Федеральное государственное автономное образовательное учреждение высшего образования
Университет ИТМО

Кафедра Вычислительной Техники

Дисциплина: Низкоуровневое программирование

Лабораторная работа №1

Выполнил: **Доморацкий Эридан Алексеевич**

Группа: **P33113**

Преподаватель: **Логинов Иван Павлович**

2020г

Задание

Before we start doing anything cool looking, we are going to ensure we won't have to code the same basic routines over and over again. As for now, we do not have anything; even getting keyboard input is a pain. So, let's build a small library for basic input and output functions.

First you have to read Intel docs [15] for the following instructions (remember, they are all described in details in the second volume):

- `xor`
- `jmp`, `ja`, and similar ones
- `cmp`
- `mov`
- `inc`, `dec`
- `add`, `imul`, `mul`, `sub`, `idiv`, `div`
- `neg`
- `call`, `ret`
- `push`, `pop`

These commands are core to us and you should know them well. As you might have noticed, Intel 64 supports thousands of commands. Of course, there is no need for us to dive there. Using system calls together with instructions listed earlier will get us pretty much anywhere.

You also have to read docs for the `read` system call. Its code is 0; otherwise it is similar to `write`. Refer to the Appendix C in case of difficulties.

Edit `lib.inc` and provide definitions for the functions instead of stub `xor rax, rax` instructions. Refer to Table 2-2 for the required functions' semantics. We do recommend implementing them in the given order because sometimes you will be able to reuse your code by calling functions you have already written.

