Marcin Bryk, Szymon Gałązka, Dawid Wojciechowski, Dawid Dominiak

**PROJEKT WSTĘPNY - UXP1A (semestr 15L)**

**Prowadzący: Grzegorz Blinowski**

**1. Treść zadania.**

Napisać wieloprocesowy system realizujący prosty system plików (w przestrzeni użytkownika, nie jądra). System powinien udostępniać następujące funkcje (zachowanie analogiczne do systemowych odpowiedników):

int simplefs\_open(char \*name, int mode);

int simplefs\_unlink(char \*name) ;

int simplefs\_mkdir(char \*name);

int simplefs\_creat (char \*name, int mode) ;

int simplefs\_read(int fd, char \*buf, int len);

int simplefs\_write(int fd, char \*buf, int len);

int simplefs\_lseek(int fd, int whence, int offset);

System powinien być zrealizowany w jednym pliku, zachowującym się jak uproszczony dysk logiczny, tj. powinien on zawierać tablicę i-node’ów, katalog główny, listę wolnych bloków, obszar na pliki. Należy zaimplementować tylko pliki zwykłe i katalogi.

Implementacja powinna pozwalać na jednoczesny dostęp wielu procesów, tak więc należy zrealizować wykluczanie i wielodostęp, nie na poziomie całego mini-systemu plików, ale możliwie “drobno-ziarniście”, tj. na poziomie poszczególnych obiektów (katalogów, tablicy inode-ów). Wykluczanie zrealizować przy pomocy potoków FIFO, bez centralnego procesu serwera.

**2. Dodatkowe założenia.**

* System obsługuje jedynie pliki zwykłe i katalogi.
* Pominięto prawa dostępu do plików na poziomie U/G/O, jedynymi uwzględnianymi prawami dostępu są prawa odczytu/zapisu (R/W), zarówno dla plików jak i dla katalogów.
* Przy dostępie do plików uwględniono jedynie tryby oznaczane przez flagi: O\_CREAT, O\_RDONLY, O\_WRONLY, O\_RDWR. Zrezygnowano z implementacji pozostałych flag trybu.
* Przyjęto statyczny rozmiar katalogów i tablicy i-node’ów oraz uproszczoną wersję metryczki pliku (patrz pkt. 5).
* Przyjęto ciągłą alokację przestrzeni dyskowej, z opcjonalnym mechanizmem relokacji (funkcjonalność dodatkowa).
* Proces może mieć otwartych wiele zwykłych plików. Zwykły plik może być otwarty przez kilka procesów jednocześnie.
* Zwykły plik może być jednocześnie otwarty przez co najwyżej 10 procesów.
* Dodatkowa funkcja simplefs\_close() w API słuząca do zamykania otwartych plików.
* Plik może zostać usunięty wtedy, gdy nie jest otwarty przez żaden proces.
* Katalog może zostać usunięty wtedy, gdy żadne pliki znajdujące się w nim nie są otwarte przez inne procesy.

**3. Opis funkcjonalności “black-box”, API**

Realizowany system plików ma udostępiać funkcje charakterystyczne dla wszystkich systemów unixowych.

Proces, który uzyska dostęp do systemu plików będzie miał możliwość wykonania podstawowych operacji:

* utworzenie lub usuniecię pliku/katalogu
* odczyt i zapis do pliku
* usunięcie pliku/katalogu

Powyższe operacje realizowane będą przez bibliotekę następujących funkcji:

int simplefs\_open(char \*name, int mode);

* Otwiera istniejący plik lub (gdy nie istnieje) tworzy i otwiera plik.
* name - ścieżka bezwzględna do pliku
* mode - tryb otwarcia (suma bitowa flag: O\_CREAT, O\_EXCL, O\_RDONLY, O\_WRONLY, O\_RDWR)
  + O\_CREAT -twórz plik, jeśli nie istnieje
  + O\_EXCL - użyte razem z O\_CREAT spowoduje, że funkcja zwróci błąd jeśli plik istnieje
  + O\_RDONLY - dostęp tylko do odczytu
  + O\_WRONLY - dostęp tylko do zapisu
  + O\_RDWR - dostęp do odczytu i zapisu
* Zwraca deskryptor otwartego pliku (wartość nieujemna), gdy wszystko się uda, -1, gdy błąd.

int simplefs\_unlink(char \*name) ;

* Usuwa plik (zwykły lub katalog wraz z jego zawartością).
* name - ścieżka bezwzględna do pliku
* Zwraca 0, gdy wszystko się uda, -1, gdy błąd.

int simplefs\_mkdir(char \*name);

* Tworzy pusty katalog.
* name - ścieżka bezwzględna do katalogu
* Zwraca 0, gdy wszystko się uda, -1, gdy błąd

int simplefs\_creat (char \*name, int mode) ;

