

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.

Студентка гр. 0382

Михайлова О.Д.

Преподаватель

Ефремов М.А.

Санкт-Петербург

2022

Цель работы.

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.

В этой работе также рассматривается приложение, состоящее из нескольких модулей, поэтому все модули помещаются в один каталог и вызываются с использованием полного пути.

Задание.

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 оверлейного сегмента

Шаг 2. Также необходимо написать и отладить оверлейные сегменты. 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 из другого каталога. Приложение должно быть выполнено успешно

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

Шаг 6. Занесите полученные результаты в виде скриншотов в отчет. Оформите отчет в соответствии с требованиями.

Выполнение работы.

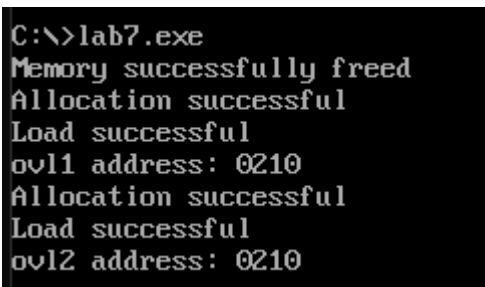
Для выполнения задания были использованы шаблоны из методических указаний, а также были добавлены следующие процедуры:

- PRINT_STRING – процедура вывода строки на экран;
- FREE_MEMORY – освобождение памяти и обработка ошибок;
- LOAD – загрузка модуля и обработка ошибок;
- PATH_READ – получение пути к файлу;
- ALLOCATION_MEM – определение размера файла OVL и выделение памяти;
- OVL_PROC – обработка файла OVL .

Шаг 1. Был написан и отлажен программный модуль .EXE, который выполняет все заданные в условии функции.

Шаг 2. Была написаны и отлажены оверлейные сегменты, выводящие адрес сегмента, в который они загружены.

Шаг 3. Была запущено отлаженное приложение.

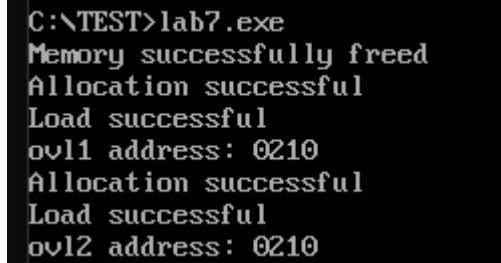


```
C:\>lab7.exe
Memory successfully freed
Allocation successful
Load successful
ovl1 address: 0210
Allocation successful
Load successful
ovl2 address: 0210
```

Рисунок 1 - Результат запуска модуля lab7.exe

В результате запуска программы видно, что оверлейные сегменты загрузились с одного и того же адреса, перекрывая друг друга.

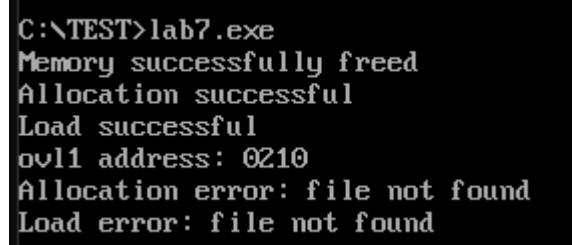
Шаг 4. Приложение было запущено из другого каталога и успешно выполнено.



```
C:\TEST>lab7.exe
Memory successfully freed
Allocation successful
Load successful
ovl1 address: 0210
Allocation successful
Load successful
ovl2 address: 0210
```

Рисунок 2 - Результат запуска приложения из другого каталога

Шаг 5. Приложение было запущено в случае, когда одного оверлея нет в каталоге. Приложение было закончено аварийно.



```
C:\TEST>lab7.exe
Memory successfully freed
Allocation successful
Load successful
ovl1 address: 0210
Allocation error: file not found
Load error: file not found
```

Рисунок 3 - Результат запуска приложения, когда файла ovl2.ovl нет в каталоге

Исходный код программы см. в приложении А.

Ответы на контрольные вопросы.

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

При обращении к оверлейному сегменту необходимо учитывать смещение 100h. Это связано с тем, что в .COM модуле присутствует PSP.

