32-bitowy tryb chroniony

Pora opuścić strefę względnego komfortu 16-bitowego trybu rzeczywistego i rzucić się w nieznane odmęty 32-bitowego trybu chronionego. Zacząć by się przydało od tego, czym w ogóle jest tryb chroniony i czym się różnić będzie od tego, na czym pracowaliśmy chwilę temu. Po pierwsze: rejestry rozszerzone zostają do 32 bitów. Do pełnego rozmiaru rejestru możemy dostać się poprzez dodanie literki e przed nazwą znanych nam rejestrów procesora. (np. eax, ebx). Rejestry nadal zachowują adresowalność w znanym z 16-bitowej architektury stylu (ax, dzielony na ah i al). Do 32 bitów zwiększone są też offsety w pamięci, więc teraz możemy adresować do 4 GB pamięci.

Problemy z przejściem z 16 na 32-bitowy tryb są dwa. Znaczy jest ich więcej, ale skupimy się na dwóch. Kluczowym z naszej perspektywy będzie to, że żegnamy się z BIOS-em. Koniec z wygodnymi przerwaniami, które nam coś wydrukują, wyświetlamy rzeczy na ekranie odnosząc się bezpośrednio do pamięci, o tym za chwilę. Najpierw musimy zająć się GDT, czyli globalną tablicą deskryptorów – bez tego nie wejdziemy w tryb 32-bitowy. Na 16 bitach możemy zaadresować do 0xffff adresów w pamięci, co przekładałoby się na 64 kilobajty – to mało, dlatego został wprowadzony mechanizm segmantacji, który pozwolił na skorzystanie z 20-bitowej szyny adresowej. W ten sposób mogliśmy się dobrać do 1 megabajta pamięci – to, szybko licząc 16 razy więcej.

Dla przykładu, chcemy umieścić zawartość rejestru dx w adresie 0x5f010. Bez segmentacji najlepsze co możemy zrobić to

```
mov [0xffff], ax
```

a w ten sposób nie jesteśmy nawet blisko. Co prawda nie ma za bardzo opcji, żeby zapisać 20 bitów na 16-bitowym rejestrze, ale dzięki wprowadzeniu segmentów pamięci możemy to osiągnąć. Wartość segment:offset liczona jest

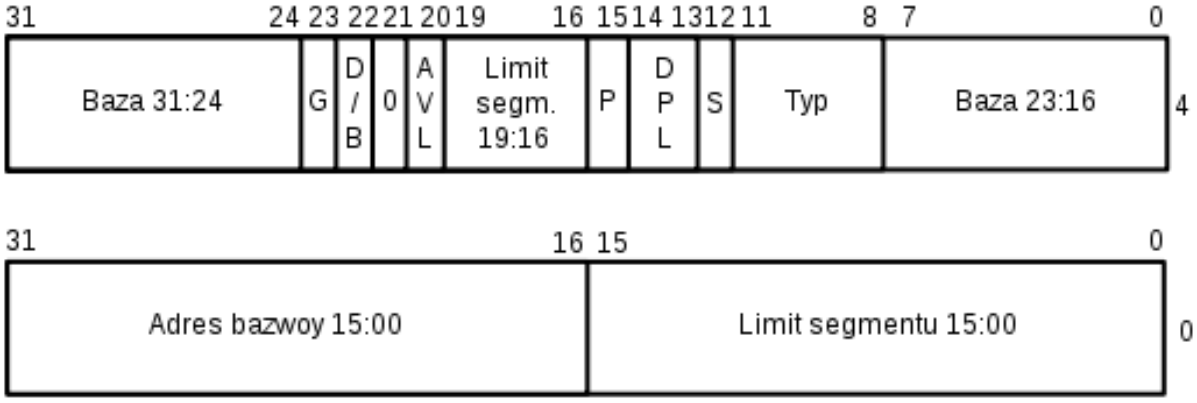
```
segment:offset = 0x1000:0xA000
adres fizyczny = 0x1000 * 0x10 + 0xA000 = 0x1A000
```

Więc wracając do pierwszego przykładu z 0x5f010:

```
mov bx, 0x5000
mov es, bx
mov [es:0xf010], ax
```

Generalnie rzecz ujmując, w 32-bitowym trybie chronionym koncept segmentów i offsetów pozostał, tak implementacja jest zupełnie inna. Gdy procesor przełącza się na ten tryb, to zamiast mnożyć wartość segmentu przez 16 i dodawać do offsetu, rejestry segmentowe (cs, ss, ds, es) zawierać będą selektor deskryptora (jego numer), który określać będzie bezpośrednio deskryptor segmentu. Easy.

Deskryptor segmentu to 64-bitowa struktura zawierająca adres bazowy segmentu, czyli to, gdzie się znajduje w pamięci fizycznej, limit segmentu, czyli rozmiar oraz flagi, które zawierają informacje o prawach dostępu, typ, status, ziarnistość czy rozmiar.



Rysunek 1 https://pl.wikipedia.org/wiki/Deskryptor_segmentu

Powyższy schemat reprezentuje faktyczną strukturę deskryptora pamięci. Proponuję zwrócić uwagę na fakt, że adres bazowy rozbity jest na trzy różne części, żeby nie było za łatwo. Tak samo 4 najstarsze bity limitu segmentu wyrzucone są z dala od młodszych 16. Nie będziemy się zbytnio zgłębiać w możliwości konfiguracji, jestem przekonany, że znajdziemy je w jakiejś dokumentacji. Zamiast tego zaimplementujemy tzw *„basic flat model”*, czyli dwa porywające się ze sobą, 4GB segmenty danych i kodu. Czyli żadnej paginacji, żadnych zabezpieczeń, cała pamięć dostępna, prawie jakbyśmy byli trybie rzeczywistym. Takie rzeczy można spróbować potem zaimplementować używając języka wysokiego poziomu.

Nasz segment kodu składać się będzie z następujących informacji i flag:

- Baza - 0x0
- Limit - 0xfffff
- P (present) – 1.
- DPL – 0 (0 – najwyższy poziom uprawnień, 3-najniższy)
- S – typ deskryptora – 1 dla segmentu danych i kodu, 0 dla segmentu systemowego
- Typ –
    - Kod: 1 dla kodu, 0 dla danych
    - Zgodność: 0. Oznacza, że kod w segmencie z niższymi uprawnieniami nioe może wywołać kodu w tym segmentcie
    - Odczyt: 1. 0 jeśli execute-only.
    - Dostęp: 0. To jest ustawiane przez procesor przy dostępie.
- G – ziarnistość – 1. Ustawiona flaga mnoży limit przez 4K (16*16*16), więc segment zwiększy się do 4GB.
- D/B – 1 – segment będzie przechowywał 32-bitowy kod. 0 dla kody 16-bitowego
- L – 0 – segment 64-bitowy. Na razie pracujemy na 32 bitach, stąd 0.
- AVL – 0. Do naszego własnego użytku.

Segment danych będzie bardzo podobny z jedyną różnicą we fladze typu. Teraz w zasadzie pozostaje implementacja. Tu ważna informacja: CPU musi znać rozmiar GDT, więc musimy dopisać jeszcze do niej deskryptor, który zawierać będzie adres i rozmiar GDT.

```
gdt_start:

gdt_null:
dd 0x0
dd 0x0

gdt_code:
dw 0xffff ; Limit 0-15
dw 0x0 ; Baza 0-15
db 0x0 ; Baza 16-23
db 10011010b ; flagi typu
db 11001111b ; reszta flag, limit 16-19
db 0x0 ; baza 24-31

gdt_data:
dw 0xffff ; Limit 0-15
dw 0x0 ; Baza 0-15
db 0x0 ; Baza 16-23
db 10010010b ; flagi typu
db 11001111b ; reszta flag, limit 16-19
db 0x0 ; baza 24-31

gdt_end: ; gdt start i end sluza nam do policzenia rozmiaru GDT
         ; dla deskryptora gdt

gdt_descriptor:
dw gdt_end-gdt_start-1 ; rozmiar tablicy deskryptorów
dd gdt_start           ; adres startu


CODE_SEG equ gdt_code - gdt_start
DATA_SEG equ gdt_data - gdt_start

; to jest aby procesor wiedział, gdzie się odwołać, kiedy
; ustawimy np. DS na 0x10 (data segment)
```

Kiedy komputer bootuje to, mimo że zwykle jest wyposażony w jakiś bardziej złożony układ graficzny, zaczyna w kolorowym tekstowym trybie VGA wyświetlający tablicę 80x25 znaków. W trybie tekstowym nie ma, thanks god, konieczności deklarowania indywidualnych pikseli, ponieważ w pamięci kontrolera zapisana jest już jedna czcionka. Zatem aby wyświetlić znak wystarczy wpisać go wraz z atrybutem w odpowiednie miejsce w pamięci.

Adresem pamięci VGA jest 0xb8000. Od tego miejsca licząc na każdy znak mamy przeznaczone dwie komórki pamięci a wszystko zapisywane jest sekwencyjnie.

Zadanie 1.

Napisz funkcję drukującą, może być na podstawie tej z poprzednich zajęć, ale musi działać na pamięci, bez przerwań. Na razie nie zadziała na 32 bitach, bo nie mamy bootloadera, ale to chwila moment i już będzie śmigać.

Teraz już przesiadka na 32 bity

Faktyczny switchover jest dość prosty. Wymagać będzie od nas wyłączenia przerwań (instrukcją cli) ze względu na to, że 16-bitowe przerwania BIOS nie zadziałają w 32-bitowym trybie chronionym, więc do czasu zadeklarowania nowej tablicy wektorów przerwań

będziemy sobie musieli radzić bez nich. Następnie musimy określić procesorowi GDT, które przed chwilą wymęczyliśmy

```
lgdt [gdt_descriptor]
```

Następnie należy przestawić pierwszy bit rejestru cr0, czyli specjalnego rejestru kontrolnego procesora.

```
Mov eax, cr0
or eax, 0x1
mov cr0, eax
```

W zasadzie, to procesor już jest w 32-bitowym trybie chronionym, jak ustawimy cr0, przy czym ważne jest jeszcze wykonanie dalekiego skoku, żeby, w uproszeczniu, odetkać rury, czyli upewnić się, że procesor nie wykonuje nam równolegle dwóch instrukcji. Potem musimy jeszcze wskazać wszystkim rejestrom segmentowym na nowy segment danych z GDT. Czyli cały kod przełączenia będzie wyglądać następująco:

```
[bits 16]
switch_pm:
cli
lgdt [gdt_descriptor]

mov eax, cr0
or eax, 1h
mov cr0, eax

jmp CODE_SEG:init_pm

[bits 32]
init_pm:
mov ax, DATA_SEG
mov ds, ax
mov cs, ax
mov ss, ax
mov es, ax
mov fs, ax
mov gs, ax

mov ebp, 0x90000
mov esp, ebp

call BEGIN_PM
```

Gdzie wywoływany na końcu BEGIN_PM odnosi się już do kodu bootsectora:

```
[org 0x7c00]

mov bp, 0x9000
mov sp, bp

mov bx, MSG_REAL
call print_string       ; wywołanie funkcji druku z poprzednich zajęć

call switch_pm

jmp $

%include "gdt.asm"
%include "32bit_switch.asm"
%include "32bit_print.asm"
%include "16bit_print.asm"

[bits 32]

BEGIN_PM:
mov ebx, MSG_PROTECTED
call print_pm

jmp $


MSG_REAL db "16-bitowy tryb"
MSG_PROTECTED db "a teraz 32-bitowy tryb"

times 510-($-$$) db 0
dw 0xAA55
```

Teraz jeszcze to należy dostosować do wcześniej określonej funkcji drukującej z pamięci i jesteśmy gotowi do drogi.