CENG 311
Computer Architecture

Lecture 9

1

8086 Assembly on Linux

Asst. Prof. Tolga Ayav, Ph.D.

Department of Computer Engineering
İzmir Institute of Technology

Assemblers

- as, as86, gas, nasm ...
- We use nasm:
  - Portable: DOS, Linux, Windows versions
  - Free, widely used
- Differences between gas and nasm:
  - as, gas use AT&T syntax: % prefixes to register names.
  - Nasm use Intel syntax
  - In Nasm: Destination first, source operand second
  - In as, gas: Source operand first, destionation second
- We will be using nasm but be careful when you use inline assembly in gcc, it uses AT&T syntax.

Assembling

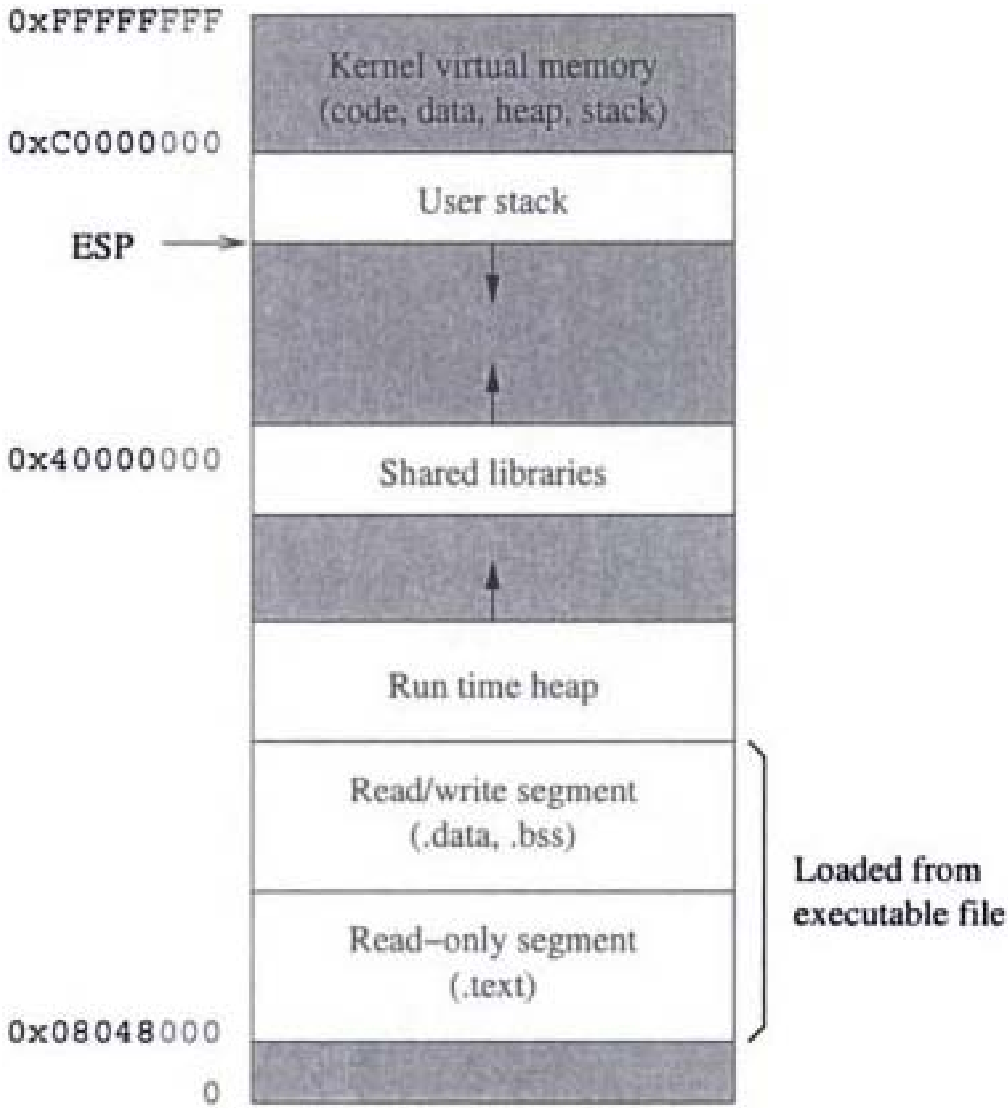
nasm –f elf hello.asm

ld –s –o hello hello.o

To run the program, type:

./hello

Linux Program Memory



Parts of the Program

.data section contains initialized variables. Constants can be defined using **equ**

db, dw, dd, dq, dt directives can be used.

```
section .data
message: db 'Hello world!'
msglength: equ 12
buffersize: dw 1024
```

Parts of the Program

.bss section contains uninitialized/zeroed variables. **resb, resw, resd, resq, rest** directives can be used.

```
section .bss
filename:     resb 255  ;reserve 255 bytes
number:       resb 1    ;reserve 1 byte
num2:         resw 1    ;reserve 1 word
realnum:      resq 10   ;reserve an array
                        of 10 reals
```

Parts of the Program

.text section contains the program in machine language.

declaration global _start tells the kernel where program execution begins.

```
section .text
    global _start

start:
    pop ebx
    .
    .
    .
```

Linux System Calls

- Linux system calls are called in exactly the same way as DOS system calls.
- You put the system call number in EAX (we're dealing with 32-bit registers here, remember)
- You set up the arguments to the system call in EBX, ECX, etc.
- You call the relevant interrupt (for DOS, 21h; for Linux, 80h)
- The result is usually returned in EAX
- There are six registers that are used for the arguments that the system call takes. The first argument goes in EBX, the second in ECX, then EDX, ESI, EDI, and finally EBP, if there are so many. If there are more than six arguments, EBX must contain the memory location where the list of arguments is stored - but don't worry about this because it's unlikely that you'll use a syscall with more than six arguments. The wonderful thing about this scheme is that Linux uses it consistently – all system calls are designed this way, there are no confusing exceptions.

System Call for Program Exit

```
mov   eax, 1  ;the exit syscall number
mov   ebx, 0  ;exit code of 0
int   80h
```

List of Linux Syscalls (There are totally 190 syscalls):

http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

Some Linux Syscalls

%eax	Name	Source	%ebx	%ecx	%edx
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	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int
8	sys_creat	fs/open.c	const char *	int	-
9	sys_link	fs/namei.c	const char *	const char *	-
10	sys_unlink	fs/namei.c	const char *	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-
12	sys_chdir	fs/open.c	const char *	-	-
13	sys_time	kernel/time.c	int *	-	-

Hello World!

```
section .data
hello: db 'Hello world!',0xA ; 0xA linefeed character
helloLen: equ $-hello

section .text
  global _start

_start:
  mov eax,4 ; The system call for write (sys_write) mov
  ebx,1 ; File descriptor 1 - standard output
  mov ecx,hello ; Put the offset of hello in ecx
  mov edx,helloLen

  mov eax,1 ;
  mov ebx,0 ; Exit with return code of 0 (no error)
  int 80h
```

Initial Stack Layout

argc	[dword] argument counter (integer)
argv[0]	[dword] program name (pointer)
argv[1] **...** **argv[argc-1]**	[dword] program args (pointers)
NULL	[dword] end of args (integer)

argc and argv[0] are always present

Command Line Arguments

- Getting the arguments in C:

```
int main(int argc, char **argv)
{
 while(argc--)
     printf(“%s\n”, argv[argc]);

 return 0;
}
```

Command Line Arguments

Popping:

```
  pop eax ;get argument counter
  pop ebx ;get our name (argv[0])
.arg:
  pop ecx ;pop all arguments
  test ecx,ecx
  jnz .arg
```

Command Line Arguments

Direct accessing:

```
pop eax ;get argument counter
pop esi ;start of arguments
mov edi,[esp+eax*4] ;end of args
```

“Procedures” and Jumping

This is a procedure:

```
fileWrite:      mov eax,4
                mov ebx,[filedesc]
                mov ecx,stuffToWrite
                mov edx,[stuffLen]
                int 80h
                ret
```

When coding, procedures and labeling have no difference. The only thing is that:

- You must call a procedure! Otherwise you must jump to the label.

