Sicurezza Informatica

Lez. 1
Elementi del linguaggio Assembler

Obiettivo

- Fornire gli strumenti necessari per poter analizzare le criticità in termini di sicurezza di alcune importanti componenti software:
  - Applicazioni scritte in C (ad es. Sistema operativo, web server, telnet, ecc.)
  - Applicazioni WEB
  - Codice Maligno

Strategia

- Studiare in maniera molto approfondita, le principali tecniche di intrusione sinora individuate nei diversi ambiti di riferimento

Programma

- Concetti preliminari:
  - Linguaggio Assembler (GAS)
  - Reverse engineering
- Attacchi ai sistemi operativi:
  - Buffer overflow
  - Integer Overflow
- La sicurezza delle applicazioni WEB:
  - SQL Injection
  - Cross site scripting
- Il codice maligno
  - Studio e analisi di un worm
- Mobile security
  - Analisi di malware per sistemi mobili

Prerequisiti

- Sicurezza delle reti e dei calcolatori
- Sistemi operativi
- Reti di calcolatori
- Linguaggio C
- Linux
- Un portatile

Esami/testi

- Esame:
  - 2 (forse 3) prove intermedie scritto + laboratorio
- Testi: materiale in inglese tratto da manuali, riviste e articoli specializzati, segnalati sul sito web del corso
- http://security.dico.unimi.it/sicurezza1314/sec2.shtml

Data Sizes

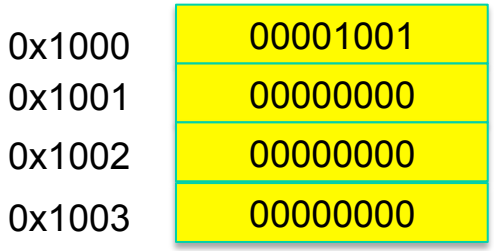
- Three main data sizes
  - Byte (b): 1 byte
  - Word (w): 2 bytes
  - Long (l): 4 bytes
- Separate assembly-language instructions
  - E.g., addb, addw, and addl

Declaring variables

- **.byte**
  - Bytes take up one storage location for each number. They are limited to
  - numbers between 0 and 255.
- **.int**
  - Ints (which differ from the **int** instruction) take up two storage locations for each number. These are limitted to numbers between 0 and 65535.9
- **.long**
  - Longs take up four storage locations. This is the same amount of space the registers use, which is why they are used in this program. They can hold numbers between 0 and 4294967295.
- **.ascii**
  - The **.ascii** directive is to enter in characters into memory. Characters each take up one storage location (they are converted into bytes internally). So, if you gave the directive **.ascii "Hello there\0"**, the assembler would reserve 12 storage locations (bytes).

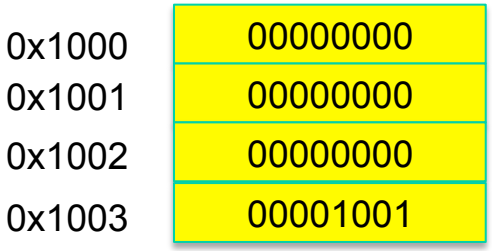
Little endian

- Intel is a little endian architecture
- Least significant byte of multi-byte entity is stored at lowest memory address
- “Little end goes first”
- Es.: l’intero 9 all’indirizzo 1000



Big endian

- Some other systems use big endian
- Most significant byte of multi-byte entity is stored at lowest memory address
- “Big end goes first”
- Es.: l’intero 9 all’indirizzo 1000



Esempio

```
int main(void) {
int i=0x003377ff, j;
unsigned char *p = (unsigned char *) &i;
for (j=0; j<4; j++)
   printf("Byte %d: %x\n", j, p[j]);
}
```

OUTPUT Little endian: ?

OUTPUT Big endian: ?

