

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №7
по дисциплине «Операционные системы»
Тема: Построение модуля оверлейной структуры

Студент гр. 9383

Нистратов Д.Г.

Преподаватель

Ефремов М.А.

Санкт-Петербург

2021

Постановка задачи.

Исследование возможности построения загрузочного модуля оверлейной структуры. Исследуется структура оверлейного сегмента и способ загрузки и выполнения оверлейных сегментов. Для запуска вызываемого оверлейного модуля используется функция 4b03h прерывания int 21h. Все загрузочные и оверлейные модули находятся в одном каталоге.

Последовательность действий

Шаг 1. Для выполнения лабораторной работы необходимо написать и отладить программный модуль типа .EXE, который выполняет функции:

- 1) Освобождает память для загрузки оверлеев.
- 2) Читает размер файла оверлея и запрашивает объем памяти, достаточный для его загрузки.
- 3) Файл оверлейного сегмента загружается и выполняется.
- 4) Освобождается память, отведенная для оверлейного сегмента.
- 5) Затем действия 1)-4) выполняются для следующего сегмента.

Шаг 2. Также необходимо написать и отладить оверлейный сегмент. Оверлейный сегмент выводит адрес сегмента, в который он загружен.

Шаг 3. Запустите отлаженное приложение. Оверлейные сегменты должны загружаться с одного и того же адреса, перекрывая друг друга.

Шаг 4. Запустите приложение с другого каталога. Приложение должно быть выполнено успешно.

Шаг 5. Запустите приложение в случае, когда одного оверлея нет в каталоге. Приложение должно закончиться аварийное.

Шаг 6. Ответьте на контрольные вопросы.

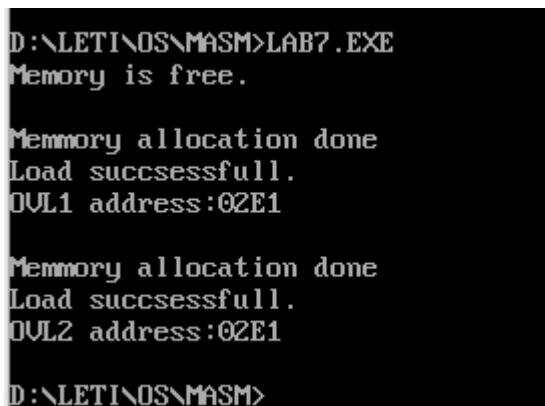
Выполнение работы.

Шаг 1. Был написан и отлажен программный модуль типа .EXE, который выполняет функции:

- 1) Освобождает память для загрузки оверлеев.
- 2) Читает размер файла оверлея и запрашивает объем памяти, достаточный для его загрузки.
- 3) Файл оверлейного сегмента загружается и выполняется.
- 4) Освобождается память, отведенная для оверлейного сегмента.
- 5) Затем действия 1)-4) выполняются для следующего сегмента.

Шаг 2. Был описан и отлажен оверлейный сегмент, выводящий адрес сегмента, в который они загружены.

Шаг 3. Программа была запущена в каталоге с исходными файлами, также была проведена проверка запуска двух оверлейных сегментов с одного участка памяти, перекрывая друг друга. См. Изображение 1



```
D:\LETI\OS\MASM>LAB7.EXE
Memory is free.

Memory allocation done
Load successfull.
OVL1 address:02E1

Memory allocation done
Load successfull.
OVL2 address:02E1

D:\LETI\OS\MASM>
```

Изображение 1 – Работа программы lab7.asm

Шаг 4. Программа была запущена из другого каталога. См. Изображение 2

```
D:\LETI\OS>LAB7.EXE
Memory is free.

Memmory allocation done
Load successfull.
OVL1 address:02E1

Memmory allocation done
Load successfull.
OVL2 address:02E1
```

Изображение 2 – запуска программы lab7.asm из другого каталога

Шаг 5. Программа была запущена с удаленным файлом ovl2.ovl, чтобы убедиться в работоспособности вывода критических ошибок. См. *Изображение 3*

```
D:\LETI\OS>LAB7.EXE
Memory is free.

Memmory allocation done
Load successfull.
OVL1 address:02E1

File not found.
File not found.

D:\LETI\OS>_
```

Изображение 3 – отсутствие файла ovl2.ovl при запуске программы lab7.asm

Шаг 6. Ответы на вопросы:

1. Как должна быть устроена программа, если в качестве оверлейного сегмента использовать модуль .COM?