Выполнение

```
; vim: syntax=nasm

; Выполняет syscall с сохранением регистров
do_syscall:
    push rcx
    push rsp

    syscall

    pop rsp
    pop rcx
    ret


; Принимает код возврата и завершает текущий процесс
exit:
    mov rax, 60
    syscall
    ret


; Принимает указатель на нуль-терминированную строку, возвращает её длину
string_length:
    xor rax, rax

.loop:
    cmp byte [rdi + rax], 0
    je .end

    inc rax
    jmp .loop

.end:
    ret


; Принимает указатель на нуль-терминированную строку, выводит её в stdout
print_string:
```

```
    push rax
    push rdi
    push rsi
    push rdx

    call string_length
    mov rdx, rax
    mov rax, 1
    mov rsi, rdi
    mov rdi, 1
    call do_syscall

    pop rdx
    pop rsi
    pop rdi
    pop rax
    ret


; Принимает код символа и выводит его в stdout
print_char:
    push rax
    push rdi
    push rsi
    push rdx

    dec rsp
    mov byte [rsp], dil
    mov rax, 1
    mov rdi, 1
    mov rsi, rsp
    mov rdx, 1
    call do_syscall
    inc rsp

    pop rdx
    pop rsi
    pop rdi
    pop rax
    ret


; Переводит строку (выводит символ с кодом 0xA)
print_newline:
    push rdi

    mov dil, 0x0a
    call print_char

    pop rdi
    ret


; Выводит беззнаковое 8-байтовое число в десятичном формате
; Совет: выделите место в стеке и храните там результаты деления
; Не забудьте перевести цифры в их ASCII коды.
print_uint:
    push rax
    push rbx
    push rcx
    push rdx
    push rdi
    xor rcx, rcx
    mov rax, rdi

    dec rsp
    mov byte [rsp], 0

    cmp rax, 0
    je .loop_body

.loop:
    cmp rax, 0
    je .end

.loop_body:
    xor rdx, rdx
    mov rbx, 10
```

```
    div rbx
    add dl, '0'
    dec rsp
    mov byte [rsp], dl
    inc cx
    jmp .loop

.end:
    mov rdi, rsp
    call print_string
    add rsp, rcx

    inc rsp
    pop rdi
    pop rdx
    pop rcx
    pop rbx
    pop rax
    ret


; Выводит знаковое 8-байтовое число в десятичном формате
print_int:
    push rdi

    cmp rdi, 0
    jge .print

    push rdi
    mov rdi, '-'
    call print_char
    pop rdi
    neg rdi

.print:
    call print_uint

    pop rdi
    ret


; Принимает два указателя на нуль-терминированные строки, возвращает 1 если они равны, 0 иначе
string_equals:
    push rbx

    xor rax, rax
.loop:
    mov bl, byte [rsi + rax]
    cmp byte [rdi + rax], bl
    jne .fail

    cmp byte [rdi + rax], 0
    je .succ

    inc rax
    jmp .loop

.fail:
    xor rax, rax
    jmp .end

.succ:
    mov rax, 1

.end:
    pop rbx
    ret


; Читает один символ из stdin и возвращает его. Возвращает 0 если достигнут конец потока
read_char:
    push rdi
    push rsi
    push rdx

    dec rsp
    xor rax, rax
    mov rdi, 0
```

```
    mov rsi, rsp
    mov rdx, 1
    call do_syscall

    cmp rax, 0
    je .stream_end

    xor rax, rax
    mov al, byte [rsp]
    jmp .end

.stream_end:
    xor rax, rax
    jmp .end

.end:
    inc rsp
    pop rdx
    pop rsi
    pop rdi
    ret


; Принимает: адрес начала буфера, размер буфера
; Читает в буфер слово из stdin, пропуская пробельные символы в начале, .
; Пробельные символы это пробел 0x20, табуляция 0x9 и перевод строки 0xA.
; Останавливается и возвращает 0 если слово слишком большое для буфера
; При успехе возвращает адрес буфера в rax, длину слова в rdx.
; При неудаче возвращает 0 в rax
; Эта функция должна дописывать к слову нуль-терминатор
read_word:

.whitespaces:
    call read_char

    cmp rax, ' '
    je .whitespaces
    cmp rax, 0x9
    je .whitespaces
    cmp rax, 0xA
    je .whitespaces

    xor rdx, rdx
.loop:
    cmp rdx, rsi
    je .fail

    cmp rax, ' '
    je .succ
    cmp rax, 0x9
    je .succ
    cmp rax, 0xA
    je .succ
    cmp rax, 0
    je .succ

    mov byte [rdi + rdx], al
    inc rdx

    call read_char
    jmp .loop

.fail:
    xor rax, rax
    jmp .end

.succ:
    mov rax, rdi
    mov byte [rdi + rdx], 0

.end:
    ret


; Принимает указатель на строку, пытается
; прочитать из её начала беззнаковое число.
; Возвращает в rax: число, rdx : его длину в символах
; rdx = 0 если число прочитать не удалось
```

```
parse_uint:
    push rbx

    xor rdx, rdx
    xor rax, rax
    xor rbx, rbx
.loop:
    mov bl, byte [rdi + rdx]
    sub bl, '0'
    jl .end
    cmp bl, 9
    jg .end

    push rdx
    mov rdx, 10
    mul rdx
    pop rdx
    add rax, rbx

    inc rdx
    jmp .loop

.end:
    pop rbx
    ret


; Принимает указатель на строку, пытается
; прочитать из её начала знаковое число.
; Если есть знак, пробелы между ним и числом не разрешены.
; Возвращает в rax: число, rdx : его длину в символах (включая знак, если он был)
; rdx = 0 если число прочитать не удалось
parse_int:
    push rdi

    cmp byte [rdi], '-'
    je .negative

    cmp byte [rdi], '+'
    je .positive

    call parse_uint
    jmp .end

.negative:
    inc rdi
    call parse_uint
    neg rax
    inc rdx
    jmp .end

.positive:
    inc rdi
    call parse_uint
    inc rdx
    jmp .end

.end:
    pop rdi
    ret


; Принимает указатель на строку, указатель на буфер и длину буфера
; Копирует строку в буфер
; Возвращает длину строки если она умещается в буфер, иначе 0
string_copy:
    push rbx
    xor rax, rax

.loop:
    cmp rax, rdx
    je .fail

    mov bl, byte [rdi + rax]
    mov byte [rsi + rax], bl
    cmp byte [rsi + rax], 0
    je .end
```

```
    inc rax
    jmp .loop

.fail:
    xor rax, rax

.end:
    pop rbx
    ret
```

Вывод

При выполнении данной лабораторной работы были изучены такие аспекты низкоуровневого программирования, как обращение к системным процедурам посредством инструкции процессора syscall, работа со стеком, работа с регистрами процессора архитектуры Intel x86 и соглашения о вызове процедур.