* Tworzy nowy plik. Po jego utworzeniu otwiera go w trybie do zapisu.
* name - ścieżka bezwzględna do pliku
* mode - tryb
* Zwraca deskryptor otwartego pliku, gdy wszystko się uda, -1, gdy błąd.

int simplefs\_read(int fd, char \*buf, int len);

* Odczytuje dane z otwartego pliku i zapisuje je do bufora.
* Przesuwa wskaźnik pozycji czytania/pisania w pliku o tyle bajtów, ile zdołał odczytać.
* fd - deskryptor otwartego pliku
* buf - bufor na odczytane dane
* len - długość bufora buf (w bajtach)
* Zwraca liczbę odczytanych bajtów (0 oznacza EOF), gdy się wszystko uda, -1, gdy błąd.

int simplefs\_write(int fd, char \*buf, int len);

* Zapisuje dane z bufora do otwartego pliku.
* Przesuwa wskaźnik pozycji czytania/pisania w pliku o tyle bajtów, ile zdołał zapisać.
* fd - deskryptor otwartego pliku
* buf - bufor z danymi do zapisu
* len - długość bufora z danymi (w bajtach)
* Zwraca liczbę zapisanych bajtów, gdy się wszystko uda, -1, gdy błąd.

int simplefs\_lseek(int fd, int whence, int offset);

* Ustawia aktualną pozycję wskaźnika do pisania/czytania w pliku.
* Ustawia aktualną pozycję wskaźnika do pisania/czytania w pliku (wartość nieujemna, wyrażająca przesunięcie w bajatch od początku pliku).
* fd - deskryptor otwartego pliku
* whence - flaga określająca pozycje w pliku, względem której funkcja przesunie wskaźnik czytania/pisania o wartość offset
* offset - przesunięcie względem zadanej pozycji
* Zwraca nowy wskaźnik pozycji w pliku, gdy wszystko się uda, -1, gdy błąd.

int simplefs\_close(int fd);

* Zamyka otwarty plik.
* fd - deskryptor otwartego pliku
* Zwraca 0, gdy wszystko się udało, -1, gdy błąd.

**4. Mechanizm synchronizacji dostępu do systemu plików**

Procesy starające się o dostęp do systemu plików tworzą jednokierunkowy pierścień, komunikując się za pomocą potoków nazwanych FIFO. W pierścieniu krąży specjalny komunikat, zwany dalej tokenem. Otrzymanie tokena jest równoznaczne z zajęciem sekcji krytycznej przez proces, a jego nadanie – z jej zwolnieniem. Procesy nie posiadające tokena oczekują na jego otrzymanie.

Założenia:

· W pierścieniu może jednocześnie istnieć tylko 1 token.

· Każdy proces korzysta z 2 kolejek FIFO – jednej do czytania, a drugiej do pisania.

· Każdy proces odpowiada za zarządzania jednym potokiem FIFO (tym, z którego czyta) – jego tworzenie i usuwanie. Nazwa tej prywatnej kolejki ma postać: fifo\_pid\_procesu.

· Wyjątkiem jest działający w tle proces fsinit, którego potok o nazwie fifo\_init jest znany wszystkim procesom starającym się o podłączenie do pierścienia. Proces stanowiąc pierwszy element pierścienia, ma za zadanie pośredniczyć w przekazywaniu tokena, ułatwiać podłączanie się nowych procesów do pierścienia i odtwarzać token w przypadku jego zagubienia. Musi zostać uruchomiony przed rozpoczęciem korzystania z systemu plików.

· Wszelkie komunikaty dotyczące poleceń reorganizacji struktury pierścienia są uznawane za realizowane w momencie otrzymania przez ich nadawcę (tj. przepropagowaniu przez cały pierścień).



**Schemat podłączania się procesu do pierścienia**

1. Proces, który chce podłączyć się do pierścienia, otwiera kolejkę fifo\_init w trybie do odczytu i nadaje komunikat LINK\_fifo\_pid, gdzie fifo\_pid to prywatna kolejka procesu.

2. Gdy ostatni proces w pierścieniu otrzyma taki komunikat, zamyka aktualną kolejkę, do której pisał (jeśli takowa istnieje) i podłącza się do kolejki wskazanej w treści komunikatu.

3. Jeśli poprzednio w pierścieniu był jedynie proces fsinit, to przekazuje on token nowo podłączonemu procesowi.

**Schemat odłączania się od pierścienia**

1. Proces który wykonał swoje zadanie w sekcji krytycznej nadaje komunikat UNLINK\_fifo\_pid, gdzie fifo\_pid to nazwa potoku, do którego pisał odłączający się proces.

2. Gdy powyższy komunikat dotrze do procesu fsinit (jest to proces poprzedzający proces odłączający się, gdyż zawsze po zakończeniu pracy w sekcji krytycznej, proces posiadający token odłącza się od pierścienia) otwiera on wskazany w komunikacie proces w trybie do zapisu, chyba że jest to jego własna kolejka fifo\_init.

3. Następnie fsinit przesyła komunikat do procesu odłączającego się, ten przekazuje token kolejnemu procesowi w pierścieniu i odłącza się od pierścienia.

**Sytuacje wyjątkowe:**

W czasie istnienia pierścienia może dojść do sytuacji, że jeden lub więcej procesów uczestniczących w komunikacji uległo nieprzewidzianemu zamknięciu. W takiej sytuacji procesy wykrywają nieoczekiwane zamknięcie procesu piszącego do ich prywatnego potoku i nadają specjalny komunikat o błędzie, który jest dla nich sygnałem do ponownego podłączenia się do pierścienia. Wówczas proces fsinit generuje nowy token, a wszelkie procesy zawieszone na starym pierścieniu muszą zostać ponownie uruchomione.

**5. System plików: struktura i organizacja.**

**Założenia:**

* Wszystkie pliki i katalogi w systemie plików znajdują się w katalogu głównym (root: /).
* W jednym katalogu może znajdować się do 128 plików
* I-node’ów naszym systemie będzie 512.

Realizowany system plików będzie składał się z poniższych sekcji:

* **Super block** – zawiera informacje na temat systemu pliku, tzn.: liczby wszystkich bloków danych, liczby wolnych bloków danych, liczby i-node’ów, przesunięcie pierwszego bloku sekcji *i-node map* względem początku systemu, przesunięcie pierwszego bloku sekcji *block map* względem początku systemu, przesunięcie pierwszego bloku sekcji *i-node table* względem początku systemu, przesunięcie bloku sekcji *blocks* względem początku systemu.
* **I-node map** – mapa bitowa, gdzie każdy bit to jeden inode; bit o wartości 1 – zajęty i-node, 0 – wolny i-node.
* **Block map** – mapa bitowa, gdzie każdy bit to jeden blok danych; bit o wartości 1 – zajęty blok, 0 – wolny blok.
* **I-node table** - tablica metryczek plików.
* **Blocks** – bloki danych o rozmiarze 4 kB.

Każdy plik w systemie jest reprezentowany przez jego **metryczkę (inode)**, która zawiera:

* informację o typie pliku (plik zwykły lub katalog)
* 7 wskaźników na bloki z danymi pliku
* wskaźnik na blok pośredni (blok ze wskaźnikami na bloki z danymi pliku)
* określenie praw dostępu (zapisu i/lub odczytu)
* informację o rozmiarze pliku w bajtach
* informację o rozmiarze pliku w blokach danych

**Katalog** będzie zawierał wpisy plików, jakie się w nim znajdują. Wpisy te będą mapować nazwy plików na indeksy ich i-node’ów.

**Deskryptorem otwartego pliku** będzie indeks jego i-node’a w sekcji *inode table*.

struct inode {

char filetype; // typ pliku (katalog bądź zwykły plik)

char mode; // prawa dostępu

unsigned short nblocks; // liczba bloków danych zajętych przez plik

unsigned int filesize; // rozmiar pliku w bajtach

unsigned short blocks[8]; // 7 wskaźników na bloki danych + 1 wskaźnik na blok pośredni

} // 24 bajty

Informacje na temat aktualnej pozycji czytania/pisania oraz tryb dostępu procesów do plików będą znajdować się w specjalnych plikach tymczasowych (o nazwie w postaci *deskryptor\_pliku.tmp*) odrębnych dla każdego pliku. Będą one zawierać takie informacje jak:

* PID procesu
* aktualna pozycja czytania/pisania
* tryb dostępu procesu do pliku

**6. Zarys koncepcji implementacji**

Projekt zostanie wykonany w języku C. Środowiskiem deweloperskim i uruchomieniowym będzie Linux Mint. Przyjęta koncepcja architektury zakłada implemetację biblioteki funkcji umożliwiających podstawowe operacje na systemie plików. Ponadto w oddzielnym module zaimplementowany zostanie mechanizm synchronizacyjny wraz z inicjalizacją potoku oraz znacznika dostępu. Opisany w punkcie 4. mechanizm synchronizacji determinuje fakt istnienia dodatkowego procesu usługowego. W celu skorzystania z systemu plików konieczne jest więc uruchomienie procesu demona, który zainicjuje kolejkę oraz znacznik.

Schemat korzystania z implementowanego systemu plików:

1. Uruchomienie procesu usługowego (w tle).
2. Uruchomienie procesów żądających dostępu do systemu.

Implementacja systemu i procesu usługowego będzie zgodna ze strukturalnym paradygmatem programowania.

Budowanie i kompilacja aplikacji przy użyciu GNU Compiler Collection oraz makefile. W celu sprawnej pracy całego zespołu jako system kontroli wersji wybrany został Git, a repozytorium kodu powstanie na portalu GitHub.