В записи значений регистров в стек надо записать значения регистра CS в DS, так как адрес сегмента данных совпадает с сегментом кода. Также необходимо добавить 100h, т.к сегменты настроены в PSP.

Заключение.

В ходе лабораторной работы были изучены структуры оверлейного сегмента, способы загрузки и выполнения оверлейных сегментов. А также были написаны программы выполняющие погрузку нескольких модулей.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab7.asm

```
AStack  SEGMENT STACK
```

```
        DW 64 DUP(?)
```

```
AStack  ENDS
```

```
DATA SEGMENT
```

```
    parametr_block dw 0
```

```
        db 0
```

```
        db 0
```

```
        db 0
```

```
    CMD_LINE db 1h,0DH
```

```
    PATH_STR db 128 dup(0), '$'
```

```
    OVL_ADDRESS dd 0
```

```
    OVL1_FILE db "ovl1.ovl", 0
```

```
    OVL2_FILE db "ovl2.ovl", 0
```

```
    CURRENT_FILE dw 0
```

```
    KEEP_SS dw 0
```

```
    KEEP_SP dw 0
```

```
    KEEP_PSP dw 0
```

```
    TEMP db 64 dup(0)
```

```
    MEMORY_BLOCK_ERROR db "Memory control block destroyed. ", 0DH, 0AH, '$'
```

```
    LOW_MEMORY db "Not enough memory. ", 0DH, 0AH, '$'
```

```
    WRONG_PTR db "Invalid memory block address. ", 0DH, 0AH, '$'
```

```
    MEMORY_FREE_SUCCESS db "Memory is free. ", 0DH, 0AH, 0AH, '$'
```

```
    WRONG_FUNC_NUMBER db "Wrong function number.", 0DH, 0AH, '$'
```

```
    FILE_NOT_FOUND_LOAD db "File not found.", 0DH, 0AH, '$'
```

```
    PATH_NOT_FOUND_LOAD db "Path not found.", 0DH, 0AH, '$'
```

```
    TOO_MUCH_FILES db "Too much files opened.", 0DH, 0AH, '$'
```

```
    ACCESS_ERROR db "Access error.", 0DH, 0AH, '$'
```

```
    NOT_ENOUGH_MEMORY db "Not enough memory.", 0DH, 0AH, '$'
```

```
    WRONG_ENVIRONMENT db "Wrong environment string.", 0DH, 0AH, '$'
```

FILE_NOT_FOUND_ALL db "File not found.", 0DH, 0AH, '\$'

PATH_NOT_FOUND db "Path not found.", 0DH, 0AH, '\$'

LOAD_SUCCESSFUL db "Load successfull.", 0DH, 0AH, '\$'

ALLOCATION_SUCCESSFUL db "Memmory allocation done", 0DH, 0AH, '\$'

KEEP_FLAG db 0

KEEP_DATA db 0

DATA ENDS

CODE SEGMENT

ASSUME CS:CODE, DS:DATA, SS:AStack

WRITE proc near

push ax

mov ah, 9h

int 21h

pop ax

ret

WRITE endp

FREE_MEMORY proc near

push ax

push bx

push dx

mov ax, offset MAIN_ENDS

mov bx, offset KEEP_DATA

add ax, bx

mov bx, 10h

xor dx, dx

div bx

mov bx, ax

add bx, dx

add bx, 100h

mov ah, 4ah

```

int 21h

jnc MEM_S_P
mov KEEP_FLAG, 1

cmp ax, 7
je M_B_D
cmp ax, 8
je L_M
cmp ax, 9
je I_A

M_B_D:
    mov dx, offset MEMORY_BLOCK_ERROR
    call WRITE
    jmp memory_free_end

L_M:
    mov dx, offset LOW_MEMORY
    call WRITE
    jmp memory_free_end

I_A:
    mov dx, offset WRONG_PTR
    call WRITE
    jmp memory_free_end

MEM_S_P:
    mov dx, offset MEMORY_FREE_SUCCESS
    call WRITE

memory_free_end:
    pop dx
    pop bx
    pop ax
    ret

FREE_MEMORY endp

PATH_FIND proc near
    push ax
    push si
    push es