Example: Find the sum of 4 integers

```
SECTION .data
x:
   dd 1
   dd 5
   dd 2
   dd 18
sum:
   dd 0
SECTION .text
   mov eax,4 ; EAX will serve as a counter
   mov ebx,0 ; EBX will store the sum
   mov ecx, x ; ECX will point to the current
top:
   add ebx, [ecx]
   add ecx,4 ; move pointer to next element
   dec eax ; decrement counter
   jnz top ; if counter not 0, then loop again
done:
   mov [sum],ebx ; done, store result in "sum"
```

Hello World - 2

- Write a program namely **writef**
- The program takes some arguments from the command line.
- The first arg is the filename, the rest of them are strings to be written into the file.

Usage:

./writef output.txt Hello World!

Hello World - 2

```
section .data
        hello     db          'Hello, world!',10      ; Our dear string
        helloLen  equ         $ - hello               ; Length of our dear string
section .bss
 address:   resw      1



section    .text
   global _start

_start:
        pop       ebx                      ; argc (argument count)
        pop       ebx                      ; argv[0] (argument 0, the program name)
        pop       ebx                      ; The first real arg, a filename

        mov       eax,8                    ; The syscall number for creat() (we already
have the filename in ebx)
        mov       ecx,00644Q               ; Read/write permissions in octal (rw_rw_rw_)
        int       80h                      ; Call the kernel
                                           ; Now we have a file descriptor in eax

        test      eax,eax                  ; Lets make sure the file descriptor is valid
        js        skipWrite  ;
```

Hello World – 2 (cont’d)

filewrite "procedure"

```
            call        fileWrite

skipWrite:
            mov         ebx,eax             ; If there was an error, save the errno in ebx
            mov         eax,1               ; Put the exit syscall number in eax
            int         80h                 ; Bail out
fileWrite:
            mov         ebx,eax             ; sys_creat returned file descriptor into eax, now
move into ebx
            mov         eax,4               ; sys_write
                                            ; ebx is already set up
            mov         ecx,hello  ; We are putting the ADDRESS of hello in ecx
            mov         edx,helloLen        ; This is the VALUE of helloLen because it's a
constant (defined with equ)

            int         80h

            mov         eax,6               ; sys_close (ebx already contains file descriptor)
            int         80h
            ret
```

main() { }

- Consider the following program "1.c":

```
main()
{
}
```

- gcc –S –masm=intel 1.c
- Output "1.s" will be:

```
section .text
global main

main:
        push    ebp
        mov     ebp, esp
        pop     ebp
        ret
```

Passing parameters to functions

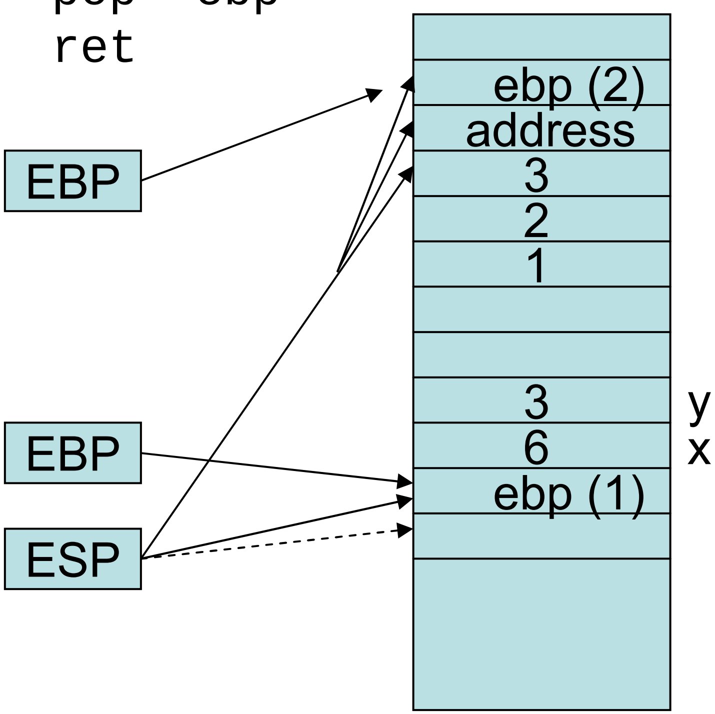
- Consider the following program “2.c”:

```
func(int a, int b, int c)
{
  return a+b+c;
}
main()
{
 int x, y=3;
 x=func(y,2,1);
}
```

