Tworzenie systemu pliku w oparciu o FUSE

Informacje o FUSE zaczerpnięte z:

Oficjalna dokumentacja API FUSE
http://fuse.sourceforge.net/doxygen/index.html
Przykładowe kody źródłowe
http://fuse.sourceforge.net/doxygen/hello_8c.html
http://fuse.sourceforge.net/doxygen/fioc_8c.html
http://fuse.sourceforge.net/doxygen/fusexmp___fh_8c.html
Nieoficjalny opis FUSE na Harvey Mudd College
https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201001/
Nieoficjalny opis FUSE na stronie IBM
http://www.ibm.com/developerworks/linux/library/l-fuse/

Strona tego projektu na GitHubie:
https://github.com/milleniumbug/wtfs

Wstęp

Tradycyjnie systemy plików działają się na poziomie jądra systemu (kernel space).
W przypadku Linuksa wprowadza to sporo ograniczeń:

- kod musi być pisany w C (w teorii niekoniecznie, w praktyce używając czegokolwiek innego walczysz z całym światem i robisz sobie pod górkę)
- wewnętrzne API nie jest stabilne między wersjami, co narzuca dodatkowy koszt na utrzymanie.
- bug w systemie plików może spowodować niestabilność całego systemu, a nawet zagrożenie bezpieczeństwa.

FUSE umożliwia tworzenie systemów plików na poziomie użytkownika (user space).
Dzięki niemu omijamy wyżej wspomniane ograniczenia:

- nasz system plików może być napisany w dowolnym języku - bindingi do Pythona i Javy już są.
- API systemowe po stronie użytkownika jest stabilne (założenie “don’t break the userspace” Linusa)
- błąd w systemie plików nie zagraża bezpieczeństwu całego systemu.

Na FUSE składa się fuse (moduł jądra), oraz libfuse które służy do komunikacji z daemonem FUSE.

Tworzenie systemu plików

Prosty system plików można stworzyć dzięki pomocy libfuse -
udostępniana jest struktura `fuse_operations`, do która zawiera wskaźniki na funkcje odpowiadające poszczególnym operacjom na systemie plików.
Tworzymy zmienną tego typu, uzupełniamy adresami do naszych funkcji, i wołamy `fuse_main`. Resztą zajmuje się libfuse i FUSE po stronie kernela.

W części przypadków są to odwzorowania 1:1 do funkcji systemowych (`read` woła `op.read`), ale nie zawsze - `close` woła `op.flush`, a później `op.release` - przyczyną jest obsługa duplikacji deskryptorów plików przez funkcję systemową `dup`
Problemem jest dostęp do danych utworzonych przez użytkownika - większość bibliotek w C przekazuje callbackom adres `void*` który jest przekazywany im przez argument.
Tutaj nie jest on przekazywany przez argument, lecz zwracany przez funkcje `fuse_get_context()`, jak oni opisują, z powodu zaszłości historycznych.

MBR

Pojedynczy wpis w tablicy partycji posiada następujące pola:

- status
- adres CHS początku partycji (nie ma znaczenia w dzisiejszych czasach)
- typ partycji
- adres CHS końca partycji (nie ma znaczenia w dzisiejszych czasach)
- adres LBA początku partycji
- ilość sektorów

Wypełnianie większości tych wpisów nie jest moją odpowiedzialnością jako twórca systemu plików (ewentualnie mogą mi się przydać do odczytu), za wyjątkiem typu partycji.
Tutaj jako system plików musimy nadać jakąś sensowną wartość polu "typ partycji". Na przykład NTFS ma zarezerwowany numer 07h, zaś FAT32 0Bh.
Niestety większość wpisów jest zarezerwowana, ponadto nie ma żadnej organizacji która przydziela te numery, co oznacza że większość numerów się nakłada - tak na przykład NTFS i exFAT mają ten sam numer 07h, co oznacza że te numery i tak się będą powtarzać i sprawdzenie numeru nie jest pewnym sposobem na sprawdzenie rodzaju partycji.
Wikipedia wspomina o istnieniu Alternative OS Development Partition Standard, które rzekomo ma zalecać typ 7Fh, lecz źródła prowadzą do strony błędu 404. Niemniej jednak dwie listy typów partycji które sprawdziłem (http://www.osdever.net/documents/partitiontypes.php, https://www.win.tue.nl/~aeb/partitions/partition_types-1.html) wspominają o tym typie że jest nie używany, co utwierdza mnie w przekonaniu że jest to dobry wybór na typ partycji.

