



Zadanie 2.3

2. System plików FAT12/16

3. Projekt: czytnik dysku, woluminu, katalogu głównego i plików (3.0)



Treść	Pliki	Historia	Pomoc
Termin 2020-12-20 zostało 10 dni	Trudność 	Próba 13 Punkty: 5	Ocena maszyny Czas testu: 16sek

Czytnik systemów plików, wymagania na ocenę 3,0

Celem projektu jest zbudowanie parsera kontenerów plikowych, zapisanych w formacie FAT12/16. We wszystkich trzech projektach operacje te ograniczone są jedynie do wejścia (operacje zapisu nie są wymagane).

Aby zrealizować projekt przygotuj zestaw funkcji API do realizacji następujących funkcjonalności:

1. Otwieranie, czytanie i zamykanie urządzenia blokowego (w formie pliku).
2. Otwieranie i zamykanie woluminu FAT12/16.
3. Otwieranie, przeszukiwanie, czytanie oraz zamykanie plików w systemie FAT.
4. Otwieranie, czytanie i zamykanie katalogów.

W wersji projektu na ocenę **3,0** operacje z punktu **3 i 4** mogą być ograniczone wyłącznie do obsługi katalogu głównego - operacje na podkatalogach nie są testowane.

Za wyjątkiem punktu **1** przyjmij, że testowany system plików jest określany na podstawie Twojego numeru albumu, zgodnie z wyrażeniem:

```
int bits_per_fat_entry = (student_id % 2 == 1) ? 12 : 16;
```

Obsługa błędów

Wszystkie funkcje zwracają kody błędów w sposób zgodny ze standardem POSIX, tj. za pomocą stałej `errno`, patrz [errno - Linux manual page](#). Kody błędów oznaczone są stałymi całkowitymi w pliku `errno.h`. Opis POSIX określa jedynie klasę błędu a dokładna przyczyna zależy od konkretnej funkcji. Dla przykładu `ENONMEM` oznacza brak pamięci, miejsca docelowego, `EINVAL` oznacza błędny argument (czegoś/gdzieś).

Przykład obsługi błędu:

```
int fd = open("fat16_test.img", O_RDONLY);
if (fd == -1) { // OR NULL if tested function returns a pointer
    perror("open");
    printf("Error code: %d\n", errno);
    return 1; // Things just went south.
}
```

Często popełnianym błędem jest sprawdzanie kodu błędu `errno` bez wcześniejszego sprawdzenia wyniku funkcji (w przykładzie to `fd`). Specyfikacja `errno` stanowi, iż w przypadku sukcesu swojej operacji wywoływana funkcja **nie ma obowiązku** ustawiania/zmiany kodu błędu na **0**. Zatem należy najpierw sprawdzić wynik wywoływanej funkcji, a dopiero potem (gdy ten oznacza błąd) sprawdzić zawartość `errno`.

Więcej informacji dostępnych jest także poleceniem `man 3 errno` w Twojej powłoce Linuxa (np. w udostępnionej maszynie wirtualnej).

Otwieranie, czytanie i zamykanie urządzenia blokowego (w formie pliku)

Wolumin danych jest plikiem, podczas gdy systemy plików pracują z urządzeniami blokowymi, gdzie podstawową jednostką wymiany danych jest blok. Długość takiego bloku to najczęściej 512 bajtów. W przypadku nośników danych bloki te zwane są sektorami. Istotne jest zatem, aby przygotować API "ukrywające" plikowy charakter woluminu pod postacią funkcji udostępniających jego zawartość w formie wspomnianych bloków. Oznacza to również, że dostęp do pliku-woluminu za pomocą funkcji `fopen()`, `fread()`, `fclose()`, itp. jest możliwy **wyłącznie** w funkcjach z tej sekcji, czyli w `disk_open_from_file()`, `disk_read()` oraz `disk_close()`. Użycie funkcji plikowych w pozostałych funkcjach jest **niedozwolone**.

Dzięki temu przygotowany parser będzie mógł być zastosowany wszędzie tam, gdzie dostępne jest API dostarczające bloki o zadanych indeksach.

Dodatkowe informacje można znaleźć tutaj: <https://pineight.com/ds/block/>, https://en.wikipedia.org/wiki/Device_file#BLOCKDEV.

