Zadanie 1

Napisz frontend sterownika urządzenia blokowego[1,2]. Przykładowa struktura sterownika urządzenia:

```
struct dev_t
{
        // informacje podstawowe
        FILE* f; // plik źródłowy

        int block_size; // wielkość bloku
        int block_count; // liczba bloków
};

dev_t* device_open(const char* file_name)
{
        // tutaj należy otworzyć plik wejściowy
        // przypisać wielkość bloku
        // oraz określić liczbę bloków

        // lub NULL, jeśli nie udało się otworzyć
}

int device_close(dev_t* device)
{
        // tutaj należy zamknąć urządzenie i zwolnić pamięć dev_t

        return 0; // kod błędu
}

int device_read(dev_t* device, void* buffer, int start_block, int count)
{
        /*
        Wczytaj z urządzenia [device] bloki do bufora [buffer].
        Wczytać należy [count] bloków, zaczynajc od [start_block]
        */
}

int device_write(dev_t* device, cosnst void* buffer, int start_block, int count)
{
        /*
        Zapisz do urządzenia [device] bloki z bufora [buffer].
        Zapisać należy [count] bloków, zaczynajc od [start_block].
        */
}
```

Oczywiście nie jest to rzeczywiste urządzenie, a jedynie symulowane za pomocą pliku. Stąd właśnie mowa o frontendzie – zestawie funkcji (API) pozwalających na dostęp do urządzenia. W rzeczywistych realizacjach funkcje takie nie odwołują się do pliku a do rzeczywistego sprzętu (backend).
Typ symulowanego urządzenia to **nośnik dyskowy** o standardowej wielkości sektora = 512 bajtów. Jest to też wielkość bloku urządzenia blokowego. Zaimplementować należy podstawę: 4 funkcje API, pozwalające otwierać oraz zamykać dostęp do urządzenia a także czytać i pisać do niego.

- Funkcja `device_open` ma otwierać plik i zwracać wskaźnik do struktury kontrolnej urządzenia. Jeśli otwracie się nie uda, funkcja musi zwrócic NULL. Ponadto funkcja powinna też określać liczbę oraz wielkość bloków. W przypadku zadania pliki mają równo 32MB.
- Funkcja `device_close` ma zamykać urządzenie, co oznacza zamknięcie pliku. Musi ona również porządkować pamięć poprzez zwalnianie wszystkich zaalokowanych przez API sterownika bloków pamięci.
- Funkcja `device_read` odpowiada za wczytywanie bloków z urządzenia do pamięci (bufora). Funkcja ma odczytywać `count` bloków, zaczynać do `start_block`. Oczywiście bufor `buffer` musi zostać wcześniej zaalokowany na odpowiednią wielkość. Ważna jest odpowiednia interpretacja parametrów `start_block` i `count` oraz wartości zwracanej. Jeśli programista chce odczytać `count`=10 i `start_block`=20 z urządzenia które ma 35 bloków, to operacja powiedzie

1 https://en.wikipedia.org/wiki/Device_file#Block_devices
2 https://pineight.com/ds/block/

się w pełni i 10 bloków (5120 bajtów) zostanie odczytanych do bufora, a funkcja zwróci 10. Jeśli jednak programista zechce odczytać 40 bloków, to funkcja musi: zwrócić błąd (np. poprzez -1) albo wczytać tylko tyle, ile może (czyli 15 bloków). W tym przypadku zalecam zwracanie błędu.
- Funkcja `device_read` działa identycznie, z tym że bloki w buforze zapisywane są do urządzenia (pliku).

Zadanie 2

Dane są dwa pliki: volume_Cluster512.img oraz volume_Cluster1024.img. Są to woluminy dyskowe o wielkości 32MB. Zastosowany system plików to FAT16[3]. Główna struktura woluminy złożona jest z następujących elementów:

- Sektor rozruchowy[4,5] (1 blok – 512 bajtów). Zawiera on informacje podstawowe o woluminie, wielkość systemu plików, położenie, liczbę tablic FAT itd.
- Dwie tabele FAT – W pyrzpadku fat16 każdy wpis ma po 2 bajty (uint16_t) i oznacza numer klastra, podstawowej jednostki alokacji pamięci w systemie plików FAT.
- Katalogu głównego
- Klastrów (danych plików).

