Własny system operacyjny cz.1

Czyli pół godziny instalowania, aby zobaczyć czarny ekran

Zacząć musimy od absolutnej podstawy, czyli doinstalowania potrzebnych pakietów, bo mamy na razie prawie gołego linuxa, a to, co będziemy robili wymaga odpalenia okienka, a co za tym idzie – interfejsu graficznego. Co prawda da się to wszystko zrobić w trybie tekstowym, ale w wypadku każdego, nawet najmniejszego fakapu zmuszeni będziemy do rebootu maszyny.
Na początek, musimy przełączyć się na roota, bo czeka nas sporo instalowania:

```
su root
```

Zaczniemy od instalacji znanego nam już z WIA2 nasma. Sorry, będzie asembler, bez tego nic nie napiszemy, ale będę tłumaczył, także ez.

```
apt install nasm
```

Następnie musimy zainstalować qemu – jest to szybki, a zarazem prosty open-source'owy emulator, który pozwala na odpalenie wielu systemów równolegle na jednej maszynie – chyba, że ktoś pracuje na maku z M1, wtedy nie pozwala na nic. Pracować będziemy w trybie emulacji systemu, co np. pozwoli na współpracę z urządzeniami I/O i w ogóle, super sprawa.

```
$ apt install qemu-system
```

Teoretycznie wystarczyć powinno

```
$ apt install qemu-system-x86
```

ale jakby ktoś chciał w przyszłości pobawić się na innej architekturze, to może się przydać więcej pakietów.

Właściwie, to w tym momencie można zbudować jakiś plik binarny z bootsectorem, załadować go do qemu i mieć własny, ekhm, system, ale bez możliwości odpalenia wielu terminali sobie co najwyżej zablokujemy system i będziemy musieli od zera odpalać maszynę, więc:

```
$ apt update
$ apt install task-xfce-desktop
```

To trochę potrwa i zajmie ok. pół GB na dysku. Gdy instalacja się zakończy musimy jeszcze ustawić domyślny target na graficzny:

```
$ systemctl set-default graphical.target
```

Po tej serii poleceń wystarczy w zasadzie tylko zrebootować system i możemy śmigać na GUI

A jak już jesteśmy w GUI, to nie pozostaje nic innego, jak zacząć prace nad systemem operacyjnym.

Boot

Zaczniemy od BIOS-u, czyli **Basic Input/Ouput System**. Zajmuje się on inicjalizacją hardware'u, gdy system się bootuje. Jest on już domyślnie zainstalowany na płycie PC i zarazem jest pierwszym softwarem, który jest uruchamiany podczas włączania komputera. Emulator QEMU używa open-source'owego SeaBIOS. Gdy komputer jest uruchamiany BIOS przejmuje kontrolę, inicjalizuje hardware (domyślnie klawiaturę i ekran) i testuje pamięć. Potem odczytywana jest z CMOS data i czas, a następnie wczytywany jest system operacyjny. I tu pojawia się problem – skąd BIOS wie, co jest systemem i skąd ma go wziąć? Bez wdawania się na razie w szczegóły tego, jak działa GRUB, wczytywany jest pamięci 512-bajtowy segment z pierwszego dysku i wykonuje kod tam zamieszczony. Zawarte tam instrukcje determinują resztę procesu bootowania. Ten 512-bajtowy segment nazywany jest MBR – **Master Boot Record**.

MBR składa się z trzech podstawowych sekcji – pierwsze 446 bajtów jest zarezerwowane na kod programu. Następne 64 bajty zawierają tabelę partycji, a to pozostawia nam dokładnie dwa bajty na kluczowy element MBR, czyli tzw. *Magic numer (AA55).* Bez tych dwóch bajtów na końcu w zasadzie nie wydarzy się nic szczególnego, ponieważ MBR uznany zostanie na nieważny i nic się nie wydarzy.

Wykonywany kod to bootloader – ładuje on jądro systemu, GRUB-a, kolejny program rozruchowy, albo to, co znajduje się w /boot na linuxach.

No to teraz pora na trochę kodowania: napiszemy upośledzony bootloader, który będzie wykonywał nieskończoną pętlę. Najpierw musimy utworzyć plik .asm, który sobie zaraz nasmem zbudujemy do pliku binarnego.

```
loop:
     jmp loop

times 510-($-$$) db 0
dw 0xaa55
```

