

# Rozwój implementacji systemów plików

---

Politechnika Śląska

Autor: Andrzej Dereszowski

Promotor: prof. dr hab. inż. Andrzej Grzywak

Konsultant: mgr inż. Piotr Kasprzyk

9 listopada 2001 roku

# Spis treści

<b>1</b>	<b>Cel i zakres pracy</b>	<b>4</b>
<b>2</b>	<b>Systemy plików a systemy operacyjne</b>	<b>5</b>
2.1	Rola systemów plików . . . . .	5
2.2	Różnorodność systemów plików . . . . .	6
2.3	Problemy z implementacją obsługi systemów plików . . . . .	7
<b>3</b>	<b>System plików jako część systemu operacyjnego</b>	<b>8</b>
3.1	Proces dostępu do pliku . . . . .	8
3.2	Przełącznik systemów plików . . . . .	9
<b>4</b>	<b>System plików HTFS i system operacyjny Linux</b>	<b>10</b>
4.1	Charakterystyka systemu plików HTFS . . . . .	10
4.2	Obsługa systemów plików w jądrze Linuksa . . . . .	15
<b>5</b>	<b>Założenia dotyczące implementacji systemu plików HTFS w systemie operacyjnym Linux</b>	<b>18</b>
<b>6</b>	<b>Implementacja systemu plików HTFS w Linuksie</b>	<b>21</b>
6.1	Instalacja systemu operacyjnego SCO Open Server 5 . . . . .	21
6.2	Porównanie plików nagłówkowych z konkretnym obrazem systemu plików . . . . .	22
6.3	Narzędzia w trybie użytkownika . . . . .	25
6.4	Kod systemowy do jądra Linuksa . . . . .	26
6.4.1	Specyfikacja zewnętrzna . . . . .	26
6.4.2	Specyfikacja wewnętrzna . . . . .	27
6.4.3	Testowanie . . . . .	31
6.4.4	Źródła i licencja . . . . .	32
6.5	Implementacja kodu do jądra Linuksa 2.2 . . . . .	32
<b>7</b>	<b>Wnioski i kierunki dalszych prac</b>	<b>33</b>

---

Zał. 1 Analiza struktur systemu plików HTFS	35
Zał. 2 Wirtualny System Plików Linuksa (VFS)	39
Zał. 3 Instrukcja kompilacji, uruchomienia i użytkowania	46
Zał. 4 Spis literatury	51

# Rozdział 1

## Cel i zakres pracy

Temat niniejszej pracy to „Rozwój implementacji systemów plików”. Celem *naukowym* pracy jest zbadanie, jak się implementuje system plików jednego systemu operacyjnego w ramach innego systemu operacyjnego.

Przeglądając archiwa Usenetu można natknąć się na pytanie dotyczące istnienia kodu implementującego HTFS (domyślny system plików SCO Open Server 5) w Linuksie. Jednak okazuje się, że dotychczas nic takiego w sieci Internet nie było dostępne. Dlatego celem *użytkowym* stała się implementacja kodu systemowego realizującego ten cel. Jest ona jednocześnie praktycznym przykładem uzupełniającym cel *naukowy* pracy.

W kolejnych rozdziałach system plików zostanie przedstawiony jako część systemu operacyjnego. Wyjaśniony zostanie powód istnienia tak wielu systemów plików, opisane zostaną przyczyny powodujące to, że implementacja dodatkowych systemów plików w systemach operacyjnych nie jest trywialna. Prześladowany zostanie proces dostępu do informacji za pośrednictwem systemu plików i jego komunikacja z sąsiednimi warstwami systemu operacyjnego. Wyszczególnione zostaną mechanizmy, na których skoncentrować powinna się osoba zamierzająca implementować obsługę dodatkowego systemu plików w danym systemie operacyjnym. Dalsze rozdziały opisują jądro Linuksa w zakresie obsługi systemów plików, system plików HTFS oraz sam proces implementacji nowego systemu plików w Linuksie.

# Rozdział 2

## Systemy plików a systemy operacyjne

Każdy system operacyjny musi dać swoim klientom (aplikacjom) możliwość przechowywania informacji na pamięciach masowych. Aplikacja może zażądać zapisania, odczytania, skasowania danych. Zadaniem systemu operacyjnego jest natomiast realizacja tego żądania.

### 2.1 Rola systemów plików

Pamięć masowa to jeden z najważniejszych zasobów systemowych, które system operacyjny udostępnia aplikacjom. Od strony sprzętowej pamięci masowe można podzielić według wielu kryteriów. Ze względu na sposób dostępu, można wyszczególnić dwie główne klasy urządzeń:

- urządzenia o dostępie sekwencyjnym (np. taśmy magnetyczne)
- urządzenia o dostępie swobodnym (np. dyski twarde)

Urządzenia o dostępie sekwencyjnym cechują się tym, że nie ma tam swobodnego dostępu do fragmentów informacji zapisanej na nośniku (albo fragmenty te są bardzo duże). Chcąc odczytać fragment informacji, trzeba odczytać całość, co zazwyczaj bywa czasochłonne. Natomiast od urządzeń o dostępie swobodnym można zażądać dostępu do mniejszych, stałej wielkości fragmentów informacji, na których opiera się wewnętrzny podział nośnika, bez konieczności oczekiwania na odczyt całości danych. Ta cecha powoduje, że istnieje możliwość podziału i katalogowania informacji w taki sam sposób, jak kataloguje się akta w oznaczonych półkach. Jednak pamiętanie kolejnych numerów fragmentów nośnika, gdzie zapisano daną informację, byłoby dość

kłopotliwe. Z pomocą przychodzi system plików - tworzy abstrakcyjną warstwę „półek i akt” w celu automatycznego katalogowania informacji.

## 2.2 Różnorodność systemów plików

Na rynku istnieje bardzo wiele różnych systemów plików. Główne powody takiego stanu rzeczy to:

- nie wymyślono jeszcze uniwersalnego systemu plików spełniającego naraz wszystkie oczekiwania użytkowników. Jedne systemy plików obsługują wydajniej pracę nad dużej wielkości plikami (np. pliki video), inne będą się lepiej nadawały do pracy z dużą ilością małych plików (np. serwery poczty elektronicznej). Z innej strony - do serwerów plików nadają się systemy plików z rozbudowanym mechanizmem praw dostępu dla poszczególnych użytkowników, a dla domowego komputera wystarczy uproszczone sprawdzanie praw dostępu, gdyż zyska się wtedy na szybkości. Systemy plików mogą zależeć również od nośnika, na przykład na nośniku jednokrotnego zapisu, takim jak CD-ROM, stosuje się uproszczony system plików ponieważ wiadomo, że nie będzie się tam dało zapisywać fragmentami
- różny stopień dopasowania systemu plików do innych części systemu operacyjnego
- wolnorynkowa konkurencja - z każdym systemem operacyjnym dostarczany jest jakiś system plików, a firmy starają się, aby to właśnie ich własny system operacyjny (a zarazem przychodzący z nim system plików) był najlepszy

Taka różnorodność powoduje, że zadanie przenoszenia danych między systemami operacyjnymi nie jest proste. Oczywiście przenieść dane między systemami operacyjnymi można na różne sposoby. Gdy systemy są połączone w sieć, można skorzystać z sieciowych systemów plików (np. *NFS*, *SMB*) lub protokołów transmisji plików (np. *FTP*). Jeśli powyższe jest niewykonalne, można także zastosować specjalne narzędzia do przenoszenia plików (np. *tar*, *pkzip*). Jeśli jednak rozwiązanie te są zbyt mało wydajne, zbyt uciążliwe, nie spełniają w pełni oczekiwań (np. podczas przenoszenia traczone są prawa dostępu do plików dla poszczególnych użytkowników) lub w ogóle niemożliwe (np. niektóre implementacje tych samych protokołów pod różnymi systemami operacyjnymi są niekompatybilne), zostaje jeszcze jedno wyjście. Można spróbować bezpośrednio użyć danego systemu plików z danym systemem operacyjnym, jeżeli tylko stworzono kod implementujący taką możliwość.

Proces projektowania i implementacji systemu plików to bardzo złożone i ciągle zmieniające się zagadnienie. Kod, który powstaje w wyniku implementacji danego systemu plików, jest oprogramowaniem systemowym (co oznacza, że jest częścią systemu operacyjnego). Kod ten jest ściśle związany z budową tego systemu operacyjnego i nie ma tutaj możliwości tworzenia kodu przenośnego.

## 2.3 Problemy z implementacją obsługi systemów plików

Największym problemem bywa zazwyczaj skąpa dokumentacja lub jej całkowity brak (np. system plików NTFS z MS Windows NT/2000). W takim przypadku zostaje tylko technika *reverse engineeringu*. Oznacza to śledzenie zmian na dysku podczas dokonywania operacji na systemie plików i wyciąganie z tego odpowiednich wniosków.

Jak już zostało wspomniane, jednym z powodów istnienia przeróżnych systemów plików jest ich zgodność (dopasowanie) z wewnętrzną budową systemu operacyjnego. Bywa to również powodem problemów z implementacją. Przy procesie implementacji trzeba dodawać brakujące mechanizmy, ignorować nadmiarowe lub dopasowywać do siebie istniejące (np. budowa systemu plików Uniksa, który jest idealnie dopasowany do samego systemu operacyjnego i systemu plików FAT, jest zupełnie inna).

Wiele zależy również od tego, w jakie mechanizmy dodawania nowych systemów plików został wyposażony system operacyjny. Może zdarzyć się na przykład tak, że twórcy systemu nie przewidywali rozszerzania systemu operacyjnego o dodatkowe systemy plików. Wtedy pozostają już tylko różne programistyczne triki, które wymagają doskonałej znajomości danego systemu operacyjnego.

# Rozdział 3

## System plików jako część systemu operacyjnego

### 3.1 Proces dostępu do pliku

Osoba implementująca obsługę systemu plików powinna znać ogólną budowę systemów operacyjnych, przynajmniej w zakresie ich części bezpośrednio związanych z systemami plików. W ten sposób może ona także wyodrębnić mechanizmy systemu operacyjnego, z którymi będzie musiała się szczególnie zapoznać chcąc użyć ich w celu dodania obsługi nowego systemu plików. Dlatego warto prześledzić drogę, jaką przechodzi żądanie związane z transmisją danych między aplikacją a pamięcią masową przez różne części systemu operacyjnego (dalej mowa będzie o dysku twardym, jednak te zasady stosują się do dowolnego urządzenia o swobodnym dostępie do medium). Tabela 1 przedstawia kolejno warstwy, przez które przechodzi standardowo żądanie odczytu lub modyfikacji danych na dysku z systemem plików.

System Operacyjny	Aplikacja użytkownika
	Funkcja biblioteczna
	<i>Przetłacznik systemów plików</i>
	Konkretny system plików
	System buforowania
	Sterownik dysków
Sprzęt	Kontroler dysków twardech
	Specjalizowane układy dysku twardego
	Nośnik

Tabela 1: Warstwy, przez które przechodzi żądanie dostępu do danych na dysku

Najpierw aplikacja wywołuje funkcję biblioteczną - np. *read()* (żądanie odczytu) i *write()* (żądanie zapisu). Funkcja taka, po zinterpretowaniu swoich argumentów, przekazuje sterowanie jądro systemu poprzez *wywołanie systemowe*, które przełącza kontekst procesora w tryb jądra. Następnie sterowanie przechodzi do funkcji jądra odpowiedzialnej za odszukanie odpowiedniego systemu plików, na którym ma być wykonane dane polecenie. Podsystem, który jest odpowiedzialny za to zadanie, będzie dalej nazywany *przełącznikiem systemów plików*. Teraz żądanie przechodzi do systemu buforowania.

W przypadku żądania odczytu danych, bufor są przeszukiwane w celu znalezienia żądanej informacji. Gdy w buforach brakuje odpowiednich danych, proces zostaje uśpiony do momentu dostarczenia ich przez niższe warstwy. Dane te zostają następnie zwrócone systemowi buforowania. Proces zostaje obudzony (jeśli wcześniej został uśpiony) i przełączony w tryb użytkownika (wraca z *wywołania systemowego*), a dostarczone dane znajdują się już w jego przestrzeni adresowej.

