

CS Basics

8) Strings

E. Benoist & C. Grothoff
Fall Term 2018-19

Strings

- Loops on Strings
 - Strings in assembly
 - STOre String by Byte
 - MOVe String by Bytes
- Loops
 - LOOP Mnemonic
 - REP prefix
- Command-line arguments
- Links with C Programs
 - Linking with standard C Lib

Loops on Strings

- ▶ **Some registers have a specific “agenda”**
 - ▶ RAX and RDX for the multiplication (implicit parameters)
 - ▶ RCX is a “Counter”
- ▶ **Some should contain specific addresses of strings (input / output)**
 - ▶ RSI “Source Index”
 - ▶ RDI “Destination Index”

Example Copy of a String (Cont.)

```
Exit:   mov eax,1           ; Code for Exit\  
→ Syscall  
       mov ebx,0         ; Return a code\  
       → of zero  
       int 80H           ; Make kernel \  
       → call
```

STOre String by Byte

The STOSB mnemonic

- ▶ **STOre String by Byte (STOSB)**
 - ▶ Does work only with “implicit” parameters
 - ▶ RDI must contain the address of a destination string (*Destination Index*)
 - ▶ RCX must contain the size of the string (*C* is for *Counter*)
 - ▶ AL must contain a value to be inserted in the string at EDI
- ▶ **Action of STOSB**
 - ▶ The byte value in AL is copied to the memory address stored in RDI
 - ▶ RDI is incremented by 1
- ▶ **Direction**
 - ▶ Normal direction if Direction Flag (DF) is not set
 - ▶ If Direction Flag (DF) is set: EDI is decremented
 - ▶ Mnemonics to change DF:
 - ▶ CLD to clear the DF
 - ▶ STD to set the DF

The STOSx mnemonics

- ▶ **STOS exists in other sizes**
 - ▶ STOSW (Words: 2 bytes),
Value is copied from AX
RDI is adjusted by 2 (incremented or decremented with respect to DF)
 - ▶ STOSD (Double Words: 4 bytes),
Value is copied from EAX
RDI is adjusted by 4
 - ▶ STOSQ (Quad Words: 8 bytes)
Value is copied from RAX
RDI is adjusted by 8

Reset of a string using STOSB

- ▶ The following code will put a spaces in the string

```
mov rcx, TARGETLEN ; The counter receives \
→the length of the string
mov rsi, InitString ; The source index \
→receives Initial String
→
mov rdi, TargetString ; The destination index\
→ receives
;the address of Target String
dec rcx ; Don t erase last char, it is a \n
.loopcopystring:
mov bl, 'm' ; Move a char containing "space" \
→into BL
mov byte [rdi], bl; The element in BL is \
→copied into memory at RDI
inc rdi ; We go forward in strings
dec rcx ; The counter is decremented
jnz .loopcopystring ;Loop as long as RCS is \
→not 0
```

Berner Fachhochschule | Haute école spécialisée bernoise | Berne University of Applied Sciences

13

MOVE String by Bytes

Berner Fachhochschule | Haute école spécialisée bernoise | Berne University of Applied Sciences

Copy a string using STOSB

- ▶ We copy each character in the first string into the second using STOSB

```
;; Init registers to start the copy
mov rcx, INITLEN ; counter receives length \
→of string
mov rsi, InitString ; The source index \
→receives Initial String
→
mov rdi, TargetString ; The destination \
→index receives the address of Target String
cld ; ensure direction for stosb is OK
.loopcopystring:
mov al, byte[rsi] ; From source at [rsi] to \
→AL
stosb ; From AL to [RDI] (and increment RDI)
inc rsi ; We need to take care of RSI
dec rcx ; The counter is decremented
jnz .loopcopystring ; Loop as long as RCS is \
→not 0
```

Berner Fachhochschule | Haute école spécialisée bernoise | Berne University of Applied Sciences

14

MOVE String by Bytes: MOVSB

- ▶ **Fast Block Copies**
 - ▶ Memory data at RSI (Source Index)
 - ▶ Is copied to RDI (Destination Index)
 - ▶ Copies one byte