GPT

Pojedynczy wpis w tablicy partycji GPT zawiera następujące pola:

- GUID typu partycji
- GUID partycji
- adres LBA początku partycji
- adres LBA końca partycji
- atrybuty
- nazwa partycji

Tutaj, podobnie jak w przypadku MBR musimy wypełnić pole "typ partycji". Jest pewne ulepszenie, gdyż mamy tutaj do dyspozycji 16 bajtów, a nie 1. 16 bajtów jest wystarczająco żeby to wystarczająco dużo żeby wygenerować unikalny identyfikator dla każdego atomu we wszechświecie, co oznacza że prawdopodobieństwo kolizji jest znikome. I tym samym wygenerowałem dla nowego systemu plików GUID: `{8b0934c2-9df4-11e5-a644-080027d32d84}`.
Atrybuty są nowe dla GPT, jest to maska bitowa o długości 8 bajtów, ich opis dla poszczególnych bitów:

0 - czy jest to partycja systemowa (tak na przykład oznaczona jest stumegabajtowa partycja "Zarezerwowane dla systemu", którą Windows tworzy)
1 - czy firmware EFI ma ignorować zawartość tej partycji i z niej nie czytać
2 - odpowiednik flagi bootowania z BIOS (w ramach kompatybilności)
3 do 47 - zarezerwowane przez EFI do przyszłego użycia
48 do 63 - do użytku wewnętrznego, specyficzne dla danego typu partycji

Dostęp do partycji z poziomu zwykłego programu w userspace

Partycja jest dostępna jako urządzenie blokowe, które widoczne jest jako specjalny plik.
Użytkownik z uprawnieniami administratora może otworzyć plik, np. `open("/dev/sdb3", ...)` i używać zdobyty deskryptor pliku do manipulacji danymi na dysku.
Na pliku dyskowym mogę wykonać sporą część operacji które mogę wykonać na zwykłym pliku, na przykład mogę odczytać dowolny fragment lub odnaleźć jego wielkość:

```
#define _FILE_OFFSET_BITS 64
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <inttypes.h>

int main()
{
    const char* path = "/dev/sdb3";
    int fd = open(path, O_RDWR);
    printf("open: %s\n", strerror(errno));
    char buf[4096] = { 0 };
    read(fd, buf, sizeof buf);
    printf("read: %s\n", strerror(errno));
    for(int i = 0; i < 4096; ++i)
        printf("%02hhX ", buf[i]);
    off_t newoff = lseek(fd, 0, SEEK_END);
    printf("lseek: %s\n", strerror(errno));
    printf("%" PRIdMAX "B\n", (intmax_t)newoff);
    printf("%" PRIdMAX "KB\n", (intmax_t)newoff/1024);
    printf("%" PRIdMAX "MB\n", (intmax_t)newoff/1024/1024);
    printf("%" PRIdMAX "GB\n", (intmax_t)newoff/1024/1024/1024);
    printf("%" PRIdMAX "TB\n", (intmax_t)newoff/1024/1024/1024/1024);
    close(fd);
}
```

Tutaj użyto funkcji systemowej `read`, ale można również mapować plik do pamięci, co jest przydatne w pisaniu systemu plików - algorytmy działające na strukturach w pamięci można przerobić żeby działały na danych na dysku poprzez zamianę funkcji która przydziela pamięć - zamiast wołać `malloc` bądź `::operator new`, zawołam `mmap`.

Przydzielanie wolnych bloków

Jednym z zadań systemu plików jest przydzielanie wolnych bloków. Do tego potrzebna jest pewna strategia przydzielania pamięci, która musi spełniać następujące wymagania:

- konserwatywne (nie przydziel bloku jeżeli nie jesteś 100% pewien że jest wolny)
- szybkie
- pozostawiające niewielką fragmentację
- wykorzystująca ponownie zwolnione bloki

Do tego celu można wykorzystać istniejące metody przydzielania pamięci - albo z użyciem listy wolnych bloków, albo za pomocą 'buddy system'.

Struktura systemu plików

W większości nowych systemów plików można wyróżnić następujące elementy:

- metadane dla partycji (BPB, EBPB)
- metadane dla plików (File Allocation Table w FAT32, MFT w NTFS, tablica i-węzłów w ext4)
- dane plików

W zależności od systemu plików możemy dojść do różnic co jest określane jako plik - w FAT plik posiada nazwę, w rodzinie systemów plików ext nazwa jest zapisywana jako dane katalogu i odnosi się do konkretnego i-węzła. Ilość nazw jest zliczana wewnątrz i-węzła. W swoim systemie plików stosuję podejście jakie stosuje ext - nazwy są zapisywane wewnątrz katalogów.

Co realizować

Odczyt katalogów i plików struktur w pamięci jest wykonany - konieczna jest możliwość odczytu z dysku. System później można rozszerzyć najpierw o zapis do plików, później o tworzenie nowych plików.