WOJSKOWA AKADEMIA TECHNICZNA

SYSTEMY WBUDOWANE

Prowadzący: **Paweł Janicki**

Autor sprawozdania: **Pol Grzegorz**

Grupa szkoleniowa: **I7X3S1**

Numer ćwiczenia: **T2**

Data oddania: **14.06.2009r.**

1. Treść zadania

Dokonać modyfikacji kodu źródłowego zawartego w zbiorze ***PMTUT.asm*** tak aby:

- tekst 'I'm the Interrupt Handler - returning' w trybie PM wyświetlony został z procedury obsługi przerwania w wyniku wywołania rozkazu "***INT <NR>***", gdzie jest numerem porządkowym studenta na liście grupy.

- początek bufora strony tekstowej w trybie PM znajdował się pod offsetem logicznym ***<offs> = 0x10000 * <NR> + 0xB8000***. Należy utworzyć/zmodyfikować istniejącą strukturę deskryptora aby w wyniku następującej modyfikacji instrukcji:

  zamiany linii: `add edi,0b8000h; physical adress of text screen`
  na: `add edi,<offs>`

  znajdującej się w procedurze ***write_msg_pm*** zachować pierwotny efekt działania programu. W procedurze ***write_msg_pm*** dopuszcza się jedynie modyfikację linii:

  ```
  mov ax,core32_idx ; in protected mode, we have to use
  ```

- bufor strony tekstowej w PM rozpoczynał się adresem liniowym ***0x80000000 (2GB)***. Należy włączyć i odpowiednio wykorzystać mechanizm stronicowania (część niewykonana)

  Mój nr na liście to 5.

2. Zadanie 1

2.1 Mapa istotnych obszarów pamięci w przestrzeni adresowej fizycznej i logicznej

Przestrzeń adresowa fizyczna jest uzależniona w naszym przypadku od 32 bitowej maszyny wirtualnej, której przestrzeń wynosi 4GB. Sam adres fizyczny powstaje w wyniku translacji o jedno miejsce adresu logicznego.

Przestrzeń adresowa logiczna w trybie rzeczywistym wynosi około 1 MB. Adresem logicznym nazywamy dwuwymiarową strukturę składającą się z pary liczb. Pierwsza z nich określa selektor, a druga określa przemieszczenie wewnątrz segmentu.

W modyfikowanym przez nas programie możemy rozróżnić dwa segmenty kodu:

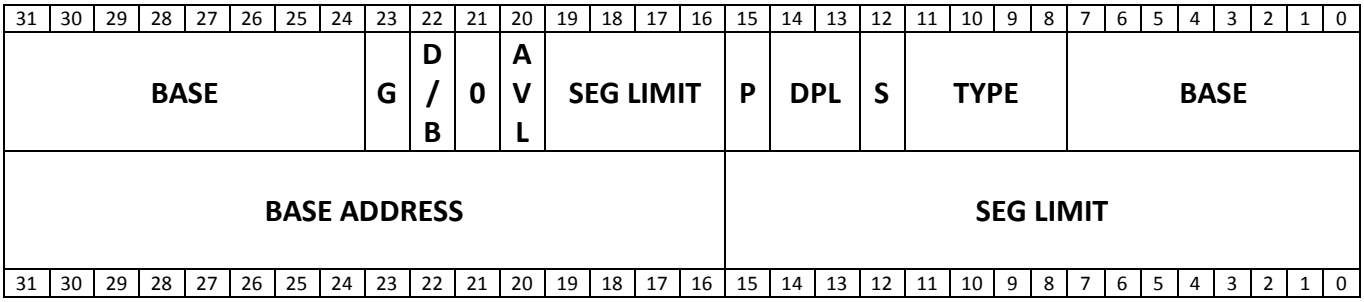
- CODE16 – jak sama nazwa wskazuje jest to 16bitowy segment kodu, który jest używany w trybie rzeczywistym. Jego początek jest punktem wejściowym programu
- CODE32 – 32 bitowy segment kodu, który w porównaniu z CODE16 używany jest w trybie chronionym. Pod jego adres początkowy wykonywany jest skok w procedurze przejścia