- gcc –S –O0 –masm=intel 2.c
- Output “2.s” will be:

Passing parameters to functions

```
func:
  push ebp
  mov  ebp, esp
  mov  eax, DWORD PTR [ebp+8]
  add  eax, DWORD PTR [ebp+12]
  add  eax, DWORD PTR [ebp+16]
  pop  ebp
  ret
```




```
main:
    push ebp
    mov  ebp, esp
    sub  esp, 28
    mov  DWORD PTR [ebp-8], 3
    mov  DWORD PTR [esp+8], 1
    mov  DWORD PTR [esp+4], 2
    mov  eax, DWORD PTR [ebp-8]
    mov  DWORD PTR [esp], eax
    call func
    mov  DWORD PTR [ebp-4], eax
    leave
    ret
```

`enter` and `leave` instructions

`enter` and `leave` instructions are used to reserve a stack space and releases it. They are equivalent to:

```
enter n,0=
  push ebp
  mov ebp, esp
  sub esp, n

leave=
  mov esp, ebp
  pop ebp
```

Global and local variables

```
int var=5;
main()
{
 int a=7;
 int b=9;
 int c=11;
 int d=13;
}
```

gcc –S –O0 –masm=intel glblcl.c

```
section .data
  var: dw 5
section .text
global main

main:
 enter 16,0
 mov word [ebp-4], 7
 mov word [ebp-8], 9
 mov word [ebp-12], 11
 mov word [ebp-16], 13

 leave
 mov eax,1
 mov ebx,0
 int 80h
```

Compiling/Linking
C and Assembly programs - 1

```
extern void puts2(char *, int);

main()
{
 char s[50];
 sprintf(s,"Hello World 3\n");

 puts2(s,strlen(s));
}
```

gcc –c ex4.c

Compiling/Linking
C and Assembly programs - 2

```
section .text
 global puts2

puts2:
       push ebp
       mov ebp,esp
       mov eax,4
       mov ebx,1
       mov ecx, dword [ebp+8]
       mov edx, dword [ebp+12]

       int 80h

       pop ebp
       ret
```

nasm –f elf ex5.asm

Compiling/Linking
C and Assembly programs - 3

ayav@ub910TA:~$ ***gcc –c ex4.c***

ayav@ub910TA:~$ ***nasm –f elf ex5.asm***

ayav@ub910TA:~$ ***gcc –o example ex4.o ex5.o***

ayav@ub910TA:~$ ***./example***

Hello World 3

ayav@ub910TA:~$

Why do we still need to write Assembly programs?

```c
#include<stdio.h>
int sub(int a, int b)
{
 return a-b;
}
main()
{
 int x=11;
 x=sub(x,1);
 printf(“%u\n”,x);
}
```

gcc –S exp6.c

```
        .
        .
        .
sub:
        push ebp
        mov ebp,esp
        mov eax,DWORD PTR [ebp+12]
        mov edx,DWORD PTR [ebp+8]
        mov ecx,edx
        mov ecx,eax
        mov eax,ecx
        pop ebp
        ret
        .
        .
        .
```