GAS Instruction Format

- General format:
  - `[prefix] opcode operands`
- Prefix used only in String Functions
- Operands represent the direction of operands
  - Single operand instruction: `XXX src`
  - Two operand instruction :`XXX src dest`
    - XXX represents the instruction opcode
    - src & dest represent the source and destination operands respectively

Loading and Storing Data

- Data can be stored in:
  - Register
  - Variable
- Variables are stored in memory
- Registers are “special” memory locations directly accessible by the processor
- The processor can only manipulate data inside registers
  - We need instruction to load from and store to memory

Esempio istruzioni dichiarative

```
.section .data                    # section declaration

msg:        .ascii "Introduci il numero:\n"   # our dear string
            len = . - msg                     # lunghezza messaggio
dieci:      .long  10
nrochar:    .word 0
zero:       .byte 0
num:        .long 0
num2:       .long 0
.section .bss
            .lcomm buf, 11
```

General purpose registers

32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Accessing data

- Processors have many ways to access data known as “addressing modes”

- Register addressing: simply moves data in or out of a register
  - Example: `movl %edx, %ecx`
  - Choice of register(s) embedded in the instruction
  - Copy value in register EDX into register ECX

Immediate Addressing

- Immediate mode is used to load direct values into registers. For example, if you wanted to load the number 12 into **%eax**, you would simply do the following:

```
movl $12, %eax
```

- Notice that to indicate immediate mode, we used a dollar sign in front of the number. If we did not, it would be direct addressing mode, in which case the value located at memory location 12 would be loaded into **%eax** rather than the number 12 itself

Direct Addressing

- Load or store from a particular memory location
  - Memory address is embedded in the instruction
  - Instruction reads from or writes to that address
- `movl 2000, %ecx`
  - Four-byte variable located at address 2000
  - Read the four bytes value contained at location 2000
  - Load the value into the ECX register
- Can use a label for (human) readability
- E.g., “i” to allow “movl i, %eax”

Indirect Addressing

- Load or store from a previously-computed address
  - Register with the address is an operand in the instruction
  - Instruction reads from or writes to that address
- Example: `movl (%eax), %ecx`
  - EAX register stores a 32-bit address (e.g., 2000)
  - Read long-word variable stored at that address
  - Load the value into the ECX register
- Dynamically allocated data referenced by a pointer
- The “(%eax)” essentially dereferences a pointer

Base pointer addressing

- Load or store with an offset from a base address
  - Register storing the base address
  - Fixed offset also embedded in the instruction
  - Instruction computes the address and does access
- Example: `movl 8(%eax), %ecx`
  - EAX register stores a 32-bit base address (e.g., 2000)
  - Offset of 8 is added to compute address (e.g., 2008)
  - Read long-word variable stored at that address
  - Load the value into the ECX register

Indexed Addressing Example

```
int a[20];
…
int i, sum=0;
for (i=0; i<20; i++)
    sum += a[i];
```

eax = ??
ebx = ??
ecx = ??

```
    movl $0, %eax
    movl $0, %ebx
sumloop:
    movl a(,%eax,4), %ecx
    addl %ecx, %ebx
    incl %eax
    cmpl $19, %eax
    jle sumloop
```

Summary

- Immediate addressing: data stored in the instruction itself
  - `movl $10, %ecx`
- Register addressing: data stored in a register
  - `movl %eax, %ecx`
- Direct addressing: address stored in instruction
  - `movl foo, %ecx`
- Indirect addressing: address stored in a register
  - `movl (%eax), %ecx`
- Base pointer addressing: includes an offset as well
  - `movl 4(%eax), %ecx`
- Indexed addressing: instruction contains base address, and specifies an index register and a multiplier (1, 2, 4, or 8)
  - `movl 2000(,%eax,1), %ecx`

Arithmetic Instructions

- Simple instructions
  - `add{b,w,l} source, dest` dest = source + dest
  - `sub{b,w,l} source, dest` dest = dest – source
  - `inc{b,w,l} dest` dest = dest + 1
  - `dec{b,w,l} dest` dest = dest – 1
  - `neg{b,w,l} dest` dest = ~dest + 1
  - `cmp{b,w,l} source1, source2` source2 – source