Выводы.

В ходе работы были исследованы возможности построения загрузочного модуля оверлейной структуры. Также были исследованы структура оверлейного сегмента и способ загрузки и выполнения оверлейных сегментов

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab7.asm

```
AStack SEGMENT  STACK
                DW 128 DUP(?)
AStack ENDS
```

```
DATA SEGMENT
```

```
FILE1 db 'ovl1.ovl', 0
FILE2 db 'ovl2.ovl', 0
PATH db 128 dup(0)
PROG dw 0
DTA db 43 dup(0)
CL_POS db 128 dup(0)
OVL_ADDR dd 0
KEEP_PSP dw 0
KEEP_SS dw 0
KEEP_SP dw 0
```

```
EOF db 0Dh, 0Ah, '$'
```

```
MEM_CMB db 'Memory error: control block destroyed', 0Dh, 0Ah,
'$'
```

```
MEM_NOT_ENOUGH db 'Memory error: not enough memory to execute
the function', 0Dh, 0Ah, '$'
```

```
MEM_WRONG_ADDR db 'Memory error: invalid block address', 0Dh,
0Ah, '$'
```

```
MEM_FREE_SUCCESS db 'Memory successfully freed', 0Dh, 0Ah, '$'
```

```
WRONG_FUNC_NUM db 'Load error: invalid function number', 0Dh,
0Ah, '$'
```

```
FILE_NOT_FOUND db 'Load error: file not found', 0Dh, 0Ah, '$'
```

```
ROUT_NOT_FOUND db 'Load error: route not found', 0Dh, 0Ah, '$'
```

```
TOO_MUCH_FILES db 'Load error: too much open files', 0Dh, 0Ah,
'$'
```

```
NO_ACCESS db 'Load error: no access', 0Dh, 0Ah, '$'
```

```
LOAD_MEM_ERROR db 'Load error: not enough memory', 0Dh, 0Ah,
'$'
```

```
WRONG_ENV db 'Load error: wrong environment', 0Dh, 0Ah, '$'
```

```
LOAD_OVL_SUCCESS db 'Load successful', 0Dh, 0Ah, '$'
```

```
ALLOC_FILE_NOT_FOUND db 'Allocation error: file not found',
0Dh, 0Ah, '$'
```

```
ALLOC_ROUTE_NOT_FOUND db 'Allocation error: route not found',
0Dh, 0Ah, '$'
```

```
ALLOC_SUCCESS db 'Allocation successful', 0Dh, 0Ah, '$'
```

```
data_end db 0
```

```
flag db 0
```

```
DATA ENDS
```

```

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

PRINT_STRING PROC near
    push ax
    mov ah, 09h
    int 21h
    pop ax
    ret
PRINT_STRING ENDP

FREE_MEMORY PROC near
    push ax
    push bx
    push cx
    push dx

    mov ax, offset data_end
    mov bx, offset proc_end
    add bx, ax
    mov cl, 4
    shr bx, cl
    add bx, 2bh
    mov ah, 4ah
    int 21h
    jnc free_memory_success
    mov flag, 1

mem_error_7:
    cmp ax, 7
    jne mem_error_8
    mov dx, offset MEM_CMB
    call PRINT_STRING
    jmp end_free_memory

mem_error_8:
    cmp ax, 8
    jne mem_error_9
    mov dx, offset MEM_NOT_ENOUGH
    call PRINT_STRING
    jmp end_free_memory

mem_error_9:
    cmp ax, 9
    mov dx, offset MEM_WRONG_ADDR
    call PRINT_STRING
    jmp end_free_memory

free_memory_success:
    mov dx, offset MEM_FREE_SUCCESS
    call PRINT_STRING

end_free_memory:
    pop dx
    pop cx
    pop bx