Oczekiwane API ma mieć następującą postać:

```
struct disk_t;
struct disk_t* disk_open_from_file(const char* volume_file_name);
int disk_read(struct disk_t* pdisk, int32_t first_sector, void* buffer, int32_t sectors_to_read);
int disk_close(struct disk_t* pdisk);
```

Zawartość struktury `disk_t` należy zaprojektować tak, aby pozwalała opisać plik-dysk w sposób pozwalający pracować na nim funkcjom `disk_read()` oraz `disk_close()`. Będzie ona stanowiła formę deskryptora oraz uchwytu urządzenia. *Pamiętaj, że w danej chwili w programie może być otwartych więcej niż 1 plik-dysk.*

```
struct disk_t* disk_open_from_file(const char* volume_file_name);
```

Funkcja przyjmuje nazwę pliku z obrazem urządzenia blokowego, o długości bloku równej **512** bajtów, na blok i otwiera go.

Wartość zwracana: W przypadku sukcesu funkcja zwraca wskaźnik na strukturę/deskryptor urządzenia - `disk_t`. W przypadku błędu funkcja zwraca `NULL` i ustawia `errno` odpowiednim kodem błędu:

- `EFAULT` - `volume_file_name` nie wskazuje na nazwę pliku (`NULL`),
- `ENOENT` - brak pliku `volume_file_name`,
- `ENOMEM` - brak pamięci.

Funkcja `disk_open_from_file()` nie może wczytywać całego pliku `volume_file_name` do pamięci. W przypadku niniejszych projektów te pliki mają wielkość ok kilku MB, jednak w realnych zastosowaniach ich odpowiednikami są fizyczne nośniki danych. Te mogą dochodzić do kilku TB.

```
int disk_read(struct disk_t* pdisk, int32_t first_sector, void* buffer, int32_t sectors_to_read);
```

Funkcja wczytuje `sectors_to_read` bloków, zaczynając od bloku `first_sector`. Źródłem jest urządzenie `pdisk` i wczytane bloki zapisywane są do bufora `buffer`.

Funkcja powiedzie się wyłącznie wtedy, gdy istnieje możliwość wczytania **wszystkich** `sectors_to_read` bloków, zaczynając od `first_sector`. Dla przykładu, jeżeli urządzenie udostępnia 30 bloków (0 - 29), to odczyt 10 bloków z pozycji 20 jest możliwy, podczas gdy odczyt 11 bloków z tej samej pozycji już nie jest.

```
012345678901234567890123456789  
11111111112222222222  
|---10---| <- sukces  
|---11----| <- porażka
```

Wartość zwracana: Funkcja zwraca liczbę wczytanych bloków (zawsze `sectors_to_read`) w przypadku sukcesu. W przypadku błędu funkcja zwraca -1 i ustawia kod błędu w `errno`:

- `EFAULT` - wskaźniki nie wskazują na poprawne struktury/bufory (`NULL`).
- `ERANGE` - operacja odczytu nie jest możliwa do wykonania - położenie bloku nie pokrywa się z możliwościami urządzenia (m.in. patrz przykład).

```
int disk_close(struct disk_t* pdisk);
```

Funkcja zamkniemy urządzenie `pdisk` i zwalnia pamięć struktury.

Wartość zwracana: Wartość 0 w przypadku powodzenia. W przypadku błędu funkcja zwraca -1 i ustawia kod błędu w `errno`:

- `EFAULT` - wskaźniki nie wskazują na poprawne struktury/bufory (`NULL`).

Otwieranie i zamykanie woluminu FAT12/16

W celu pracy z woluminem FAT przygotuj następujące API:

```
struct volume_t;  
struct volume_t* fat_open(struct disk_t* pdisk, uint32_t first_sector);  
int fat_close(struct volume_t* pvolume);
```

Zadaniem struktury `volume_t` jest przechowywanie wszystkich niezbędnych informacji o otwartym woluminie, jego geometrii, buforach podrzycznych, urządzeniu; słowem wszystko, co będzie niezbędne do pracy funkcjom plikowym oraz katalogowym. Pamiętaj również, że w danej chwili w Twoim programie może być otwarty wiele woluminów równocześnie.

```
struct volume_t* fat_open(struct disk_t* pdisk, uint32_t first_sector);
```

Funkcja otwiera i sprawdza wolumin FAT, dostępny na urządzeniu `pdisk`, zaczynając od sektora `first_sector`. Informacja o długości woluminu nie jest tutaj potrzebna; jest ona dostępna w supersektorze FAT.

