Министерство образования Республики Беларусь

Учреждение Образования

«Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Лабораторная работа № 7

по дисциплине «Программирование на языке ассемблера»

на тему «Загрузка и выполнение программ. Работа с памятью»

вариант №8

|  |  |  |
| --- | --- | --- |
| Выполнил  студент гр.250502  Грибовская А.А. |  | Проверил  Туровец Н.О. |

Минск 2023

**Цель работы:** Ознакомиться с загрузкой и выполнением программ, рассмотреть работу с памятью.

**Теоретические сведения**

Для выполнения работы требуется рассмотреть следующие элементы языка ассемблера и операционной системы:

*1. Управление памятью.*

При запуске программы в DOS, ей выделяется все доступная память, поэтому доступно использование, например, памяти за концом программы практически до конца сегмента кода. Однако для загрузки других программ из текущей потребуется свободная память и наиболее простой вариант ее получения – сократить выделенный текущей программе блок памяти до минимума с помощью функции DOS 4Ah.

-- Функция DOS 4Ah (INT 21h) – изменить размер блока памяти:

Ввод:

АН = 4Ah,

ВХ = новый размер в 16-байтных параграфах,

ES = сегментный адрес модифицируемого блока.

Вывод:

если CF = 1, то есть ошибки (в AX – код ошибки):

-- 07 – блоки управления памятью разрушены,

-- 08 – не хватает памяти (при увеличении, в ВХ = максимальный размер, доступный для этого блока),

-- 09 – ES содержит неверный адрес.

Если CF = 0, то операция выполнена успешно.

Также доступно выделение и удаление дополнительных блоков памяти:

-- Функция DOS 48h (INT 21h) – выделить блок памяти:

Ввод:

АН = 48h

ВХ = размер блока в 16-байтных параграфах.

Эта функция с ВХ = FFFFh используется для определения размера самого большого доступного блока памяти.

Вывод:

если CF = 1, то есть ошибки (в AX – код ошибки):

-- 07 – блоки управления памятью разрушены,

-- 08 – не хватает памяти (ВХ = размер максимального доступного блока).

Если CF = 0, то операция выполнена успешно,

АХ = сегментный адрес выделенного блока.

-- Функция DOS 49h (INT 21h) – освободить блок памяти:

Ввод:

АН = 49h

ES = сегментный адрес освобождаемого блока

Вывод:

если CF = 1, то есть ошибки (в AX – код ошибки):

-- 07 – блоки управления памятью разрушены,

-- 09 – ES содержит неверный адрес.

Если CF = 0, то операция выполнена успешно.

*2. Загрузка и выполнение программ.*

Для загрузки и выполнения программ требуется использовать функцию DOS 4Bh (INT 21h) – загрузить и выполнить программу:

Ввод:

АН = 4Bh,

AL = подфункции:

AL = 00h — загрузить и выполнить;

AL = 01h — загрузить и не выполнять:

DS:DX – адрес ASCIZ-строки с полным именем программы,

ES:BX – адрес блока параметров ЕРВ:

+00h (слово) – сегментный адрес окружения, которое будет скопировано для нового процесса (или 0, если используется текущее окружение);

+02h (4 байта) – адрес командной строки для нового процесса;

+06h (4 байта) – адрес первого FCB для нового процесса;

+0Ah (4 байта) – адрес второго FCB для нового процесса;

+0Eh (4 байта) – здесь будет записан SS:SP нового процесса после его завершения (только для AL = 01);

+12h (4 байта) – здесь будет записан CS:IP (точка входа) нового процесса после его завершения (только для AL = 01).

AL = 03h – загрузить как оверлей:

DS:DX — адрес ASCIZ-строки с полным именем программы,

ES:BX — адрес блока параметров:

+00h (слово) – сегментный адрес для загрузки оверлея,

+02h (слово) – число, которое будет использовано в командах, использующих непосредственные сегментные адреса (обычно то же самое число, что и в предыдущем поле или 0 для com-файлов).

AL = 05h – подготовиться к выполнению (DOS 5.0+):

DS:DX – адрес следующей структуры:

+00h(слово) – 00h,

+02h (слово):

бит 0 – exe-программа,

бит 1 – программа-оверлей,

+04h (4 байта) – адрес ASCIZ-строки с именем новой программы;

+08h (слово) – сегментный адрес PSP новой программы;

+0Ah (4 байта) – точка входа новой программы;

+0Eh (4 байта) – размер программы, включая PSP.