- ▶ **Example Copy String MySource to MyDest**

```
mov RSI, MySource ; Install source address \
→into register RSI
mov RDI, MyDest ; Install destination \
→address into register RDI
mov RCX, MYSOURCELEN ; copy the length of \
→MYSOURCE into the counter
```

MyLabel:

```
movsb ; Copy the content of memory at RSI into \
→ memory at address RDI
dec RCX
jnz MyLabel ; Loop as long as RCX not zero
```

- ▶ **Move other sizes**
 - ▶ Words (2 bytes) MOVSW
 - ▶ Double Words (4 Bytes) MOVSD
 - ▶ Quad Words (8 Bytes) MOVSQ

Berner Fachhochschule | Haute école spécialisée bernoise | Berne University of Applied Sciences

16

Example: Copy a string using MOVSB

```
mov rcx, INITLEN      ; The counter ↘  
→receives the length of the string  
mov rsi, InitString  ; The source ↘  
→index receives Initial String ↘  
→  
mov rdi, TargetString ; The ↘  
→destination index receives  
; the address of Target String
```

.loopcopystring:

```
movsb                ; From [RSI] to ↘  
→ [RDI] (and increment RDI and RSI)  
dec rcx              ; The counter ↘  
→is decremented  
jnz .loopcopystring ; Loop as long ↘  
→as RCS is not 0
```

LOOP Mnemonic

Loops

LOOP

▶ Loop using the Counter (RCX)

- ▶ The following pieces of code are equivalent

▶ Using Jnz

```
mov RCX, 5 ; Set the counter to 5  
Label1:  
....  
....  
dec RCX ; Decrements the counter  
jnz Label1 ; Loop as long as RCX is not 0
```

▶ Using loop

```
mov RCX, 5 ; Set the counter to 5  
Label1:  
....  
....  
loop Label1 ; Decrements RCX and loop as ↘  
→long as RCX is not 0
```

REP prefix

REP prefix

- ▶ **REP is not an instruction**
 - ▶ It is a prefix to be placed in front of an instruction
 - ▶ Repeat the instruction by decrementing RCX (the counter)
 - ▶ as long as RCX is not 0
- ▶ **The following two codes are functionally equivalent**
 - ▶ Using explicit instructions

Clear:

```
mov byte [rdi], al ; write the value ↘  
→ of AL to memory  
inc edi ; Move one step forward in ↘  
→ the memory  
dec rcx ; Decrement the counter  
jnz Clear ; Loop until RCX becomes 0
```

- ▶ Using implicit instructions

```
rep stosb ; All in one !!!
```

REP with MOVSB

- ▶ **Example Copy String MySource to MyDest**

```
mov RSI, MySource ; Install source ↘  
→ address into register RSI  
mov RDI, MyDest ; Install ↘  
→ destination address into register RDI  
mov RCX, MYSOURCELEN ; copy the ↘  
→ length of MYSOURCE into the counter  
rep movsb ; Copy the block
```

- ▶ **REP uses a counter RCX**

- ▶ RCX is decremented by 1 (one)
- ▶ Even if using MOVSW, MOVSD or MOVSQ

Command-line arguments

Command-line arguments

▶ Starting a program in Linux

```
bie@mymachine> ./myprogram arg1 arg2 arg3
```

- ▶ Arguments are separated with spaces

▶ Enter args in debugger DDD

- ▶ Insider the gdb console (at the bottom of the window)
- ▶ type:


```
(gdb) set args arg1 arg2 arg3
```
- ▶ Then click on "run"

Linux Stack

▶ Stack is initialized by Linux at the begining of your program

Contains:

- ▶ Fully qualified pathname of the executable
- ▶ Command-line arguments
- ▶ Current state of Linux environment variables
- ▶ **RSP (Stack Pointer) points to the top of the stack**
 - ▶ By the way: it is at the high end of the memory, since Stack is growing from high addresses to low addresses
- ▶ **Stack contains (from the top to the bottom)**
 - ▶ Count of arguments (1 even if no arguments, it is "invocation text")
 - ▶ Address of the invocation text (terminated by 00h)
 - ▶ Address(es) of the command-line arguments (each terminated by 00h)
 - ▶ One null pointer (4 bytes of binary 0)
 - ▶ Addresses of environment variables (terminated by 00h)
 - ▶ Actual data refered by addresses

Linux stack at program startup

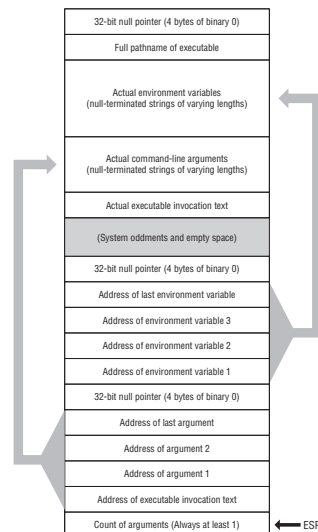


Figure 11-4: Linux stack at program startup

Instruction SCAn String by Bytes: SCASB

▶ We use SCASB to search for the end of string marker (00h)

- ▶ Address of the first Byte to be searched is in RDI
- ▶ Value to be searched is placed in AL (or AX, EAX, RAX depending on the instruction: SCASW, SCASD, SCASQ)
- ▶ Maximum count is placed in RCX (the counter)

▶ Action

- ▶ The string at RDI is checked and one byte is compared to AL

▶ Repeat as long as not equal: REPNE (i.e. REPeat Not Equal)

```
REPNE SCASB ; ECX is decremented until the
              ;value is found RDI is incremented
```

In the end RDI contains the address of the searched elements (0 in our case)

Example: show all arguments

```
; Executable name : SHOWARGS1
; Version         : 1.0
; Created date    : 4/17/2009
; Last update     : 5/19/2009
; Author          : Jeff Duntemann
; Description     : A simple program in assembly for Linux, using NASM 2.05,
;                 demonstrating the way to access command line arguments on the stack.
;
; Build using these commands:
;   nasm -f elf -g -F stabs showargs1.asm
;   ld -o showargs1 showargs1.o
;
SECTION .data                ; Section containing initialised data
    ErrMsg db "Terminated_with_error.",10
    ERRLEN equ $-ErrMsg

SECTION .bss                 ; Section containing uninitialized data

; This program handles up to MAXARGS command-line arguments. Change the
; value of MAXARGS if you need to handle more arguments than the default 10.
; In essence we store pointers to the arguments in a 0-based array, with the
; first arg pointer at array element 0, the second at array element 1, etc.
; Ditto the arg lengths. Access the args and their lengths this way:
;   Arg strings:      [ArgPtrs + <index reg>*4]
;   Arg string lengths: [ArgLens + <index reg>*4]
; Note that when the argument lengths are calculated, an EOL char (10h) is
; stored into each string where the terminating null was originally. This
; makes it easy to print out an argument using sys_write. This is not
; essential, and if you prefer to retain the 0-termination in the arguments,
; you can comment out those lines as indicated.
```

Berner Fachhochschule | Haute école spécialisée bernoise | Berne University of Applied Sciences

29

Example (Cont.)

```
MAXARGS equ 10                ; Maximum # of args we support
ArgCount: resd 1              ; # of arguments passed to program
ArgPtrs:  resd MAXARGS        ; Table of pointers to arguments
ArgLens:  resd MAXARGS        ; Table of argument lengths

SECTION .text                 ; Section containing code

global _start                ; Linker needs this to find the entry point!

_start:
    nop                      ; This no-op keeps gdb happy...

; Get the command line argument count off the stack and validate it:
    pop ecx                  ; TOS contains the argument count
    cmp ecx,MAXARGS          ; See if the arg count exceeds MAXARGS
    ja Error                 ; If so, exit with an error message
    mov dword [ArgCount],ecx ; Save arg count in memory variable

; Once we know how many args we have, a loop will pop them into ArgPtrs:
    xor edx,edx              ; Zero a loop counter

SaveArgs:
    pop dword [ArgPtrs + edx*4] ; Pop an arg into the memory table
    inc edx                  ; Bump the counter to the next argument
    cmp edx,ecx              ; Is the counter = the argument count?
    jb SaveArgs              ; If not, loop back and do another
```