Funkcja musi przeprowadzić podstawowe testy supersektora, aby stwierdzić czy jest to poprawny supersektor danego systemu plików (pamiętaj, że test generuje obrazy FAT w zależności od numeru Twojego albumu).

Wartość zwracana: W przypadku sukcesu funkcja zwraca wskaźnik na strukturę/deskryptor woluminu FAT. W przypadku błędu funkcja zwraca `NULL` i ustawia `errno` odpowiednim kodem błędu:

- `EFAULT` - `pdisk` nie wskazuje na strukturę opisującą urządzenie dyskowe (`NULL`).
- `ENOMEM` - brak pamięci.
- `EINVAL` - struktura supersektora, geometria, tablica fat, itp są uszkodzone (np. różne tablice FAT, brak sygnatury 55AA),
• oraz błędy zgodne z funkcją `disk_read()`, jeżeli nie powiedzie się jej odczyt bloków/sektorów z woluminu.

Na potrzeby tej funkcji (i następnych) można skorzystać z kodów źródłowych, przygotowanych w ramach zadań 2.1 Czytanie łańcucha klastrów oraz 2.2 Odczyt danych z katalogu.

```
int fat_close(struct volume_t* pvolume);
```

Funkcja zamkniemy wolumin `pvolume` i zwalnia pamięć struktury.

Wartość zwracana: Wartość 0 w przypadku powodzenia. W przypadku błędu funkcja zwraca -1 i ustawia kod błędu w `errno`:

- `EFAULT` - wskaźniki nie wskazują na poprawne struktury/bufory (`NULL`).

Otwieranie, przeszukiwanie, czytanie oraz zamykanie plików w systemie FAT

Przygotuj funkcje API oraz odpowiednie struktury, wymagane do realizowania podstawowych operacji odczytu plików.

```
struct file_t;  
struct file_t* file_open(struct volume_t* pvolume, const char* file_name);  
int file_close(struct file_t* stream);  
size_t file_read(void* ptr, size_t size, size_t nmemb, struct file_t* stream);  
int32_t file_seek(struct file_t* stream, int32_t offset, int whence);
```

Zadaniem struktury `file_t` jest tożsame z tymi strukturami `volume_t` oraz `disk_t`. Ta jednak odnosi się do konkretnego pliku. Strukturę taką należy traktować jako uchwyt/deskryptor otwartego (konkretnego) pliku. Wskaźnik na strukturę `file_t` generowany jest funkcją `file_open()`, która otwiera wybrany plik, oraz niszczony funkcją `file_close()`, która go plik zamyka.

Jeżeli widzisz analogię między `file_t*`, `file_open()` i `file_close()` a znany Ci `FILE*`, `fopen()` i `fclose()` to obserwacja ta jest w pełni uzasadniona. Zadaniem przygotowywanych przez Ciebie funkcji jest oddanie charakteru odpowiedników z Biblioteki Standardowej, danych standardem POSIX.

```
struct file_t* file_open(struct volume_t* pvolume, const char* file_name);
```

Funkcja otwiera plik `file_name`. Nawiązując do podobieństwa z funkcją `fopen()` w Bibliotece Standardowej (POSIX), funkcja `file_open()` otwiera plik `filename` wyłącznie w trybie binarnym i tylko do odczytu.

Wartość zwracana: Funkcja zwraca wskaźnik na `file_t`. W przypadku błędów funkcja zwraca `NULL` i ustawia odpowiedni kod błędu w `errno`:

- `EFAULT` - `pvolume` nie wskazuje na strukturę opisującą wolumin FAT (`NULL`).
- `ENOMEM` - brak pamięci.
- `ENOENT` - brak wpisu o nazwie `file_name`,
- `EISDIR` - wpis `file_name` jest katalogiem (nie jest plikiem, np. posiada atrybut `DIRECTORY` lub `VOLUME`).

```
int file_close(struct file_t* stream);
```

Funkcja zamkniemy plik `stream` i zwalnia pamięć struktury. Działanie funkcji jest tożsame z `fclose` z Biblioteki Standardowej (POSIX).

Wartość zwracana: Wartość 0 w przypadku powodzenia. W przypadku błędu funkcja zwraca -1 i ustawia kod błędu w `errno`:

- **EFAULT** - wskaźniki nie wskazują na poprawne struktury/bufory (**NULL**)

```
int32_t file_seek(struct file_t* stream, int32_t offset, int whence);
```

Funkcja ustawia pozycję w pliku `stream`. Nowa pozycja w pliku, mierzona w bajtach, wyznaczana jest poprzez dodanie przesunięcia `offset` do pozycji określonej parametrem `whence`. Jeżeli parametr `whence` jest dany jako `SEEK_SET`, `SEEK_CUR`, `SEEK_END`, to `offset` traktowany jest jako przesunięcie odpowiednio względem początku pliku (`SEEK_SET`), bieżącej pozycji w pliku (`SEEK_CUR`) lub końca pliku (`SEEK_END`).