```

```

        pop ax
        ret

FREE_MEMORY ENDP

PATH_READ 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

find_two_zeros:
    inc si
    mov dl, es:[si-1]
    cmp dl, 0
    jne find_two_zeros
    mov dl, es:[si]
    cmp dl, 0
    jne find_two_zeros

    add si, 3
    mov bx, offset PATH

not_point:
    mov dl, es:[si]
    mov [bx], dl
    cmp dl, '.'
    je b_loop

    inc bx
    inc si

    jmp not_point

b_loop:
    mov dl, [bx]
    cmp dl, '\'
    je break
    mov dl, 0h
    mov [bx], dl
    dec bx
    jmp b_loop

break:
    pop dx
    mov di, dx
    push dx

```



```

        inc bx

new_file:
        mov dl, [di]
        cmp dl, 0
        je end_path_read
        mov [bx], dl
        inc di
        inc bx
        jmp new_file

end_path_read:
        mov [bx], byte ptr '$'
        pop dx
        pop di
        pop bx
        pop es
        pop si
        pop ax

        ret
PATH_READ endp

LOAD PROC near
        push ax
        push bx
        push cx
        push dx
        push ds
        push es

        mov ax, DATA
        mov es, ax
        mov bx, offset OVL_ADDR
        mov dx, offset PATH
        mov ax, 4b03h
        int 21h

        jnc load_success

load_error_1:
        cmp ax, 1
        jne load_error_2
        mov dx, offset WRONG_FUNC_NUM
        call PRINT_STRING
        jmp end_load

load_error_2:
        cmp ax, 2
        jne load_error_3
        mov dx, offset FILE_NOT_FOUND
        call PRINT_STRING
        jmp end_load

load_error_3:

```

```

        cmp ax, 3
        jne load_error_4
        mov dx, offset ROUT_NOT_FOUND
        call PRINT_STRING
        jmp end_load

load_error_4:
        cmp ax, 4
        jne load_error_5
        mov dx, offset TOO_MUCH_FILES
        call PRINT_STRING
        jmp end_load

load_error_5:
        cmp ax, 5
        jne load_error_8
        mov dx, offset NO_ACCESS
        call PRINT_STRING
        jmp end_load

load_error_8:
        cmp ax, 8
        jne load_error_10
        mov dx, offset LOAD_MEM_ERROR
        call PRINT_STRING
        jmp end_load

load_error_10:
        cmp ax, 10
        mov dx, offset WRONG_ENV
        call PRINT_STRING
        jmp end_load

load_success:
        mov dx, offset LOAD_OVL_SUCCESS
        call PRINT_STRING

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

        call OVL_ADDR
        mov es, ax
        mov ah, 49h
        int 21h

end_load:
        pop es
        pop ds
        pop dx
        pop cx
        pop bx
        pop ax
        ret

```

LOAD ENDP

ALLOCATION_MEM proc near

```
    push ax
    push bx
    push cx
    push dx

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

    jnc allocation_success
    cmp ax, 2
    je alloc_error_2
    cmp ax, 3
    je alloc_error_3
```

```
alloc_error_2:
    mov dx, offset ALLOC_FILE_NOT_FOUND
    call PRINT_STRING
    jmp end_allocation
```

```
alloc_error_3:
    mov dx, offset ALLOC_ROUTE_NOT_FOUND
    call PRINT_STRING
    jmp end_allocation
```

```
allocation_success:
    push di
    mov di, offset DTA
    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_ADDR, ax
    mov dx, offset ALLOC_SUCCESS
    call PRINT_STRING
```

end_allocation:

```

        pop dx
        pop cx
        pop bx
        pop ax
        ret

ALLOCATION_MEM ENDP

OVL_PROC proc near
    push dx
    call PATH_READ
    mov dx, offset PATH
    call ALLOCATION_MEM
    call LOAD
    pop dx

    ret
OVL_PROC ENDP

Main PROC FAR
    push ds
    push ax
    mov ax, DATA
    mov ds, ax

    mov KEEP_PSP, es
    call FREE_MEMORY
    cmp flag, 0
    jne final
    mov dx, offset FILE1
    call OVL_PROC
    mov dx, offset FILE2
    call OVL_PROC

final:
    mov ah, 4Ch
    int 21h
Main ENDP
proc_end:
CODE ENDS
END Main

```