Why do we still need to write Assembly programs?

```
      .
      .
      .
sub:
      push ebp
      mov ebp,esp
      mov eax,DWORD PTR [ebp+12]
      mov edx,DWORD PTR [ebp+8]
      mov ecx,edx
      sub ecx,eax
      mov eax,ecx
      pop ebp
      ret
      .
      .
      .
```

Return value

instead, we can write
exp7.asm to implement function *sub*:

```
section .text
global sub
sub:
      push ebp
      mov eax,dword [ebp+8]
      mov ebp,esp
      sub eax,dword [ebp+8]
      pop ebp
      ret
```

Our code is shorther than the one gcc produced.

Note: how return value is passed to the caller program

Pointer arguments

```
#include<stdio.h>

void signal(int *s)
{
 (*s)++;
}

main()
{
 int sem=0;
 signal(&sem);
 printf("%u\n",sem);
}
```

```
section .text
global signal
signal:
        push ebp
        mov ebp,esp
        mov eax,dword [ebp+8]
        mov edx, [eax]
        inc edx
        mov [eax],edx
        pop ebp
        ret
```

Inline Assembly

AT&T vs. Intel Syntax

- GNU C compiler uses AT&T syntax

Intel Code		AT&T Code	
mov	eax,1	movl	$1,%eax
mov	ebx,0ffh	movl	$0xff,%ebx
int	80h	int	$0x80
mov	ebx, eax	movl	%eax, %ebx
mov	eax,[ecx]	movl	(%ecx),%eax
mov	eax,[ebx+3]	movl	3(%ebx),%eax
mov	eax,[ebx+20h]	movl	0x20(%ebx),%eax
add	eax,[ebx+ecx*2h]	addl	(%ebx,%ecx,0x2),%eax
lea	eax,[ebx+ecx]	leal	(%ebx,%ecx),%eax
sub	eax,[ebx+ecx*4h-20h]	subl	-0x20(%ebx,%ecx,0x4),%eax

Basic inline assembly

```
asm("movl %ecx %eax");
```

/* moves the contents of ecx to eax */

```
__asm__("movb %bh (%eax)");
```

/*moves the byte from bh to the memory pointed by eax */

Basic inline assembly

```
__asm__ ("movl %eax, %ebx\n\t"
     "movl $56, %esi\n\t"
     "movl %ecx,
  $label(%edx,%ebx,$4)\n\t"
     "movb %ah, (%ebx)");
```

We touch some registers and exit without fixing them.
This may cause big trouble

Extended inline assembly

```
asm ( assembler template
        : output operands /* optional */
        : input operands /* optional */
        : list of clobbered registers /* optional */
  );
```

Extended inline assembly

```
int a=10, b;
asm ("   movl %1, %%eax;
         movl %%eax, %0;"
    :    "=r"(b) /* output */
    :    "r"(a) /* input */
    :    "%eax" /* clobbered register*/
);
```

Some Recipes - I

```
int main(void) {

int foo = 10, bar = 15;
__asm__ __volatile__(
"addl %%ebx,%%eax"
:"=a"(foo) :"a"(foo), "b"(bar) );

printf("foo+bar=%d\n", foo);

return 0; }
```

Some Recipes – II
Move Block Macro

```
#define mov_blk(src, dest, numwords) \
__asm__ __volatile__ ( \
"cld\n\t" \
"rep\n\t" \
"movsl" \
:  : "S" (src), "D" (dest), "c" (numwords) \
: "%ecx", "%esi", "%edi" \
)
```

Some Recipes – III
Exit System Call

```
asm("movl $1,%%eax; /* SYS_exit is 1 */
xorl %%ebx,%%ebx;
    /* Argument is in ebx, it is 0 */
int $0x80" /* Enter kernel mode */
);
```

Some Recipes – IV
Exit System Call

```
int reason;

asm("movl $1,%%eax; /* SYS_exit is 1 */
int $0x80" /* Enter kernel mode
: : “b” (reason)
: “%eax” “%ebx”
);
```