Wartość zwracana: Funkcja zwraca bezwzględną (po przesunięciu) pozycję w pliku `stream`, wyrażoną w bajtach. W przypadku błędu funkcja zwraca `-1` i ustawia kod błędu w `errno`.

- **EFAULT** - wskaźnik nie wskazuje na poprawną strukturę pliku (**NULL**),
 - **EINVAL** - punkt odniesienia w przestrzeni bajtowej pliku jest nieprawidłowy (**whence**),
 - **ENXIO** - żadana pozycja w pliku nie istnieje (nie może być ustaliona).

```
size_t file_read(void *ptr, size_t size, size_t nmemb, struct file_t *stream)
```

Funkcja `file_read` odczytuje `nmemb` elementów danych, każdy o długości `size` bajtów. Odczyt danych rozpoczyna się na pozycji oraz z pliku, danego wskaźnikiem `ptr`. Wczytywane dane są zapisywane do bufora, danego wskaźnikiem `ptr`. Po wczytaniu danych pozycja w pliku przesuwana jest o liczbę wczytanych bajtów.

Zwróć uwagę, że funkcja `file_read()` próbuje wczytać tyle danych ile tylko się da -- aż do końca pliku przy czym nie więcej niż `nmemb` elementów (lub `nmblocks` sekcji bajtów). Nicco inaczej działa funkcja `disk_read()`. W jej przypadku obowiązuje zasada *wszystko albo nic* (brak choć jednego z żądanych bloków oznacza porażkę).

Wartość zwracana: W przypadku sukcesu funkcja `file_read()` zwraca liczbę wczytanych elementów danych o długości `size`. Liczba ta jest równa liczbie bajtów wczytyanych z pliku `stream` tylko i wyłącznie wtedy, gdy `size` elementu jest równy `1`. W przypadku napotkania końca pliku funkcja zwraca liczbę `odczyt` elementów, które udało się wczytać.

W skrajnym przypadku funkcja zwróci `0` co oznacza, że udało się wczytać `0` elementów. Ma to miejsce w sytuacji żądania odczytu danych, podczas gdy żadnych nie ma.

Wanneer de huid van functionerende mensen - bijvoorbeeld kinderen -

- **EFAULT** - wskaźniki nie wskazują na poprawne struktury pliku/bufory (**NULL**),
 - **ERANGE** - próba odczytania danych spoza fizycznego obszaru urządzenia (wynik błędu funkcji `disk_read()`),
 - **ENXIO** - próba odczytu danych z spoza przestrzeni woluminu
 - Klasycznym przypadkiem błędu **ENXIO** jest błąd w geometrii woluminu. Jeżeli test oczekuje, aby `file_read()` zwróciła ten kod błędu, podczas gdy ta wykona operację odczytu z powodzeniem, to świadczy to o złym wyznaczeniu geometrii woluminu i czytaniu nieprawidłowych obszarów dysku.

Otwieranie, czytanie i zamykanie katalogów

Ostatnim elementem API systemu plików FAT jest funkcjonalność, pozwalająca na przeglądanie listy katalogów. W przypadku wersji projektu na ocenę 3.0 możliwości tych funkcji są testowane jedynie dla katalogu głównego.

```
struct dir_t;
struct dir_t* dir_open(struct volume_t* pvolume, const char* dir_path);
int dir_read(struct dir_t* pdir, struct dir_entry_t* pentry);
int dir_close(struct dir_t* pdir);
```

Znaczenie struktury `dir_t` jest tożsame z poprzednimi (`file_t` itd.). Struktura ta odpowiada za przechowywanie informacji o katalogu, otwartym funkcją `dir_open()`. Można powiedzieć, że `dir_t` jest deskryptorem/uchwytem otwartego katalogu. Do niszczenia deskryptorów służy funkcja `dir_close()` a do uzupełniania kolejnych pozycji z katalogu funkcja `dir_readdir()`.

Odpowiedniki tych funkcji w standardzie POSIX to `opendir()`, `readdir()` oraz `closedir()`.