Mull/Div

- Multiply
  - `mul (unsigned) or imul (signed)`
    - Performs signed multiplication and stores the result in the second operand. If the second operand is left out, it is assumed to be **%eax**, and the full result is stored in the double-word **%edx**:**%eax**
- Divide
  - `div (unsigned) or idiv (signed)`
  - Divides the contents of the double-word contained in the combined **%edx**:**%eax** registers by the value in the register or memory location specified. The **%eax** register contains the resulting quotient, and the **%edx** register contains the resulting remainder

Bitwise logic instructions

- Simple instructions
  - `and{b,w,l} source, dest` dest = source & dest
  - `or{b,w,l} source, dest` dest = source | dest
  - `xor{b,w,l} source, dest` dest = source ^ dest
  - `not{b,w,l} dest` dest = ~dest
  - `sal{b,w,l} source, dest` dest = dest << source
  - `sar{b,w,l} source, dest` dest = dest >> source

Control Flow

- We obtain contro flow using two instructions:

```
cmpl $0, %eax
je   end_loop
```

- The first one is the cmpl instruction which compares two values, and stores the result of the comparison in the status register EFLAGS. Notice that the comparison is to see if the second value is greater than the first
- The second one is the flow control instruction JUMP which says to jump to the end_loop depending on the values stored in the status register and on the condition expressed

Types of Jumps

- je: Jump if the values were equal
- jg: Jump if the second value was greater than the first value
- jge: Jump if the second value was greater than or equal to the first value
- jl: Jump if the second value was less than the first value
- jle: Jump if the second value was less than or equal to the first value
- jmp:Jump no matter what. This does not need to be preceeded by a comparison

I/O

- Initially we will use system calls for performing the basic I/O operations
- Linux system calls are called in the following way:
  - 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 Linux, 80h)
  - The result is usually returned in EAX

Exit Syscall

- Some example code always helps:

```
mov    eax,1    # The exit syscall number
mov    ebx,0    # Have an exit code of 0
int    80h      # Interrupt 80h
```

- But how do you find out what these system calls are, and what they do, and what arguments they take? Firstly, all the syscalls are listed in /usr/include/asm/unistd.h

Important Linux Syscalls

%eax	Name	%ebx	%ecx	%edx	Note
1	exit	Return value			Exits the program
3	read	File descriptor (0 for stdin)	Buffer start	Buffer size	Read into the given buffer
4	write	File descriptor (1 for stout)	Buffer start	Buffer size	Writes the buffer to the file descriptor

syscall write

Function Definition

```
size_t write(int fildes, const void *buf, size_t nbytes);
```

Field	Description
int fildes	The file descriptor of where to write the output. You can either use a file descriptor obtained from the open system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
const void *buf	A null terminated character string of the content to write.
size_t nbytes	The number of bytes to write. If smaller than the provided buffer, the output is truncated.
return value	Returns the number of bytes that were written. If value is negative, then the system call returned an error.