Berner Fachhochschule | Haute école spécialisée bernoise | Berne University of Applied Sciences

30

Example (Cont.)

```
; With the argument pointers stored in ArgPtrs, we calculate their lengths:
    xor eax,eax              ; Searching for 0, so clear AL to 0
    xor ebx,ebx              ; Pointer table offset starts at 0
ScanOne:
    mov ecx,0000ffffh        ; Limit search to 65535 bytes max
    mov edi,dword [ArgPtrs+ebx*4] ; Put address of string to search in EDI
    mov edx,edi              ; Copy starting address into EDX
    cld                      ; Set search direction to up-memory
    repne scasb              ; Search for null (0 char) in string at edi
; Comment out the following line if you need null-terminated arguments:
    mov byte [edi-1],10       ; Store an EOL where the null used to be
    sub edi,edx              ; Subtract position of 0 from start address
    mov dword [ArgLens+ebx*4],edi ; Put length of arg into table
    inc ebx                  ; Add 1 to argument counter
    cmp ebx,[ArgCount]       ; See if arg counter exceeds argument count
    jb ScanOne               ; If not, loop back and do another one
```

Berner Fachhochschule | Haute école spécialisée bernoise | Berne University of Applied Sciences

31

Example (Cont.)

```
; Display all arguments to stdout:
    xor esi,esi              ; Start (for table addressing reasons) at 0
Showem:
    mov ecx,[ArgPtrs+esi*4] ; Pass offset of the message
    mov eax,4                ; Specify sys_write call
    mov ebx,1                ; Specify File Descriptor 1: Standard Output
    mov edx,[ArgLens+esi*4] ; Pass the length of the message
    int 80H                  ; Make kernel call
    inc esi                  ; Increment the argument counter
    cmp esi,[ArgCount]       ; See if we've displayed all the arguments
    jb Showem                ; If not, loop back and do another
    jmp Exit                  ; We're done! Let's pack it in!

Error:
    mov eax,4                ; Specify sys_write call
    mov ebx,1                ; Specify File Descriptor 2: Standard Error
    mov ecx,ErrMsg           ; Pass offset of the error message
    mov edx,ERRLEN           ; Pass the length of the message
    int 80H                  ; Make kernel call

Exit:
    mov eax,1                ; Code for Exit Syscall
    mov ebx,0                ; Return a code of zero
    int 80H                  ; Make kernel call
```

Berner Fachhochschule | Haute école spécialisée bernoise | Berne University of Applied Sciences

32

Links with C Programs

Links with C Programs

- ▶ **Gnu is not Unix, but Unix is written in C**
 - ▶ Most of Unix libraries are in C
- ▶ **Access C libraries in Assembly**
 - ▶ Call C functions in Assembly
 - ▶ Send parameters
 - ▶ Link files
- ▶ **Advantages**
 - ▶ We can access to all the C functions (large set of functions)
 - ▶ We see exactly how it works in C.

Compiling code

- ▶ **Compiling using gcc**
 - ▶ Compiling C code

```
gcc eatc.c -o eatc
```

Compiling and linking in one step using gcc (because output is a file without suffix)
 - ▶ Files created by GCC
 - ▶ .C a C file after preprocessing (handles #includes and #defines)
 - ▶ .S an assembler file that is compiled with gas
 - ▶ .o an object file that needs to be linked (also with standard C libraries)
 - ▶ executable

