

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

ОТЧЕТ
по лабораторной работе №4
по дисциплине «Операционные системы»
Тема: Обработка стандартных прерываний

Студент гр. 0382

Кондратов Ю.А.

Преподаватель

Ефремов М.А.

Санкт-Петербург

2022

Цель работы.

В архитектуре компьютера существуют стандартные прерывания, за которыми закреплены определенные вектора прерываний. Вектор прерываний хранит адрес подпрограммы обработчика прерываний. При возникновении прерывания, аппаратура компьютера передает управление по соответствующему адресу вектора прерывания. Обработчик прерываний получает управление и выполняет соответствующие действия.

В лабораторной работе № 4 предлагается построить обработчик прерываний сигналов таймера. Эти сигналы генерируются аппаратурой через определенные интервалы времени и, при возникновении такого сигнала, возникает прерывание с определенным значением вектора. Таким образом, управление будет передано функции, чья точка входа записана в соответствующий вектор прерывания.

Постановка задачи.

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

- 1) Проверяет, установлено ли пользовательское прерывание с вектором 1Ch.
- 2) Устанавливает резидентную функцию для обработки прерывания и настраивает вектор прерываний, если прерывание не установлено, и осуществляется выход по функции 4Ch прерывания int 21h.
- 3) Если прерывание установлено, то выводится соответствующее сообщение и осуществляется выход по функции 4Ch прерывания int 21h.
- 4) Выгрузка прерывания по соответствующему значению параметра в командной строке /un. Выгрузка прерывания состоит в восстановлении стандартного вектора прерываний и освобождении памяти, занимаемой резидентом. Затем осуществляется выход по функции 4Ch прерывания int 21h.

Для того, чтобы проверить установку прерывания, можно поступить следующим образом. Прочитать адрес, записанный в векторе прерывания.

Предположим, что этот адрес указывает на точку входа в установленный резидент. На определенном, известном смещении в теле резидента располагается сигнатура, некоторый код, который идентифицирует резидент. Сравнив известное значение сигнатуры с реальным кодом, находящимся в резиденте, можно определить, установлен ли резидент. Если значения совпадают, то резидент установлен. Длину кода сигнатуры должна быть достаточной, чтобы сделать случайное совпадение маловероятным. Программа должна содержать код устанавливаемого прерывания в виде удаленной процедуры. Этот код будет работать после установки при возникновении прерывания. Он должен выполнять следующие функции:

- 1) Сохраняет стек прерванной программы (регистры SS и SP) в рабочих переменных и восстановить при выходе.
- 2) Организовать свой стек.
- 3) Сохранить значения регистров в стеке при входе и восстановить их при выходе.
- 4) При выполнении тела процедуры накапливать общее суммарное число прерываний и выводить на экран. Для вывода на экран следует использовать прерывание int 10h, которое позволяет непосредственно выводить информацию на экран.
- 5) Функция прерывания должна содержать только переменные, которые она использует

Шаг 2. Запустите отлаженную программу и убедитесь, что резидентный обработчик прерывания 1Ch установлен. Работа прерывания должна отображаться на экране, а также необходимо проверить размещение прерывания в памяти. Для этого запустите программу ЛР 3, которая отображает 4 карту памяти в виде списка блоков МСВ. Полученные результаты поместите в отчет.

Шаг 3. Запустите отлаженную программу еще раз и убедитесь, что программа определяет установленный обработчик прерываний. Полученные результаты поместите в отчет.

Шаг 4. Запустите отлаженную программу с ключом выгрузки и убедитесь, что резидентный обработчик прерывания выгружен, то есть сообщения на экран не выводятся, а память, занятая резидентом освобождена. Для этого также следует запустить программу ЛР 3. Полученные результаты поместите в отчет.

Шаг 5. Ответьте на контрольные вопросы

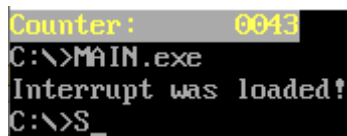
Используемые функции

1. `custom_int` – созданный обработчик прерывания;
2. `load_int` – функция, выполняющая загрузку обработчика прерывания в память;
3. `unload_int` – функция, выполняющая выгрузку обработчика прерывания из памяти;
4. `find_cmd_flag` – функция для определения флага командной строки;
5. `is_loaded` – функция, определяющая загружен ли созданный обработчик прерывания в память

Выполнение работы.

Исходный код модуля представлен в приложении А.