Istotne obszary pamięci fizycznej:

- 0x0 – 0x400 – wektory przerwań w trybie rzeczywistym
- 0xB8000 – pamięć wideo
- reszta adresów jest względna

2.2 Opis zdefiniowanych struktur deskryptorowych

Przy realizacji zadania trzeba było zdefiniować deskryptor przerwania. Deskryptor jest to obszar w pamięci w systemie związany z aktualnie wykonywanym zadaniem. Podstawowym typem deskryptora w architekturze x86 jest deskryptor segmentu (ang. Segment Descriptor) umieszczony jest on w lokalnej lub globalnej tablicy deskryptorów (ang. Global Descriptor Table - GDT, Local Descriptor Table - LDT). Przez GDT lub LDT prowadzą wszystkie odwołania do pamięci. W trybie chronionym (protected mode) pracy procesora x86 (od procesora i386) mamy do czynienia z tablicą deskryptorów przerwań (ang. Interrupt Descriptor Table, IDT) łączącą każdy wektor wyjątku lub przerwania z deskryptorem bramy (deskryptory bram to deskryptory pozwalające na kontrolowany dostęp do segmentów kodu o różnych stopniach uprzywilejowana) dla procedury lub zadania obsługującym dany wyjątek lub przerwanie.



Deskryptor segmentu

Deskryptory grupowane są w tablice, tzn. kolejno następujące po sobie deskryptory, np:

```
dummy_descriptor segment_descriptor <0,0,0,0,0,0>
code32_descriptor segment_descriptor <0ffffh,0,0,9ah,0cfh,0>
data32_descriptor segment_descriptor <0ffffh,0,0,92h,0cfh,0>
core32_descriptor segment_descriptor <0ffffh,0,0,92h,0cfh,0>
code16_descriptor segment_descriptor <0ffffh,0,0,9ah,0,0>
data16_descriptor segment_descriptor <0ffffh,0,0,92h,0,0>
```

Tablica ta zawiera sześć deskryptorów: pusty deskryptor (zawiera same zera), 32 bitowy deskryptor segmentu danych, 32 bitowy deskryptor segmentu kodu, core32_descriptor, 16 bitowy deskryptor segmentu kodu oraz 16 bitowy deskryptor segmentu danych.

Pierwszy deskryptor pusty można załadować do selektora jednak odwołanie się do pamięci spowoduje błąd. Kolejne trzy deskryptory są bardzo podobne i opisują 4GB przestrzeni pamięci od adresu zerowego, do 0xFFFFFFFF, o 32 bitowej długości słowa, z ustawionym bitem obecności, najwyższym poziomem uprzywilejowania (DPL=0), wyzerowanym bitem dostępu (A, ang. accessed), z ustawionym bitem granulacji (pole limit deskryptora mnożone jest przez 4KB) i rozszerzaniem w górę (offset musi być większy od pola base). Wyróżnia się code32_descriptor który posiada pole type ustawione na 101 – oznacza to, że jest to segment kodu (wykonanie i odczyt), pozostałe dwa segmenty posiadają ustawione pole type na 001 co oznacza segment danych (odczyt i zapis).

Deskryptory code16_descriptor oraz data16_descriptor to odpowiednio deskryptory segmentów kodu i danych o 16 bitowej długości słowa, długości 64KB (pole segment limit ustawione na 0xFFFF, bit granulacji wyzerowany więc długość określa rozmiar przestrzeni adresowej w bajtach) – reszta pól ustawiona została identycznie jak dla segmentów 32 bitowych.