How gcc builds Linux executables

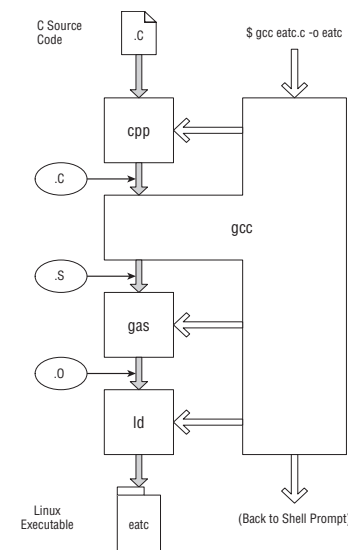


Figure 12-1: How gcc builds Linux executables

Linking with standard C Lib

Structure of a hybrid C-assembly program

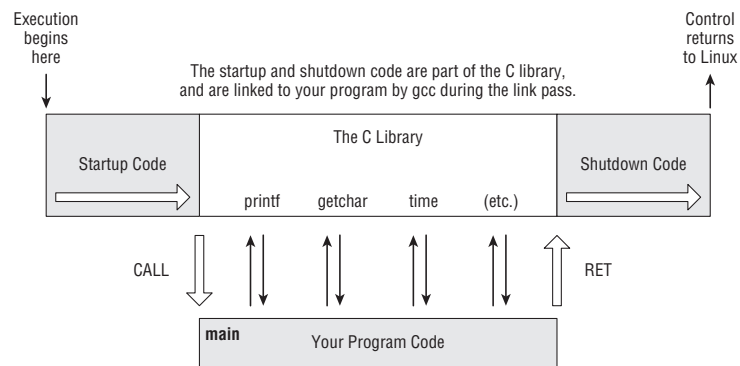


Figure 12-2: Structure of a hybrid C-assembly program

Linking to the Standard C Library

▶ Your program will call standard functions

- ▶ Printing strings (`printf`)
- ▶ Reading chars (`getchar`)
- ▶ Accessing time (`time`)

Need to link the standard C Library `glibc`

▶ Or any of your own C functions

- ▶ Need to link the `.o` produced from your C source
- ▶ Rule: only one `main:` label (assembly) or C `main` function in one executable

▶ Use `gcc` to link the files with the `glibc`

- ▶ It knows exactly what to link

C Calling Conventions

▶ You can call C functions with **CALL** instruction

- ▶ You must follow rules of C calling conventions

▶ Conventions (32bits)

- ▶ A procedure must preserve the values of `EBX`, `ESP`, `EBP`, `ESI`, and `EDI`
All other registers may be altered: you have to save registers prior to call a function
- ▶ Return value is stored in `EAX` (32-bit) or `EAX` and `EDX` (64-bit)
Strings (and the like) are returned by reference (the address in `EAX`)
- ▶ Parameters passed to procedures are pushed onto the stack in *reverse order*
Example: for function `MyFunc(foo, bar, bas)`, `bas` is pushed onto the stack first, then `bar` and `foo` last
- ▶ Procedures will not remove parameters from the stack, the caller must do that after the procedure returns
- ▶ The label at which the program starts must be `main` (case sensitive) and **not** `_start`

Calling conventions in 64-bit architecture (simplified!)

- ▶ **Arguments are stored in registers (not primarily on stack)**
 - ▶ Once arguments are classified, the registers get assigned (in left-to-right order) for passing as follows:
 - ▶ If the class is MEMORY, pass the argument on the stack.
 - ▶ If the class is INTEGER, the next available register of the sequence RDI, RSI, RDX, RCX, R8 and R9 is used
 - ▶ RAX must be set to 0: `xor rax,rax`
 - ▶ so RDI, RSI, RDX, RCX, R8 and R9 are the registers in order used to pass parameters to any libc function from assembly.
 - ▶ RDI is used for first parameter, RSI for 2nd, RDX for 3rd and so on.
 - ▶ Then `call` instruction should be given.

Call the function puts()

- ▶ **puts() sends characters to the standard output**
 - ▶ Label `main` (and not `._start`) as starting point
 - ▶ store RBP on the stack
 - ▶ save RSP into RBP
 - ▶ Must preserve RBX, RSI and RDI
 - ▶ Push arguments for the function (address of message)
 - ▶ Call the function
 - ▶ Restore the saved values
- ▶ `Puts()` expects use of `0` as a string delimiter!