Do wykonania (proste):

- Zapoznaj się z budową sektora rozruchowego dla systemu plików FAT16.
- Zapoznaj się ze strukturą systemu plików FAT16.
- Napisz program wyświetlający podstawowe informacje o systemie na podstawie sektora rozruchowego.
- **Kod realizujący system plików ma korzystać z funkcji napisanych w zadaniu 1. Dzięki temu, jeśli podmienić kod tych funkcji na np. korzystający z fizycznych nośników pamięci, Wasz system plików nadal będzie działał bez najmniejszej modyfikacji.**
- Wyznacz liczbę strumieni plików (ang. *file streams* / *cluster streams*).
- Wyznacz liczbę zajętych oraz wolnych klastrów. Ile to będzie bajtów?
  - Tutaj uwaga: plik volume_Cluster512.img oparty jest na klastrach o wielkości 512 bajtów (1 sektor/blok) podczas gdy volume_Cluster1024.img posiada klaster o wielkości 1024 bajtów.

Do wykonania (średnio trudne):

- Wyświetl listę plików i katalogów z katalogu głównego.
- Wyświetl listę plików i katalogów z \kat2\kat21\Bioroid
-
- Wyświetl listę wszystkich plików i katalogów (może być w formie drzewa).
- Wyświetl listę wszystkich plików i katalogów wraz z atrybutami (daty modyfikacji/utworzenia, atrybuty dostępu, wielkość, klaster początkowy).

Do wykonania (trudne):

Napisz własne implementacje funkcji: `fopen`, `fread`, `fgets`, `ftell`, `fseek` no i własną wersję struktury FILE.

- Wyświetl zawartość pliku \zadania\zad.c
- Skopiuj plik \mmedia\mtr.mp4 na swój dysk i odtwórz. Czy otwiera się poprawnie?
- Zawartość całego woluminu do dowolnego katalogu na lokalnym dysku.

---

3 https://en.wikipedia.org/wiki/File_Allocation_Table#FAT16

4 https://en.wikipedia.org/wiki/Volume_boot_record
http://www.pcguide.com/ref/hdd/file/structVolume-c.html

5 http://www.ntfs.com/fat-partition-sector.htm

Do wykonania (bardzo trudne i równie ciekawe):
Napisz własne implementacje funkcji: `fwrite` oraz `fputs`.

- Zapisz w woluminie dowolne pliki o wielkościach: <512, == 512, 512-1024, ==1024, 2000, 4000, 1MB.
- Odczytaj je następnie i sprawdź, czy funkcje zapisu dobrze działają.
- Napisz program do kopiowania danych dysk lokalny ↔ wolumin .img.

Dodatkowe (proste):

- Napisz program, który dla podaje z klawiatury liczby megabajtów, utworzy pusty wolumin z systemem plików w formacie FAT16. Jest to nic innego niż odpowiednik polecenia *format* :)

Dodatkowe informacje:
Istnieje cała gama systemów plików, zarówno bardziej jak i mniej skomplikowanych niż FAT16. Z rodziny FATxx jest jeszcze 12 i 32, ale nie różnią się one mocno ponad długość indeksu w tabelach FAT (12 bitów i 32 bity). Ponadto w trakcie studiów literaturowych natkniecie się na pojęcie LFN (ang. *Long File Name*). Otóż w pierwszych systemach DOS (fat16,fat12) pliki miały 8 znaków nazwy + 3 znaki rozszerzenia. Wraz z nadejściem systemu Windows95 pojawiły się długie nazwy plików (ze znakami narodowymi - diakrytycznymi) do długości 255 bajtów. Ponieważ systemu FAT nie posiadają wspomagania dla takich nazw, utworzono specjalny sposób generowania danych w listach katalogów (taki „sposób” to popularny „hack”, czyli zmuszenie urządzenia/mechanizmu/algorytmu do robienia rzeczy, do których nie był zaprojektowany). Przygotowane woluminy są ich pozbawione, ale nic nie stoi na przeszkodzie, aby taką funkcjonalność zaimplementować. Wolałbym jednak, abyście w pierwszej kolejności zrealizowali zapis do woluminów.

Woluminy zostały przygotowane za pomocą programu ImDisk: http://www.ltr-data.se/opencode.html/
Ogromną liczbę informacji na temat systemu plików znajdziecie na wszelkich stronach poświęconych pisaniu amatorskich systemów operacyjnych (osdev, bona fide os development, itp…). Wynika to z faktu, że FAT jest jednym z najprostszych systemów plików do implementacji. Przykładowe implementacje dostępne są na Githubie, aczkolwiek wysokiej jakości to one nie mają :).