W przypadku żądania modyfikacji danych, sterowanie zwracane jest z wywołania niemal natychmiast - system buforowania asynchronicznie zapisze dane na medium później, zależnie od swojego algorytmu buforowania (przekazując żądanie niższym warstwom).

Przechodząc do niższych warstw, żądanie trafia do odpowiedniego sterownika dysków w jądrze. Ten przekazuje je kontrolerowi dysków (np. IDE, SCSI). Kontroler komunikuje się z dyskiem, którego specjalizowane układy realizują dostęp do medium.

Poszczególne systemy operacyjne mogą nieco odbiegać od opisanego schematu (np. system buforowania może być zaraz po *przełączniku systemów plików*, a przed konkretnym systemem plików w tabeli 1), jednak ogólna zasada pozostaje ta sama.

## 3.2 Przełącznik systemów plików

Osoba implementująca system plików powinna szczegółowo zapoznać się z *przełącznikiem systemów plików* (wyróżniony w tabeli 1). Jest to część systemu operacyjnego, która łączy go z systemami plików. To właśnie od konstrukcji tego mechanizmu będzie zależał sposób implementacji dodatkowego systemu plików. W kolejnym rozdziale zostanie przedstawiony *przełącznik systemów plików* w systemie operacyjnym Linux.

## Rozdział 4

# System plików HTFS i system operacyjny Linux

Jak już zostało wspomniane w rozdziale 1, jednym z celów niniejszej pracy jest implementacja systemu plików HTFS w systemie operacyjnym Linux. Przed implementacją trzeba dokładnie zapoznać się z mechanizmami Linuksa dotyczącymi możliwości implementacji dla niego dodatkowych systemów plików. Należy również zgromadzić i dokonać analizy jak największej ilości informacji dotyczących HTFS.

Pierwszy zostanie przedstawiony system plików HTFS, na jego podstawie zostanie bowiem wprowadzonych kilka ogólnych pojęć dotyczących systemów plików stosowanych w Uniksach, a także w Linuksie. Następnie przedstawiony zostanie Linux od strony obsługi systemów plików.

Omawiane w tym punkcie rozwiązanie zaprojektowane jest dla jąder z serii 2.4, ponieważ są w tym zakresie dużo nowocześniejsze niż poprzednia stabilna seria (2.2). Ponieważ Linux 2.2 jest jednak wciąż szeroko stosowany, został także przedstawiony krótki opis rozwiązania dla serii 2.2. Znajduje się on w punkcie 6.5. Na przykładzie porównania mechanizmów w jądrach z tych dwóch serii zostało pokazane, jak twórcy tego systemu operacyjnego starają się usprawnić i uprościć metody implementacji nowych systemów plików.

### 4.1 Charakterystyka systemu plików HTFS

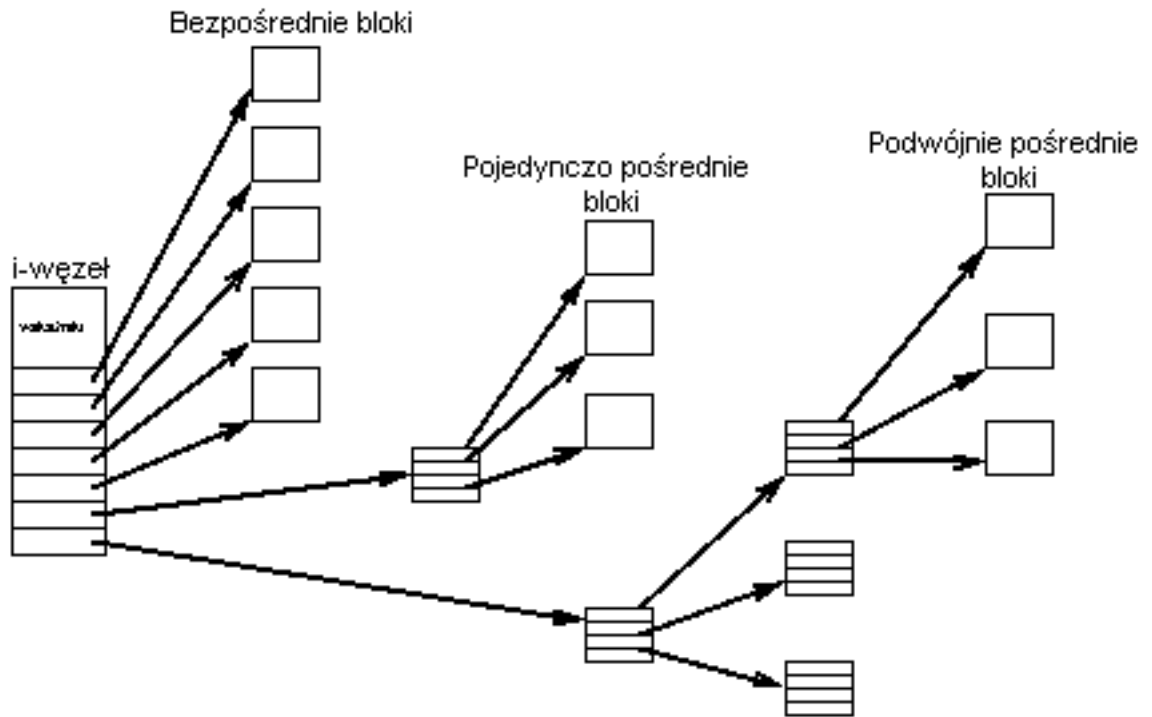
System plików High Throughput FileSystem (HTFS) jest oparty na systemie plików UNIX SYSV R4. Zastosowany tam system plików (określany dalej jako SYSVFS) to protoplasta systemów plików używanych w dzisiejszych implementacjach systemu Unix (i systemów z nim zgodnych) i napisano o nim wiele. Tutaj zostaną przytoczone tylko najważniejsze informacje. Dokład-

niejsze informacje znajdzie Czytelnik w [6] i [2]. Używane dalej określenia: modelowy lub standardowy system plików w Uniksie można odnieść bezpośrednio do SYSVFS.

Wszystkie podstawowe informacje dotyczące systemu plików zapisane są w superbloku (ang. *superblock*). Jest to najważniejszy blok w systemie plików, gdzie zawarte są najbardziej podstawowe informacje, pozwalające właściwie zinterpretować całą resztę bloków. Zwykle jest to drugi blok na urządzeniu. Pierwszy blok jest zazwyczaj zarezerwowany dla bloku startowego (ang. *boot block*), z którego może być uruchomiony system operacyjny.

SYSVFS, a także HTFS, oparty jest na i-węzłach (ang. *i-node*, tłumaczenie wg [6]). I-węzeł to unikalny numer, opisujący jakąś strukturę na dysku (plik, katalog, kolejkę FIFO, urządzenie znakowe lub blokowe, dowiązanie symboliczne). Wszystkie i-węzły są zgromadzone w tablicy i-węzłów w określonym miejscu na dysku. Informacja, gdzie znajduje się to miejsce, zawarta jest w superbloku. Mając numer i-węzła, można ten i-węzeł odczytać z tablicy i-węzłów, a tam zawarte są *metadane*. *Metadane* to informacje organizacyjne systemu plików (np. superblok, i-węzły) W i-węźle zawarte są informacje dotyczące tego, w jaki sposób dotrzeć do danych należących do pliku, na jakich warunkach otrzyma się do nich dostęp a także różne inne, np. typ struktury opisywanej przez i-węzeł, właściciel i grupa, czasy ostatniej modyfikacji i dostępu, wreszcie wskaźniki do bloków z danymi, jeśli typ struktury tego wymaga.

Na rysunku 1 jest pokazane, w jaki sposób zorganizowane są bloki danych należące do i-węzła.

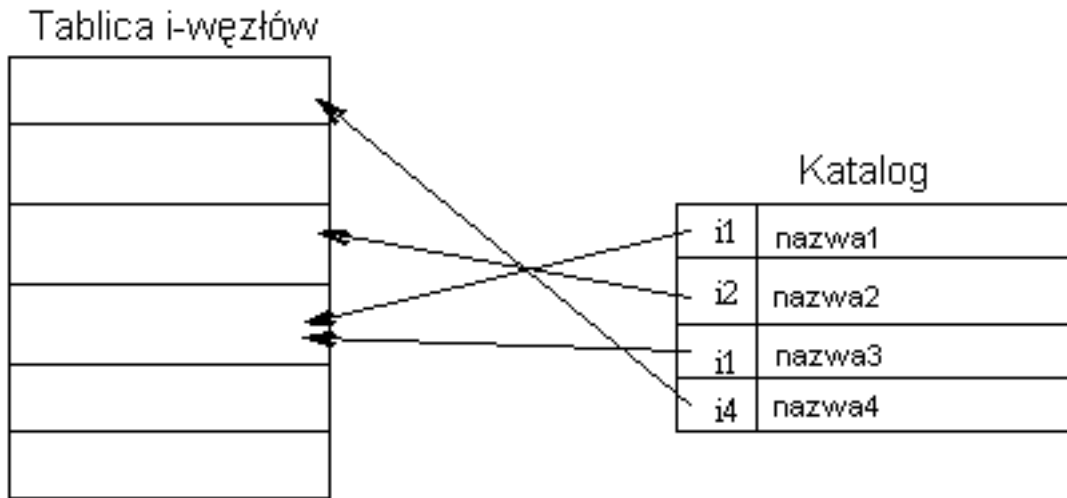


Rysunek 1: Wskaźniki do danych pliku w i-węźle w systemie Unix

To standardowe uniksowe rozwiązanie. Dostęp do małych plików, nie przekraczających 10 bloków, jest bezpośredni. Pliki przekraczające 10 bloków adresowane są pośrednio przez jeden blok ze wskaźnikami do bloków danych. Gdy plik przekracza wielkość 10 bloków + ilość wskaźników do bloków mieszczących się w jednym bloku, używany jest kolejny poziom bloków pośrednich. Uzyskuje się wtedy właściwość powodującą to, że im mniejszy jest plik, tym szybciej można się do niego dostać (zwiększa to wydajność np. w przypadku operowania na wielu małych plikach). W przypadku wielkości bloku równej 2048 kilobajtów, maksymalny rozmiar pliku wyniesie  $(10 + 32 + 32 * 32) * 2048$ , co daje około 2 gigabajty (znane ograniczenie systemu plików *ext2*).

W porównaniu do rysunku 1, w HTFS jest jeszcze jeden poziom wskaźnika - potrójnie pośredni. Oczywiście daje to możliwość przechowywania dużo większych plików.

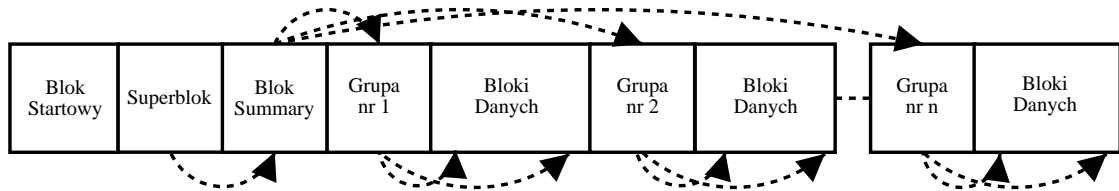
Wpisy katalogowe (ang. *directory entry*) w takim systemie plików zawierają tylko nazwę i numer i-węzła, do którego odnosi się dany plik (rysunek 2).



Rysunek 2: Zależność między wpisami katalogowymi a i-węzłami w systemie Unix

Dzięki temu przeszukiwanie katalogów z wieloma plikami jest szybsze niż w systemach, gdzie wpis katalogowy zawiera od razu wszystkie metadane dotyczące obiektu, z którym jest związany. Jest to też wygodne przy rozmieszczaniu bloków danych względem metadanych, ponieważ struktura i-węzła jest zawsze tego samego rozmiaru (a w większości nowszych systemów plików struktura wpisu katalogowego jest zmiennego rozmiaru, ze względu na dużą maksymalną długość nazwy pliku). Powoduje to też między innymi to, że możliwa jest sytuacja, że do jednego i-węzła odnosić się może więcej wpisów katalogowych (dowiązania twarde), albo nawet taka, że żaden wpis nie wskazuje na dany i-węzeł (po wykonaniu operacji *unlink()*, gdy liczba dowiązań twardych spadnie do zera, a jakieś procesy jeszcze korzystają z obiektu, na który wskazuje tenże i-węzeł. Ułatwia to znacznie obsługę wielodostępu do obiektu związanego z i-węzłem i zwalnia z konieczności implementacji przymusowych blokad dostępu do obiektu, jak to ma miejsce np. w MS Windows z systemem plików FAT).

