Boot sektor

Czyli co się dzieje zanim coś się zacznie dziać?

Start komputera

Tryb 16 bitowy (Real Mode)

Tryb 32-bitowy (Protected Mode)

Piszemy system operacyjny

DEMO 7

MAGIC!

Rozmowa z hardware

Hello!

DEMA 2 & 3

DEMO 8

DEMO 1

Dziękuję za uwagę

DEMO 4

Źródła

DEMO 6

DEMO?

Tak, DEMO 5

Boot sektor

Czyli co się dzieje zanim coś się zacznie dziać?

Artur Pacholec

Start komputera

Gdy włączamy komputer, nie mamy systemu operacyjnego (duh!)

System będzie dopiero załadowany z dysku (twardy, dyskietka, usb).

BIOS po zrobieniu swoich testów hardware'u zabierze się za szukanie systemu operacyjnego.

Jednak BIOS nie zna pojęcia systemu plików, nie wie co to plik.

BIOS widzi jedynie sektory.

Słodkie, słodkie sektory.

BIOS próbuje więc załadować pierwszy sektor (zwykle 512 bajtów) dysku oznaczonego jako główny (w ustawieniach BIOSu).

Jednak procesor nie widzi różnicy między programem a danymi więc skąd wiadomo czy dany dysk posiada w ogóle bootsektor?

BIOS iteruje po zainstalowanych dyskach (w określonej kolejności) i na końcu pierwszego (index 0) sektora szuka dwóch bajtów:

```
0xAA55
```

Kiedy je znajdzie, jest bardzo z siebie zadowolony i zaczyna wykonywać kod znajdujący się w sektorze 0 tego dysku.

DEMO 1

Nie, większego się nie dało :(

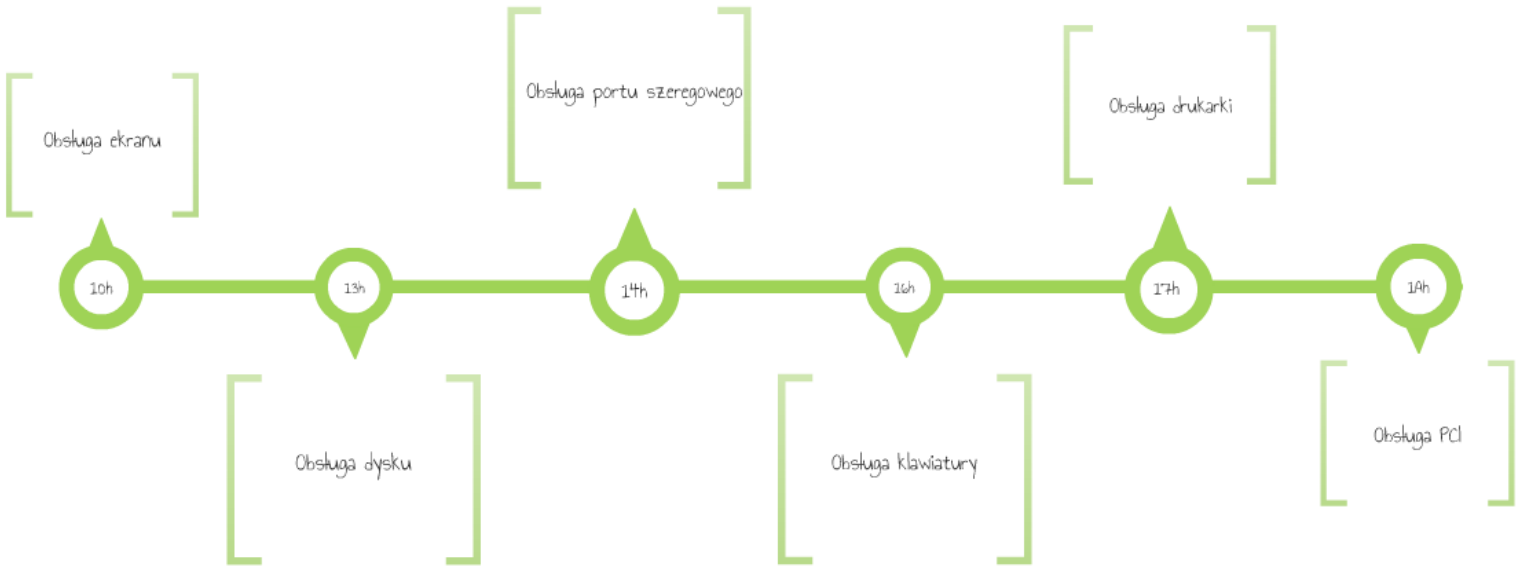
Tryb 16-bitowy (Real Mode)