Example of function call (32-bit)

```
; Source name      : EATCLIB.ASM
; ...
; Build using these commands:
;   nasm -f elf -g -F stabs eatclib.asm
;   gcc eatclib.o -o boiler
[SECTION .data]    ; Section containing initialised data
    EatMsg: db "Eat,at,Joe's!",0

[SECTION .bss]    ; Section containing uninitialized data\
→
[SECTION .text]  ; Section containing code
extern puts      ; Simple "put string" routine from clib
global main      ; Required so linker can find entry point

main:
    push ebp     ; Set up stack frame for debugger
    mov ebp,esp
    push ebx     ; Must preserve ebp, ebx, esi, & edi
    push esi
    push edi
    ;; Everything before this is boilerplate; use it for all \
→ordinary apps!
```

Example of function call (Cont.)

```
    push EatMsg ; Push address of message on the stack
    call puts   ; Call clib function for displaying strings
    add esp,4   ; ; Stack cleanup 1 parameter x 4 bytes

    ;; Everything after this is boilerplate; use it for all \
→ordinary apps!
    pop edi    ; Restore saved registers
    pop esi
    pop ebx
    mov esp,ebp ; Destroy stack frame before returning
    pop ebp
    ret        ; Return control to Linux
```

Example of function call (64-bit)

```
[SECTION .data] ; Section containing initialised data
EatMsg: db "Eat_at_Joe's!" ,10,0 ; We do not need the length because of c ↘
→library
[SECTION .text] ; Section containing code
extern puts ; Simple "put string" routine from clib
global main ; Required so linker can find entry point
main:
    push rbp ; Set up stack frame for debugger
    mov rbp, rsp
    push rbx ; Must preserve registers
    push rsi
    push rdi
    ;; Everything before this is boilerplate; use it for all ordinary apps!
    mov rdi, EatMsg ; Write address of message in RDI
    call puts ; Call clib function for displaying strings
    ;; Everything after this is boilerplate; use it for all ordinary apps!
    pop rdi ; Restore saved registers
    pop rsi
    pop rbx
    mov rsp, rbp ; Destroy stack frame before returning
    pop rbp
    ret ; Return control to Linux
```

Call of printf()

▶ The C function printf() - print and format

- ▶ output text
- ▶ Format the numbers inside it
- ▶ Takes as input: a string with place holders, values to be put inside the string

▶ Example

```
printf('%d+=%d=%d', 2,3,5); // prints ↘
→out 2+3=5
```

▶ Formatting codes

- ▶ %c character
- ▶ %d integer in decimal
- ▶ %s string in a string
- ▶ %x integer in hexa
- ▶ %% a percent symbol

Example of call of printf()

▶ Call in assembly 32-bit

```
push 5
push 3
push 2
push mathmsg ; the address of the ↘
→string
call printf
add esp, 16 ; Stack cleanup 3 ↘
→parameters x 4 bytes
```

Example of call of printf()

▶ Call in assembly 64-bit

Parameters in RSI and RDI

```
mov rdi, MathMsg ; Write ↘
→address of message in RDI
mov rsi, 50 ; Second ↘
→parameter is 50
mov rdx, 40 ; Third ↘
→parameter is 40
mov rcx, 30 ; Fourth ↘
→parameter is 30
xor rax, rax ; reset of ↘
→ RAX before the call
call printf ; Call clib ↘
→function for displaying strings
```

Value returned

- ▶ **Must be read in RAX**

Conclusion

- ▶ **New syntax for manipulating strings**
 - ▶ STOSB, MOVSB
 - ▶ use implicit arguments
 - ▶ can be integrated in loops LOOP, REP, REPNE
- ▶ **Call to the C standard library**
 - ▶ Require to link with gcc
 - ▶ Need to have another start label (`main`)
 - ▶ Parameters are written inside specific registers
 - ▶ Result is written in RAX

Bibliography

- ▶ This course corresponds to chapters *11* and *12* of the course book:
Assembly Language Step by Step (3rd Edition)