Przykład użycia:

```
struct dir_t* pdir = open_dir(pvolume, "/");
struct dir_entry_t entry;
while(read_dir(pdir, &entry) == 0) {
    printf("%15s size=%d dir=%d\n", entry.name, entry.size, entry.is_directory)
}
dir_close(pdir);
```

```
struct dir_t* dir_open(struct volume_t* volume, const char* dir_path);
```

Funkcja otwiera katalog, dany ścieżką `dir_path`. W przypadku projektu na ocenę **3,0** testowany jest wyłącznie wariant otwierania katalogu głównego, czyli

Zwróć uwagę, że w systemach POSIX-owych katalogi rozdzielone są znakiem "/" podczas gdy w systemie plików FAT, jako wywodzącym się z linii systemów operacyjnych DOS (MS-DOS, PC-DOS), katalogi rozdzielane są symbolem "\",.

- EFAULT - `pvolume` nie wskazuje na strukturę opisującą wolumin FAT (`NULL`),
 - ENOMEM - brak pamięci,

```
int dir_close(struct dir_t* pdir);
```

Funkcja zamkna katalog `pdir` i zwolni pamieć struktury. Działanie funkcji jest tożsame z `closedir()` z Biblioteki Standardowej.

- wartosc zwrocona:** Wartosc 0 w przypadku powodzenia. W przypadku b

Funkcja `dir_read()` zapisuje informacje o kolejnym poprawnym wpisie w katalogu `pdir` do struktury informacyjnej `dir_entry_t`, dostarczonej wskaznikiem `ptrenty`. Pierwsze uruchomienie `dir_read()`, zaraz po otwarciu katalogu funkcja `dir_open()`, zwróci informacje o pierwszym pliku katalogu z

Zadaniem funkcji `dir_read()` jest wczytywanie jedynie informacji o plikach i katalogach. Etykiety woluminu, usunięte pliki, terminatory czy inne niewłaściwe elementy funkcja ignoruje, kontynuując poszukiwanie kolejnego wpisu lub końca katalogu.

Wartość zwracana: Funkcja zwraca wartość `0` jeżeli udało się wczytać kolejny wpis z katalogu `pdir` do `pentry`. Wartość `1` jest zwracana, jeżeli w katalogu `pdir` nie ma więcej wpisów do przeczytania. W takim przypadku zawartość `pentry` pozostaje nienaruszona.

W przypadku błędu funkcja zwraca `-1` i ustawia kod błędu w `errno`:

- `EFAULT` - wskaźniki nie wskazują na poprawne struktury/bufory (`NULL`),
- `EIO` - jeżeli nie udało się wczytać wpisu katalogu: w systemie FAT wpis ma `32` bajty,
- `ENXIO` - próba odczytu danych z spoza przestrzeni woluminu.

Struktura informacyjna `dir_entry_t` ma posiadać następujące pola:

- `name` - nazwa pliku/katalogu (w przypadku pliku nazwa ma być z kropką i rozszerzeniem), bez nadmiarowych spacji i zakończona terminatorem,
- `size` - rozmiar pliku/katalogu w bajtach,
- `is_archived` - wartość atrybutu: plik zarchiwizowany (`0` lub `1`),
- `is_READONLY` - wartość atrybutu: plik tylko do odczytu (`0` lub `1`),
- `is_system` - wartość atrybutu: plik jest systemowy (`0` lub `1`),
- `is_hidden` - wartość atrybutu: plik jest ukryty (`0` lub `1`),
- `is_directory` - wartość atrybutu: katalog (`0` lub `1`).

Informacje o strukturze systemu plików FAT można znaleźć tutaj:

- <http://read.pudn.com/downloads77/ebook/294884/FAT32%20Spec%20%28SDA%20Contribution%29.pdf>
- https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system,
- <http://www.c-jump.com/CIS24/Slides/FAT/lecture.html>,
- <https://wiki.osdev.org/FAT>.

Przykładowe obrazy:

- FAT12: [1](#) [2](#) [3](#) [4](#) [5](#)
- FAT16: [1](#) [2](#) [3](#) [4](#) [5](#)

Uwagi

- W zadaniu nie jest testowana funkcja `main()`. Można ją wykorzystać do swoich testów.
- Wszystkie struktury oraz prototypy, wymagane specyfikację zadania, należy umieścić w pliku nagłówkowym `file_reader.h`.
- Dane przekazane do funkcji są poprawne logicznie oraz fizycznie.