Poniżej przedstawiam interpretację deskryptora przerwania:

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="16">OFFSET</td><td>P</td><td colspan="2">DPL</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="8">RESERVED</td></tr>
<tr><td colspan="16">SELEKTOR SEGMENTU</td><td colspan="16">OFFSET</td></tr>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

Deskryptor przerwania

W kodzie deskryptor przerwania reprezentowany jest przez strukturę interrupt_descriptor.

interrupt_0 interrupt_descriptor <small demo_int,code32_idx,0,8eh,0>
interrupt_descriptor 5 dup (<small demo_int,code32_idx,0,8eh,0>)

Pierwsze pole tej struktury to mniej znaczące 16 bitów wskaźnika na procedurę obsługi przerwania, kolejne to selektor segmentu (tryb PM więc używamy code32_idx), nieużywany bajt, następnie występuje bajt 0x8e, który za chwilę zostanie dokładniej opisany, na końcu bardziej znacząca cześć wskaźnika do procedury obsługi przerwania.

Bajt 0x8e po zamianie na postać binarną przedstawia się następująco: 1 00 0 1110b
Najstarszy bit jest ustawiony co odpowiada za flagę P (ang. present) – obecność poprawnego segmentu. Kolejne dwa bity to pole DPL (ang. Descriptor Privilege Level) – 0 oznacza najwyższy poziom uprzywilejowania. Kolejny bit jest nieużywany, pozostałe cztery bity opisują rodzaj furtki – w zdefiniowanym przypadku to 32 bitowe przerwanie (ang. 32-Bit Interrupt Gate).

2.3 Opis procedury przejścia w tryb *protected*

W celu przejście w tryb chroniony należy:

1. **Stworzyć globalną tablicę deskryptorów – GDT, zdefiniować deskryptory.**

```
label global_descriptor_table fword ; here begins the GDT

gdt_start dw gdt_size,0,0 ; val for GDT reg
dummy_descriptor segment_descriptor <0,0,0,0,0,0>
code32_descriptor segment_descriptor <0ffffh,0,0,9ah,0cfh,0>
data32_descriptor segment_descriptor <0ffffh,0,0,92h,0cfh,0>
core32_descriptor segment_descriptor <0ffffh,0,0,92h,0cfh,0>
code16_descriptor segment_descriptor <0ffffh,0,0,9ah,0,0>
data16_descriptor segment_descriptor <0ffffh,0,0,92h,0,0>

gdt_size=$-(offset dummy_descriptor)
code32_idx = 08h ; offset of 32-bit code segment in GDT
data32_idx = 10h ; offset of 32-bit data segment in GDT
core32_idx = 18h ; offset of 32-bit core segment in GDT
code16_idx = 20h ; offset of 16-bit code segment in GDT
data16_idx = 28h ; offset of 16-bit data segment in GDT
```

2. **Stworzyć tablicę deskryptorów przerwań – IDT, zdefiniować deskryptory**

```
label interrupt_descriptor_table fword ; here begins the IDT
idt_start dw idt_size,0,0
interrupt_0 interrupt_descriptor <small demo_int,code32_idx,0,8eh,0>
idt_size=$-(offset interrupt_0)
```

3. **Zamaskowanie/zablokowanie przerwań. Załadowanie adresów tablic deskryptorów do rejestrów GDTR, IDTR.**

```
cli ; disable interrupts
lgdt [fword ds:global_descriptor_table] ; load GDT register
lidt [fword ds:interrupt_descriptor_table] ; load IDT register
```

4. **W rejestrze CR0 ustawić bit „Protection Enable”.**

```
mov eax,cr0 ; get CR0 into EAX
or al,1 ; set Protected Mode bit
mov cr0,eax ; after this we are in Protected Mode!
```

5. **Wykonać skok na początek procedury PM.**

```
db 0eah ; opcode for far jump (to set CS correctly)
dw small start32,code32_idx
```