HTFS został rozszerzony w porównaniu do SYSVFS o podział na grupy i-węzłów (rysunek 3).



Rysunek 3: Schemat systemu plików HTFS

Podział taki związany jest z poprawą wydajności. Polega on na rozmieszczeniu bloków z metadanymi w różnych miejscach dysku. Chodzi tutaj o czas dostępu - bloki danych należące do danego i-węzła umieszczone są wtedy blisko samej struktury i-węzła i wskaźników pośrednich do bloków. Jest to szczególnie ważne w przypadku dużych plików, do których dostęp jest realizowany właśnie przez wiele pośrednich wskaźników. Rozwiązanie to jest rozszerzeniem standardowego systemu plików Uniksa. Podobne rozwiązania (podziały na grupy i-węzłów, grupy bloków itp.) są stosowane w prawie wszystkich obecnie stosowanych systemach plików - EXT2 i EXT3, NTFS, REISERFS, XFS i innych, poza FAT. W najnowocześniejszych systemach plików (np. XFS firmy SGI) podział na grupy związany jest też z możliwością równoległej modyfikacji metadanych w różnych grupach bez utraty integralności systemu plików.

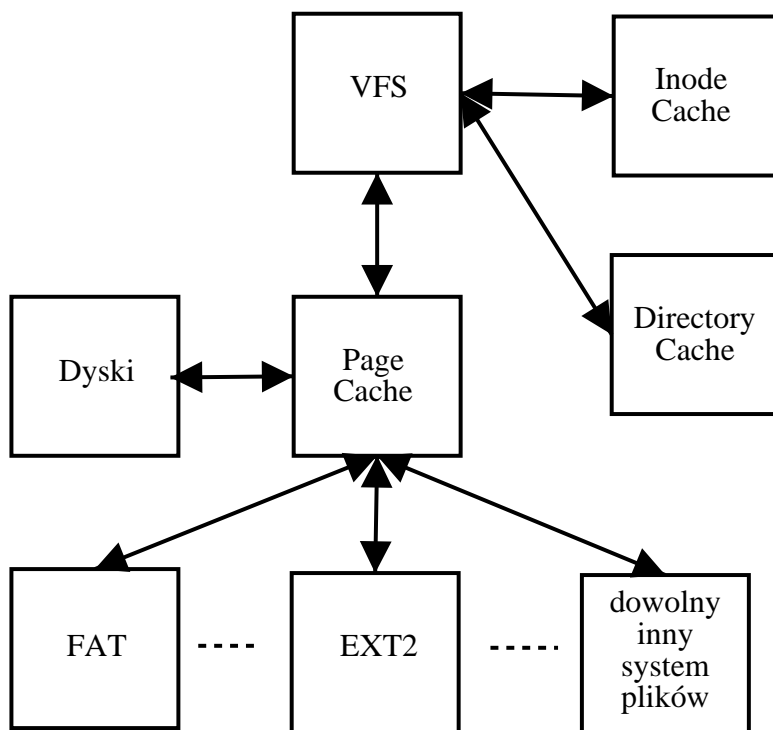
Strzałki na rysunku 3 obrazują, w jaki sposób system operacyjny dostaje się do kolejnych struktur metadanych systemu plików, zanim wreszcie dotrze do bloków z danymi. W superbloku jest wskaźnik do bloku, gdzie znajdują się wskaźniki do poszczególnych grup i-węzłów (blok *summary*). Mając grupę i-węzłów, wybiera się z niej szukany i-węzeł, a mając i-węzeł odczytuje się bloki danych, na które ten wskazuje.

System plików HTFS ma jeszcze jedno rozszerzenie w porównaniu do SYSVFS. Jest to księgowanie (ang. *journaling*). Mechanizm ten służy do uodpornienia systemu plików przed utratą integralności w przypadku niespodziewanych awarii zasilania lub innych sytuacji powodujących nieprawidłowe zamknięcie systemu. Ponieważ dotyczy to implementacji zapisu, istota działania tego mechanizmu nie będzie tutaj opisywana.

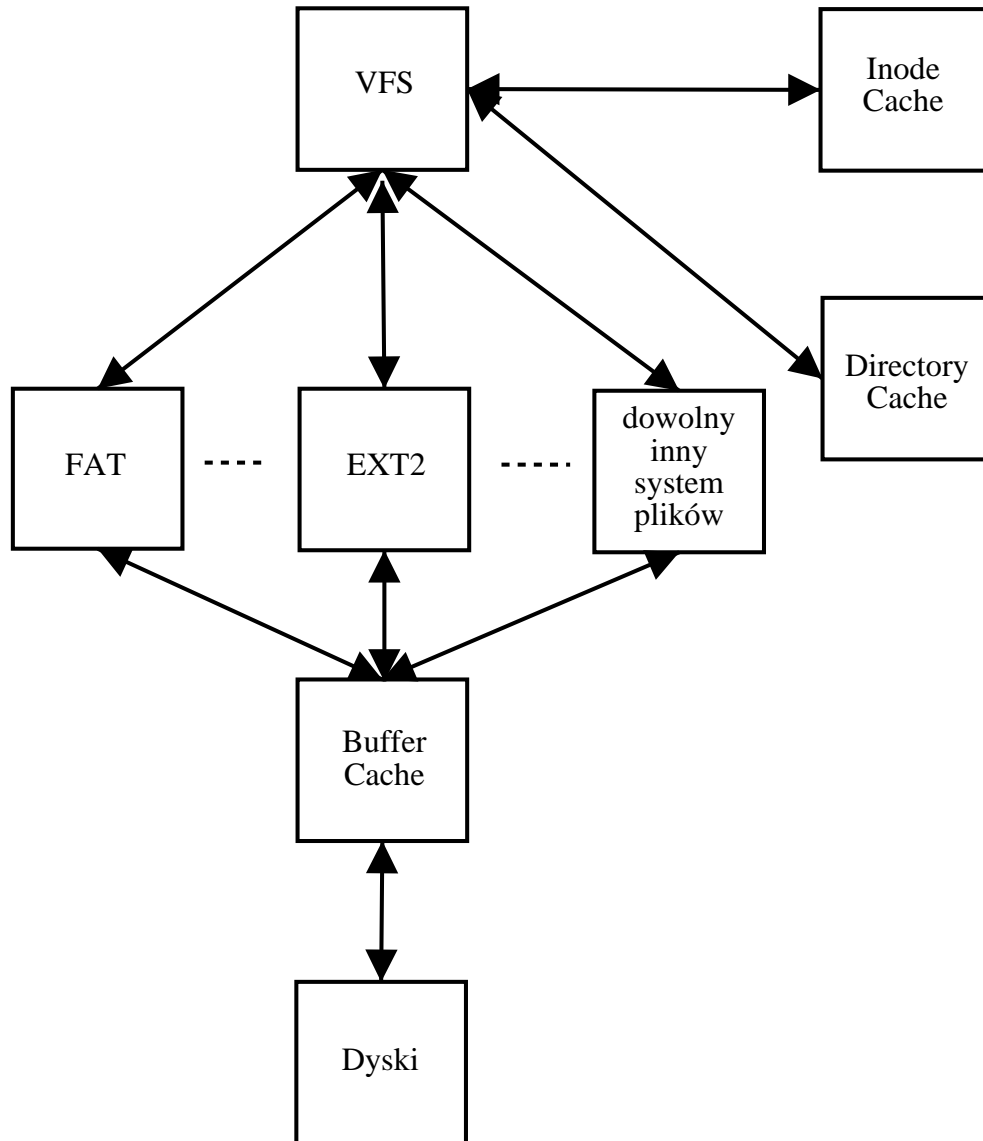
Struktury systemu plików HTFS i szczegółowe informacje dotyczące ich wykorzystania są opisane w załączniku 1.

## 4.2 Obsługa systemów plików w jądrze Linuksa

Za obsługę i przełączanie systemów plików w Linuksie jest odpowiedzialny interfejs VFS (określany dalej również jako Wirtualny System Plików). Na rysunkach 4 i 5 przedstawiona została interakcja VFS z innymi częściami jądra Linuksa 2.4 z wykorzystaniem mechanizmu *pagecache*, jak i Linuksa 2.2 przy użyciu mechanizmu *buffercache*.



Rysunek 4: Schemat działania VFS (Wirtualnego Systemu Plików) w Linuksie 2.4



Rysunek 5: Schemat działania VFS (Wirtualnego Systemu Plików) w Linuksie 2.2

VFS można by określić jako jeden wielki system plików, scalający w jedno wiele rzeczywistych systemów plików, znajdujących się na urządzeniach. Struktury VFS (i-węzeł, superbloki) rezydują tylko w pamięci RAM (nie ma ich fizycznie na żadnym dysku), a zadaniem implementacji rzeczywistego systemu plików jest dostarczenie mechanizmów do wczytania z dysku odpowiedniej struktury i przekształcenia jej w taki sposób, by pasowała do

struktur VFS. Ponieważ VFS skonstruowany jest podobnie, jak standardowy (rzeczywisty) system plików Uniksa (oparty jest na i-węzłach), oczywistym jest, że bardziej będą do niego pasowały takie właśnie systemy plików. Jednak jest on na tyle elastyczny, że przy odrobinie pomysłowości można do niego dopasować również systemy plików oparte o inne struktury (od dawna jest dostępna pełna implementacja systemu plików FAT, a także implementacja tylko do odczytu systemu plików NTFS). Wada tego rozwiązania jest taka, że obsługa systemów plików, które dość znacznie odbiegają strukturą od systemu plików Uniksa, jest mniej wydajna niż w systemie operacyjnym, z którego pochodzi dany system plików (można to zaobserwować na systemie plików FAT).

Z punktu widzenia użytkownika dołączenie i korzystanie z systemu plików wygląda następująco: przy starcie systemu, pod katalog główny („/”) VFS montowany (z ang. *mount*) jest pierwszy system plików. Struktura katalogów tego systemu plików staje się strukturą katalogów VFS, a następne systemy plików można podpinąć pod dowolne z katalogów tej struktury. Użytkownik mający odpowiednie prawa może tego dokonać poleceniem *mount*. Np. polecenie *mount -t vfat /dev/hda1 /mnt/win*, jeśli się powiedzie, spowoduje zamontowanie systemu plików Windows 9x pod katalog */mnt/win*. Użytkownik widzi jeden wielki system plików i z jego punktu widzenia podział na części składowe (rzeczywiste systemy plików) jest niewidoczny.

Różnica pomiędzy Linuksem 2.2 i 2.4 polega na tym, że ten drugi posiada mechanizm *pagecache*. Przy wykorzystaniu mechanizmu *pagecache*, implementacja systemu plików jest zwolniona z konieczności bezpośredniej komunikacji ze sterownikami urządzeń i samodzielnej obsługi buforowania. Jest to wyraźny krok w stronę uproszczenia sposobu implementowania systemów plików. Dokładniejszy opis i porównanie tych dwóch mechanizmów znajdzie Czytelnik w załączniku 2, w punkcie 6.5 a także w [1].

*Inode cache* i *directory cache* to mechanizmy VFS do buforowania metadanych (i-węzłów i wpisów katalogowych). Nie jest to tematem tej pracy i nie będzie tu omawiane (informacje Czytelnik znajdzie w [1] i [2]).

Funkcje Wirtualnego Systemu Plików są przez jądro eksportowane - znajdują się w tablicy symboli. Dlatego implementację taką można zrealizować jako moduł (czyli kod jądra, dołączany na życzenie w dowolnym czasie do działającego już jądra). Implementacja HTFS w Linuksie, przedstawiona w dalszych rozdziałach, korzysta właśnie z tego mechanizmu.

Dalsze szczegółowe informacje znajdzie Czytelnik w załączniku 2 a także w rozdziale 6.

# Rozdział 5

## Założenia dotyczące implementacji systemu plików HTFS w systemie operacyjnym Linux

Poniżej zostały przedstawione etapy, przez które należy przejść w celu zrealizowania celu niniejszej pracy.

### 1. Instalacja systemu operacyjnego SCO Open Server 5

Caldera International (producent SCO Open Server) udostępnia darmową licencję dla pojedynczych użytkowników. Taką licencję, wraz z niezbędnymi kodami, można uzyskać ze strony *www.caldera.com*. Na instalację i badania przygotowano dyski 2GB i 130MB. Po instalacji uzyskuje się gotowy system plików do badania a także pliki nagłówkowe.

### 2. Przegląd i interpretacja plików nagłówkowych

Po instalacji, w katalogu */usr/include* (standardowy katalog z plikami nagłówkowymi w Uniksach) powinny znaleźć się odpowiednie pliki nagłówkowe, które będzie można przenieść programem *tar* do Linuksa w celu włączenia potrzebnych struktur do tworzonego kodu.

### 3. Porównanie plików nagłówkowych z konkretnym obrazem systemu plików

Na tym etapie, używając odpowiedniego binarnego edytora (np. *hexedit*) i własnych narzędzi, w oparciu o odpowiednie pliki nagłówkowe będą prowadzone starania w celu właściwego zinterpretowania metadanych znajdujących się na dysku.

#### 4. Narzędzia w trybie użytkownika i wstępne testy

Tutaj zaplanowano stworzenie prostych narzędzi w trybie użytkownika, spełniających funkcje uniksowych poleceń *ls* i *cat*, które oferują możliwość poruszania się po systemie plików i przeprowadzenia wstępnych testów poprawności zinterpretowanych metadanych oraz poprawności powstałego kodu. Już tutaj można wymienić wady i zalety takiego rozwiązania:

zalety:

- możliwość kompilacji także pod innymi Uniksami, z niewielką zmianą kodu także pod innymi systemami operacyjnymi
- nie ma szczególnej potrzeby dbałości o aktualność kodu (funkcje biblioteki *libc* definiowane przez standard Posix)

wady:

- ograniczenie do kilku napisanych narzędzi
- wolne działanie
- konieczność przepisania dużej ilości kodu już zawartego w jądrze

Implementacja takiego kodu ma jeszcze jedną zaletę z punktu widzenia implementującego. Pomaga zapoznać się dobrze z nowym systemem plików przed pisaniem kodu będącego częścią systemu operacyjnego. Poprawianie błędów jest znacznie łatwiejsze, jeśli chodzi o zwykły kod w trybie użytkownika, w przeciwieństwie do kodu systemu operacyjnego.

#### 5. Kod systemowy do jądra Linuksa

Po poznaniu systemu plików HTFS i doświadczalnym sprawdzeniu nabytej wiedzy, będzie możliwość zaimplementowania obsługi HTFS w jądrze Linuksa. Oto zalety i wady tego rozwiązania:

zalety:

- szybkie działanie
- wykorzystanie istniejących w Linuksie mechanizmów implementacji różnych systemów plików (VFS)
- nie trzeba pisać części kodu wspólnego dla różnych systemów plików

- możliwość skorzystania z kodu do obsługi systemu plików SYSVFS (który zaimplementowano już w Linuksie)
- możliwość korzystania z olbrzymiej bazy oprogramowania linuksowego na nowo zaimplementowanym systemie plików

wady:

- nie można przenieść do innych systemów
- należy dbać o uaktualnienie wraz z rozwojem jądra
- błędy w kodzie mogą spowodować niestabilność całego systemu

Jądra z serii 2.2 i 2.4 różnią się systemem buforowania. (w 2.4 można co prawda korzystać ze starego kodu buforującego, ale wprowadzono także inny, nowocześniejszy) Celem podstawowym będzie implementacja kodu dla serii 2.4, ponieważ linia ta jest nowsza i będzie powoli wypierać serie 2.2. Jednak ze względu na ciągle szerokie zastosowanie serii 2.2, w ramach pracy zostanie zaimplementowany kod również dla tych jąder.

## **6. Testy kodu do jądra systemu operacyjnego Linux**

Na tym etapie powinny zostać przeprowadzone gruntowne testy dotyczące poprawności działania zaimplementowanego kodu. Testy muszą być dokładne, ponieważ jest to część jądra i jako taka może spowodować poważne szkody w systemie. Testy te powinny obejmować różnej wielkości partycje z systemem plików HTFS, jak i kompilacje w otoczeniu różnych wersji jądra. Planowane jest udostępnienie kodu przez sieć Internet i zaangażowanie w testy osób, które chciałyby skorzystać z tego rozwiązania.

## Rozdział 6

# Implementacja systemu plików HTFS w Linuksie

### 6.1 Instalacja systemu operacyjnego SCO Open Server 5

Przy instalacji okazało się, że Open Server posiada własny podział na partycje, niezależny od dosowego programu *fdisk* (partycje te zostały tam określone jako *divvy slices*). Podziałem tym zarządza się programem *divvy*, który jest standardowo instalowany wraz z systemem. Takie rozwiązanie zostało wprowadzone, ponieważ Open Server wymaga do instalacji 4 partycji, co w owych czasach (gdy dosowy *fdisk* nie miał jeszcze możliwości zakładania partycji typu *extended*) uniemożliwiłoby zainstalowanie na takim komputerze innego systemu operacyjnego. Dlatego dzielono dysk *fdiskiem*, a następnie jedną z partycji używano do instalacji Open Servera, a program *divvy* dzielił ją na swój własny sposób.

Niestety brak jest jakiegokolwiek dokumentacji, jak ten podział jest fizycznie dokonywany na dysku. Problem ten został rozwiązany prowizorycznie, o czym będzie mowa dalej.

Rozwiązanie, które zastosowano wcześniej w celu zbadania samego systemu plików HTFS, to założenie tego systemu bezpośrednio na drugim dysku (lub partycji dosowej) bez używania *divvy*. (komendą *mkfs -t HTFS inny dysk*). Daje to pewność, że badany jest właśnie HTFS, a nie nagłówki podziału na *divvy slices*.

Następnie znaleziono systemowe pliki nagłówkowe, gdzie były również te dotyczące systemu plików HTFS (w katalogu */usr/include/sys*). Programem *tar*, przy użyciu dyskietki, zostały przeniesione do systemu Linux, w którym badania były kontynuowane.

## 6.2 Porównanie plików nagłówkowych z konkretnym obrazem systemu plików

W punkcie tym trzeba było, na założonym systemie plików, odczytać i zinterpretować informacje uzyskane z plików nagłówkowych. Pierwszą rzeczą, którą należy w takiej sytuacji zrobić, to dowiedzieć się, gdzie leży superblok, a stąd na jakiej wielkości blokach (dalej zwanych blokami logicznymi) oparty jest założony system plików (taka informacja jest w superbloku). Podczas prób okazało się, że struktura superbloku jest zawarta w pierwszym bloku systemu plików, lecz przesunięta jest o 512 bajtów. Wynika to stąd, że standardowe urządzenia blokowe (dyski IDE, SCSI, dyskietka) operują na blokach o rozmiarze 512 bajtów (dalej nazywanych fizycznymi), a pierwszy blok fizyczny zarezerwowano tutaj na blok startowy. Superblok znajdujemy na podstawie jego magicznej liczby (patrz punkt 7), obliczając jej pozycję w bajtach w obrębie całej struktury superbloku + 512 bajtów bloku startowego. Następnie odczytujemy z superbloku najważniejsze informacje (struktura jest przedstawiona w załączniku 1). Pierwszą rzeczą do odczytania jest rozmiar bloku logicznego. HTFS założony na dysku o rozmiarze 130MB miał wielkość bloku 1024 bajty. Następnie odczytano informacje o liczbie grup i-węzłów (w tym przypadku była to tylko 1 grupa). Te informacje to najważniejsze parametry, które będą potrzebne przy każdej następnej operacji. Chcąc teraz dostać się do konkretnej ścieżki (np. `/bin/sh` - to ścieżka prowadząca do pliku wykonywalnego z domyślnym interpreterem poleceń), wykonujemy kolejno następujące operacje <sup>1</sup>:

- *block* - numer bloku
- *offset* - przesunięcie w bajtach wewnątrz bloku
- *inode* - numer i-węzła
- *blocksize* - rozmiar bloku systemu plików

1. Numer bloku i pozycję wewnątrz bloku, gdzie leży blok summary (składający się ze struktur `htfs_igsum` - każda opisująca jedną grupę), obliczamy następująco:

$$block = \frac{((inode/2^{22}) * sizeof(struct htfs_igsum))}{blocksize} \quad (6.1)$$

---

<sup>1</sup>Algorytm zakłada, że podana ścieżka istnieje. W przeciwnym wypadku wyszukiwanie kończy się niepowodzeniem; niezalezieniem wpisu w katalogu lub błędnym typem i-węzła (różnym od typu *katalog*)

$$offset = ((inode/2^{22}) * sizeof(struct htfs_igsum)) \bmod blocksize \quad (6.2)$$

I-węzeł główny (ang. *root inode*), wskazujący na katalog „/”, ma numer 2. Przy pierwszym przejściu przez ten algorytm taką właśnie wartość należy podstawić we wzorach jako *inode*. Przy kolejnych przejściach jest to numer i-węzła odczytany ze struktury *htfs\_direct* (jak to opisano dalej).

2. Ze struktury *htfs\_igsum* dostajemy informację o numerze pierwszego bloku, gdzie leżą struktury i-węzłów (*htfs\_dinode32*). Chcąc odczytać i-węzeł o podanym numerze korzystamy z następujących wzorów:

$$block = \frac{inode * sizeof(struct htfs_dinode32)}{blocksize} \quad (6.3)$$

$$offset = inode * sizeof(struct htfs_dinode32) \bmod blocksize \quad (6.4)$$

3. Teraz następuje odczyt i-węzła. Sprawdzany jest jego typ. Jeśli jest to dowiązanie symboliczne, podąża się za jego wskazaniem, a następnie skacze się początku algorytmu z nową ścieżką (np. gdyby */bin/sh* wskazywał na */bin/bash*, należy teraz szukać tego drugiego). Zmienna *inode* znowu przyjmuje wartość 2.

W przeciwnym wypadku

4. Odczytując i-węzeł, dostaje się wskaźniki do bloków, na które ten wskazuje <sup>2</sup>. Jeśli jest to koniec ścieżki, to cel został osiągnięty (w podanym przykładzie byłby to i-węzeł wskazujący na plik *sh* w katalogu *bin*). Jeśli nie, to sprawdzamy typ i-węzła. Jeżeli jest różny od typu *katalog*, algorytm kończy się z błędem.

W przeciwnym wypadku

5. należy przeglądać wpisy katalogowe w aktualnym i-węźle którego danymi są bloki zawierające struktury *htfs\_direct*. Chcąc teraz dotrzeć do odpowiedniego wpisu, przegląda się wpisy po kolei porównując nazwy (w przykładzie najpierw szuka się ciągu *bin*). Ponieważ w HTFS długość nazw plików w rekordzie *htfs\_direct* (pole *name*) jest zmienna,

---

<sup>2</sup>Chyba, że dany typ i-węzła nie posiada bloków danych, jak na przykład urządzenie czy kolejka FIFO

sama długość tego rekordu jest także zmienna. Pole `d_namlen` określa długość w znakach pola `name` (z którym porównuje się szukany ciąg znaków), natomiast pole `d_reclen` określa długość rekordu (długość rekordu jest przypuszczalnie zaokrąglana w taki sposób, żeby nie obejmowała 2 bloków - końca jednego i początku drugiego). Dlatego do następnego wpisu katalogowego należy przesunąć się o długość pola `d_reclen`. Jeśli wpisu nie znaleziono, algorytm kończy się z błędem.

W przeciwnym wypadku

6. po znalezieniu odpowiedniej nazwy, korzysta się z pola `d_ino`, w którym zawarty jest numer i-węzła danego wpisu katalogowego, po czym następuje skok do początku algorytmu, przy czym zmienna `inode` przyjmuje wartość aktualnie znalezionej nazwy i-węzła.

Mając strukturę i-węzła, można dostać się do bloków danych należących do tego i-węzła. Poniżej przedstawiony jest algorytm, który dostaje na wejściu numer bloku należącego do i-węzła, a zwraca odpowiadający mu numer bloku systemu plików.

- `i_block` - numer bloku i-węzła
- `fs_block` - numer bloku systemu plików
- `sizeof(daddr_t)` - rozmiar bloku na dysku (w bajtach)
- $ipb = \frac{blocksize}{sizeof(daddr_t)}$  (ilość wskaźników do bloków przypadających na jeden blok)
- `addr = inode.d32_addr`
- nawiasy kwadratowe `[]` oznaczają indeksowanie tablicy w pamięci
- nawiasy klamrowe `{}` oznaczają indeksowanie wskaźnika w bloku na dysku

1. jeśli `i_block < 10`

$$fs\_block = addr[i\_block]$$

zwróć `fs_block`

w przeciwnym wypadku

- 2.

$$i\_block = i\_block - 10$$

jeśli  $i\_block < ipb$

$$fs\_block = addr[10]\{i\_block\}$$

zwróć  $fs\_block$

w przeciwnym wypadku

3.

$$i\_block = i\_block - ipb$$

jeśli  $i\_block < ipb^2$

$$fs\_block = addr[11]\left\{\frac{i\_block}{ipb}\right\}\{i\_block \bmod ipb\}$$

zwróć  $fs\_block$

w przeciwnym wypadku

4.

$$i\_block = i\_block - ipb^2$$

$$fs\_block = addr[12]\left\{\frac{i\_block}{ipb^2}\right\}\left\{\frac{i\_block \bmod ipb^2}{ipb}\right\}\{i\_block \bmod ipb^2 \bmod ipb\}$$

zwróć  $fs\_block$

w przeciwnym wypadku zgłoś błąd.

## 6.3 Narzędzia w trybie użytkownika

Ta część kodu obejmuje dwa prymitywne narzędzia *sco\_ls* i *sco\_cat*, odpowiadające standardowym uniksowym poleceniom *cat* i *ls* (kod ten znajduje się w pliku *htfs\_linux.tar.gz* i jest dalej nazywany pakietem *htfs\_linux*). Został on napisany w języku C i korzysta ze standardowych funkcji biblioteki *libc*. W obecnej postaci powinien dać się skompilować na większości systemów uniksowych. Służył on głównie do zapoznania się z systemem plików HTFS, dlatego jego funkcjonalność jest ograniczona (głównym celem był kod systemowy do jądra Linuksa). Sposób kompilacji i użycia tych narzędzi opisany został w załączniku 3.

Pakiet jest bardzo przydatny w przypadku, gdy korzystamy z partycji podzielonej przez *divvy* (patrz punkt 6.1). Do pakietu został dołączony program *hdinfo\_sco* autorstwa Krzysztofa G. Baranowskiego, który wyświetla informacje o podziale partycji na *divvy-slices*. Ponieważ informacje o wyszukiwanych partycjach *divvy* okazały się niedokładne (np. partycję *divvy* z

systemem plików HTFS odnaleziono o kilka bloków dalej, niż by to wynikało z informacji wyświetlonych przez *hdinfo\_sco*), w skład pakietu *htfs\_linux* wchodzi też program *readsuper\_sco*, który, korzystając z wyników działania *hdinfo\_sco* lokalizuje dokładnie, gdzie znajduje się początek partycji HTFS (przeszukuje po kolei 512-bajtowe bloki w poszukiwaniu superbloku HTFS, poczynając od bloku wskazywanego przez *hdinfo\_sco*). Program tworzy w bieżącym katalogu plik *sco\_fsstart.blk*, w którym zapisana jest nazwa urządzenia i blok startowy (w blokach wielkości 512 bajtów), od którego zaczyna się system plików HTFS. Z informacji w tym pliku korzystają potem *sco\_ls* i *sco\_cat* (dzięki czemu nie trzeba im podawać parametrów za każdym wywołaniem). Ta informacja będzie również bardzo cenna przy korzystaniu z modułu do jądra przy użyciu urządzenia *loop* (patrz punkty 6.4 i 7).

Testy objęły kilka stworzonych partycji HTFS i wszystkie wykryte usterki zostały poprawione. Nie zaimplementowano podążania za dowiązaniem symbolicznym, ponieważ wymagało to stworzenia dużej ilości kodu, która istnieje już w jądrze Linuksa i działa bardzo sprawnie z HTFS (patrz punkt 6.4). Wskazania dowiązań symbolicznych są jednak wyświetlane przez *sco\_ls*, dlatego użytkownik może ręcznie dostać się do obiektu, na który wskazuje dowiązanie. Można też stworzyć zestaw skryptów lub dodać kod rozszerzający funkcjonalność pakietu *htfs\_linux*.

## 6.4 Kod systemowy do jądra Linuksa

Ponieważ to główny cel użyteczny tej pracy, stworzony kod systemowy będzie opisany nieco dokładniej. Kod ten został zaimplementowany jako moduł do jądra Linuksa z serii 2.4. Moduł to część jądra Linuksa ładowana do pamięci wykonującego się już jądra w określonych warunkach (jeśli zajdzie potrzeba). Moduł to plik o rozszerzeniu *.o* w formacie binarnym ELF. Podczas ładowania do pamięci jądra (można to zrobić z linii poleceń komendą *insmod* będąc użytkownikiem *root* lub mając odpowiedni przywilej) następuje proces dynamicznej konsolidacji. Polega to na zastępowaniu (relokacji) adresów w tablicy symboli modułu adresami wolnej pamięci, w której jądro umieści struktury modułu. Odwołania do globalnej tablicy symboli jądra zostają natomiast zastąpione przez właściwe adresy pobrane z tej tablicy.

### 6.4.1 Specyfikacja zewnętrzna

Moduł *htfs.o* to standardowy moduł Linuksa do obsługi systemu plików. Jego użycie następuje w momencie wywołania polecenia *mount* z opcją *-t htfs*. Dokładniejszy opis jest zawarty w punkcie 7 oraz na stronie elektronicznego

podręcznika *mount(8)*.

### 6.4.2 Specyfikacja wewnętrzna

W skład kodu źródłowego modułu *htfs.o* wchodzi następujące pliki:

```
COPYING
README
Makefile
fs.h.diff
linux/htfs_fs.h
linux/htfs_fs_i.h
linux/htfs_fs_sb.h
Mdir.c
file.c
inode.c
namei.c
symlink.c
```

Pliki *COPYING* i *README* to pliki tekstowe. Plik *Makefile* to plik wejściowy dla polecenia *make*. *fs.h.diff* to plik w formacie *diff* (wynik działania polecenia *diff*), który jest plikiem wejściowym dla polecenia *patch*. Reszta to pliki źródłowe w języku C. Poniżej znajduje się dokładniejszy opis plików.

- **README**: krótki opis, jak skompilować, zainstalować i korzystać z modułu *htfs.o*
- **COPYING**: plik z licencją, na jakiej rozpowszechniany jest moduł (patrz punkt 6.4.4)
- **Makefile**: plik wejściowy dla polecenia *make*, które służy do zautomatyzowania procesu kompilacji

Kolejne pliki dotyczą już ściśle kodu źródłowego w języku C. Kolejność ich opisywania odpowiada kolejności powstawania w procesie implementacji.

- ***fs.h.diff***

Plik ten zawiera modyfikacje, które należy wykonać w pliku *include/linux/fs.h* przed kompilacją. Polegają one na rozszerzeniu unii *u* w strukturze *super\_block* (patrz punkt 4.1) o pole `struct htfs_inode_info htfs_i` oraz w strukturze *inode* o pole `struct htfs_inode_info htfs_i`, żeby wygodnie odwoływać się do informacji specyficznych dla systemu plików HTFS. Plik ten jest używany w poleceniu *patch*, a zmodyfikowany plik *fs.h* jest umieszczany w bieżącym katalogu.

- *htfs\_fs.h*

Plik nagłówkowy zawierający struktury systemu plików HTFS

- *htfs\_fs\_i.h* i *htfs\_fs\_sb.h*

Pliki nagłówkowe zawierające struktury specyficzne odpowiednio dla i-węzła i superbloku systemu HTFS, a które mają być umieszczone w pamięci po odczytaniu i-węzła lub superbloku (są to struktury `htfs_sb_info` i `htfs_inode_info` wspomniane wyżej przy opisywaniu pliku *fs.h.diff*).

- *inode.c* Ten plik zawiera metody operujące na superbloku i i-węzłach. Zaimplementowano następujące z nich:

```
static struct super_operations htfs_sops = {
    read_inode:    htfs_read_inode,
    put_super:     htfs_put_super,
    statfs:       htfs_statfs
};

struct address_space_operations htfs_aops = {
    readpage: htfs_readpage,
    bmap: htfs_bmap
};
```

Poniżej przedstawiono najważniejsze z funkcji:

- `htfs_read_super()` Ta funkcja jest wywoływana podczas wywołania systemowego *mount* (gdy system plików zostanie zamontowany). Funkcja ta odczytuje do bufora superblok (bufor ten zostanie zwolniony przy odmontowaniu systemu plików, w funkcji `htfs_put_super()`) i bada, czy rzeczywiście jest to system plików HTFS (przez porównanie z *magiczną liczbą*). Po pomyślnym wykonaniu tych operacji następuje wypełnianie pól struktury `super_block`. Wypełniane są między innymi takie pola jak rozmiar bloku, a także specyficzne pola dla HTFS, jak ilość grup i-węzłów czy numer bloku *summary*. Blok *summary* jest także, w celu przyspieszenia operacji na systemie plików, odczytywany do bufora w pamięci. Są tu także pola wypełniane różnymi wartościami bezpośrednio obliczonymi z innych pól tej samej struktury `super_block`; to również zostało wprowadzone jako rozwiązanie podnoszące wydajność kodu. Tutaj jest także przypisywany wskaźnik do odpowiednich metod operujących na superbloku

(struktura typu `super_operations`). Na końcu funkcja przydziela miejsce w buforach wpisów katalogowych (ang. *directory entry cache*) na główny katalog („/”) systemu plików i łączy go z odpowiednim i-węzłem (określonym przez `HTFS_ROOT_INO` i równym 2). I-węzeł wypełniany jest dopiero, gdy zajdzie potrzeba jego odczytu.

– `htfs_read_inode()`

To funkcja spełniająca podobną rolę jak `htfs_read_super()` dotyczy jednak wypełniania struktury `inode`. Funkcja zna numer i-węzła, który ma odczytać z dysku, i jej zadaniem jest wyszukanie odpowiedniego i-węzła na dysku i wypełnienie struktury `inode` odczytanymi informacjami. Funkcja wylicza numer bloku i pozycję i-węzła wewnątrz bloku według wzorów z punktu 6.2, odczytuje blok i wypełnia pola w strukturze `inode` otrzymanymi informacjami. Następnie, zależnie od typu i-węzła (katalog, plik, dowiązanie symboliczne lub plik specjalny) przypisuje do i-węzła odpowiednie metody<sup>3</sup>. Na końcu funkcja zwalnia bufor, do którego wczytany został blok z dysku.

– `htfs_get_block()` i `htfs_block_bmap()`

Funkcja `htfs_block_bmap()` dostaje na wejściu strukturę i-węzła oraz numer bloku wewnątrz tego i-węzła, zwraca natomiast odpowiadający mu numer bloku w systemie plików. Algorytm został opisany w punkcie 6.2. Funkcja `htfs_get_block()` dodatkowo wiąże z tym blokiem bufor i oznacza go jako zamapowany. Mając te funkcje zaimplementowane, można bardzo łatwo skonstruować metody dla struktury `address_space_operations` (w przypadku `htfs_fs` implementowane są tylko metody `readpage()` i `bmap()`). Kluczową metodę `readpage()` definiuje się następująco:

```
static int htfs_readpage(struct file *file, struct page *page)
{
    return block_read_full_page(page, htfs_get_block);
}
```

W podobny sposób implementuje się `bmap()` - metodę służącą do mapowania plików do przestrzeni użytkownika. W dalszej części tego punktu będzie opisane, jak skorzystać z tego mechanizmu przy definiowaniu metod do odczytu pliku (struktura `file_operations`).

---

<sup>3</sup>Nie wykryto tutaj, gdzie zapisane są informacje dotyczące plików specjalnych. Dlatego przyjęto, że każdy plik specjalny otrzyma numery *minor* i *major* równe 0

Metoda `statfs` jest wywoływana w celu uzyskania informacji o systemie plików i jest tutaj zaimplementowana tylko częściowo.

- *file.c*

W tym pliku określone są metody operujące na strukturze `file_operations`:

```
struct file_operations htfs_file_operations = {
    read:                generic_file_read,
    mmap:                generic_file_mmap,
};
```

Funkcje `generic_file_read()` i `generic_file_mmap()` to standardowe funkcje w jądrze, które przekazują obsługę odczytu i mapowania do metod określonych w strukturze `address_space_operations`. W efekcie, przy żądaniu odczytu, wywoływana jest metoda `readpage()` (określona w pliku `inode.c`), która przekazuje sterowanie do funkcji `block_read_full_page()` (`fs/buffer.c`). Funkcja ta przydziela stronę pamięci i przypisuje do niej bufory (struktury `buffer_head`), odpowiadające kolejnym blokom pliku. Później wszystkie bufory są asynchronicznie wczytywane do pamięci (funkcja `submit_bh()` zdefiniowana w pliku `drivers/block/ll_rw_blk.c`). Jeśli chodzi o zapis, występują pewne komplikacje. Zainteresowani znajdą o tym informacje w [1].

- *dir.c*

W pliku zaimplementowano następujące metody:

```
struct file_operations htfs_dir_operations = {
    read:                generic_read_dir,
    readdir:            htfs_readdir
};
```

`generic_read_dir()` to standardowa funkcja jądra, która pozwala procesom na odczyt katalogu tak, jak zwykłego pliku (funkcja biblioteczna `read()` wywołana na deskrytorze pliku będącym katalogiem). Funkcja `htfs_readdir` umożliwia odczytanie kolejnego wpisu katalogowego (korzystając ze struktury `file` i zapamiętanej tam pozycji w pliku). Funkcja jest używana, gdy proces chce otrzymać listę plików (lub jej część) w danym katalogu. Algorytm tutaj zastosowany został opisany w punkcie 6.2.

- *namei.c*

Zdefiniowano tutaj jedną metodę:

```
struct inode_operations htfs_dir_inode_operations = {
    lookup:          htfs_lookup
};
```

Jest to metoda, która implementuje wyszukanie nazwy pliku we wpisie katalogowym. Algorytm został przedstawiony w punkcie 6.2. Jest ona bardzo często wywoływana - procesy, które chcą zmienić swoją bieżącą ścieżkę lub otworzyć jakikolwiek plik, wywołują tę funkcję.

- *symlink.c*

Plik określa następujące metody:

```
struct inode_operations htfs_symlink_inode_operations = {
    readlink:      htfs_readlink,
    follow_link:   htfs_follow_link,
};
```

Obie metody zostają wywołane przy operacjach na dowiązaniach symbolicznych. Mają za zadanie zinterpretować, na jaką ścieżkę wskazuje i-węzeł będący dowiązaniem symbolicznym. Następnie wywołują funkcje VFS (odpowiednio `vfs_readlink()`, która powoduje zwrócenie wskazania procesowi i `vfs_follow_link()` zwracająca i-węzeł wskazywany przez dowiązanie symboliczne).

### 6.4.3 Testowanie

Kod został skompilowany i przetestowany na jądrach w wersji 2.4.3, 2.4.5 i 2.4.7.

Pierwszy przeprowadzony test polegał na sprawdzeniu poprawności kodu (wykrycie pomyłek programisty). Używano różnych rozmiarów partycji z systemem HTFS i różnych układów plików i struktur katalogów. Testowanie polegało na wykonywaniu różnych operacji w ten sposób, żeby wszystkie zaimplementowane metody zostały wywołane.

Drugi test to sprawdzenie poprawności zastosowanych algorytmów. Sprawdzano integralność odczytywanych danych (porównanie danych odczytanych

oryginalnym kodem na SCO Open Server 5 i kodem stworzonym dla Linuksa). Użyto różnych rozmiarów plików i różnych struktur katalogów.

W przypadku uwag lub wykrycia błędów, można porozumieć się z autorem pisząc na adres [deresz@truml.art.pl](mailto:deresz@truml.art.pl).

#### 6.4.4 Źródła i licencja

Program został stworzony przy wykorzystaniu i zmodyfikowaniu kodu zawartego w standardowym jądrze Linuksa (*fs/sysv - Linux SystemV/Coherent filesystem routines*). Autorzy tego kodu to: *Linus Torvalds, Bruno Haible, Pascal Haible, Remy Card, Doug Evans* i *Krzysztof G. Baranowski*. Kod ten rozprowadzany jest na licencji GNU General Public License (GPL). Tekst licencji zawarty jest w pliku *COPYING* opisanym w punkcie 6.4.2. Licencja zobowiązuje autora modułu *htfs\_fs* do udostępnienia stworzonego kodu również na licencji GPL. Najnowsza wersja jest dostępna w sieci Internet pod adresem [http://deresz.dhs.org/~deresz/filesystems/htfs\\_fs/](http://deresz.dhs.org/~deresz/filesystems/htfs_fs/)

### 6.5 Implementacja kodu do jądra Linuksa 2.2

Ponieważ Linux w wersji 2.2 jest ciągle powszechnie stosowany, zdecydowano się na przeniesienie kodu również dla tej serii. Modyfikacja polegała na zrezygnowaniu z mechanizmu *pagecache*, którego seria 2.2 nie posiada, i użycie *buffercache*. *Buffercache* jest zorientowany na wczytywanie pojedynczych bloków systemu plików (a nie całej strony pamięci jak w przypadku *pagecache*). Jest to mechanizm mniej wydajny, ponieważ przydzielanie i utrzymywanie struktur pamięci jest dużo bardziej skomplikowane w przypadku operowania na strukturach mniejszych niż strona. *Buffercache* jest też zbyt mało ogólny, żeby mógł być podporządkowany bezpośrednio VFS, dlatego kod operowania na buforach musi być zaimplementowany w konkretnym systemie plików (np. w pliku *file.c* zaimplementowano prosty mechanizm *read-ahead*, czyli wczytywania kilku bloków naprzód, co w Linuksie 2.4 jest realizowane przez *pagecache*). Ta część kodu nie będzie tu jednak dokładniej analizowana. Zainteresowani mogą znaleźć informacje w [2], [4] i [5].

# Rozdział 7

## Wnioski i kierunki dalszych prac

Zasadniczy cel pracy został zrealizowany. Stworzone zostały dwie wersje kodu systemowego, dla jąder Linuksa z serii 2.4 i 2.2. Kod ten pozwala na wygodną pracę i przetwarzanie danych na systemie plików HTFS wszystkimi dostępnymi dla platformy Linux narzędziami. Oprócz tego powstały narzędzia w trybie użytkownika pozwalające na odczyt pojedynczych plików i wyświetlania zawartości katalogów. Te narzędzia pozwalają także na dokładne zlokalizowanie początku systemu plików HTFS w obrębie partycji *divvy*, co jest ważnym parametrem dla kodu systemowego. Wiadomość o stworzeniu tego oprogramowania została wysłana na odpowiednie grupy dyskusyjne dotyczące Linuksa, a stamtąd osoby chcące skorzystać z tego kodu mogą pobrać adres, z którego kod jest dostępny przez sieć Internet. Osoby te mogą, w razie problemów lub wykrycia błędów, kontaktować się pocztą elektroniczną z autorem, który udzieli niezbędnego wsparcia lub wprowadzi poprawki.

Rozwiązanie z kodem systemowym wydaje się być najbardziej odpowiednie do implementacji obsługi dodatkowego systemu plików w systemie operacyjnym. Jest ono najwygodniejsze i cechuje się najlepszą wydajnością.

Do implementacji pozostała obsługa partycji *divvy* oraz obsługa systemu plików HTFS w trybie do zapisu i odczytu. Brakuje także identyfikacji numerów *major* i *minor* dla plików specjalnych określających urządzenia znakowe i blokowe. Na przeszkodzie stoi jednak brak dokumentacji. Brak dokumentacji to największy problem, jeśli chodzi o przedstawione w niniejszej pracy zagadnienia. I odwrotnie, mnóstwo dokumentacji i otwartość źródeł powoduje, że system operacyjny Linux ma wielkie walory edukacyjne - na jego podstawie można nauczyć się nie tylko budowy samego Linuksa, ale funkcjonowania systemów operacyjnych w ogóle. Jeśli chodzi o możliwości implementacji systemów plików w Linuksie, widać, że jego twórcy, wraz z rozwojem jądra,

starają się w miarę możliwości uprościć sposób dołączania nowych systemów plików (patrz porównanie serii 2.2 i 2.4 - punkt 6.5). Kod dotyczący obsługi systemów plików w Linuksie jest też ciągle rozwijany. Świadczy o tym fakt, że jeden z celów niniejszej pracy, implementacja obsługi stron kodowych CP1250 i CP852 dla dysków z systemem plików Windows 9x, zdezaktualizował się, ponieważ został już wprowadzony przez inne osoby pracujące nad rozwojem jądra Linuksa.

# Załącznik 1

## Analiza struktur systemu plików HTFS

Oto wybrane pola superbloku systemu plików HTFS32 <sup>1</sup>:

```
typedef struct htfs_igsum {
    long   gs_free;           /* total free inodes in group */
    long   gs_sblk;          /* starting block address of group */
    long   gs_nblks;         /* fs blocks in group */
    long   gs_inum;          /* first inumber of group */
    sco_daddr_t gs_mapbn;    /* free inode segment bitmap block */
} htfs_igsum_t;

typedef struct htfs_ighdr {
    long   gh_nigrp;         /* total number of inode groups */
    ulong  gh_tiblks;        /* total fs blocks occuppied by inodes */
    sco_daddr_t gh_sumbn;    /* summary block number */
} htfs_ighdr_t;

struct htfs_super_block {
    ulong      s_ysize;       /* size in blocks of i-list */
    sco_daddr_t s_fsize;     /* size in blocks of entire volume */
    short      s_nfree;      /* number of addresses in s_free */
    sco_daddr_t s_free[HTFS_NICFREE];
}
```

---

<sup>1</sup>Tak dokładnie nazywa się system plików zakładany standardowo w SCO Open Server 5, choć w plikach nagłówkowych są struktury opisujące też wcześniejsze wersje HTFS. W ramach niniejszej pracy zaimplementowano właśnie HTFS32.

```

                                /* free block list */
    htfs_ighdr_t  si32_ighdr; /* number of inode groups */
    sco_time_t   s_time;     /* last super block update */
    sco_daddr_t  s_tfree;    /* total free blocks*/
    ino16_t      s_tinode;   /* total free inodes */
    char         s_fname[6]; /* file system name */
    ino32_t      s_logino;   /* i-number of the log file */
    long        s_state;     /* file system state */
    long        s_magic;     /* magic number to indicate new file system */
    long        s_type;     /* type of new file system */
};

#define HTFS_INOTOGRP (1 << 22)
                                /* inodes per inode group */

```

W polu `si32_ighdr` (struktura `htfs_ighdr_t` jest przedstawiona powyżej struktury `htfs_super_block`) zawarta jest całkowita liczba grup oraz wskaźnik do bloku *summary*, w którym leżą struktury `htfs_igsum_t`, opisujące każdą grupę z osobna. Są tam pola opisujące kolejno liczbę wolnych i-węzłów w grupie, początkowy blok, w którym zaczyna się tablica i-węzłów związana z daną grupą, liczba bloków zajęta przez tę tablicę i-węzłów, numer pierwszego wolnego i-węzła, a także blok bitów opisujący wolne i-węzły. I-węzeł główny w HTFS (wskazujący na katalog główny „/”) to i-węzeł numer 2. Liczba i-węzłów na grupę wynosi  $2^{22}$  (stała `HTFS_INOTOGRP`).

Kolejne pola w strukturze `htfs_super_block` to całkowita liczba bloków zajmowana przez i-węzły, całkowita liczba bloków zajmowana przez system plików, rozmiar tablicy, w której jest lista wolnych i-węzłów oraz sama tablica, opisywana wyżej struktura `htfs_ighdr_t`, data ostatniej zmiany superbloku, liczba wszystkich wolnych bloków, liczba wszystkich wolnych i-węzłów, nazwa systemu plików, numer i-węzła, w którym leży dziennik transakcji (dotyczy księgowania), stan systemu plików (sprawdzony lub z możliwością wystąpienia braku integralności - wtedy może być uruchomiony mechanizm dokończenia transakcji lub program sprawdzający integralność), magiczna liczba (ang. *magic number*), po której można sprawdzić, czy rzeczywiście jest to HTFS oraz typ systemu plików (tutaj istotny jest HTFS32).

Pola związane z liczbą wolnych bloków i i-węzłów oraz księgowaniem (ang. *journaling*) nie są tutaj opisywane, gdyż celem pracy było napisanie implementacji tylko do odczytu.

Jak już wspomniano, w strukturze `htfs_igsum_t` jest pole `gs_sblk` wskazujące na pierwszy blok, w którym leży tablica i-węzłów. Pojedyncza struk-

tura i-węzła wygląda następująco:

```
struct htfs_dinode32 {
    ushort  d32_mode;      /* mode and type of file */
    short   d32_nlink;     /* number of links to file */
    ulong_t d32_uid;      /* owner's user id */
    ulong_t d32_gid;      /* owner's group id */
    quad    d32_size;     /* number of bytes in file */
    daddr_t d32_addr[13]; /* disk block addresses */
    time_t  d32_atime;    /* time last accessed */
    time_t  d32_mtime;    /* time last modified */
    time_t  d32_ctime;    /* time status last changed */
    time_t  d32_crtime;   /* time file was created */
};
```

Pola kolejno opisują typ i-węzła oraz jego prawa dostępu, liczbę dowiązań twardych, właściciela i grupę, długość (jeśli konieczna), adresy do bloków na dysku (w przypadku dowiązania symbolicznego znajduje się tam ścieżka dostępu do pozycji katalogowej, na którą dowiązanie to wskazuje. Jest to kolejna cecha, różniąca ten system plików od SYSVFS, gdzie w przypadku dowiązania symbolicznego znajdował się tu wskaźnik do bloku, w którym zawarta była ścieżka dostępu. Rozwiązanie to jest szybkie, ale ogranicza nam długość ścieżki dostępu. W systemie EXT2 dowiązania symboliczne są podzielone - niewidocznie dla użytkownika - na dowiązania szybkie i wolne, odpowiadające kolejno rozwiązaniu zastosowanemu w HTFS i w SYSVFS). Na końcu umieszczono czasy dostępu, modyfikacji, zmiany i utworzenia. Adresy do bloków na dysku w tablicy `d32_addr` to kolejno 10 bezpośrednich adresów, jeden adres pośredni, jeden podwójnie pośredni i jeden potrójnie pośredni

Oto struktura wpisu katalogowego:

```
struct htfs_direct {
    ino32_t    d_ino;      /* inode number of entry */
    ushort    d_reclen;   /* length of this record */
    uchar     d_namlen;   /* length of string in d_name */
    char      d_name[255]; /* name with length <= 255 */
} htfs_direct_t;
```

Pola kolejno opisują numer i-węzła związanego z tym wpisem, długość tego rekordu (która jest zależna od długości nazwy). Następna jest długość nazwy (maksymalnie 255 znaków), a na końcu występuje już sama nazwa. Tu również widać cechę różniącą ten system plików od SYSVFS, w którym rozmiar tego rekordu był stały (a maksymalna długość nazwy pliku wynosiła 14 znaków).

## Załącznik 2

# Wirtualny System Plików Linuksa (VFS)

Aby powiadomić jądro Linuksa o zamiarze skorzystania z nowego systemu plików, należy taki system plików zarejestrować. Służy do tego funkcja `register_filesystem()`, jak na poniższym przykładzie z pliku `fs/htfs/inode.c`:

```
static DECLARE_FSTYPE_DEV(htfs_fs_type, "htfs", htfs_read_super);

static int __init init_htfs_fs(void)
{
    return register_filesystem(&htfs_fs_type);
}
```

Dzięki temu wywołanie komendy `mount -t htfs urządzenie katalog` powoduje, że do jądra zostanie przekazana informacja, że to właśnie ten typ systemu plików ma być użyty.<sup>2</sup>

Funkcja `unregister_filesystem()` służy do poinformowania jądra, że dany system plików nie będzie już używany i jest wywoływana przy odładowaniu modułu odpowiedzialnego za obsługę systemu plików (jeśli dany kod jest implementowany jako moduł).

```
static void __exit exit_htfs_fs(void)
{
    unregister_filesystem(&htfs_fs_type);
}
```

---

<sup>2</sup>Polecenie `mount` może też załadować automatycznie odpowiedni moduł, jeżeli nazywa się on identycznie z typem systemu plików - w tym przypadku musiałyby to być plik `htfs.o`.

`htfs_read_super` to wskaźnik do funkcji, która zostanie wywołana, gdy po raz pierwszy jakiś proces zażąda dostępu do systemu plików. Ma ona za zadanie odczytać podstawowe informacje z superbloku (lub innej struktury, zależnej od systemu plików), odpowiednio je zinterpretować i przekazać je jądro w formie dla niego zrozumiałej. Musi też ustawić katalog główny systemu plików na odpowiadający mu i-węzeł, zarezerwować dla niego pamięć i przekazać go VFS. Jeżeli operacja ta zakończy się sukcesem, jądro jest już w stanie obsłużyć żądania użytkowników związane z plikami leżącymi na danym systemie plików. Jako jeden z parametrów, funkcja ta dostaje wskaźnik do struktury jądra, zdefiniowanej w `include/linux/fs.h`, którą ma za zadanie wypełnić:

```

struct super_block {
    struct list_head      s_list;          /* Keep this first */
    kdev_t                s_dev;
    unsigned long         s_blocksize;
    unsigned char         s_blocksize_bits;
    unsigned char         s_lock;
    unsigned char         s_dirt;
    struct file_system_type *s_type;
    struct super_operations *s_op;
    struct dquot_operations *dq_op;
    unsigned long         s_flags;
    unsigned long         s_magic;
    struct dentry          *s_root;
    wait_queue_head_t     s_wait;

    struct list_head      s_dirty;        /* dirty inodes */
    struct list_head      s_files;

    struct block_device   *s_bdev;
    struct list_head      s_mounts;      /* vfsmount(s) of this one */
    struct quota_mount_options s_dquot; /* Diskquota specific options */

    union {
        struct minix_sb_info   minix_sb;
        struct ext2_sb_info    ext2_sb;
        struct htfs_sb_info    htfs_sb;
        ..... wszystkie systemy plików, które potrzebują
        informacji sb-private ...
    };
};

```

```
        void                *generic_sbp;
    } u;
    ...
};
```

Część z pól zostaje wypełniona przez VFS, część musi wypełnić funkcja `htfs_read_super()`, a część może zostać nie wypełniona (np. pola używane do modyfikacji danych na systemie plików w przypadku implementacji tylko do odczytu). Oto opis pól z pominięciem tych, które nie będą istotne dla implementacji tylko do odczytu:

- `s_list`: podwójnie powiązana lista wszystkich aktywnych superbloków. To wewnętrzna struktura jądra, używana, gdy potrzebuje ono wyszukać odpowiedni superblok.
- `s_dev`: numer *major* i *minor* urządzenia odpowiadającemu systemowi plików
- `s_blocksize`, `s_blocksize_bits`: rozmiar bloku systemu plików (nie urządzenia) oraz  $\log_2(\text{blocksize})$  - ta liczba przydaje się do szybkich operacji przesuwania bitów, czego często używa się w implementacjach systemu plików.
- `s_type`: wskaźnik do struktury `file_system_type`, która jest tworzona przez makro `DECLARE_FSTYPE_DEV` przedstawione wyżej przy omawianiu funkcji `register_filesystem()`
- `s_op`: wskaźnik do struktury `super_operations`, która zawiera specyficzne dla systemu plików metody do operacji na i-węzłach (np. zapis, odczyt). Odpowiednia funkcja `read_super()` (powyżej była to funkcja `htfs_read_super`) ma za zadanie poprawnie zainicjować tę strukturę. Jest ona przedstawiona poniżej.
- `s_root`: wskaźnik do głównego i-węzła systemu plików wskazującego na katalog główny („/”). Jak już zostało wspomniane, wskaźnik ten ustawiany jest w funkcji `read_super()`.
- `s_files`: lista wszystkich otwartych plików na tym systemie plików
- `s_bdev`: wskazuje urządzenie blokowe, na którym leży system plików. Wirtualne systemy plików (takie jak np. *proc*, *devpty*) nie leżą na urządzeniach blokowych i w takich przypadkach pole to nie jest istotne.

Unia `u` zawiera informacje specyficzne dla superbloku danego systemu plików (jeżeli jest to konieczne)

Oto struktura `super_operations` (*include/linux/fs.h*):

```
struct super_operations {
    void (*read_inode) (struct inode *);
    void (*write_inode) (struct inode *, int);
    void (*put_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*statfs) (struct super_block *, struct statfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);
};
```

Dla trybu tylko do odczytu należy zaimplementować następujące metody:

- `read_inode()`: metoda ma za zadanie odczytać i-węzeł z systemu plików. Dostaje ona od VFS numer i-węzła (w strukturze `inode` z pliku *include/linux/fs.h*), odczytuje odpowiedni blok z dysku, po czym wypełnia resztę struktury `inode`.
- `put_super()`: metoda jest wywoływana przy odmontowaniu systemu plików. Tutaj powinno się zwolnić pamięć przydzieloną na bloki i struktury w `read_super()`
- `statfs()` metoda wywoływana jest dla uzyskania przez użytkownika różnych informacji dotyczących systemu plików (typ systemu plików, rozmiar bloku, liczba bloków itp.)

Reszcie wskaźników możemy nie przypisywać wartości - (domyślnie inicjalizowane są na wartość `NULL`), co oznacza dla jądra, że nie zaimplementowano danej metody. Jeśli VFS posiada dla danej metody zdefiniowaną metodę ogólną, wtedy właśnie ona zostanie użyta.

W Linuksie 2.4 wprowadzono mechanizm *pagecache*. Jest to mechanizm buforowania fizycznych stron pamięci. Był on już stosowany w systemie UNIX SVR4. Zastępuje on stosowany w Linuksie 2.2 mechanizm *buffercache*, który był zorientowany na buforowanie pojedynczych bloków systemu plików, niezależnie od przydzielania pojedynczych stron pamięci fizycznej. Różnica między SVR4 a Linuksem 2.4 polega na tym, że w tym pierwszym *pagecache* był stosowany tylko do systemów plików, a w Linuksie jest on bardziej

ogólny (jest także wykorzystywany przez VFS do współpracy z podsystemem pamięci wirtualnej w celu mapowania plików do przestrzeni użytkownika, do obsługi plików wymiany, wykorzystują go też niektóre drivery, np. *loopback*). Dlatego zamiast struktury *inode*, używa on struktury *address\_space* (*include/linux/fs.h*) zdefiniowanej następująco:

```
struct address_space {
    struct list_head          pages;
    unsigned long             nrpages;
    struct address_space_operations * a_ops;
    void *                    host;
    struct vm_area_struct *   i_mmap;
    struct vm_area_struct *   i_mmap_shared;
    spinlock_t                i_shared_lock;
};
```

Dla implementacji systemu plików wykorzystującego *pagecache* ważna jest struktura *address\_space\_operations* (*include/linux/fs.h*), która zdefiniowana została następująco:

```
struct address_space_operations {
    int (*writepage)(struct page *);
    int (*readpage)(struct file *, struct page *);
    int (*sync_page)(struct page *);
    int (*prepare_write)(struct file *,
        struct page *, unsigned, unsigned);
    int (*commit_write)(struct file *,
        struct page *, unsigned, unsigned);
    int (*bmap)(struct address_space *, long);
};
```

W przypadku implementacji tylko do odczytu stworzyć należy tylko funkcje *readpage* i *bmap* (tą drugą gdy chcemy umożliwić procesom mapowanie plików do ich wirtualnej pamięci). Resztę, tak jak w przypadku struktury *super\_operations*, należy pominąć. Wskaźnik do struktury *address\_space* jest osiągalny poprzez strukturę *inode* (*include/linux/fs.h*), i właśnie w ten sposób dokonywane są operacje na i-węzłach.

VFS posiada zdefiniowane standardowe funkcje obsługi niektórych metod, które sprawdzają się w wielu systemach plików. W większości wystarczy zaimplementować funkcję typu *get\_block\_t* (*include/linux/fs.h*). Funkcja ta dostaje wskaźnik do struktury i-węzła VFS, numer bloku i wskaźnik do bufora, do którego ma za zadanie wprowadzić blok o podanym numerze, należący

do tego i-węzła. Mając tę funkcję, VFS może już, za pomocą standardowo zdefiniowanych metod, zażądać odczytania odpowiedniej porcji danych wskazywanych przez dany i-węzeł. W pliku *include/linux/fs.h* znajduje się struktura *file\_operations* (*include/linux/fs.h*):

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *,
                 unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned
                     long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *,
                     unsigned long, loff_t *);
};
```

Struktura *file\_operations* to część struktury *file* (*include/linux/fs.h*), która jest otwartą instancją i-węzła (jest ona i-węzłem z punktu widzenia procesu użytkownika). W przypadku implementacji tylko do odczytu, trzeba zdefiniować metody *read* i *mmap*. Jeśli zdefiniowana była funkcja typu *get\_block\_t*, można tym wskaźnikom przypisać standardowo zdefiniowane w VFS funkcje *generic\_file\_read()* i *generic\_file\_mmap()*. Jest to duża różnica w porównaniu do jąder z serii 2.2, gdzie trzeba było definiować wszystkie metody z osobna (a jeśli implementuje się również zapis, funkcji tych jest znacznie więcej).

Wskaźnik do struktury *inode\_operations* jest częścią struktury *inode*.

```
struct inode_operations {
    int (*create) (struct inode *, struct dentry *, int);
```

```
struct dentry * (*lookup) (struct inode *,struct dentry *);
int (*link) (struct dentry *,struct inode *,struct dentry *);
int (*unlink) (struct inode *,struct dentry *);
int (*symlink) (struct inode *,struct dentry *,const char *);
int (*mkdir) (struct inode *,struct dentry *,int);
int (*rmdir) (struct inode *,struct dentry *);
int (*mknod) (struct inode *,struct dentry *,int,int);
int (*rename) (struct inode *, struct dentry *,
               struct inode *, struct dentry *);
int (*readlink) (struct dentry *, char *,int);
int (*follow_link) (struct dentry *, struct nameidata *);
void (*truncate) (struct inode *);
int (*permission) (struct inode *, int);
int (*revalidate) (struct dentry *);
int (*setattr) (struct dentry *, struct iattr *);
int (*getattr) (struct dentry *, struct iattr *);
int (*attrctl) (struct inode *, struct attr_op *, int);
int (*acl_get) (struct dentry *, struct acl *, struct acl *);
int (*acl_set) (struct dentry *, struct acl *, struct acl *);
};
```

Zdefiniować należy osobną strukturę `inode_operations` dla każdego typu i-węzła (katalog, dowiązanie symboliczne, urządzenie itp.). W większości przypadków nie ma konieczności definiowania `inode_operations` dla zwykłego pliku (wszystko obsługuje standardowymi funkcjami VFS, tak jak to było opisane wyżej). VFS ma też zdefiniowane standardowe funkcje obsługi dowiązań symbolicznych.

Konieczne jest zdefiniowanie `inode_operations` dla katalogu. W przypadku trybu tylko do odczytu interesująca jest tylko metoda `lookup()`. Metoda ma zwrócić kolejną pozycję w katalogu (wskazywaną przez podaną jako parametr strukturę `inode` jako pozycja w pliku). Zwracana jest ona w postaci struktury `dentry` (*include/linux/dcache.h*). W strukturze tej mamy m.in. numer i-węzła, który jej odpowiada. Może to być na przykład kolejny katalog. W ten sposób umożliwia się przeglądanie pozycji w katalogach oraz przemieszczanie się po drzewie katalogów.

# Załącznik 3

## Instrukcja kompilacji, uruchomienia i użytkowania

### Kod w trybie użytkownika (*htfs\_linux*)

Pakiet *htfs\_linux* znajduje się w pliku *htfs\_linux.tar.gz*. Zacząć należy od rozpakowania tego pliku komendą:

```
$ tar xzf htfs_linux.tgz
```

Kompilacja pakietu wymaga w systemie obecności kompilatora języka C, narzędzia *make* oraz biblioteki *libc* lub zgodnej. *make* oczekuje, że kompilator będzie dostępny przy wywołaniu jako *gcc*, ale można to zmienić w pliku *Makefile*. W celu skompilowania pakietu należy wydać polecenie:

```
$ cd htfs_linux  
$ make
```

Po poprawnej kompilacji i konsolidacji w bieżącym katalogu powinny zostać utworzone pliki *hdinfo\_sco*, *readsuper\_sco*, *sco\_ls* i *sco\_cat*. Zakładając, że dysk (lub partycja) jest podzielony dodatkowo programem *divvy* (patrz punkt 6.1), wywołuje się polecenie:<sup>3</sup>:

```
$ ./hdinfo_sco urządzenie
```

---

<sup>3</sup>Jeśli HTFS nie jest umieszczony wewnątrz partycji *divvy*, pomija się wywołanie *hdinfo\_sco* i parametr *sektor* w wywołaniu *readsuper\_sco*

gdzie *urządzenie* to dysk podzielony programem *divvy*.<sup>4</sup> Poniżej przedstawiono przykład dla testowego urządzenia - dysku 4,1 GB ze standardową instalacją SCO Open Server 5 (w trakcie instalacji wybrano opcję *Whole disk for Unix*, co oznacza, że podział jest dokonany jedynie za pomocą programu *divvy*, z pominięciem wcześniejszego podziału programem *fdisk* na partycje *dosowe*). Na testowej maszynie urządzenie było identyfikowane jako */dev/hdc/*:

```
$ ./hdinfo_sco /dev/hdc
Partition info for /dev/hdc
=====
Sector=63          Offset=32256          Bytes=15047168      Type=0x63
{invalid SCO badtrk table (4321:668) @63+44b}
{SCO divvy/badtrk tables @63+42b/63+44b}
SCO behaviour when heads and sectors are both odd is unknown
doubling cylinder offset to realign to 1Kbyte boundary.
this *might* work, please mail siwne@iinet.net.au either way...
SCO divvy offset (HDIO_GETGEO) 2*945 = 1890b
SCO /dev/hdc/1 <Sector=1890      Offset=967680      Bytes=15728640      Type=0x63>
SCO /dev/hdc/2 <Sector=32610     Offset=16696320    Bytes=249561088     Type=0x63>
SCO /dev/hdc/3 <Sector=520034    Offset=266257408   Bytes=-274896896    Type=0x63>
SCO /dev/hdc/d <Sector=8371734    Offset=-8639488    Bytes=10240         Type=0x63>
SCO /dev/hdc/c <Sector=1890      Offset=967680      Bytes=15046656      Type=0x63>
```

Na wyjściu dostaje się wszystkie partycje *divvy*, istniejące na danym urządzeniu, razem z ich prawdopodobnymi<sup>5</sup> początkami w 512-bajtowych sektorach. Partycja HTFS to zazwyczaj partycja numer 3 (oznaczona jako */dev/hdc/3* na powyższym wydruku) Następnie wywołuje się:

```
$ ./readsuper_sco urządzenie sektor
```

gdzie *sektor* bierze się z wyniku zwróconego przez *hdinfo\_sco*. Polecenie to wyszuka dokładny początek partycji HTFS (patrz punkt 6.3). W testowym przykładzie należało wywołać:

```
$ ./readsuper_sco /dev/hdc 520034
Offset: 520034
```

<sup>4</sup>Pamiętać należy o prawie odczytu do danego urządzenia

<sup>5</sup>Niestety podział na partycje *divvy* nie został dostatecznie zbadany i wciąż nie wiadomo, jak dokładnie zachowuje się program *divvy* przy różnych parametrach CHS dysku twardego.

```
Actually i don't know how exactly divvy places the filesystems
inside it's slice, so we are going to check some blocks forward...
.....
```

```
Okay, magic signature found. Filesystem starts at block 566339,
displaying some informations.
```

```
Dumping start block to sco_fsstart.blk.
```

```
Version: 1
```

```
Type: 2 (1024 byte blocks)
```

```
Name:
```

```
Inode table size (in blocks): 122686
```

```
Size (in blocks): 3925850
```

```
Number of inode groups: 1
```

```
Number of fs blocks occupied by inodes: 122683
```

```
Summary block number: 122685
```

Dostajemy tu informację, że HTFS zaczyna się od bloku 566339. Ta informacja będzie bardzo ważna, gdy konieczne jest zamontowanie systemu plików HTFS poprzez urządzenie *loopback* z parametrem *offset* (patrz punkt 7) W bieżącym katalogu zostanie utworzony plik *sco\_fsstart.blk*, który zawiera nazwę urządzenia i wspomniany początek partycji HTFS. Jeśli *readsuper\_sco* nie znajdzie początku HTFS na danej partycji, należy spróbować z inną.

Teraz można już korzystać z poleceń *sco\_ls* i *sco\_cat*. Sposób skorzystania z tych narzędzi jest następujący:

```
$ ./sco_ls [-l] katalog
```

Spowoduje to wyświetlenie zawartości katalogu *katalog*. Opcjonalny parametr *-l* powoduje wyświetlenie pełnej informacji dotyczącej pozycji w katalogu.

```
$ ./sco_cat plik
```

To polecenie spowoduje wyświetlenie na standardowym wyjściu zawartości pliku *plik*.

Ewentualne usterki należy zgłaszać na adres [deresz@truml.art.pl](mailto:deresz@truml.art.pl)

## Moduł do jądra Linuksa (*htfs\_fs*)

Moduł do jądra znajduje się w pliku *htfs\_fs.tar.gz*. Jest on aktualnie przeznaczony dla architektury 32-bitowej x86 (*ia32* - procesory firmy Intel i zgodne).

Plik rozpakowuje się poleceniem:

```
$ tar xjf htfs_fs-wersja.tar.bz2
```

Następnie należy wybrać wersję jądra, dla której ma się skompilować pakiet. W tym celu wykonuje się polecenie

```
$ cd wersja_jadra
```

gdzie *wersja\_jadra* to *2.4* lub *2.2*. Następnie można przystąpić do kompilacji, którą rozpoczyna się poleceniem:

```
$ cd htfs_fs-wersja  
$ make
```

Jeśli kompilacja przebiegła poprawnie, utworzony zostanie plik *htfs.o*, który jest modulem do jądra. Moduł ten należy zainstalować do katalogu */lib/modules*. W tym celu należy uzyskać uprawnienia użytkownika *root*, następnie polecenie:

```
# make install
```

zainstaluje moduł we właściwe miejsce.

W przypadku, gdy mamy system plików HTFS bez podziału na partycje *divvy*, wystarczy wywołać:

```
# mount -t htfs urządzenie katalog
```

co spowoduje zamontowanie urządzenia *urządzenie* z systemem plików HTFS pod katalog *katalog*.

Gdy urządzenie jest podzielone programem *divvy*, również istnieje możliwość zamontowania HTFS. W tym celu należy wkompiłować w jądro obsługę urządzenia *loopback* (lub też skompilować ją do modułu). Następnie należy uzyskać pozycję w blokach, od której zaczyna się HTFS na danym urządzeniu (opisano to w punkcie 7). Teraz wywołuje się komendę:

```
# mount -t htfs urządzenie katalog -o loop,offset=pozycja,blocksize=512
```

Gdzie *pozycja* to pozycja w blokach 512-bajtowych, od których zaczyna się HTFS na urządzeniu *urządzenie*. W przykładzie z punktu 7 pozycja przyjąłaby wartość *566339*.

W obydwu przypadkach w katalogu *katalog* powinno się otrzymać drzewo katalogów systemu plików HTFS, po którym można się poruszać i odczytywać zawarte tam dane.

Ewentualne usterki należy zgłaszać na adres *deresz@truml.art.pl*

# Załącznik 4

## Spis literatury

- [1] Tigran Aivazian: *Linux Kernel 2.4 Internals*,  
<http://www.linuxdoc.org/LDP/lki>
- [2] David A. Rustling: *The Linux Kernel*, <http://www.linuxdoc.org/LDP/tlk>
- [3] Michael K. Johnson: *Kernel Hacker's Guide*,  
<http://www.linuxdoc.org/LDP/khg>
- [4] Alessandro Rubini: *Linux Device Drivers, 2nd Edition*, O'Reilly (2001),  
<http://www.xml.com/ldd/chapter/book/>
- [5] M. Beck i inni: *Linux Kernel Internals*, Personal Education Deutschland GmbH (1998), wydanie polskie: *Linux Kernel - Jądro Systemu*, MIKOM (1999)
- [6] Jerzy Marczyński: *UNIX użytkowanie i administrowanie*, Helion (1995)