call kernel
   mov [handle], eax         ; move file handle to memory

_read:

   mov eax, 3                ; system call (sys_read)
   mov ebx, [handle]         ; file handle
   mov ecx, buffer           ; buffer
   mov edx, 1                ; read byte count
   int 0x80                  ; call kernel
   cmp eax, 0
   je _exit

   mov eax, 4                ; system call (sys_write)
   mov ebx, 1                ; stdout
   mov ecx, buffer           ; buffer
   mov edx, 1                ; write byte count
   int 0x80                  ; call kernel

   jmp _read
```