```



```

push bx
push di
push dx

mov ax, KEEP_PSP
mov es, ax
mov ax, es:[2Ch]
mov es, ax
xor si, si
FOUND_ZERO:
inc si
mov dl, es:[si-1]
cmp dl, 0
jne FOUND_ZERO
mov dl, es:[si]
cmp dl, 0
jne FOUND_ZERO

add si, 3
mov bx, offset PATH_STR
LOOP_FINDER:
mov dl, es:[si]
mov [bx], dl
cmp dl, '.'
je LOOP_BREAK

inc bx
inc si

jmp LOOP_FINDER
LOOP_BREAK:
mov dl, [bx]
cmp dl, '\'
je END_LOOP
mov dl, 0h
mov [bx], dl
dec bx
jmp LOOP_BREAK

```

```

END_LOOP:
    pop dx
    mov di, dx
    push dx
    inc bx
NEW_LOOP:
    mov dl, [di]
    cmp dl, 0
    je END_PATH_FIND
    mov [bx], dl
    inc di
    inc bx
    jmp NEW_LOOP
END_PATH_FIND:
    mov [bx], byte ptr '$'
    pop dx
    pop di
    pop bx
    pop es
    pop si
    pop ax
    ret
PATH_FIND endp

```

```

ALLOCATION_MEMORY proc near
    push ax
    push bx
    push cx
    push dx

    push dx
    mov dx, offset TEMP
    mov ah, 1ah
    int 21h
    pop dx
    xor cx, cx
    mov ah, 4eh
    int 21h

```

jnc MEM_ALLOCATED

cmp ax, 2

je F_N_F_A

cmp ax, 3

je P_N_F_A

F_N_F_A:

mov dx, offset FILE_NOT_FOUND_ALL

call WRITE

jmp MEM_ALL_END

P_N_F_A:

mov dx, offset PATH_NOT_FOUND

call WRITE

jmp MEM_ALL_END

MEM_ALLOCATED:

push di

mov di, offset TEMP

mov bx, [di+1ah]

mov ax, [di+1ch]

pop di

push cx

mov cl, 4

shr bx, cl

mov cl, 12

shl ax, cl

pop cx

add bx, ax

add bx, 1

mov ah, 48h

int 21h

mov word ptr OVL_ADDRESS, ax

mov dx, offset ALLOCATION_SUCCESSFUL

call WRITE

MEM_ALL_END:

pop dx

```
    pop cx
    pop bx
    pop ax
    ret
ALLOCATION_MEMORY endp
```

```
MAIN_HANDLER proc near
```

```
    push ax
    push bx
    push cx
    push dx
    push ds
    push es
```

```
    mov ax, data
    mov es, ax
    mov bx, offset OVL_ADDRESS
    mov dx, offset PATH_STR
    mov ax, 4b03h
    int 21h
```

```
    jnc LOADED_SUCCESS
    cmp ax, 1
    je WRONG_FUNC_NUM
    cmp ax, 2
    je FILE_NOT_FOUND_ERR
    cmp ax, 3
    je DISK_ERR_FOUND_ERR
    cmp ax, 4
    je NOT_EN_MEM
    cmp ax, 6
    je ACCESS_ERR_MSG
    cmp ax, 8
    je NON_EN_MEM
    cmp ax, 10
    je FORMAT_ERROR_MSG
```

```
WRONG_FUNC_NUM:
```

```
    mov dx, offset WRONG_FUNC_NUMBER
```

```

    call WRITE
    jmp END_HANDLER
FILE_NOT_FOUND_ERR:
    mov dx, offset FILE_NOT_FOUND_LOAD
    call WRITE
    jmp END_HANDLER
DISK_ERR_FOUND_ERR:
    mov dx, offset PATH_NOT_FOUND_LOAD
    call WRITE
    jmp END_HANDLER
NOT_EN_MEM:
    mov dx, offset TOO_MUCH_FILES
    call WRITE
    jmp END_HANDLER
ACCESS_ERR_MSG:
    mov dx, offset ACCESS_ERROR
    call WRITE
    jmp END_HANDLER
NON_EN_MEM:
    mov dx, offset NOT_ENOUGH_MEMORY
    call WRITE
    jmp END_HANDLER
FORMAT_ERROR_MSG:
    mov dx, offset WRONG_ENVIRONMENT
    call WRITE
    jmp END_HANDLER
LOADED_SUCCESS:
    mov dx, offset LOAD_SUCCESSFUL
    call WRITE

    mov ax, word ptr OVL_ADDRESS
    mov es, ax
    mov word ptr OVL_ADDRESS, 0
    mov word ptr OVL_ADDRESS+2, ax

    call OVL_ADDRESS
    mov es, ax
    mov ah, 49h