Шаг 1. В результате выполнения шага 1 был написан исполняемый модуль типа .exe, который совершает все необходимые по условию действия. Результат корректной работы модуля представлен на рисунке 1.



```
Counter: 0013
C:\>MAIN.exe
Interrupt was loaded!
C:\>S_
```

Рисунок 1 – Результат работы модуля main.exe

Шаг 2. Для выполнения шага 2 был взят модуль main1 из лабораторной работы 3. Результат выполнения модуля main1.com представлен на рисунке 2. Из рисунка видно, что резидентный обработчик прерывания действительно находится в основной памяти.

```

Counter: 0142
C:\>MAIN.EXE
Interrupt was loaded!
C:\>MAIN1.com
Available mem size: 647792
Extended mem size: 246720
MCB: 1, addr: 016F, owner PSP: 0008, size: 16, SD/SC:
MCB: 2, addr: 0171, owner PSP: 0000, size: 64, SD/SC:
MCB: 3, addr: 0176, owner PSP: 0040, size: 256, SD/SC:
MCB: 4, addr: 0187, owner PSP: 0192, size: 144, SD/SC:
MCB: 5, addr: 0191, owner PSP: 0192, size: 944, SD/SC: MAIN
MCB: 6, addr: 01CD, owner PSP: 01D8, size: 144, SD/SC:
MCB: 7, addr: 01D7, owner PSP: 01D8, size: 647792, SD/SC: MAIN1

```

Рисунок 2 – Результат работы модуля main1.com

Шаг 3. Отлаженный модуль main.exe был запущен ещё раз, в результате было выведено сообщение о том, что обработчик уже находится в памяти (см. рисунок 3).

```

C:\>MAIN.EXE
Interrupt has already been loaded!

```

Рисунок 3 – Результат вторичного запуска модуля main.exe

Шаг 4. При запуске модуля с ключом выгрузки было выведено сообщение о том, что пользовательский обработчик успешно выгружен из памяти. Как можно видеть из рисунка 4, после ещё одного запуска модуля main.exe обработчик снова был успешно загружен в память.

```

C:\>MAIN.EXE /un
Interrupt was unloaded!
C:\>main.exe
Interrupt was loaded!

```

Рисунок 4 – Демонстрация успешной выгрузки пользовательского обработчика

Контрольные вопросы.

Сегментный адрес недоступной памяти:

1. Как реализован механизм прерывания от часов?

Аппаратура генерирует сигналы через определённые временные интервалы (18 раз в секунду). После каждого сигнала происходит прерывание с номером 1ch. При замене обработчика этого прерывания в таблице на пользовательскую функцию, после каждого сигнала будет выполняться именно она.

2. Какого типа прерывания использовались в работе?

Программные (21h, 10h) и аппаратные (1ch).

Выводы.

В процессе выполнения данной лабораторной работы был изучен механизм работы аппаратного таймера, проведена работа по созданию резидентного пользовательского обработчика прерывания.

ПРИЛОЖЕНИЕ А.

Исходный код модулей

main.asm:

```
AStack    SEGMENT STACK
           DB 256 dup (?)
AStack    ENDS

DATA SEGMENT
    flag    DB 0
    load_msg DB 'Interrupt was loaded!$'
    unload_msg DB 'Interrupt was unloaded!$'
    in_mem_msg DB 'Interrupt has already been loaded!$'
    not_loaded_msg DB 'Interrupt wasnt loaded!$'
DATA ENDS

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

custom_int PROC far
    jmp    int_start

    int_sign DW 7777h
    PSP      DW ?
    old_cs   DW 0
    old_ip   DW 0
    old_ss   DW 0
    old_sp   DW 0
    old_ax   DW 0
    count_msg DB 'Counter:      0000'
    msg_len = $ - count_msg
    int_stack DB 128 dup (?)
    stack_end:

int_start:
    mov     old_ss, SS
    mov     old_sp, SP
    mov     old_ax, AX
```

```

mov     AX, CS
mov     SS, AX
mov     SP, OFFSET stack_end
push    BX
push    CX
push    DX
push    DS
push    ES
push    SI
push    DI
push    BP

mov     AH, 03h
mov     BH, 0
int     10h
push    DX

mov     AH, 02h
mov     BH, 0
mov     DX, 0
int     10h

push    BP
push    DS
push    SI
mov     DX, SEG count_msg
mov     DS, DX
mov     SI, OFFSET count_msg
mov     CX, 5
inc_loop:
mov     BP, CX
dec     BP
mov     AL, byte ptr [SI+BP+13]
inc     AL
mov     [SI+BP+13], AL
cmp     AL, 3Ah
jne     oklab
mov     AL, 30h
mov     byte ptr [SI+BP+13], AL

```



```

        loop    inc_loop
oklab:
        pop     SI
        pop     DS

        push    ES
        mov     DX, SEG count_msg
        mov     ES, DX
        mov     BP, OFFSET count_msg
        mov     AH, 13h
        mov     AL, 1
        mov     BH, 0
        mov     CX, msg_len
        mov     DX, 0
        int     10h
        pop     ES
        pop     BP

        mov     AH, 02h
        mov     BH, 0
        pop     DX
        int     10h

        pop     BP
        pop     DI
        pop     SI
        pop     ES
        pop     DS
        pop     DX
        pop     CX
        pop     BX
        mov     AX, old_ss
        mov     SS, AX
        mov     SP, old_sp
        mov     AX, old_ax
        mov     AL, 20h
        out     20h, AL
        iret

int_end:

```

```
custom_int ENDP
```

```
load_int PROC
```

```
    push    AX
    push    CX
    push    DX

    mov     AH, 35h
    mov     AL, 1Ch
    int     21h
    mov     old_ip, BX
    mov     old_cs, ES

    push    DS
    mov     DX, OFFSET custom_int
    mov     AX, SEG custom_int
    mov     DS, AX
    mov     AH, 25h
    mov     AL, 1Ch
    int     21h
    pop     DS

    mov     DX, OFFSET int_end
    mov     CL, 4
    shr     DX, CL
    inc     DX
    mov     AX, CS
    sub     AX, PSP
    add     DX, AX
    xor     AX, AX
    mov     AH, 31h
    int     21h
    pop     DX
    pop     CX
    pop     AX
    ret
```

```
load_int ENDP
```

```
unload_int PROC
```

```

push    AX
push    DX
push    SI
push    ES

cli

push    DS
mov     AH, 35h
mov     AL, 1Ch
int     21h
mov     SI, OFFSET old_cs
sub     SI, OFFSET custom_int
mov     DX, ES:[BX+SI+2]
mov     AX, ES:[BX+SI]
mov     DS, AX
mov     AH, 25h
mov     AL, 1Ch
int     21h
pop     DS
mov     AX, ES:[BX+SI-2]
mov     ES, AX
push    ES
mov     AX, ES:[2Ch]
mov     ES, AX
mov     AH, 49h
int     21h
pop     ES
mov     AH, 49h
int     21h
sti

pop     ES
pop     SI
pop     DX
pop     AX

ret

unload_int ENDP

find_cmd_flag PROC
push    AX

```

```

        mov     AL, ES:[82h]
        cmp     AL, '/'
        jne     nparam
        mov     AL, ES:[83h]
        cmp     AL, 'u'
        jne     nparam
        mov     AL, ES:[84h]
        cmp     AL, 'n'
        jne     nparam
        mov     flag, 1
nparam:
        pop     AX
        ret
find_cmd_flag ENDP

is_loaded PROC
        push    AX
        push    DX
        push    SI
        mov     flag, 1
        mov     AH, 35h
        mov     AL, 1Ch
        int     21h
        mov     SI, OFFSET int_sign
        sub     SI, OFFSET custom_int
        mov     DX, ES:[BX+SI]
        cmp     DX, 7777h
        je      loaded_lab
        mov     flag, 0
loaded_lab:
        pop     SI
        pop     DX
        pop     AX
        ret
is_loaded ENDP

PRINT_STRING PROC
        push    AX
        mov     AH, 09h

```

```

        int      21h
        pop      AX
        ret
PRINT_STRING ENDP

MAIN PROC far
        mov      AX, data
        mov      DS, AX
        mov      PSP, ES
        mov      flag, 0
        call     find_cmd_flag
        cmp      flag, 1
        je       unload_lab

        call     is_loaded
        cmp      flag, 0
        je       not_loaded_lab
        mov      DX, OFFSET in_mem_msg
        call     PRINT_STRING
        jmp      final_lab
not_loaded_lab:
        mov      DX, OFFSET load_msg
        call     PRINT_STRING
        call     load_int

        jmp      final_lab

unload_lab:
        call     is_loaded
        cmp      flag, 0
        jne      already_loaded_lab
        mov      DX, OFFSET not_loaded_msg
        call     PRINT_STRING
        jmp      final_lab
already_loaded_lab:
        call     unload_int
        mov      DX, OFFSET unload_msg
        call     PRINT_STRING

```

```
final_lab:
    mov     AX, 4C00h
    int     21h
MAIN      ENDP
CODE      ENDS
END       MAIN
```