Example

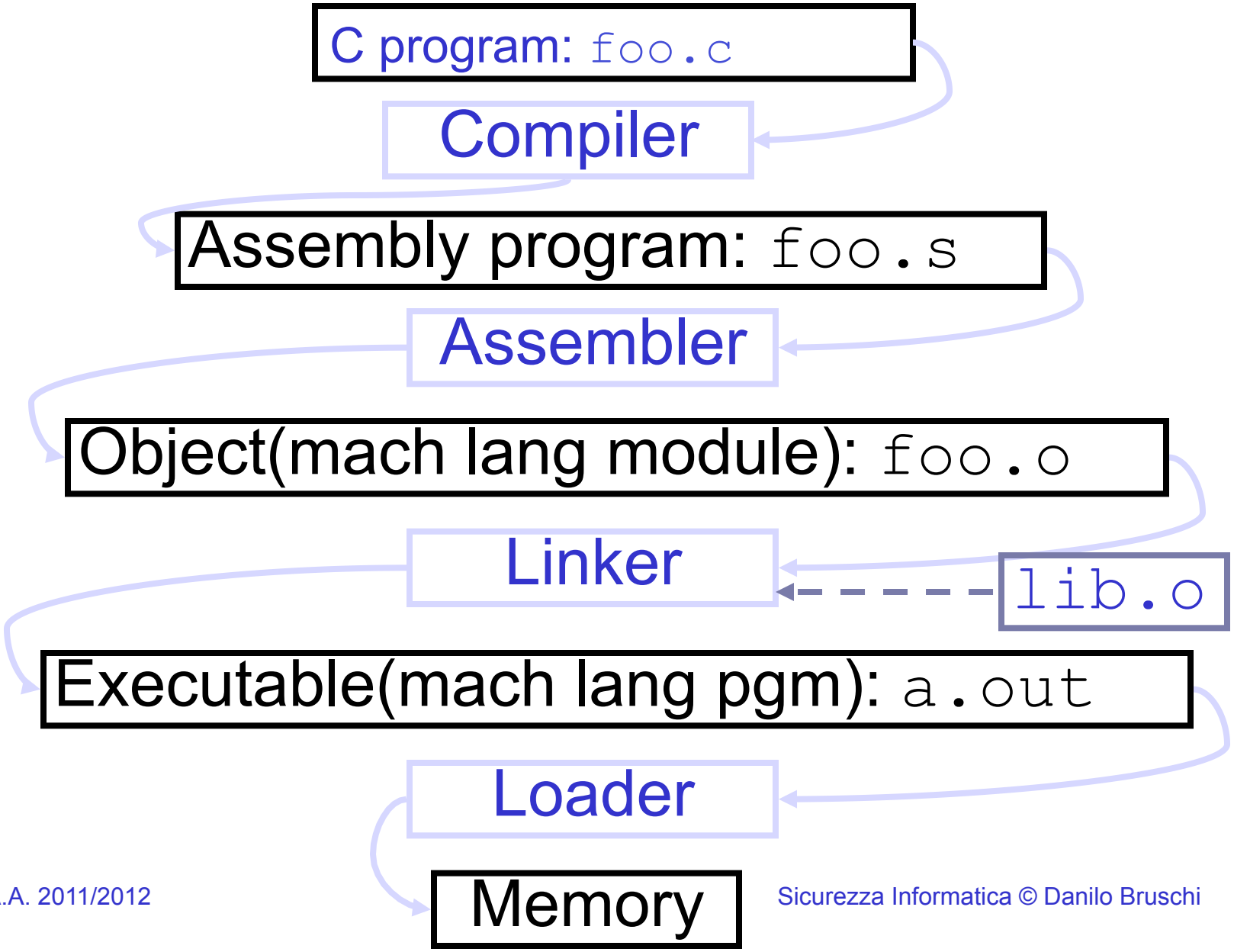
```
.text                   # section declaration
                        # we must export the entry point to the ELF linker or
.global _start          # loader. They conventionally recognize _start as
                        # entry point. Use ld -e foo to override the default
_start:
# write our string to stdout
        movl    $len,%edx           # third argument: message length
        movl    $msg,%ecx           # second argument: pointer to msg
        movl    $1,%ebx             # first argument: file handle (stdout)
        movl    $4,%eax             # system call number (sys_write)
        int     $0x80               # call kernel and exit
        movl    $0,%ebx             # first argument: exit code
        movl    $1,%eax             # system call number (sys_exit)
        int     $0x80               # call kernel
.data                               # section declaration
msg:    .ascii  "Hello, world!\n"   # our dear string
        len = . - msg               # length of our dear string
```

From .s to executables

From assembler to executable

- In order to be executed by a processor an assembler program has to be translated in machine language
- In order to accomplish such a task we need the following tools:
  - Assembler
  - Linker
  - Loader