Dla zachowania kompatybilności wstecznej komputery z mikropocesorami z rodziny x86 startują zawsze w trybie emulacji 8086.

Tryb ten nie obsługuje ochrony pamięci.

Dostęp mamy tylko do rejestrów 16-bitowych (więc także adresowanie jest 16b).

Tak samo jak DOS czy Linux, w BIOSie mamy dostęp do pewnych przerwań których działanie zależy od wartości rejestrów AH i AL.



Moje ulubione to 15h AH: 00h - Włącz odtwarzacz kaset.

Obsługa ekranu

Obsługa dysku

Obsługa portu szeregowego

14h

16h

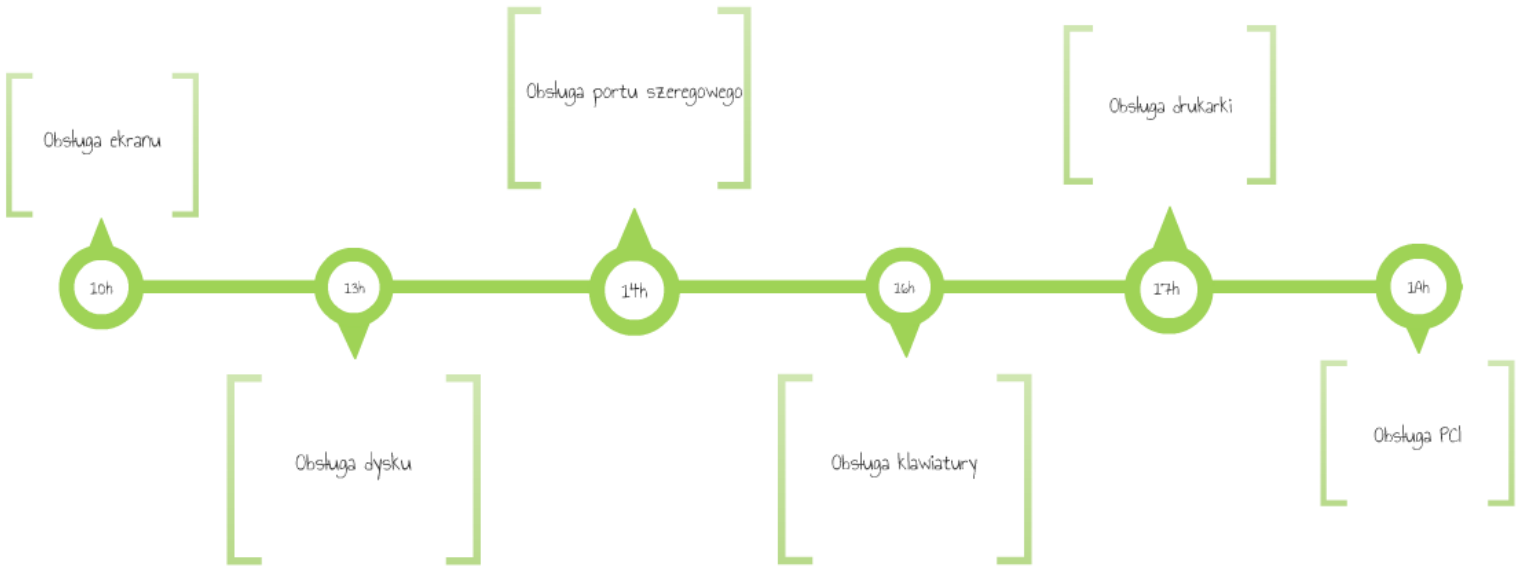
Obsługa klawiatury