Вывод:

Если CF = 1, то произошла ошибка (в АХ = код ошибки):

-- 02h – файл не найден,

-- 05h – доступ к файлу запрещен,

-- 08h – не хватает памяти,

-- 0Ah – неправильное окружение,

-- 0Bh – неправильный формат.

Если CF = 0, то операция успешно выполнена:

ВХ и DX изменены.

Особенности:

Для подфункций 00 и 01 требуется, чтобы было достаточно свободной памяти для загрузки программы, поэтому com-программы должны воспользоваться функцией DOS 4Ah для уменьшения отведенного им блока памяти до минимально необходимого.

При вызове подфункции 03, DOS загружает оверлей в память, выделенную текущим процессом, так что exe-программы должны убедиться, что ее достаточно.

Эта функция игнорирует расширение файла и различает exe- и com-файлы по первым двум байтам заголовка («MZ» для exe-файлов).

*3. Оверлейные модули.*

Оверлей – это часть исполняемой программы (обычно процедура, хотя это может быть полностью самостоятельная программа со своими сегментами данных и стека), которая по мере необходимости загружается в определенную область памяти. Различные оверлейные модули могут загружаться в одно и то же место, перекрывая предыдущий код, что позволяет экономить память, но снижает быстродействие программы при частых загрузках.

**Код программы (.exe)**

.model small

.stack 100h

.data

blockEPB dw 0

cmd\_offset dw offset cmd, 0

fcb1 dw 005ch, 0

fcb2 dw 006ch, 0

cmd db 10,' '

cmd\_text db 125 dup (0), 0dh, '$'

EPBlen dw $-blockEPB *; вычисляет длину сегмента данных и сохраняет ее*

path db "lab7.exe", 0 *; путь к файлу*

num dw ?

cmdLen db 0

minus dw 0

no\_cmd\_msg db "you must pass one argument to the command line!", '$'

error\_msg db "invalid comand line argument! it must be number from 1 to 255", '$'

realloc\_error\_msg db "error while realloc occured!", '$'

run\_error\_msg db "error while running new copy occured!", '$'

badNumFlag dw ?

badCMDflag dw ?

NumBuffer dw 0

string db 7, 7 dup('$')

string\_end = $ - 1

start\_msg db "programm started. the number of copy is ", '$'

end\_msg db "programm ended. the number of copy is ", '$'

data\_segment\_size = $ - blockEPB *; размер сегмента данных*

.code

start:

mov ax, es *; значение текущего сегмента кода*

mov bx, (code\_segment\_size/16+1)+(data\_segment\_size/16+1)+256/16+256/16 *; для выделения нового сегмента данных, размер сегмента кода и размер сегмента данных, учитывается размер стека и размер дополнительных сегментов, всё это складывается и увеличивается на 1*

mov ah, 4Ah *; выделение памяти*

int 21h

jc reallocError

mov ax, @data

mov es, ax *; чтобы переключить работу программы на новый сегмент данных*

xor ch, ch

mov cl, ds:[80h] *; длина командной строки (количество аргументов)*

cmp cl, 1

jl noCMD

dec cl *; длина командной строки уменьшается на 1, так как первый аргумент это имя программы*

mov cmdLen, cl

mov si, 81h ; ->cmd

inc si *; пропускаем пробел между именем и первым аргументом*

lea di, cmd\_text

rep movsb *;копируем содержимое командной строки в переменную*

mov ds, ax

call workWithCMD

cmp badCMDflag, 1

je exit

lea dx, start\_msg

call output

lea dx, cmd\_text

call output

dec num

cmp num, 0

je exit

call getNewCMD

mov bx,offset blockEPB

mov ax,ds

mov word ptr[blockEPB+4],ax ; сегмент командной строки

mov ax,cs

mov word ptr[blockEPB+8],ax ; сегмент 1-го fcb(file control block)

mov word ptr[blockEPB+12],ax ;сегмент второго fcb

mov ax, 4B00h *; для запуска новой копии программы*

lea dx, path *; загружается адрес переменной, содержащей путь к файлу*

lea bx, blockEPB

int 21h *; для запуска новой копии программы с обновленным аргументом командной строки*

jc runProgError

lea dx, end\_msg

call output

inc num

call numberToString

lea dx, [di+1]

call output

call new\_line

exit:

mov ax, 4C00h *; для завершения программы*

int 21h

noCMD:

mov ds, ax

lea dx, no\_cmd\_msg

call output

jmp exit

reallocError:

lea dx, realloc\_error\_msg

call output

jmp exit

runProgError:

lea dx, run\_error\_msg

call output

jmp exit

getNewCMD proc

push ax

push si

lea si, cmd\_text

l1:

mov al, [si]

cmp al, 0 *; сравнение al с 0 терминатором (конец строки)*

je decNumber

inc si

jmp l1

decNumber:

dec si

mov al, [si]

cmp al, '0'

je prevSym

back:

sub al, 1 *; вычисляется длина строки*

mov [si], al

jmp exitGetNewCMD

prevSym:

mov al, '9'

mov [si], al

dec si

mov al, [si]

cmp al, '1'

je nine

jmp back

nine:

mov al, '9'

mov [si], al

inc si

mov al, 0

mov [si], al

exitGetNewCMD:

pop si

pop ax

ret

endp getNewCMD

new\_line proc

mov dl, 10

mov ah, 2

int 21h

mov dl, 10

mov ah, 2

int 21h

ret

endp new\_line

workWithCMD proc

push dx

mov badCMDflag, 0

call stringToNumber

cmp badNumFlag, 1

je cmdError

cmp num, 255

jg cmdError

cmp num, 1

jl cmdError

exitWorkWithCMD:

pop dx

ret

cmdError:

mov badCMDflag, 1

lea dx, error\_msg

call output

jmp exitWorkWithCMD

endp workWithCMD

output proc

push ax

mov ah, 09h *; для вывода строки*

int 21h

pop ax

ret

endp output

stringToNumber proc

push ax

push bx

push dx

push si

mov badNumFlag, 0

lea si, cmd\_text

mov num, 0

makeNum:

xor ax, ax

mov al, [si]

cmp al, 0

je exitStringToNumber

cmp al, '0'

jl badNum

cmp al, '9'

jg badNum

sub al, '0'

mov bx, ax

mov ax, num

mov dx, 10

mul dx

mov num, ax

add num, bx

inc si

jmp makeNum

exitStringToNumber:

pop si

pop dx

pop bx

pop ax

ret

badNum:

mov badNumFlag, 1

jmp exitStringToNumber

endp stringToNumber

numberToString proc

push ax

push cx

push dx

mov ax, num

std

lea di, string\_end - 1

mov cx,10

repeat:

xor dx,dx

idiv cx *; целочисленное деление*

xchg ax,dx *; обменивает значениями содержимое регистров*

add al,'0'

stosb *; для сохранения байта*

xchg ax,dx

or ax,ax

jne repeat

pop dx

pop cx

pop ax

ret

endp numberToString

code\_segment\_size =$-start *; размер кодового сегмента на основе разницы между текущей позицией счетчика команд и адресом начала сегмента*

end start

**Вывод программы**

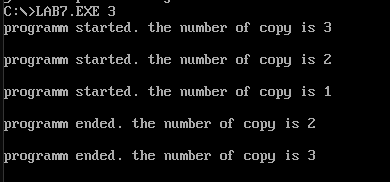
****

Рисунок 1 – Результат работы программы

**Теория**

**1. Управление памятью.**

При запуске программы в DOS, ей выделяется все доступная память, поэтому доступно использование памяти за концом программы практически до конца сегмента кода. Однако для загрузки других программ из текущей потребуется свободная память и наиболее простой вариант ее получения – сократить выделенный текущей программе блок памяти до минимума с помощью функции DOS 4Ah.

-- Функция DOS *4Ah* (INT 21h) – изменить размер блока памяти

Также доступно выделение и удаление дополнительных блоков памяти:

-- Функция DOS *48h* (INT 21h) – выделить блок памяти

-- Функция DOS *49h* (INT 21h) – освободить блок памяти

**2. Загрузка и выполнение программ.**

Для загрузки и выполнения программ требуется использовать функцию DOS *4Bh* (INT 21h) – загрузить и выполнить программу

AL = подфункции:

AL = *00*h — загрузить и выполнить

AL = *01*h — загрузить и не выполнять

AL = *03*h – загрузить как оверлей

AL = *05*h – подготовиться к выполнению (DOS 5.0+)

Особенности:

Для подфункций *00* и *01* требуется, чтобы было достаточно свободной памяти для загрузки программы, поэтому com-программы должны воспользоваться функцией DOS 4Ah для уменьшения отведенного им блока памяти до минимально необходимого.

При вызове подфункции *03*, DOS загружает оверлей в память, выделенную текущим процессом, так что exe-программы должны убедиться, что ее достаточно.

**3. Оверлейные модули.**

*Оверлей* – это часть исполняемой программы, которая по мере необходимости загружается в определенную область памяти. Различные оверлейные модули могут загружаться в одно и то же место, перекрывая предыдущий код, что позволяет экономить память, но снижает быстродействие программы при частых загрузках.

; OVERLAY1.ASM

DSEG SEGMENT ; сегмент данных оверлея

DSEG ENDS

CSEG SEGMENT PARA PUBLIC 'CODE' ; сегмент кода оверлея

OVERLAY PROC FAR ; процедура оверлея

Программа сохраняет текущее значение сегмента в ax, вычисляет адрес сегмента для нового сегмента данных и устанавливает его в bx.

int 21h используется для выделения памяти для нового сегмента данных. Если происходит ошибка, программа переходит к метке reallocError.

Затем программа настраивает es для указания на новый сегмент данных и обрабатывает аргументы командной строки.

Вызывается workWithCMD для преобразования аргумента командной строки в число и проверки на ошибки.

Если ошибок нет, программа продолжает выполнение, запуская новую копию самой себя с другим аргументом командной строки.

Программа выводит сообщения о начале и окончании, увеличивает число, преобразует его в строку и отображает новое число.

+04h (4 байта) – адрес ASCIZ-строки с именем новой программы;

+08h (слово) – сегментный адрес PSP новой программы

+12h (4 байта) – здесь будет записан CS:IP (точка входа) нового процесса после его завершения (только для AL = 01)

AL = 05h – подготовиться к выполнению (DOS 5.0+):

AL = 00h — загрузить и выполнить; AL = 01h — загрузить и не выполнять:

.model small

.stack 100h

.data

isEmpty dw 0

data\_size dw $-data ; узнаём размер данных

.code

old\_int dd 0 ;Адрес старого обработчика

new\_handler proc far

cli ;Запрещаем прерывания

pushf ;передать управление старому обработчику 10 h

push ax

in al, 21h ;Читаем маску мастера

and al, 10111111b ;Разрешаем прерывания клавиатуры

out 21h, al ;Записываем маску мастера

pop ax

call dword ptr cs:[old\_int] ;Вызываем старый обработчик

pusha ;Помещает в стек регистры общего назначения

push ds

push es

mov ax, @data

mov es, ax

mov ax, 0B800h ; video-memmory

mov ds, ax

xor si, si

mainLoop1:

cmp si, 4000 ; сравниваем индекс с пороговым значением

je continue ; если они совпали, то идём красить скобки без пары

inc si

mov al, ds:[si] ; поместили в al символ с позиции экрана ds:[si]

cmp al, 4

je colorPrev

afterColorPrev:

inc si ;Увеличивает SI на 2, т.к. пропускаем атрибуты

jmp mainLoop1

colorPrev:

mov ds:[si], 7

jmp afterColorPrev

continue:

xor si, si ; DS:SI будет указывать на экран

mainLoop:

cmp si, 4000 ; сравниваем индекс с пороговым значением

je letsColorIt ; если они совпали, то идём красить скобки без пары

mov al, ds:[si] ; поместили в al символ с позиции экрана ds:[si]

cmp al, '('

je addToStack1

cmp al, '{'

je addToStack2

cmp al, '['

je addToStack3

cmp al, ')'

je closeBracket

cmp al, '}'

je closeBracket

cmp al, ']'

je closeBracket

afterAddToStackAndOrCloseBracket:

inc si ;Увеличивает SI на 2, т.к. пропускаем атрибуты

inc si

jmp mainLoop

addToStack1:

push si ; запомнили координату

mov al, ')'

