## Университет ИТМО

## Кафедра ВТ

Лабораторная работа №1

Низкоуровневое программирование

Выполнил: Федоров Сергей

Группа: P33113

Санкт-Петербург

2020 г.

## Задание лабораторной работы:

Задача стояла в том чтобы реализовать простенькую I/O библиотеку с следующими функциями:

| **FUNCTION** | **DEFINITION** |
| --- | --- |
| *GENERAL* | *General functions* |
| exit | Accepts an exit code and terminates the process |
| string\_lenght | Accepts a pointer to a string and returns its lenght |
| *OUTPUT* | *Output functions* |
| print\_string | Accepts a point to a null-terminated string and prints it to stdout |
| print\_char | Accepts a character code directly as its first argument and prints it to stdout |
| print\_newline | Prints a character with code 0xA |
| print\_uint | Prints an unsigned 8-byte integer in decimal format |
| print\_int | Prints a signed 8-byte integer in decimal format |
| *Input* | *Input functions* |
| read\_char | Read one character from stdin and **return** it. If end of input stream occurs, return 0 |
| read\_word | Accepts a buffer address and size as arguments. Reads next word from stdin. Returns 0 if word id too big for the buffer specified, otherwise returns a buffer address |
| *Processing* | *Processing functions* |
| parse\_uint | Accepts a null-terminated string and tries to parse an unsigned number from its start. Returns number in rax, characters count in rdx |
| parse\_int | Accepts a null-terminated string and tries to parse a signed number from its start. Returns number in rax, characters count in rdx (including possible sign) |
| string\_equals | Accepts two pointers to strings and compares them. Returns 1 if they are quals, 0 otherwise. |
| string\_copy | Accepts a pointer to a string, a pointer to a buffer, and buffer's lenght. Copies string to the destination. The destination address is returned if the string fits the buffer, 0 otherwise |

**Выполнение:**

%define *stdin* 0

%define *stdout* 1

%define *system\_exit* 60

%define *system\_read* 0

%define *system\_write* 1

%define *null* 0

%define *dec\_base* 10

%define *digit\_ascii\_offset* 0x30

%define *tab* 9 *; 0x9*

%define *CR* 13 *; 0xD*

%define *new\_line* 10 *; 0xA*

%define *space* 32 *; 0x20*

%define *minus* 45 *; 0x2D*

section .text

*; GENERAL FUNCTIONS*

*; args: rdi - exit code*

exit:

mov rax, *system\_exit*

syscall

*; args: rdi - pointer to the start of the string -> returns: rax - string's length*

string\_length:

xor rax, rax

.forward\_iterate:

cmp byte[rdi + rax], *null*

je .end

inc rax

jmp .forward\_iterate

.end:

ret

*; OUTPUT FUNCTIONS*

*; args: rdi - char itself -> Side effect*

print\_char:

xor rax, rax

push rsi

push rdi

mov rsi, rsp *; WHAT to write*

mov rdx, 1 *; HOW MUCH to write*

mov rax, system\_write, *; WHICH func to use*

mov rdi, stdout, *; WHERE to write*

syscall *; JUST DO IT*

pop rdi

pop rsi

ret

*; args: rdi - pointer to the start of the string -> Side effect*

print\_string:

xor rax, rax

push rdi

call string\_length

pop rdi

mov rsi, rdi *; WHAT to write*

mov rdx, rax *; HOW MUCH to write*

mov rax, system\_write *; WHICH func to use*

mov rdi, stdout *; WHERE to write*

syscall *; JUST DO IT*

ret

*; EMPTY args -> Side effect*

print\_newline:

xor rax, rax

mov rdi, new\_line *; SET new\_line char*

jmp print\_char

*; args: rdi - unsigned integer itself -> Side effect*

print\_uint:

mov rax, rdi

push r12 *; Save calee-saved regs*

push r13 *; Save calee-saved regs*

mov r12, rsp

mov r13, dec\_base

dec rsp

mov byte[rsp], null *; Final character of null-terminated string*

.digit\_loop:

xor rdx, rdx

div r13 *; Divide current acc by decimal base*

add rdx, digit\_ascii\_offset *; Convert resulted remainder to ASCII char*

dec rsp

mov byte[rsp], dl *; Save right-est digit (1 byte) to stack*

test rax, rax *; End of number?*

jz .output

jmp .digit\_loop

.output:

mov rdi, rsp

call print\_string

mov rsp, r12 *; Restore stack pointer*

pop r13 *; Restore R13*

pop r12 *; Restore R12*

ret

*; args: rdi - signed integer itself -> Side effect*

print\_int:

test rdi, rdi *; Check if RDI is positive*

jns print\_uint *; If it is, go ahead and print it*

push rdi

mov rdi, minus *; Print minus sign*

call print\_char

pop rdi *; Restore initial value*

neg rdi *; And negate it*

jmp print\_uint *; Print negated integer*

*; INPUT FUNCTIONS*

*; EMPTY args -> returns: rax - new char*

read\_char:

push null *; Placeholder for new char*

mov rax, system\_read *; WHICH func to use*

mov rdi, stdin *; WHERE to read from*

mov rsi, rsp *; WHERE to write to*

mov rdx, 1 *; HOW MUCH to read*

syscall *; JUST DO IT*

pop rax *; Save result*

ret

*; args: rdi - buffer address, rsi - buffer size -> returns: Right(rax - buffer address, rdx - word length) or Left(rax = 0)*

read\_word:

push r14

push r15

xor r14, r14

mov r15, rsi

dec r15

.space\_init\_loop:

push rdi

call read\_char *; Read new char (preserving rdi)*

pop rdi

cmp al, space *; Compare with space*

je .space\_init\_loop

cmp al, new\_line *; Compare with new\_line*

je .space\_init\_loop

cmp al, tab *; Compare with tab*

je .space\_init\_loop

cmp al, CR *; Compare with "Carruage Return"*

je .space\_init\_loop

test al, al

jz .correct\_ending

.read\_word\_loop:

mov byte[rdi + r14], al

inc r14

push rdi

call read\_char *; Read new char (preserving rdi)*

pop rdi

cmp al, space *; Compare with space*

je .correct\_ending

cmp al, new\_line *; Compare with new\_line*

je .correct\_ending

cmp al, tab *; Compare with tab*

je .correct\_ending

cmp al, CR *; Compare with "Carruage Return"*

je .correct\_ending

test al, al *; Compare with null*

jz .correct\_ending

cmp r14, r15 *; Check if not overflown*

je .incorrect\_ending

jmp .read\_word\_loop

.correct\_ending:

mov byte[rdi + r14], null *; Append null symbol*

mov rax, rdi *; Insert results*

mov rdx, r14

jmp .ending

.incorrect\_ending:

xor rax, rax *; Set result to 0*

jmp .ending

.ending:

pop r15 *; Restoring r14-r15*

pop r14

ret

*; PROCESSING FUNCTIONS*

*; args: rdi = integer string repr address -> returns: rax - number, rdx - count of characters*

parse\_uint:

push r8

mov r8, dec\_base

xor rax, rax

xor rcx, rcx

xor rdx, rdx

xor rsi, rsi

.parse\_char\_loop:

mov sil, [rdi + rcx], *; Move to char to sil*

test sil, sil

jz .ending

cmp sil, digit\_ascii\_offset *; Check boundaries between 0x30 and 0x39 (0..9)*

jl .ending

cmp sil, digit\_ascii\_offset + 9

jg .ending

sub sil, digit\_ascii\_offset *; Convert to number*

mul r8

add rax, rsi

inc rcx

jmp .parse\_char\_loop

.ending:

mov rdx, rcx

pop r8

ret

*; args: rdi = integer string repr address -> returns: rax - number, rdx - count of characters*

parse\_int:

cmp byte[rdi], minus *; Is negative?*

je .parse\_negative

jmp parse\_uint

.parse\_negative:

inc rdi *; Skip minus*

call parse\_uint *; Parse as positive*

cmp rdx, 0 *; If nothing, then do nothing*

je .error

neg rax *; Negate positively parsed rax*

inc rdx *; Adjust char count with minus char*

ret

.error:

xor rax, rax

ret

*; args: rdi = string1 address, rsi = string2 address -> returns: rax = 1 (true) or rax = 0 (false)*

string\_equals:

.comparison\_loop:

mov al, byte[rsi] *; Take byte*

cmp al, byte[rdi] *; Compare with another*

jne .not\_equal

inc rsi *; Proceed to the next char*

inc rdi

test al, al *; Check if not null*

jnz .comparison\_loop

jmp .equal

.equal:

mov rax, 1

ret

.not\_equal:

xor rax, rax

ret

*; args: rdi = source address, rsi = destinastion address, rdx = destination size -> returns: Right(rax = destination address) or Left(rax = 0)*

string\_copy:

push rdi

push rsi

push rdx

call string\_length *; Count source string length (preserving provided args)*

pop rdx

pop rsi

pop rdi

cmp rax, rdx *; Compare source and dest sizes*

jae .length\_exceed *; If exceeds - do nothing*

push rsi

.filling\_loop:

mov dl, byte[rdi], *; Take byte from source*

mov byte[rsi], dl *; Move it to dest*

inc rdi *; Increment byte addresses*

inc rsi

test dl, dl *; Check if reached null-terminator*

jnz .filling\_loop

pop rax *; Fill rax with dest address*

ret

.length\_exceed:

xor rax, rax

ret

**Выводы:**

Учитывая то что мой опыт общения с любыми низкоуровневыми языками, а в особенности с assembly был минимален, весьма НЕ странно что этот опыт был весьма болезненным, особенно с такими сжатыми сроками.

Наблюдения:

1. Даже на базовые, казалось бы действия приходится тратить достаточно много времени и кол-ва строчек.

2. Очень много логики обычно предоставленной в стандартных библиотеках тут отсутвует, что заставляет искать и узнавать что-то на каждом шагу.

3. Видя что мы буквально программируем команды для процессора, становится очевидно, что при должно желании, можно достаточно сильно оптимизировать выполнение программы, как в плане скорости, так и в плане памяти.