Obsługa drukarki

17h

Obsługa PCI

Tak samo jak DOS czy Linux, w BIOSie mamy dostęp do pewnych przerwań których działanie zależy od wartości rejestrów AH i AL.



Moje ulubione to 15h AH: 00h - Włącz odtwarzacz kaset.

Hello?

Is it me, you're looking for?

Ponieważ jesteśmy poważnymi studentami II stopnia, proponuję zacząć od "Hello world".

DEMA 2 & 3

Jak wiadomo BIOS wczytuje z pamięci instrukcje i przekazuje je do wykonania procesorowi.

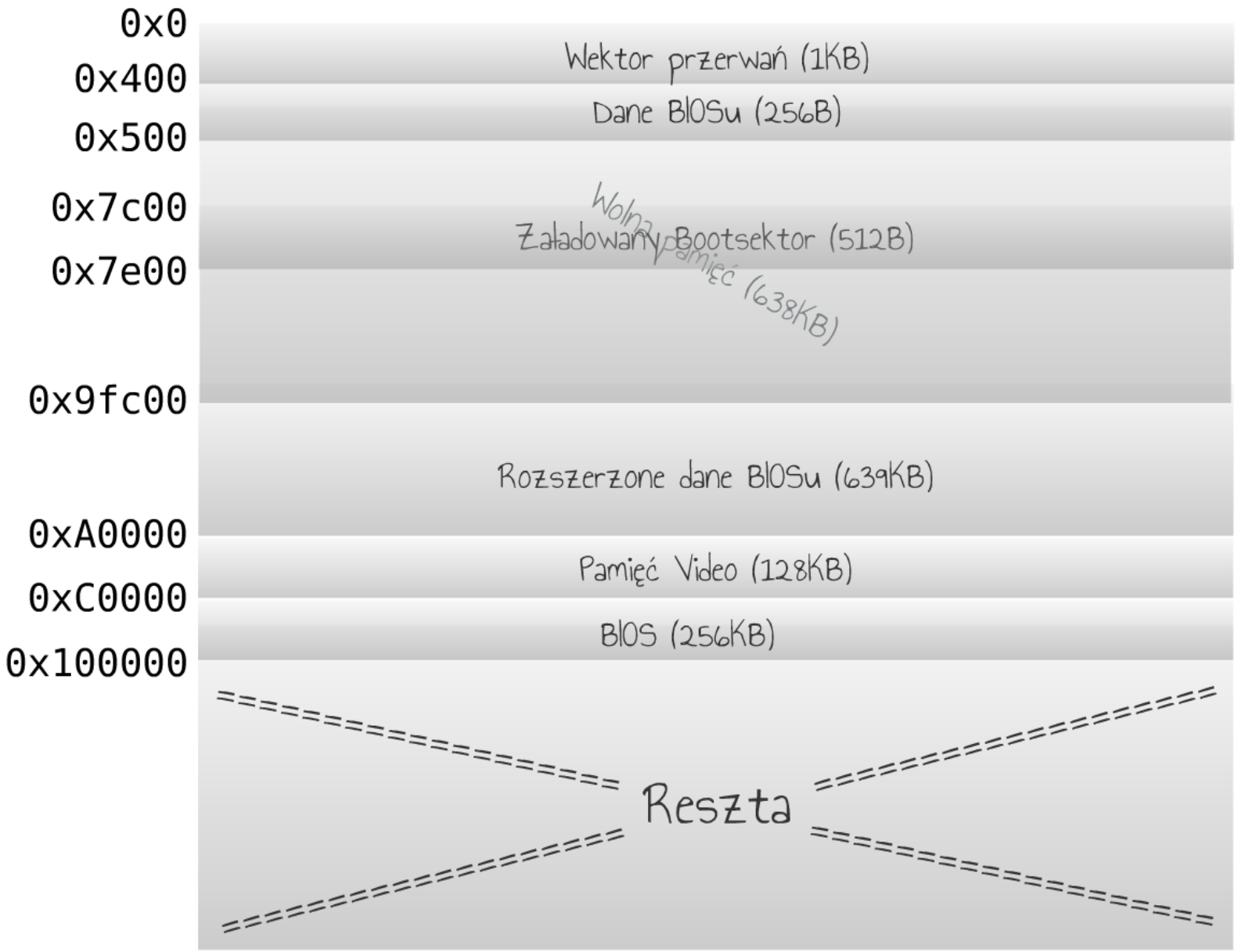
Ale gdzie dokładnie w pamięci znajduje się nasz kod? Czy pod adresem 0?