push ax ; запомнили символ

mov bx, isEmpty ;

inc bx ; увеличили счётчик скобок на 1

mov isEmpty, bx ;

jmp afterAddToStackAndOrCloseBracket

addToStack2:

push si ; запомнили координату

mov al, '}'

push ax ; запомнили символ

mov bx, isEmpty ;

inc bx ; увеличили счётчик скобок на 1

mov isEmpty, bx ;

jmp afterAddToStackAndOrCloseBracket

addToStack3:

push si ; запомнили координату

mov al, ']'

push ax ; запомнили символ

mov bx, isEmpty ;

inc bx ; увеличили счётчик скобок на 1

mov isEmpty, bx ;

jmp afterAddToStackAndOrCloseBracket

closeBracket:

mov bx, isEmpty ;

cmp bx, 0 ; если открывающей скобки нет

je colorCloseBracket ; то значит закрывающая скобка без пары => подсвечиваем её красным цветом

pop bx ; извлекаем символ последней скобки

cmp bl, al ; сравниваем символ последней закрывающей скобки

jne resetStack ; если символы скобки не совпали, то возвращаем символ, извлечённый из стека, обратно в стек и красим текущую скобку в красный цвет

mov bx, isEmpty ;

dec bx ; в противном случае уменьшаем количество скобочек на 1

mov isEmpty, bx ;

pop bx ; удаляем из стека координату открывающей скобки

jmp afterAddToStackAndOrCloseBracket

resetStack:

push bx ; возвращаем обратно символ открывающей скобки

inc si ;

mov ds:[si], 4 ; изменили цвет скобки на красный

dec si ;

push si ; запомнили координату текущей скобки

cmp al, ')' ; если это не закрывающая круглая скобка

jne otherBrackets ; то прыгаем на метку, на которой мы сделаем эту квадратную или фигурную скобку открывающей (')' - 1 = '(')

sub al, 1 ; в противном случае делаем её открывающей круглой скобкой

jmp pushMe ; прыгаем на метку, где запомним её символ

otherBrackets:

sub al, 2 ; ']' - 2 = '['

pushMe:

push ax ; запомнили символ этой скобки

mov bx, isEmpty ;

inc bx ; увеличили счётчик скобок на 1

mov isEmpty, bx ;

jmp afterAddToStackAndOrCloseBracket

colorCloseBracket:

inc si ;

mov ds:[si], 4 ; изменили цвет скобки на красный

dec si ;

jmp afterAddToStackAndOrCloseBracket

letsColorIt:

mov bx, isEmpty

colorLoop:

cmp bx, 0

je end\_handler

pop si ; извлекли координату неправильно расставленной скобки из стека

pop si

inc si ; стали на атрибут символа

mov ds:[si], 4 ; изменили цвет скобки на красный

dec bx

jmp colorLoop

end\_handler:

mov isEmpty, bx

pop es

pop ds

popa

sti ;Разрешаем прерывания

iret ;Сначала извлекате IP из стека, затем CS, и в конце регистр флагов (поэтому не вызываем перед ним popf)

new\_handler endp

start:

mov ax, @data

mov ds, ax

mov ah, 35h ;Функция считывания вектора прерывания

mov al, 09h ;leg number9

int 21h ;На выходе ES:BX - адрес обработчика прерывания

mov word ptr cs:[old\_int], bx ;Запоминаем наш обработчик

mov word ptr cs:[old\_int+2], es

mov ah, 25h ;Функция установки нового обработчика

mov al, 09h ;Номер прерывания

push cs ;Устанавливаем новый обработчик прерывания 09h

pop ds ;Помещаем в DS CS (подготавливаем сегмент для адреса нового обработчика)

mov dx, offset new\_handler ;Смещение нового обработчика

int 21h ;DS:DX - адрес нового обработчика

mov ah, 31h ;Функция, которая оставляет программу резидентной

mov al, 00h ;Код выхода

mov dx, (code\_size / 16) + (data\_size / 16) + 16 + 16 + 2 ;(Размер в 16-байтных сегментах) = размер кода + размер данных + размер стрека + размер PSP + 1 параграф для переменных code\_size, data\_size

размер резидента в 16-байтных параграфов считая от начал psp

int 21h

code\_size dw $-code ; узнаём размер кода

end start p