```

```

    int 21h
END_HANDLER:
    pop si
    pop di
    pop dx
    pop cx
    pop bx
    pop ax
    ret
MAIN_HANDLER endp

ONE_FILE_PROCESSING proc near
    push dx
    call PATH_FIND
    mov dx, offset PATH_STR
    call ALLOCATION_MEMORY
    call MAIN_HANDLER
    pop dx

    ret
ONE_FILE_PROCESSING endp

MAIN proc far
    push ds
    push ax
    mov ax,data
    mov ds,ax

    mov KEEP_PSP, es
    call FREE_MEMORY
    cmp KEEP_FLAG, 1
    je END_ERROR

    mov dx, offset OVL1_FILE
    call ONE_FILE_PROCESSING

    mov dx, offset OVL2_FILE
    call ONE_FILE_PROCESSING

```

END_ERROR:

mov ah, 4ch

int 21h

MAIN_ENDS:

MAIN endp

CODE ends

END Main

Название файла: ovl1.asm

CODE SEGMENT

ASSUME CS:CODE, DS:NOTHING, SS:NOTHING

MAIN proc far

push ax

push dx

push ds

push di

mov ax, cs

mov ds, ax

mov di, offset OVL_ADDRESS

add di, 16

call WRD_TO_HEX

mov dx, offset OVL_ADDRESS

call WRITE

pop di

pop ds

pop dx

pop ax

retf

MAIN endp

OVL_ADDRESS db "OVL1 address: ", 0AH, 0DH, 0AH, '\$'

WRITE proc near

push dx

push ax

```

    mov ah, 09h
    int 21h
    pop ax
    pop dx
    ret
WRITE endp

TETR_TO_HEX proc near
    and al,0fh
    cmp al,09
    jbe JUMP
    add al,07
JUMP:
    add al,30h
    ret
TETR_TO_HEX endp

BYTE_TO_HEX proc near
    push cx
    mov ah, al
    call TETR_TO_HEX
    xchg al,ah
    mov cl,4
    shr al,cl
    call TETR_TO_HEX
    pop cx
    ret
BYTE_TO_HEX endp

WRD_TO_HEX proc near
    push bx
    mov bh,ah
    call BYTE_TO_HEX
    mov [di],ah
    dec di
    mov [di],al
    dec di
    mov al,bh

```



```
xor ah,ah
call BYTE_TO_HEX
mov [di],ah
dec di
mov [di],al
pop bx
ret
WRD_TO_HEX endp
```

```
CODE ends
end MAIN
```

Название файла: ovl2.asm

```
CODE SEGMENT
ASSUME CS:CODE, DS:NOTHING, SS:NOTHING
```

```
MAIN proc far
    push ax
    push dx
    push ds
    push di

    mov ax, cs
    mov ds, ax
    mov di, offset OVL_ADDRESS
    add di, 16
    call WRD_TO_HEX
    mov dx, offset OVL_ADDRESS
    call WRITE

    pop di
    pop ds
    pop dx
    pop ax
    retf
MAIN endp
```

```
OVL_ADDRESS db "OVL2 address:  ", 0DH, 0AH, '$'
```

WRITE proc near

push dx

push ax

mov ah, 09h

int 21h

pop ax

pop dx

ret

WRITE endp

TETR_TO_HEX proc near

and al,0fh

cmp al,09

jbe JUMP

add al,07

JUMP:

add al,30h

ret

TETR_TO_HEX endp

BYTE_TO_HEX proc near

push cx

mov ah, al

call TETR_TO_HEX

xchg al,ah

mov cl,4

shr al,cl

call TETR_TO_HEX

pop cx

ret

BYTE_TO_HEX endp

WRD_TO_HEX proc near

push bx

mov bh,ah

```
call BYTE_TO_HEX
mov [di],ah
dec di
mov [di],al
dec di
mov al,bh
xor ah,ah
call BYTE_TO_HEX
mov [di],ah
dec di
mov [di],al
pop bx
ret
WRD_TO_HEX endp
```

```
CODE ends
end MAIN
```