Nie do końca.

BIOS działał wiele cykli zanim zaczął ładować nasz bootsektor i będzie działał jeszcze wiele cykli po załadowaniu go.

Okazuje się, że BIOS ma z góry ustalony sposób gdzie w pamięci trzymane są poszczególne komponenty.

Kod bootsektora będzie zawsze pod adresem `0x7c00`.


0x0
0x400
0x500
0x7c00
0x7e00
0x9fc00
0xA0000
0xC0000
0x100000
Wektor przerwań (1KB)
Dane BIOSu (256B)
Załadowany Bootsektor (512B)
Wolna pamięć (638KB)
Rozszerzone dane BIOSu (639KB)
Pamięć Video (128KB)
BIOS (256KB)
Reszta

DEMO 4

Jednak zazwyczaj całego systemu oepracyjnego w 512 bajtach nie zmieścisz dlatego fajnie by było móc wczytać coś z dysku i zacząć to wykonywać.

Na szczęście BIOS oferuje przerwania do obsługi dysku.

Ponieważ jesteśmy w trybie 16b to rejestry mogą maksymalnie przyjąć wartość `0xffff` co odpowiada 64 KB (65536B) - lepiej, ale wciąż mało.

Na ratunek przychodzą nam rejestry segmentowe CS, DS, ES i SS.

Nasza pamięć zostaje podzielona na sektory i jeśli chcemy wykonać instrukcję na danych w określonym sektorze, musimy podać jego indeks w odpowiednim rejestrze.

Np dla `mov AX, [0x123]` będziemy używac DS (Data Segment).

A dla instrukcji stosu będziemy używać SS (Stack Segment).

Procesor mnoży wartość danego rejstru segmentowego przez 16 i dodaje go do operandu instrukcji.

Więc jeśli ustawimy DS= `0x4D` i podamy instrukcję `mov AX, [0x20]` wartość do AX naprawdę zostanie załadowana z `0x4F0` (16 * `0x4d` + `0x20`).

W ten sposób możemy zaadresować `0xffff` * 16 + `0xffff` = ~1MB (WOW!)

Wracając do dysku...

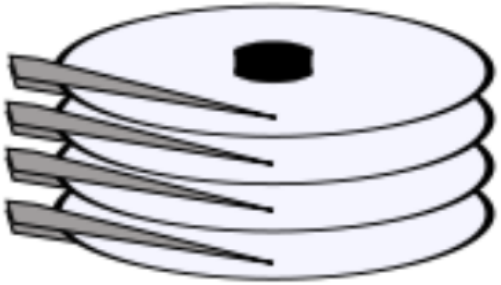
Jak zbudowany jest dysk magnetyczny mniej więcej każdy wie, talerze, głowice itd.

Dysk SSD mimo że nie ma talerzy, nie chce być antykonformistą i żyje w kłamstwie, udając że je ma.

Jednak jak dokładnie wskazać co chcemy odczytać?

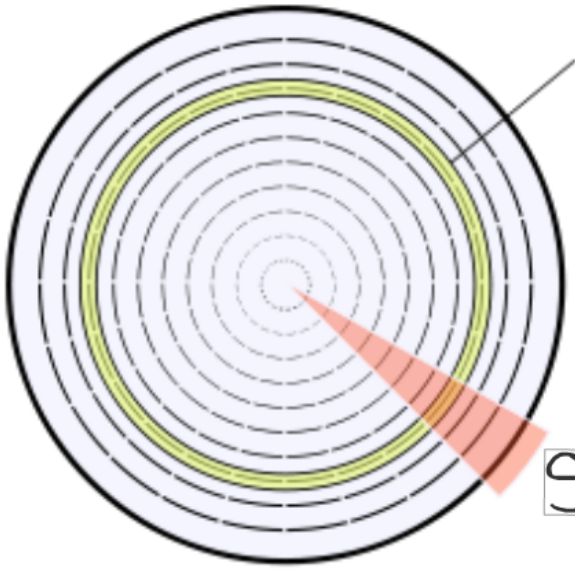
Do wskazania konkretnego obszaru dysku będziemy stosować adresowanie CHS (Cylinder-Head-Sector).

- Cylinder (cylinder) - dystans od krawędzi talerza
- Head (głowica) - strona konkretnego talerza
- Sector (sektor) - fragment cylindra



4 Talerze

8 Głowic



DEMO?

Tak, DEMO 5

Tryb 32-bitowy (Protected Mode)

Tryb 16b jest całkiem przyjemny, ale my chcemy WIĘCEJ mocy, dlatego przełączymy się w tryb 32b. Różnice między trybami:

- Rejestry 32b, czyli EAX, EBX itp.
- Dwa dodatkowe rejestry segmentowe: FS i GS.
- Możemy teraz adresować 4GB pamięci (`0xFFFFFFFF`).
- Możemy określić chronione segmenty pamięci, instrukcje z innych segmentów nie będą miały dostępu do tych segmentów.
- Możemy określić pamięć wirtualną podzieloną na strony, nieużywane strony mogą być przechowywane na dysku.

Najtrudniejsze w przełączaniu się na tryb 32b jest fakt, że musimy stworzyć w pamięci strukturę nazywaną GDT (Global Descriptor Table) która określa segmenty pamięci i ich atrybuty.

Po jej stworzeniu użyjemy odpowiedniej instrukcji procesora by wiedział gdzie nasza struktura się znajduje, następnie zmienimy bit w specjalnym rejestrze by dokonać zmiany trybu.

Wszytko by było fajnie, gdyby nie to, że tą strukturę musimy stworzyć w ASM, co nie będzie wcale ładne.

To co, jedziemy?

Chwila, jeszcze nie wspomniałem o największym plusie trybu 32b: po zmienie nie będziemy juz mieli dostępu do BIOSa. Powiedziałem plusie? Miałem na myśli minusie.

Jeśli ktoś myślał że to co dotąd robiliśmy było hardkorowe to jeszcze mało widział...

Więc jak mamy teraz coś wypisać na ekran skoro nie mamy przerwań?

Otóż po włączeniu komputera, włącza się tryb tekstowy gdzie mamy 80x25 znaków do dyspozycji.

Cały nasz ekran jest mapowany na pewien obszar pamięci, dokładnie `0xB8000`

Żeby wypisać konkretny znak w konkretnym miejscu musimy obliczyć przesunięcie danego znaku na ekranie względem pamięci i tam umieścić żądany kod ASCII.

Każdy znak na ekranie reprezentowany jest przez dwa bajty, pierwszy to kod ASCII, drugi to atrybuty jak kolor tła czy czcionki.

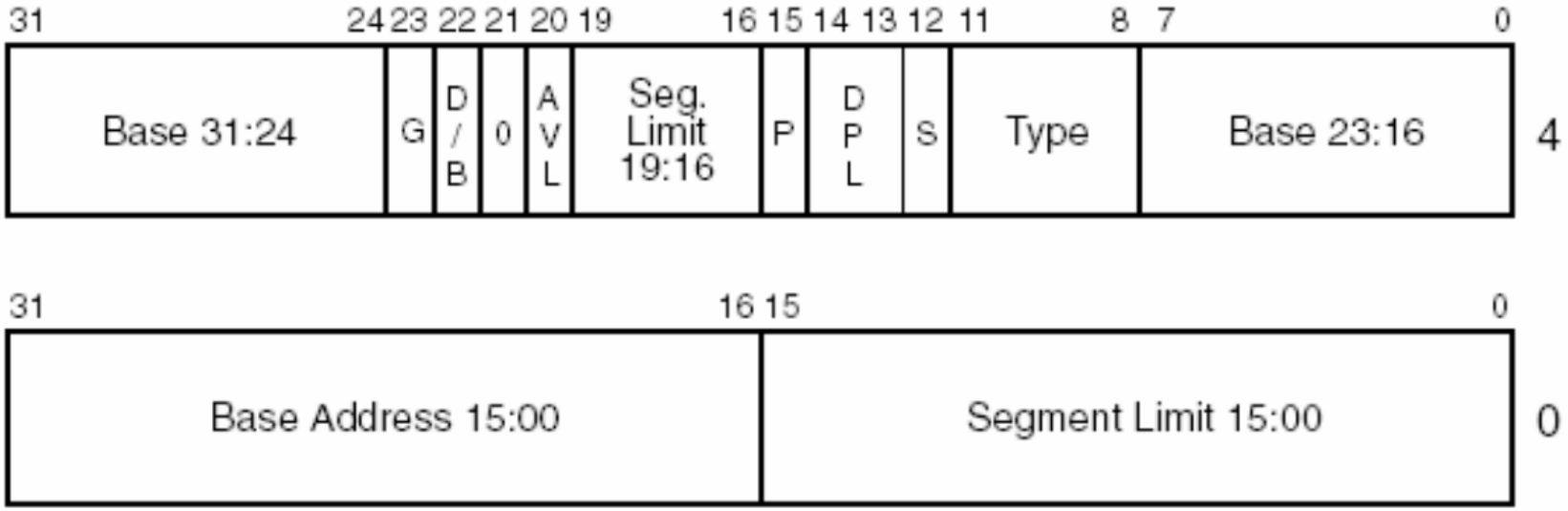
Więc ogólny wzór na pozycję znaku to `0xb8000 + 2 * (rząd * 80 + kolumna)`.

Wspominałem już o dostępie do większych adresów pamięci przez użycie rejestrów segmentowych.

W trybie 32b jest podobnie, ale inaczej, ES nie jest mnożony przez 16 i dodawany do adresu. ES będzie oznaczać indeks konkretnego deskryptora segmentu (SD) w GDT.

Deskryptor segmentu to 8-bajtowa struktura która definiuje właściwości segmentu pamięci chronionej takie jak jego początek, rozmiar i atrybuty (read only itp).

Procesor wymaga aby pierwszy segment był segmentem zerowym, jako zabezpieczenie przed błędnymi wartościami rejestrów segmentowych.



BASE - Adres segmentu

LIMIT - Rozmiar segmentu

TYPE - Rodziaj (0 - systemowy, 1 - dane/kod)

DPL - Poziom dostępu

Zwróćmy uwagę jak pięknie i logicznie rozłożone są pola BASE i LIMIT w tej strukturze.

Możliwe że są ku temu jakieś powody historyczne, ja osobiście myslę, że ktoś w Intelu nie lubi programistów...

Dla uproszczenie stworzymy tzw. płaski model pamięci: dwa segmenty zajmujące całą pamięc, nałożone na siebie. Jeden na dane, drugi na kod.

Segment kodu będzie wyglądał tak:

- Base - 0 (od początku)
- Limit - 0x2000 (bo mamy 32MB pamięci)
- Granularity - 1 (jeśli 1 to Limit jest mnozony przez 4K)
- Privilige - 0 (najwyższy poziom dostępu)
- Present - 1 (bo nie jest wirtualny)
- Type - 1
- Code - 1
- Readable - 1

Segment danych będzie wyglądał podobnie do kodu oprócz tych różnic:

- Code - 0
- Writeable - 1

Ponieważ jednak procesor nie wie dokładnie ile segmentów będzie w naszej GDT, podanie samego adresu by nie wystarczyło.

Dlatego podajemy procesorowi tzw. deskryptor GDT, który mówi gdzie zaczyna się GDT i jaki ma rozmiar.

DEMO 6

Może wreszcie coś ciekawego (?)

Piszemy system operacyjny

Jak tu doszliśmy to znaczy, że nawet się wyrabiam. He He He

Wiem, że wszyscy kochamy asm i najchętniej byśmy pisali system operacyjny tylko w nim, mieli miliardy linijek kodu i w ogóle żyć nie umierać

Jednak jako że przedmiot nie ma w nazwie "assembler" to będę grał na cheatach i przejdę na C.

Mogę? Dziękuje :)

DEMO 7

Jak to też nie będzie ciekawe to to już nie moja wina :/

Rozmowa z hardware

Pokazałem juz jeden ze sposobów radzenia sobie z brakiem BIOSu - mapowanie pamięci. Dane zapisywane do konkretnego obszaru pamięci są zapisywane do bufora urządzenia.

Bardzo dawno temu procesory miały dedykowane piny do obsługi hardware, np. pin do włączania silniczka stacji dyskietek.

Jednak to by wymagało by procesor pracował z prędkością najwolniejszego urządzenia żeby się z nim synchronizować, co przy dzisiejszych prędkościach CPU byłoby marnowaniem zasobów.

W dzisiejszych czasach mamy kontrolery urządzeń podłączone do magistrali którym mówimy co mają zapisać i odczytać, a one wykonują te operacje w swoim czasie.

W strukturze x86 rejestry tych kontrolerów są mapowane na przestrzeń adresową I/O która jest zupełnie oddzielna od głównej przestrzeni adresowej.

Używamy do tego instrukcji `in` i `out`.

Na przykład, stacja dyskietek których używamy na codzień ma przypisany adres `0x3F2`, żeby włączyć silniczek takiej stacji moznaby napisać:

```
mov DX, 0x3F2              ; Adres portu zapisujemy w DX
in AL, DX                  ; Wczytujemy zawartoć portu do AL
or AL, 00001000b           ; Zmieniamy bit silniczka
out DX, AL                 ; Aktualizujemy port
```

DEMO 8

Ostatnie (hurra), ale duże (buu)

Skoro mamy już punkt wejściowy do C to teraz z górki.

Wystarczy jeszcze tylko zaimplementować:

- Zarządzanie procesami
- Obsługę wielowątkowości
- System plików
- Zarządzanie pamięcią
- Obsługę sieci
- Sterowniki (do wszystkiego)
- Obsługę pamięci wirtualnej
- Obsługę czasu rzeczywistego
- Komunikację międzyprocesową
- Obsługę wielu użytkowników
- Interfejs graficzny
- Ochronę procesów
- Ochronę pamięci

To co, jedziemy?

Nie dzisiaj...

Dziękuję za uwagę

^ ^

—

Źródła

http://wiki.osdev.org/
http://en.wikipedia.org/
http://google.com