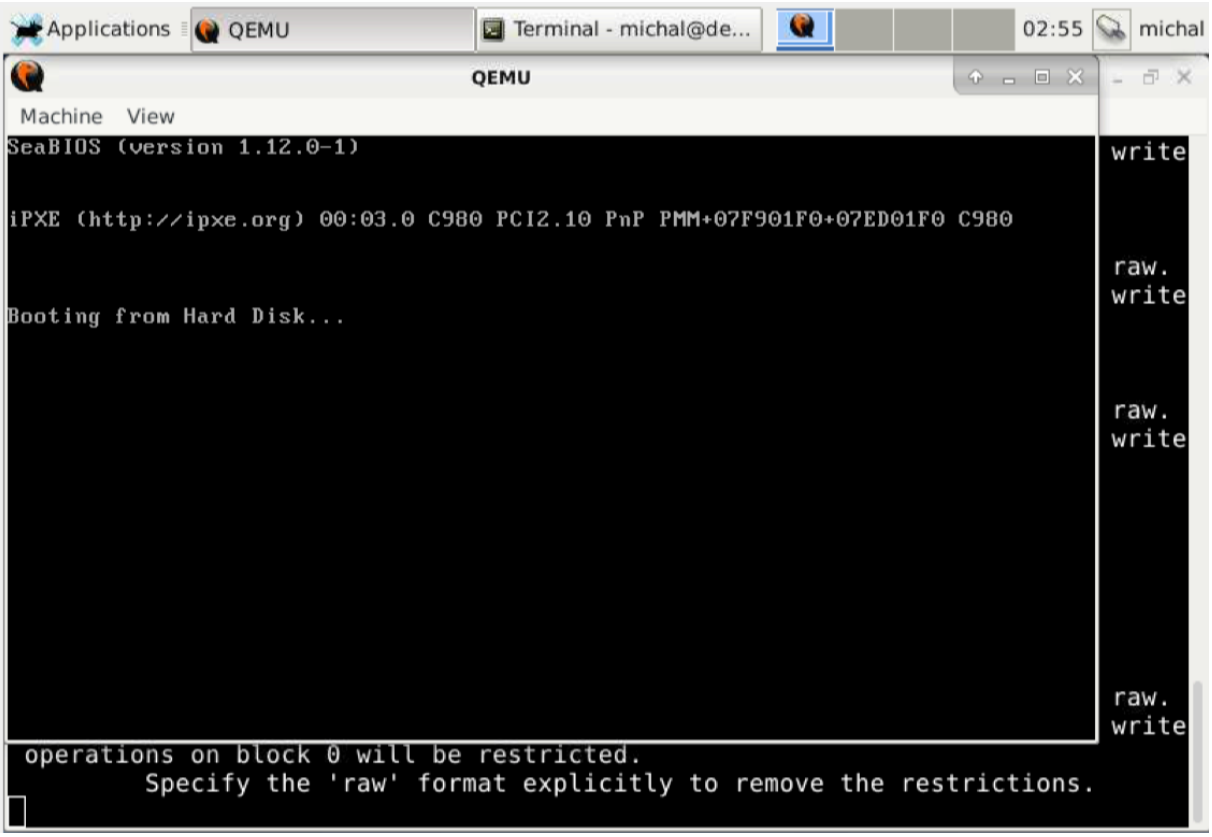
pierwsze dwie linie powinny być oczywiste, jest to pętla wykonująca się w nieskończoność. Dalej mamy instrukcję, która wypełnia nam niezapełnione bajty programu zerami – do instrukcji times podajemy wartość 510-(bieżąca_pozycja-początek_sekcji) – tyle razy zapisane zostanie podane dalej 0. Ostatnia linijka umieści nam po tych zerach, czyli na końcu pliku nasz *magic number.*

Taki plik .asm musimy zasemblować do pliku binarnego

```
$ nasm -f bin z1.asm -o z1.bin
```

Nie pozostaje nam nic innego, jak załadować ten pliczek do emulatora:

```
$ qemu-system-x86_64 z1.bin
```



I tak oto powstał nasz „system operacyjny”, który nic nie robi poza wykonywaniem nieskończonej pętli. Proponuję podejrzeć sobie, jak wygląda nasz program w wersji heksowej, użyć można do tego absolutnie najśmieszniejszego programu na świecie

```
$ xxd z1.bin
```

Przydałoby się jakoś się przekonać, że faktycznie coś robi, nie? W tym celu użyjemy przerwania procesora. Jeśli ktoś ma flashbacki z WIA2, to słusznie.
http://www.ctyme.com/intr/rb-0106.htm – przerwanie 10h/AH=0Eh, czyli wyświetlanie teletype, które automatycznie przesuwa kursor i ewentualnie przescrolluje ekran, jeśli będzie taka potrzeba.

Zadanie 1

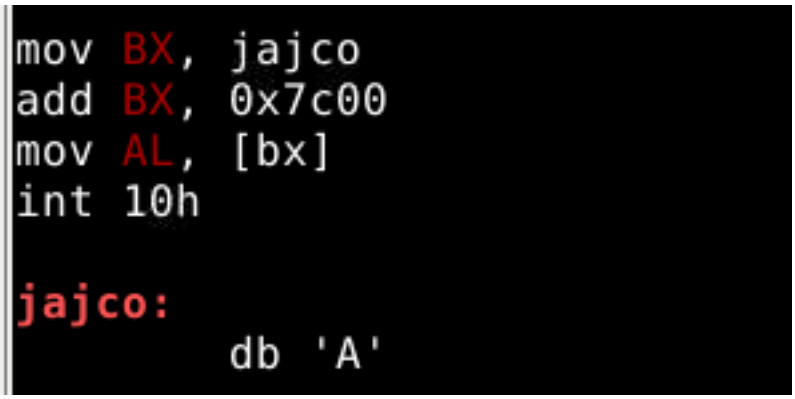
Wydrukuj coś przy bootowaniu systemu:

```
SeaBIOS (version 1.12.0-1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F901F0+07ED01F0 C980


Booting from Hard Disk...
HylaOS
```

Pamięć

Wiemy już, że te 512 bajtów programu jest gdzieś ładowane, pozostaje na ustalić, gdzie konkretnie BIOS je ładuje. Odpowiedź brzmi 0x7C00. Co za tym idzie, kiedy będziemy chcieli

wczytać jakiekolwiek dane z pamięci, to musimy mieć na uwadze, że adresy są zoffsetowane o podaną wyżej wartość.



```
mov BX, jajco
add BX, 0x7c00
mov AL, [bx]
int 10h

jajco:
        db 'A'
```

Bez offsetu kod:

```
MOV AH, 0x0E
MOV AL, [jajco]
int 10h
```

Zwróci nam jakiś losowy śmietnik. Aby ten problem rozwiązać, możemy za każdym razem dodawać wartość 0x7C00, jak na przykładzie powyżej, ale możemy również po prostu postawić znany z WIA2 offset. Tam ustawialiśmy offset na 100h, tu ustawiamy

```
org 0x7C00
```

Teraz kod z bezpośrednim adresowaniem powinien śmigać aż miło.