2.4 Implementacja i opis realizacji

```
label interrupt_descriptor_table fword ; here begins the IDT

idt_start dw idt_size,0,0
interrupt_0 interrupt_descriptor <small demo_int,code32_idx,0,8eh,0>
interrupt_descriptor 5 dup (<small demo_int,code32_idx,0,8eh,0>)

idt_size=$-(offset interrupt_0)
```

2.5 Opis realizacji

Zadanie polegało na zdefiniowaniu obsługi przerwania w trybie protected mode w taki sposób aby INT 5, czyli przerwanie o nr autora z listy porządkowej w grupie wywoływał procedurę demo_int, która odpowiada za wyświetlanie napisu „Now In Protected Mode - calling Interrupt”. W tym celu należało na miejscu 6 tablicy IDT (ponieważ numeracja zaczyna się od zera) umieścić strukturę interrupt_descriptor z odpowiednio zdefiniowanymi polami.

Polecenie 5 dup powoduje umieszczenie 5 struktur interrupt_descriptor dzięki czemu ostatnie pole tablicy IDT odpowiada wymaganemu numerowi przerwania.

3. Zadanie 2

3.1 Implementacja i opis realizacji

Dokonana modyfikacja deskryptora core32_descriptor :

```
label global_descriptor_table fword ; here begins the GDT

gdt_start dw gdt_size,0,0 ; val for GDT reg
dummy_descriptor segment_descriptor <0,0,0,0,0,0>
code32_descriptor segment_descriptor <0ffffh,0,0,9ah,0cfh,0>
data32_descriptor segment_descriptor <0ffffh,0,0,92h,0cfh,0>
; core32_descriptor segment_descriptor <0ffffh,0,0,92h,0cfh,0>
core32_descriptor segment_descriptor <0ffffh,0,0fbh,92h,0cfh,0ffh>
code16_descriptor segment_descriptor <0ffffh,0,0,9ah,0,0>
data16_descriptor segment_descriptor <0ffffh,0,0,92h,0,0>
gdt_size=$-(offset dummy_descriptor)
```

Definicja offs, oraz modyfikacja jednej linii w procedurze write_msg_pm:

```
offs = (10000h*5 + 0B8000h) ;108000h
proc write_msg_pm
      push ax esi edi es
      mov ax,core32_idx       ; in protected mode, we have to use
                              ; core memory to address the screen
      mov es,ax
      xor edi,edi
      movzx di,[esi+2] ; get Y position
      imul edi,160
      add di,[esi] ; add X position
      add di,[esi]
      ; add edi,0b8000h ; physical address of text screen
      add edi,offs ; linear address of text screen
      mov ah,[esi+4] ; get attribute byte
      add esi,5
write_loop:
      mov al,[esi]
      or al,al ; end of string?
      jz loop_end
      inc esi
      mov [es:edi],ax
```

```
        inc edi
        inc edi
        jmp write_loop
loop_end:
        pop es edi esi ax
        ret
endp write_msg_pm
```

3.2 Opis realizacji

Zadanie drugie sprowadzało się do modyfikacji definicji deskryptora core32_descriptor – jest on wskazywany przez wskaźnik tablicy core32_idx który to z kolei ładowany jest do rejestru ES w procedurze write_msg_pm. Zdefiniowany on został w taki sposób, aby zapewnić przemalowanie adresu 0x108000 (wynik działania 0x10000*5 + 0xB8000) na adres fizyczny 0xB8000. Dokonano tego dzięki ustawieniu adresu bazowego deskryptora na wartość 0xFFFB0000 – jego obliczenie można zrealizować następująco: 0xFFFFFFFF - 0x108000 + 1 + 0xB8000.Wykorzystano tutaj fakt zawinięcia się adresu po przekroczeniu wartości maksymalnej (0xFFFFFFFF ze względu na 32 bitową szynę adresową) – po dodaniu do adresu bazowego 0xFFFB0000 przesunięcia 0x108000 otrzymujemy 0x1000B8000 co po pozostawieniu jedynie 32 mniej znaczących bitów daje adres fizyczny 0xB8000.