Steps to Starting a Program



Compiling & Linking

- To assembly the program type in the command

```
as name.s -o name.o
```

- as is the command which runs the assembler, name.s is the source file, and –o name.o tells the assemble to put it's output in the file name.o which is an object file. An object file is code that is in the machine's language, but has not been completely finalized

Assembler

- Reads and Uses Directives
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File (.o files)

Assembler Directives/Pseudo

- Give directions to assembler, but do not produce machine instructions, e.g.
  - `.text`: Subsequent items put in user text segment
  - `.data`: Subsequent items put in user data segment
  - `.globl sym`: declares `sym` global and can be referenced from other files
  - `.asciiz str`: Store the string `str` in memory and null-terminate it
- Pseudo: variations of machine language instructions introduced for simplifying the programming task
  - pseudo are translated in the corresponding real instruction

Translating in machine language

- Many instructions can be assembled independently
  - `pushl %edx`
  - `leal (%eax, %eax, 4), %eax`
  - `movl $0, %eax`
  - `addl %ebx, %ecx`
- But, some make references to other data or code
  - `jne skip`
  - `pushl $msg`
  - `call printf`
- Need to fill in those references to generate a final executable binary

2 phase assembler

- Pass1: Assembler traverses assembly program to create a symbol table
  - Key: label
  - Value: information about label (Label name, which section, what offset within that section, …)
- Pass 2: Assembler traverses assembly program again to create…
  - RODATA section
  - DATA section
  - BSS section
  - TEXT section
  - Relocation record section
    - Each relocation record indicates an area that the linker must patch

Example

```
.text                   # section declaration
                        # we must export the entry point to the ELF linker or
.global _start          # loader. They conventionally recognize _start as
                        # entry point. Use ld -e foo to override the default
_start:
# write our string to stdout
        movl    $len,%edx           # third argument: message length
        movl    $msg,%ecx           # second argument: pointer to msg
        movl    $1,%ebx             # first argument: file handle (stdout)
        movl    $4,%eax             # system call number (sys_write)
        int     $0x80               # call kernel and exit
        movl    $0,%ebx             # first argument: exit code
        movl    $1,%eax             # system call number (sys_exit)
        int     $0x80               # call kernel
.data                               # section declaration
msg:    .ascii  "Hello, world!\n"   # our dear string
        len = . - msg               # length of our dear string
```

Sections

```
danilo@muffet:~$ objdump -h hello.o

hello.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA           LMA           File off
Algn
  0 .text         00000022  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         0000000e  00000000  00000000  00000058  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000068  2**2
                  ALLOC
```

Symbol Table

```
danilo@muffet:~$ objdump -t hello.o

hello.o:     file format elf32-i386

SYMBOL TABLE:
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss   00000000 .bss
0000000e l       *ABS*  00000000 len
00000000 l       .data  00000000 msg
00000000 g       .text  00000000 _start
```

Contenuto Sezioni

```
danilo@muffet:~$ objdump -s hello.o

hello.o:     file format elf32-i386

Contents of section .text:
 0000 ba0e0000 00b90000 0000bb01 000000b8  ................
 0010 04000000 cd80bb00 000000b8 01000000  ................
 0020 cd80                                 ..
Contents of section .data:
 0000 48656c6c 6f2c2077 6f726c64 210a      Hello, world!.
```

Sezione text

```
_start:

# write our string to stdout

        movl    $len,%edx       # third argument: message length
   0:   ba 0e 00 00 00          mov     $0xe,%edx
        movl    $msg,%ecx       # second argument: pointer to message to write
   5:   b9 00 00 00 00          mov     $0x0,%ecx
        movl    $1,%ebx         # first argument: file handle (stdout)
   a:   bb 01 00 00 00          mov     $0x1,%ebx
        movl    $4,%eax         # system call number (sys_write)
   f:   b8 04 00 00 00          mov     $0x4,%eax
        int     $0x80           # call kernel
  14:   cd 80                   int     $0x80

# and exit

        movl    $0,%ebx         # first argument: exit code
  16:   bb 00 00 00 00          mov     $0x0,%ebx
        movl    $1,%eax         # system call number (sys_exit)
  1b:   b8 01 00 00 00          mov     $0x1,%eax
        int     $0x80           # call kernel
  20:   cd 80                   int     $0x80
```

Relocation record

```
danilo@muffet:~$ objdump -r hello.o

hello.o:     file format elf32-i386

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000006 R_386_32          .data
```

Linker

- Symbols have to be relocated
- In most large programs, you will have several source files, and you will convert each one into an object file.
- The linker is the program that is responsible for putting the object files together and adding information to it so that the kernel knows how to load and run it.
- To link the file, enter the command

```
ld name.o -o name
```

Linker

- Combines several object (.o) files into a single executable ("linking") (when needed)
- Enable Separate Compilation of files
  - Changes to one file do not require recompilation of whole program
- Works in two phases: resolution and relocation

Linker

- Step 1: Take text segment from each .o file and put them together.
- Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.
- Step 3: Resolve References
  - Go through Relocation Table and handle each entry
  - That is, fill in all absolute addresses

Resolving References (1/2)

- Linker *assumes* first word of first text segment is at address 0x00000000.
- Linker knows:
  - length of each text and data segment
  - ordering of text and data segments
- Linker calculates:
  - absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

Resolving References (2/2)

- To resolve references:
  - search for reference (data or label) in all symbol tables
  - if not found, search library files (for example, for `printf`)
  - once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header) (.elf or PE)

Sezioni eseguibile

```
danilo@muffet:~$ objdump -h hello

hello:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000022  08048074  08048074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         0000000e  08049098  08049098  00000098  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Executing

- You can run the executable prog by typing in the command

  ./prog

- The ./ is used to tell the computer that the program isn't in one of the normal program directories, but is the current directory instead

Debugging

- In assembly language, even minor errors usually have results such as the whole program crashing with a segmentation fault error
- Therefore, to aid in determining the source of errors, you must use a source debugger
- The debugger we will be looking at is GDB - the GNU Debugger
- It can debug programs in multiple programming languages, including assembly language

Loader I

- Executable files are stored on disk
- When one is run, loader's job is to load it into memory and start it running:
  - Reads executable file's header to determine size of text and data segments
  - Creates new address space for program large enough to hold text and data segments, along with a stack segment
  - Copies instructions and data from executable file into the new address space (this may be anywhere in memory)

Loader II

- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program's arguments from stack to registers and sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call

Debugging

- To run a program under gdb you need to have the assembler include debugging information in the executable. All you need to do to enable this is to add the --gstabs option to the as command. So, you would assemble it like this:

```
as --gstabs name.s –o name.o
```

- Linking would be the same as normal
- Now, to run the program under the debugger, you would type in

```
gdb name
```

gdb

```
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or
distribute copies of it under certain conditions. Type
"show copying" to see the conditions. There is
absolutely no warranty for GDB. Type "show warranty"
for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

- At this point, the program is loaded, but is not running yet. The debugger is waiting your command. To run your program, just type in run.

Some commands

- A breakpoint is a place in the source code that you have marked to indicate to the debugger that it should stop the program when it hits that point
- To set breakpoints you have to set them up before you run the program. Before issuing the run command, you can set up breakpoints using the break command
- For example, to break on line 27, issue the command break 27. Then, when the program crosses line 27, it will stop running, and print out the current line and instruction.

Some commands

- To follow the flow of a program, keep on entering stepi (for "step instruction"), which will cause the computer to execute one instruction at a time
- To check the contents of register in GDB either use the command info register or print/ $eax to print register eax in hexadecimal, or do print/d $eax to print it in decimal
- x/ addr: print the contents of memory address addr
- For other command see the help command

Excercise

- Assembly and execute the Hello World program, modify the EXECUTABLE so that it can print “Hello, Milan!”

- Write a program for reading a number from keyboard and print it out

Homework

- Write an assembly program which reads as input two positive integer numbers and print the difference between the first and the second one (you can assume that the difference will never be negative)

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL