# Uniksowe systemy plików

Magdalena Borkowska
Michał Ejdys
Paweł Hryczuk
Małgorzata Jankowska

19 stycznia 2004

# Spis treści

# Część I

# Wstęp

# Rozdział 1

# Czym jest system plików

**System plików** to sposób organizacji danych na nośniku pamięci trwałej.

Pierwszym i najważniejszym zadaniem systemu plików jest rozmieszczenie danych, umieszczenie ich w jakiejś strukturze danych.

Drugie zadanie to utrzymanie informacji o danych, takich jak ich rozmiar, czas utworzenia lub modyfikacji itd.

Często system plików dba też o prawa dostępu do danych.

# Rozdział 2

# Dziennikowanie

W momencie montowania systemu plików zerowany jest tzw. „clean bit", co oznacza że system plików jest w użyciu. Przy wyłączaniu komputera narzędzia odmontowujące system plików czekają wszystkie dane będące w trakcie zapisywania oraz znajdujące się w cache'u zostaną zapisane na dysk. Gdy to się stanie, odmontowują system ustawiając „clean bit". Jeżeli jednak zdarzy się awaria systemu np:

- banalna przerwa w dostawie prądu
- wyjątkowo poważny błąd systemowy
- królik przegryzie kabel

na dysku mogą pojawić się niespójności. Nie zostanie również zapisany „clean bit", co spowoduje, że przy ponownym uruchamianiu komputera system uruchomi program fsck, by sprawdzić spójność danych na dysku.

A takie sprawdzanie może długo potrwać, o czym świetnie wiedzą wierni użytkownicy systemu ext2, który dziennikowania nie posiada. Żeby było ciekawiej, długość naprawy dysku rośnie proporcjonalnie do jego wielkości. Ponadto program fsck może sobie zwyczajnie nie poradzić z naprawą błędów logicznych na dysku.

Dlatego w niektórych systemach plików, np. ReiserFS, ext3, jfs, xfs stosowane jest **dziennikowanie (księgowanie)**, *z ang. journaling*.

System plików z dziennikowaniem przechowuje przebieg zmian, które mają zostać wykonane na plikach na dysku, w specjalnym dzienniku. Zapisywane zmiany są powiązane w transakcje. Jeżeli uda się zapisać do dziennika listę operacji do wykonania na dysku, system plików zapisuje zmiany na dysku. Jeżeli uda się wykonać wszystkie zmiany, to lista operacji do wykonania jest usuwana z dziennika.

Jeśli zdarzy się awaria komputera i nie wszystkie operacje wyszczególnione w dzienniku (z transakcji) zostaną wykonane, to na dysku będą występowały niespójności. Jednakże podczas montowania systemu plików zostanie sprawdzone, czy w dzienniku nie ma transakcji będących w stadium wykonywania. Jeżeli są, to zostaną wykonane.

# Część II

# VFS

# Rozdział 3

# Wprowadzenie

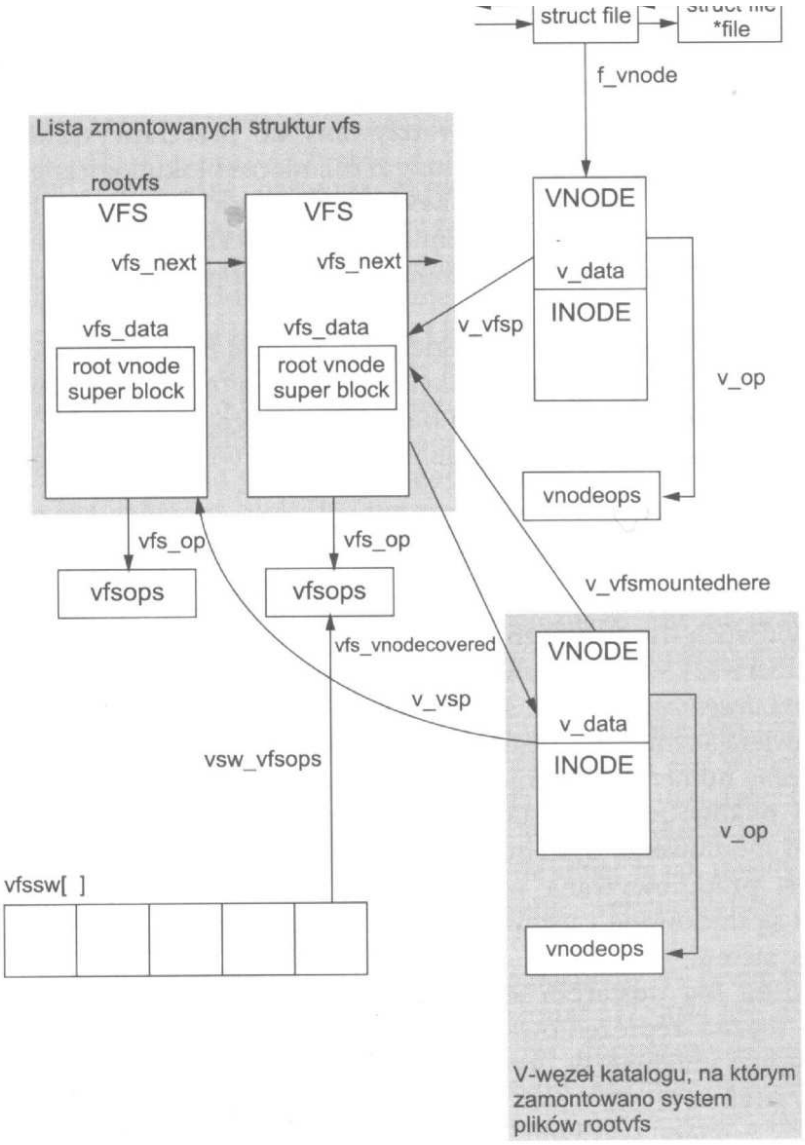
System Unix SVR3 wykorzystywał mechanizm zwany *tablicą rozdzielczą plików* (FSS - *File System Switch*). Polegał on na podziale i-węzła na dwie części: zależną i niezależną od konkretnego systemu plików. Dzięki temu jądro systemu mogło być niezależne od konkretnych implementacji systemów plików. Pomysł polegał na przechowywaniu w strukturze i-węzła operacji konkretnego systemu plików, które mogły być zaimplementowane gdzie indziej, a tutaj przezroczyście wywoływane.

Firma Sun Microsystems opracowała inne rozwiązanie, nazwane *tablicą rozdzielczą VFS* (VFS - *Virtual File System*), które zostało zaimplementowane w systemie SunOS.

Używany dzisiaj VFS powstał jako połączenie dwóch wyżej opisanych w systemie Unix SVR4 i jest niezmieniany od ponad 15 lat. Najważniejszą nowością w stosunku do Sun VFS było zdecydowane rozgraniczenie podsystemu zarządzania plikami od podsystemu zarządzania pamięcią. Dało to bardzo eleganckie rozwiązanie, jednak skomplikowało implementację nowych systemów plików.

# Rozdział 4

# Struktury VFS



## 4.1 Tablica rozdzielcza wirtualnego systemu plików

Tablica rozdzielcza wirtualnego systemu plików (VFSSW - *Virtual Filesystem Switch Table*) jest strukturą jądra systemu, która przechowuje po jednym wpisie dla każdego obsługiwanego rodzaju systemu plików.

| **Element** | **Opis** |
|---|---|
| *char *vsw_name* | napis zawierający nazwę typu systemu plików |
| *int (*vsw_init)()* | wskaźnik do funkcji inicjowania systemu plików |
| *struct vfsops *vsw_vfsops* | wskaźnik do tablicy operacji zależnych od typu systemu plików |

Element *vsw_name* to ten np. przekazywany do mount w opcji *-F*, a *vsw_init* jest wykonywane podczas inicjalizacji systemu operacyjnego. Element *vfsops* został opisany poniżej.

## 4.2 Operacje zależne od typu systemu plików

Operacje zależne od typu systemu plików (struktura *vfsops*):

| **Element** | **Związane makro** | **Opis operacji** |
|---|---|---|
| *int (vfs_mount)()* | *VFS_MOUNT* | zamontowanie systemu plików |
| *int (vfs_unmount)()* | *VFS_UNMOUNT* | odmontowanie systemu plików |
| *int (vfs_root)()* | *VFS_ROOT* | odczytanie v-węzła dla korzenia systemu plików |
| *int (vfs_statvfs)()* | *VFS_STATVFS* | odczytanie informacji statycznych o systemie plików |
| *int (vfs_sync)()* | *VFS_SYNC* | zapisanie zawartości buforów dyskowych na dysku |
| *int (vfs_vget)()* | *VFS_VGET* | znalezienie v-węzła odpowiadającego identyfikatorowi pliku |
| *int (vfs_mountroot)()* | *VFS_MOUNTROOT* | zamontowanie głównego systemu plików |

Pierwszym argumentem tych makr jest wskaźnik do struktury *vfsops* dla systemu plików, którego dotyczy wywołanie. Dzięki temu makra te zapewniają niezależny od systemu plików sposób wywoływania na nim operacji.

Jądro systemu nie zakłada nic na temat operacji wykonywanych przez konkretne funkcje realizujące powyższe operacje. Jednak zanim zostanie wykonana dostarczona przez implementację systemu plików funkcja, jądro wykonuje pewne czynności (np. przy *vfs_unmount* jest to usunięcie z pamięci podręcznej nazw plików wszystkich odwołań do nazw zawartych w odmontowywanym systemie plików).

## 4.3 Informacje o systemie plików

Struktura *ustat*, przechowywana w przestrzeni adresowej użytkownika:

| **Element** | **Opis** |
|---|---|
| *daddr_t f_tfree* | łączna liczba wolnych bloków |
| *o_ino_t f_tinode* | łączna liczba wolnych i-węzłów |
| *char f_fname[6]* | napis zawierający nazwę systemu plików |
| *char f_fpack[6]* | napis zawierający nazwę woluminu systemu plików |

Struktura *statvfs* opisuje superblok zamontowanego systemu plików i nie zależy od jego typu:

| **Element** | **Opis** |
|---|---|
| *u_long f_bsize* | rozmiar bloku systemu plików |
| *u_long f_frsize* | rozmiar fragmentów (jeśli są obsługiwane) |
| *u_long f_blocks* | łączna liczba bloków w systemie plików (jednostką jest *f_frsize*) |
| *u_long f_bfree* | łączna liczba wszystkich wolnych bloków w systemie plików (jednostką jest *f_frsize*) |
| *u_long f_bavail* | liczba wolnych w systemie plików bloków dostępnych dla zwykłego użytkownika (jednostką jest *f_frsize*) |
| *u_long f_files* | łączna liczba i-węzłów |
| *u_long f_ffree* | łączna liczba wolnych i-węzłów |
| *u_long f_favail* | liczba wolnych i-węzłów dostępnych dla zwykłego użytkownika |
| *u_long f_fsid* | identyfikator systemu plików (obecnie numer urządzenia) |
| *char f_basetype[FSTYPSZ]* | napis zawierający nazwę typu systemu plików |
| *u_long f_flag* | znaczniki: *ST_READONLY*, *ST_NOSUID*, *ST_NOTRUNC* |
| *u_long f_namemax* | maksymalna długość nazwy pliku |
| *char f_fstr[32]* | napis specyficzny dla systemu plików |

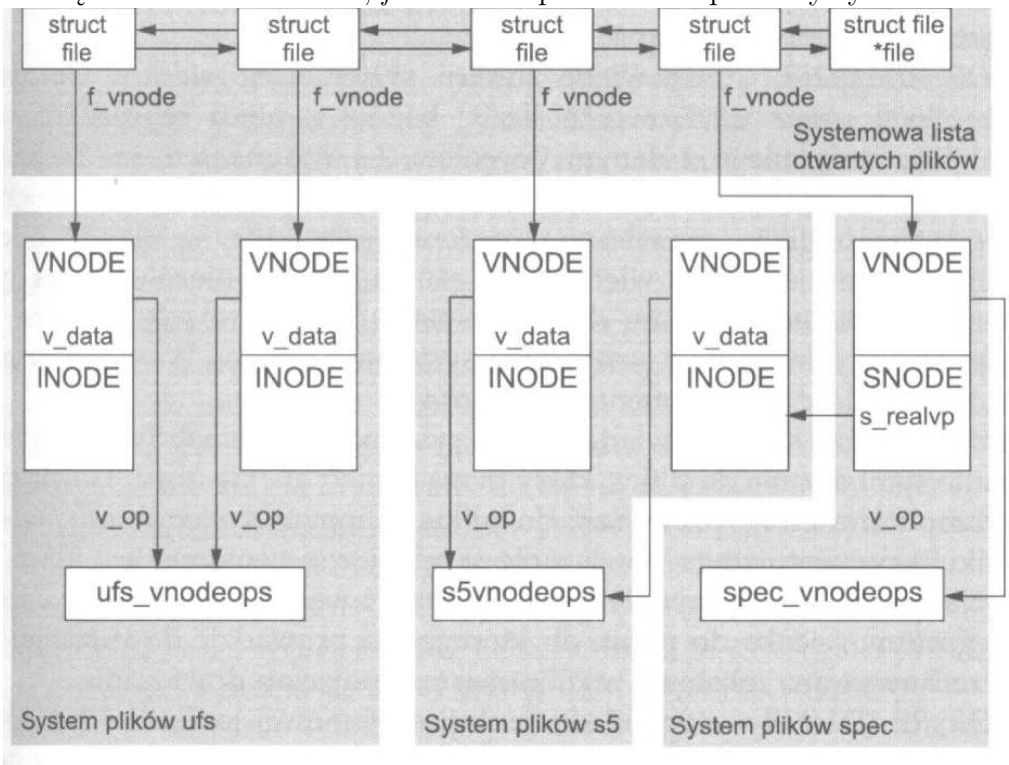
# Rozdział 5

# V-węzły

Każdemu otwartemu plikowi odpowiada w wirtualnym systemie plików v-węzeł, który zawiera wskaźnik na i-węzeł tego pliku. V-węzeł jest zapisywany przez kod zależny od rodzaju systemu plików. Wszystkie operacje na pliku są wywoływane przez jego v-węzeł.

Struktura *vnode* zawiera wszystkie informacje dotyczące zarządzania plikiem, ponieważ tylko do nich może się odwoływć jądro systemu. Ta struktura zawiera więc wskaźnik na strukturę *vnodeops*, która pamięta adresy funkcji realizujących operacje na pliku (za pośrednictwem makr, analogicznie do *vfsops*).

Powiązanie struktur *vnode*, *file* i *inode* przedstawia poniższy rysunek:



# Rozdział 6

# Odwzorowywanie ścieżek

System odwzorowuje ścieżki, aby uzyskiwać odpowiadające im v-węzły. Ścieżka jest dzielona na nazwy kolejnych katalogów, oddzielone znakiem / i przetwarzana kolejno dla każdego z nich. Tak podzielona ścieżka jest pamiętana w strukturze *pathname*.

Funkcja *lookupname()* tworzy i wypełnia strukturę *pathname*, a następnie przekazuje ją funkcji *lookuppn()*, która ją przetwarza. Oto schemat algorytmu funkcji *lookuppn()*:

```
zacznij poszukiwanie od bieżącego katalogu

if (pierwszym znakiem ścieżki jest '/') {
   usuń ukośnik ze ścieżki
   zacznij przeszukiwanie od katalogu głównego
}

loop {
   if (długość ścieżki wynosi zero) {
     wybierz bieżący katalog
     return success
   }

   if (VOP_LOOKUP() == error)
     return error

   while (v_vfsmountedhere != NULL)
     przejdź do korzenia tego systemu plików
   if (ten człon jest dowiązaniem symbolicznym) {
     utwórz nową strukturę pathname
     odczytaj dane z tego dowiązania i zapisz je w pathname
     next loop
   }
}
```

# Rozdział 7

# VFS a konkretny system plików

Jądro systemu operacyjnego jest niezależne od konstrukcji konkretnych systemów plików. Zatem chcąc wykonywać operacje manipulujące systemami plików musimy odwoływać się do specyficznych narzędzi, dostosowanych do konkretnych ich typów. Tak jest na przykład z poleceniem *mkfs*, które - w zależności od podanego parametru - wykonuje *mkfs.ext2*, *mkfs.minix*, *mkfs.reiserfs*, itd...

Z drugiej strony, niezależność ta ułatwia implementację innych operacji. Na przykład polecenie *df* wywołuje funkcję systemową *statvfs()*, która jest w jądrze zaimplementowana jako wywołanie *vfs_statvfs* struktury *vfsops*. Zatem z punktu widzenia użytkownika jest w tym przypadku nieistotne, jaki konkretny system plików jest używany.

# Rozdział 8

# Uwaga dotycząca vfs_sync

Operacja *VFS_SYNC* odpowiada funkcji systemowej *sync*, a także jest wywoływana przez demona *fsflush*. Przy wywołaniu przekazywany jest znacznik, który informuje, czy ma zostać wykonana częściowa, czy pełna operacja zapisania danych na dysku. Częściowa to zapisanie tylko znajdujących się w pamięci i-węzłów, a pełna - superbloku i buforów zawartości plików. Funkcja systemowa *sync* zleca pełną, a demon *fsflush* częściową operację zapisania danych na dysku.

Demon *fsflush* jest procesem systemowym, który, budzony co sekundę przez funkcję *clock()*, ma zapewnić jakąś formę integralności danych w podsystemie plikowym systemu operacyjnego.

# Część III

# Lokalne systemy plików

# Rozdział 9

# Wprowadzenie

Ten rozdział opisuje po krótce niektóre elementy implementacji różnych systemów plików w Uniksie. Omówione zostaną następujące systemy:

- system plików VERITAS, inaczej VxFS - zaimportowany do wielu wersji Unixa, jeden z najbardziej udanych z komercyjnie dostępnych systemów plików,
- UFS - zaprezentowany pierwszy raz w BSD UNIX jako *Fast File System*, dostępny w wielu wersjach Unixa,
- ext2 z jego następcą ext3 - wraz z rozwojem systemu Linux coraz bardziej popularny i dobrze udokumentowany,
- XFS - jeden z najbardziej wydajnych systemów plików w Unixie.

Bardziej szczegółowo został omówiony ReiserFS w wersji 4.0, która niedługo się ukaże.

# Rozdział 10

# System plików UFS

Rozdział ten opisuje system plików UFS, wcześniej znany jako *Berkley Fast File System* (FFS), mający swe korzenie w technologii BSD, chyba jeden z najbardziej opisanych systemów plików Unixa. Pierwszy raz opisany w 1984.

## 10.1 Pomysł

System plików FFS to pierwszy wydajny system plików korzystający z fizycznego ułożenia danych na dysku.

## 10.2 Dane na dysku

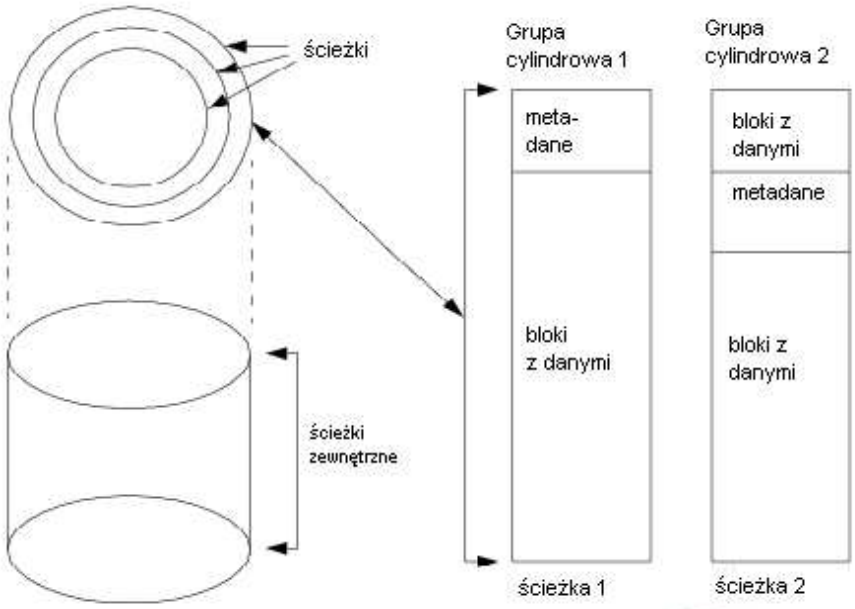
System plików został podzielony na *grupy cylindrowe*, które odpowiadają bezpośrednio cylindrycznemu układowi danych na dysku z tamtych czasów (wcześniej każdy cylinder na dysku miał taki sam rozmiar bez względu na to czy leżał blisko środka czy krawędzi dysku). Każda grupa cylindrowa zawiera kopię superbloku, stałą liczbę i-węzłów, bitmapy opisujące wolne i-węzły i bloki z danymi, zużycie bloków z danymi i same bloki z danymi. Każda taka grupa ma stałą liczbę i-węzłów, kalkulowaną tak, aby dla każdych 2048 bajtów istniał jeden i-węzeł. Osądzono, że to powinno dostarczać więcej plików niż będzie potrzebnych.

## 10.3 Księgowanie

Mechanizm księgowania jest dostepny dopiero w najnowszych wersjach UFS, opracowanych przez firmę Sun.

## 10.4 Ciekawostki

- Aby zmniejszyć problem marnotrawstwa miejsca, gdy na przykład plik jest tylko troszkę większy od rozmiaru bloku, nowy system plików wprowadził pojęcie *frag-*




*mentów*. Bloki mogą być podzielone na 2, 4 lub 8 fragmentów, których rozmiar ustalany jest w czasie tworzenia systemu.

- Dziś UFS, przystosowany do dzisiejszej technologii budowy dysków i dostosowany do potrzeb (m.in. księgowanie), dostępny jest na Solaris.

# Rozdział 11

# VERITAS

Rozwój systemu plików VERITAS rozpoczął się w latach 90-tych razem z pierwszą implementacją SVR4.0. W ciągu ostatniej dekady VxFS rozrósł się na tyle, żeby zająć miejsce dla najbardziej udanego komercyjnego uniksowego systemu plików. VxFS pisany był, aby bezpośrednio wspierać systemy: Solaris, HP-UX, AIX i Linuxa.

## 11.1 Pomysł

VxFS stosuje **ekstenty** (ang. extent). Nie bacząc na rozmiary bloków wybrane dla VxFS, które mogą mieć 1KB, 2KB, 4KB, 8KB, dane mogą być alokowane do dużych przyległych bloków zwanych ekstentami. Minimalny rozmiar ekstentu jest taki jak rozmiar bloku systemu plików, a maksymalny to maksymalny rozmiar pliku.

Domyślny algorytm używany do alokowanie ekstentów dla pliku jest bazowany na systemie Wejścia/Wyjścia. Na przykład, jeśli stworzymy plik, i będziemy sekwencyjnie pisać do pliku, to pierwszy przydzielony ekstent będzie 2 razy większy od rozmiaru pisania. Ekstent przydzielony po pierwszym pisaniu wzrasta wraz z kontynuowanym zapisem. Przez przydzielanie coraz to większych kawałków próbuje się zapewnić przyległe położenie jak największej liczby bloków. Jeśli plik jest zamykany, a ostatni zapis nie zajął całego ekstentu, ekstent jest zmniejszany i pozostałe miejsce jest zwracane do puli wolnej przestrzeni.

## 11.2 VxFS na dysku

Geometria danych na dysku rozwija się, aby wesprzeć potrzebę zwiększenia rozmiaru pliku i systemu plików. Istnieje 5 różnych wersji ułożenia systemu plików. Pierwsza wersja jest podobna do UFS, podczas gdy pozostałe wersje są istotnie różne.

Nowsze sposoby ułożenia danych na dysku rozwiązują problem stałej liczby i-węzłów i skracają czas zakładania systemu plików poprzez usytuowanie wszystkich metadanych systemu w plikach, które mogą rosnąć na żądanie.

Nowe wersje przedstawiają także pomysł *zbiorów plików*. Jest on wzięty z DCE DFS (Distributed File System), gdzie każdy *zbiór plików* wygląda dla użytkownika jak system

plików - ma i-węzeł dla roota, katalog lost+found, i hierarchię katalogów i plików, taką jak inne systemy plików. Każdy ze zbiorów jest niezależnie montowalny.

W czasie tworzenia VxFS, tworzone są dwa *zbiory plików*:

- główny zbiór (ang. *primary fileset*) - montowany, gdy wywołana jest komenda *mount*,
- zbiór strukturalny (ang. *structural fileset*) - zawierający wszystkie metadane systemu plików.

Każdy zbiór ma własną listę i-węzłów przechowywaną jako plik.
Zbiór główny zawiera wszystkie katalogi, pliki regularne i inne należące do użytkownika.
Zbiór strukturalny zawiera:

- Tablica lokacji (*Object location table* - OTL ) - wskazywana przez superblok, używana przy montowaniu, zawiera wskaźniki na struktury potrzebne do zamontowania systemu plików.
- Etykieta - zawiera superblok i jego kopie.
- Plik nagłówkowy zbioru plików - zbiór plików jest opisywany przez wejście w tym pliku, każde wejście zawiera informacje takie jak liczba i-węzłów alokowanych w zbiorze, numer i-węzła pliku zawierającego *Listę i-węzłów* i innych związanych i-węzłów.
- Lista i-węzłów - jeden plik na cały zbiór, zawiera wszystkie i-węzły alokowane w systemie,
- Jednostka alokacji i-węzłów (z ang. Inode Alocation Unit File IAU) - znowu jeden na cały zbiór, używany do zarządzania i-węzłami, zawiera mapę bitową wolnych i-węzłów, podsumowujące informacje, i rozległe informacje o operacjach.
- Dziennik.
- Pliki zawierające informacje o ekstentach.

## 11.3 Kronikowanie w VxFS

VxFS rozwiązuje problem osieroconych i zgubionych plików wprowadzając transakcyjność zmian w systemie plików. Transakcja to rekord jednej lub więcej zmian w systemie. Zmiany są najpierw pisane do *Dziennika*, (cyklicznego bufora zlokalizowanego wewnątrz systemu plików), ustawiane są wskaźniki na początek i koniec transakcji. W przypadku awarii systemu zawartość dziennika jest odtwarzana. Rekordy z dziennika mogą być odtwarzane wielokrotnie z takim samym skutkiem. To zapewnia, że dziennik może być odtworzony nawet w przypadku gdy nastąpi awaria w trakcie odtwarzania dziennika.

## 11.4 Ciekawostki

- VxFS wspiera wymuszone odmontowanie systemu plików, nawet wtedy gdy system jest zajęty.
- Administracja Online. VxFS dostarcza mechanizm pozwalający na podejmowanie operacji administracyjnych na systemie plików, takich jak zmiana rozmiaru systemu plików, w czasie gdy jest on jeszcze zamontowany i aktywny.
- Ze względu na zlokalizowanie informacji o systemie plików w plikach, zminimalizowana jest ilość informacji alokowanych przy inicjowaniu struktury. Na przykład, tylko 32 i-węzłów jest alokowanych przy tworzeniu systemu plików. Aby zwiększyć liczbę i-węzłów, plik z listą i-węzłów jest po prostu rozszerzany.

# Rozdział 12

# Ext2 i Ext3

Pierwszym systemem plików dostarczonym wraz z dystrybucją Linuxa był Minix. Minix przechowywał adresy bloków w 16-bitowych integerach, co ograniczało rozmiar systemu plików do 64MB. Stałe były też wejścia do katalogów, co limitowało rozmiar nazw plików do 14 znaków. W 1992 Minix został zastąpiony przez ext, który wspierał systemy plików do 2GB i nazwy plików do 255 znaków. I-węzły w ext nie miały bezpośredniego dostępu, wprowadzania zmian ani stempli czasowych, a do zarządzania wolnymi blokami i i-węzłami były używane listy. To powodowało fragmentację i niską wydajność.

Te błędy poprawiono tworząc dwa systemy plików *Xia* i *ext2* (modelowany na bazie BSD FFS). Postępy w ext2 przekroczyły te z Xia i to właśnie ext2 stał się defacto standardem Linuxa.

## 12.1 Pomysł

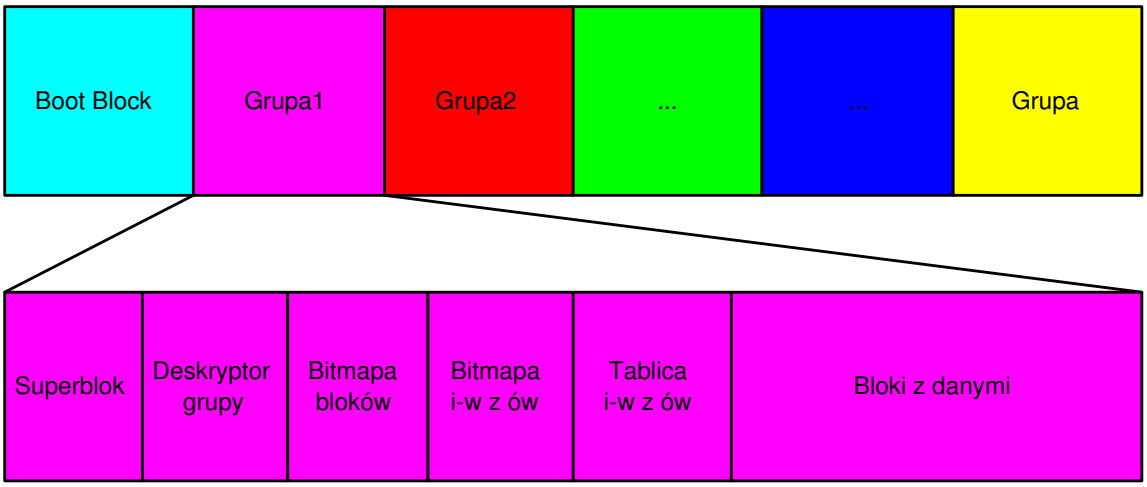
Ext2 ma jedną, bardzo ważną wadę: długi czas, jaki potrzebny jest do sprawdzenia spójności danych systemu po awarii systemu. Jeśli system plików jest duży, czas sprawdzenia systemu ext2 może zająć kilka godzin. Aby rozwiązać ten problem powstał ext3. Dzięki księgowaniu czas ten jest zależny tylko od ilości metadanych, a nie od wielkości systemu plików.

Dodatkowo ext3 wprowadza minimalną liczbę zmian do kodu ext2, który jest systemem bardzo małym i stabilnym, a co za tym idzie prostym do utrzymania i zrozumiałym.

Ext3 jest tak napisany, że możliwe jest montowanie ext3 jako ext2 i vice versa.

## 12.2 Logiczna struktura dysku

Rysunek 12.1 przedstawia logiczną strukturę dysku w ext2 i ext3, szczegółowo omówioną na wykładzie.




Rysunek 12.1: Struktura danych na dysku w ext2 i ext3

## 12.3 Księgowanie w ext3

Rzeczywista warstwa księgowania (*Journal Block Device* JBD) jest oddzielona od ext3. System plików rozumie ideę transakcji (gdzie się transakcja zaczyna i kończy), ale nie jest odpowiedzialny za samo księgowanie.

Dziennik zarządzany przez JBD tworzony jest w zwykłym pliku lub na innym urządzeniu blokowym. Do dziennika zapisywane są całe zmienione bloki,które mają być zapisane na dysk.

Ext3 dostarcza trzy tryby księgowania:

- *journal* - dziennikowanie danych i metadanych (najbezpieczniejszy, najmniej wydajny),
- *ordered* - dziennikowanie metadanych, ale po zapisaniu danych na dysku (domyślny),
- *writeback* - dziennikowane są tylko metadane (możliwość niespójności).

## 12.4 Ciekawostki

- W ext3 istnieje możliwość włączenia indeksowanego formatu katalogu do szybszego znajdowania pozycji w katalogu.
- w ext2 5 procent systemu może być zarezerwowana dla administratora, co ułatwia naprawę w przypadku pełnego systemu.
- Administrator za pomocą opcji montowania może wybrać jedną w dwóch semantyk dostępu do pliku, BSD lub SVR4. Ma to odzwierciedlenie w wyborze ID dla grupy.

W semantyce BSD tworzony plik ma takie same group id jak katalog nadrzędny. W semantyce SVR4, jeśli katalog ma ustawiony bit *ustaw group id*, nowy plik dziedziczy to id, a podkatalogi dodatkowo dziedziczą ten bit; w przeciwnym przypadku pliki i katalogi dziedziczą identyfikator grupy wołającego procesu.

# Rozdział 13

# XFS

Historia XFS'a rozpoczęła się na początku lat 90-tych. Szukano systemu, który zaspokoiłby ograniczenia pojemnościowe ówczesnych systemów. W ten sposób w 1994 razem z systemem IRIX 5.3. ukazał się XFS.

## 13.1 Pomysł

XFS jest systemem 64-bitowym gwarantującym transakcyjność wszystkich operacji. We wszystkich strukturach danych wykorzystuje B+drzewa. XFS efektywnie obsługuje wielkie pliki, katalogi, systemy plików oraz dużą liczbę plików. Wymaga to wyspecjalizowanych algorytmów i organizacji przestrzeni dyskowej.

## 13.2 Dane na dysku

XFS obsługuje w pełni 64 bitowy system plików. Wszystkie wskaźniki (bloków, i-węzłów) są 64 bitowe. Aby uniknąć w niektórych przypadkach tłumaczenia wskaźników na 64 bitowe, system plików jest podzielony na grupy alokacji (ang. *Allocation Groups* - AG). Ich rozmiar jest stały i może wynosić od 16MB do 4GB (w praktyce wykorzystuje się podziały >= 0.5GB). Grupy alokacji ułatwiają wielowątkowe zarządzanie systemem plików.

AG są autonomicznymi jednostkami systemu plików. Każda AG zawiera osobne struktury wymagane dla zarządzania jej przestrzenią. Odpowiedni rozmiar AG pozwala utrzymać struktury w optymalnym rozmiarze. W skład struktur grupy alokacji wchodzą:

- superblok - na początku AG, zawiera wszelkie podstawowe informacje o systemie,
- struktura wolnych obszarów - w ramach AG dostępna wolna przestrzeń całego systemu plików jest indeksowana w formie pary B+drzew.

- drzewo i-węzłów - w grupie alokacji i-węzły są zapamiętywane w specjalnych B+drzewach. I-węzły XFS'a nie różnią się wiele od klasycznych uniksowych i-węzłów. Również mają swój niepowtarzalny numer, według którego są indeksowane w B+drzewach.

- wykaz bloków zajmowanych przez drzewo wolnych obszarów - struktura pomocnicza przyspieszająca znajdowanie wolnej przestrzeni.

Zarządzanie wolnymi obszarami ma największy wpływ na wydajność systemu plików. XFS zastąpił bitmapę wolnych bloków przez strukturę składającą się z pary B+drzew dla każdej AG. Pierwsze drzewo to indeks położenia wolnych obszarów, drugie - ich rozmiarów. Podwójne indeksowanie pozwala na bardzo elastyczne i efektywne wyszukiwanie wolnych obszarów. Złożoność obliczeniowa operacji wyszukiwania w drzewie B+ jest rzędu O(log n), a więc znacznie mniejsza od złożoności przeszukiwania map bitowych, która w skrajnym wariancie wynosi O(n).

## 13.3 Księgowanie

Księgowanie w XFS'ie jest asynchroniczne. Modyfikacje metadanych nie są od razu zapisywane do dziennika, ale są przez pewien czas przechowywane w buforze dziennika w pamięci. Zapewnione jest przy tym, że żadne dane nie zostaną zapisane na dysk zanim nie zostanie zapisana do dziennika na dysku informacja o tych zmianach.

## 13.4 Ciekawostki

- Listy uprawnień w XFS'ie są rozbudowane w stosunku do zwykłych Unix'owych praw dostępu. Można definiować prawa dla konkretnej osoby.

# Rozdział 14

# Reiser File System

ReiserFS jest, obok ext2 i ext3 jednym z najbardziej popularnych linuxowych systemów plików. Jest domyślnym systemem plików dla SuSE, Lindows i Gentoo.

Nazwa wzięła się od nazwiska Hansa Reisera – ojca, głównego architekta, szefa projektu i programisty systemu.

Obecnie używamy Reiser3, ale w fazie zaawansowanych, ostatnich testów jest już Reiser4. Postanowiliśmy bliżej mu się przyjrzeć.

## 14.1 Dlaczego Reiser?

- Testy wykazują, że jest jednym z najszybszych systemów plików.
- Wykorzystuje dziennikowanie, z tym że w Reiserze4 zostało ono ulepszone w stosunku do Reisera3, np. przez zastosowanie tzw. ***wędrujących logów**, z ang. wandering logs.*
- Jest **systemem atomowym**, tzn. operacje albo wykonują się do końca, albo wcale – nie są przerywane w trakcie. Dzieje się to bez znaczących strat dla wydajności, gdyż używane są tu algorytmy, które nie kopiują podwójnie danych.
- Używa ***tańczących drzew** (z ang. dancing trees)*, czyli szybkich drzew zrównoważonych.
- Bardziej wydajny pamięciowo niż inne systemy plików, ponieważ ściska małe pliki razem, co pozwala oszczędzać miejsce w pamięci.
- Oparty na **wtyczkach** *z ang. plugins*, co znaczy, że jest zmodularyzowany i w związku z tym będzie dobrze współdziałał z różnymi zewnętrznymi nowinkami, które będzie można wdrożyć bez ponownego formatowania dysku.
- Zaprojektowany dla militarnego poziomu bezpieczeństwa. Łatwo jest kontrolować jego kod, wejścia do wszystkich funkcji są chronione.

## 14.2 Warstwy

Różne programy są często podzielone na warstwy, z których każda komunikuje się tylko z warstwą położoną bezpośrednio nad nią lub pod nią.


Reiser4 ma tzw. *warstwę semantyczną*, która odpowiada za nazywanie obiektów i decydowanie, co z nimi zrobić, natomiast nie zajmuje się takimi sprawami, jak ulokowanie obiektów w odpowiednim miejscu na dysku, czy też w strukturze danych – tym zajmuje się *warstwa przechowywania*.

Przy szukaniu obiektów warstwa semantyczna bierze nazwy i konwertuje je do kluczy, a warstwa przechowywania na podstawie tych kluczy znajduje odpowiednie bajty.

## 14.3 Pliki

Interakcje z plikami są wyłapywane przez *wtyczkę dla plików*. Każdy sposób interakcji z tą wtyczką nazywamy *metodą*. Wtyczka jest skonstruowana jako zbiór takich właśnie metod.

Każdemu plikowi przypisuje się *identyfikator wtyczki, z ang. pluginid*, który określa niejako typ pliku i zbiór operacji, które można na nim wykonać. Jest on odszukiwany przy każdej próbie interakcji z plikiem; w kernelu mamy tablicę wtyczek i przy pomocy identyfikatora wtyczki odszukiwana jest wtyczka odpowiednia dla tego pliku.

## 14.4 Katalogi

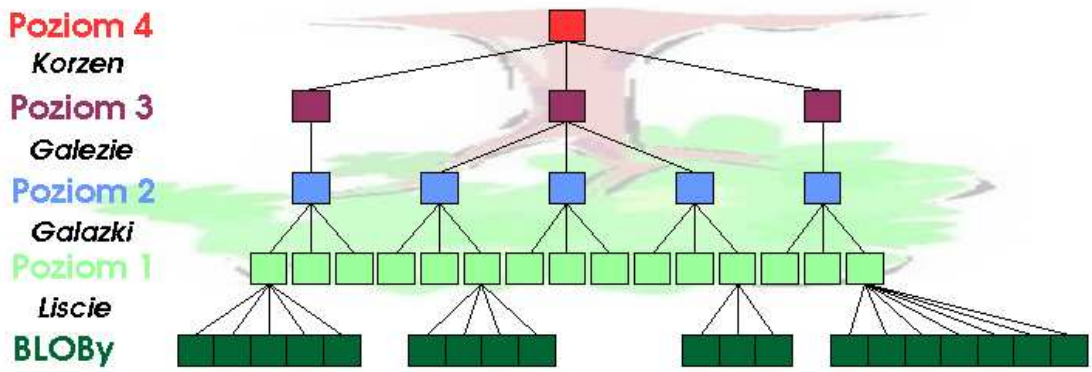
*Wtyczka do obsługi katalogów* implementuje katalogi jako zbiory wpisów katalogowych. Wpisy zawierają nazwę i klucz. Kiedy wtyczka dostaje ścieżkę do rozwinięcia, znajduje element, któremu przypisana jest dana nazwa i zwraca klucz. Klucz może być teraz użyty przez *warstwę przechowywania*, do znalezienia każdego kawałka tego, co określała nazwa.

W Reiser4 (ale nie w ReiserFS 3) obiekt może być jednocześnie plikiem i katalogiem. Jeśli otworzymy go jako plik, zobaczymy sekwencję bajtów. Jeśli zaś otworzymy go jako katalog, zobaczymy w nim inne katalogi i pliki. Na Linux Kernel Mailing List toczyła się długa dyskusja na temat tego, czy jest to technicznie możliwe do zrobienia. Linus pokazał, że jest, bez łamania VFS. Pozwolenie obiektowi być plikiem i katalogiem jednocześnie jest podstawą do zbudowania funkcjonalności obecnej w strumieniach. Aby uzyskać plik ze wszystkimi jego metadanymi Reiser4 używa wtyczki plikowej do ciała pliku i wtyczki katalogowej do odszukiwania wtyczek plikowych do obsługi metadanych.

## 14.5 Koncepcje modelowania drzewa

Dobrym sposobem na uporządkowanie informacji jest ułożenie ich w drzewo.

### 14.5.1 Drzewo w ReiserFS 3



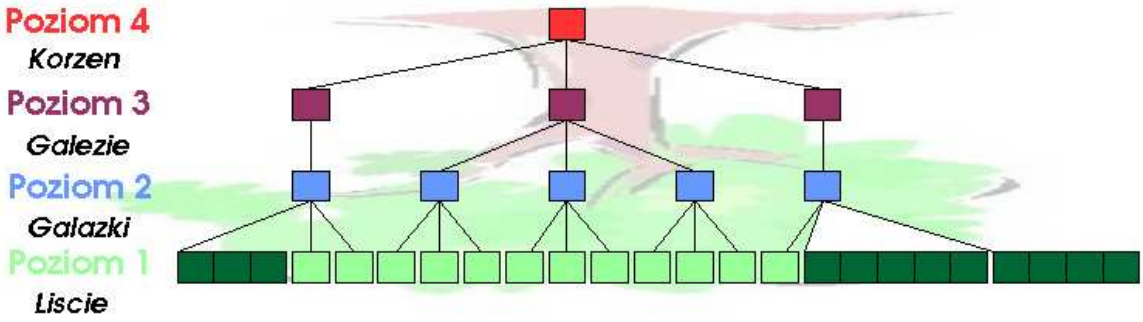

ReiserFS 3 używa B+drzew, które od B-drzew różnią się tym, że wszystkie dane są przechowywane w liściach. Krótkie informacje, takie jak nazwy katalogów, plików trzymane są bezpośrednio w liściach, a duże pliki w tzw. *BLOBach*.

**BLOB**y czyli *Binary Large OBjects* to sposób przechowywania obiektów o rozmiarze większym niż rozmiar węzła, polegający na przechowywaniu wskaźników do węzłów zawierających poszczególne fragmenty obiektu. Owe wskaźniki są zwykle przechowywane w tych węzłach, które nazywa się liśćmi :)

BLOBy sprawiają, że drzewo przestaje być zrównoważone, redukują rozdzielenie wskaźników od danych i w związku z tym – zmniejszają wydajność.

### 14.5.2 Drzewo w Reiser4

Reiser4 powraca do klasycznej definicji drzewa zrównoważonego ze względu na długości ścieżek do liści. Nie próbuje udawać, że węzły, które przechowują obiekty większe niż węzeł, jakimś cudem nie są częścią drzewa. Dzięki temu zużywa się mniej pamięci na pamiętanie wskaźników.



### 14.5.3 Gałęzie, gałązki, liście

Rodzaje węzłów w drzewie Reisera4:

**sformatowany liść** *z ang. formatted leaf* przechowywane są w nim dane, zawiera tzw. *items*, o których powiem później

| Nagłówek węzła | Wpis 0 | Wpis 1 | ... | Wpis n | Wolne miejsce | Nagłówek wpisu 0 | Nagłówek wpisu n | ... | Nagłówek wpisu 0 |
|---|---|---|---|---|---|---|---|---|---|

**niesformatowany liść** *z ang. unformatted leaf, czyli w skrócie unfleaf*, to liść, który zawiera blok rozszerzony. W związku z tym wskaźniki do niesformatowanych liści są inne niż wskaźniki do pozostałych węzłów. Są to tzw. *wskaźniki rozszerzone*. Wskaźnik rozszerzony zawiera numer pierwszego bloku bloku rozszerzonego i jego długość.

| ..... |
|---|

**węzły wewnętrzne** *z ang. internal nodes* składają się ze wskaźników do poddrzew, oddzielonych kluczami. Klucz jest równy pierwszemu kluczowi w pierwszym węźle sformatowanym poddrzewa, na które wskazuje wskaźnik. Wyróżniamy tu

- **gałązki** *twigs* to rodzice liści. Tu znajdziemy wskaźniki rozszerzone.

| Nagł. węzła | Wpis 0 Wsk. 0 | Wpis 1 Rozsz. wsk. 1 | Wpis 2 Wsk. 2 | Wpis 3 Rozsz. wsk. 3 | . | Wpis n Wsk. n | Wolne miejsce | Nagł. wpisu n | . | Nagł. wpisu 0 |
|---|---|---|---|---|---|---|---|---|---|---|

- **gałęzie** *branches* to pozostałe węzły.

| Nagłówek węzła | Wpis 0 Wskaźnik 0 | ... | Wpis n Wskaźnik n | Wolne miejsce | Nagłówek wpisu n | ... | Nagłówek wpisu 0 |
|---|---|---|---|---|---|---|---|

Dla uproszczenia kodu drzewa w Reiserze mają minimalną wysokość równą 2 i korzeń zawsze ma dzieci. Jedyną przyczyną takiego stanu rzeczy jest lenistwo twórców: nie trzeba się jakoś specjalnie troszczyć o drzewa jednowęzłowe – upraszcza to kod.

### 14.5.4 Rozmiar węzłów

W Reiserze wszystkie węzły mają ten sam rozmiar. Takie podejście ułatwia alokację nie używanego miejsca pomiędzy węzłami, ponieważ jego rozmiar jest pewną wielokrotnością rozmiaru węzła, zatem nie ma problemu z wolnym miejscem, które jest zbyt małe, aby zmieścić jeden węzeł. Ponadto dyski (stacje dysków) mają interfejs, który przyjmuje równy rozmiar bloków, co jest wygodne dla ich algorytmów korekcji błędów.

Węzły w Reiserze mają zwykle rozmiar strony. Nie ma żadnego szczytnego powodu, dlaczego wybrano ten właśnie rozmiar. Nie jest to wartość wybitnie lepsza od innych. Jest ona po prostu łatwa do wpisania w kodzie, do zaprogramowania, a twórcy Reisera nie mieli czasu na eksperymenty z innymi rozmiarami.

Skoro węzły mają równy rozmiar, to co się dzieje, jeśli chcemy przechować obiekt, który ma większy rozmiar niż rozmiar węzła? Otóż jest on krojony na mniejsze kawałki, z ang. *item*. Mają one oczywiście rozmiar jednego węzła.

Większość systemów plików przechowuje pliki w całych blokach. Z grubsza mówiąc, znaczy to, że wówczas mnóstwo miejsca jest marnowanego, szczególnie gdy mamy dużo małych plików. Ponadto niepotrzebnie wydłuża się czas operacji wejścia-wyjścia. Na przykład nie jest efektywne przechowywanie typowych obiektów bazodanowych takich jak adresy lub numery telefonów w oddzielnych plikach, ponieważ zaowocuje to stratą ponad 90% miejsca w blokach, które je pamiętają. Reiser upycha małe pliki i kawałki plików do jednego bloku. Daje to wydajność pamięciową około 94% dla małych plików.

Mimo wszystko wyrównywanie plików do 4k ma wielkie zalety dla dużych plików. Kiedy program chce operować bezpośrednio na danych pliku, bez używania wywołań systemowych, może użyć *mmap()*, aby zrobić z danych pliku część przestrzeni adresowej procesu. Z powodu pewnych detali implementacji funkcja *mmap()* potrzebuje, aby dane pliku były wyrównane do 4k i jeśli tak jest od razu, jest bardziej efektywna. W Reiserze4 domyślnie jest tak, że pliki, które są większe niż 16k, są wyrównywane do 4k. Twórcy Reisera nie zbadali jeszcze doświadczalnie, czy 16k jest tu optymalną liczbą, jednak wydaje się, że jest to co najmniej przyzwoity wybór.

## 14.6 Rozdrabnianie

Węzły w drzewie są mniejsze niż niektóre obiekty, które przechowują, z drugiej strony – większe niż inne. Więc jak dokładnie przechowujemy obiekty? Jednym ze sposobów jest rozlanie obiektów do pojemników wielkości węzła. Te pojemniki to właśnie *items*, o których wspominałam wcześniej.

Każdy taki pojemnik ma etykietkę, która zawiera:

- klucz
- offset, który mówi, gdzie w węźle zaczyna się zawartość pojemnika
- wielkość zawartości
- identyfikator wtyczki, który wyznacza typ zawartości

Te "pojemniki" pozwalają uniknąć zaokrąglania obiektów do 4k.
A tak wyglądają:

| Zawartość (ciało) | ... | Etykietka (nagłówek) | | | |
|---|---|---|---|---|---|
| | | Klucz | Offset | Długość | Identyfikator wtyczki |

### 14.6.1 Rodzaje pojemników

**informacyjny** (*static_stat_data*) przechowuje:

- właściciela
- prawa dostępu

- czas ostatniego dostępu
- czas utworzenia
- czas ostatniej modyfikacji
- rozmiar
- liczbę dowiązań do pliku

**katalogowy** (*cmpnd_dir_item*) przechowuje:

- wpisy katalogowe
- klucze do plików, których wpisy dotyczą

**pośredni (wskaźnikowy)** (*pointers, extent pointers*) przechowuje listę wskaźników (zwykłych lub rozszerzonych) do bloków na dysku

**bezpośredni** (*bodies*) przechowuje części plików, które nie są wystarczająco duże, aby je przechowywać w niesformatowanych liściach

*Jednostka* to coś, co musimy w całości zmieścić w pojemniku, czego nie możemy rozlać do wielu pojemników.

- Dla pojemników typu bezpośredniego jednostką jest bajt.
- Dla pojemników typu katalogowego jednostkami są wpisy katalogowe. Wpisy zawierają nazwę i klucz pliku.
- Dla pojemników typu informacyjnego wszystkie informacje, które przechowują (czyli właściciel, prawa itd.) są niepodzielną jednostką.

### 14.6.2 Tańczące drzewa – szybsze niż drzewa zrównoważone

Wyobraźmy sobie spójny ciąg węzłów *dirty* (zmodyfikowane, nie zapisane). Po angielsku taki ciąg jest nazywany *slum*, my go sobie nazwiemy *szaszłykiem* (gdzie każdy kawałek mięska lub czegokolwiek obrazuje pewną porcję danych).

Kawałki mięska można wyjadać, a następnie resztę dosuwać do siebie, czyli ściskać.

Drzewa zrównoważone tradycyjnie używają ustalonego kryterium do zdecydowania, czy węzły powinny być ściśnięte razem do mniejszej liczby węzłów dla zachowania miejsca. Ściskanie może się odbywać nawet po każdej modyfikacji. Takie podejście może co prawda dać większą oszczędność miejsca, jednakże, im więcej sąsiadów chcemy ścisnąć, tym większe jest prawdopodobieństwo, że będziemy musieli tych sąsiadów odczytywać z dysku. To może bardzo rzutować na wydajność czasową systemu.

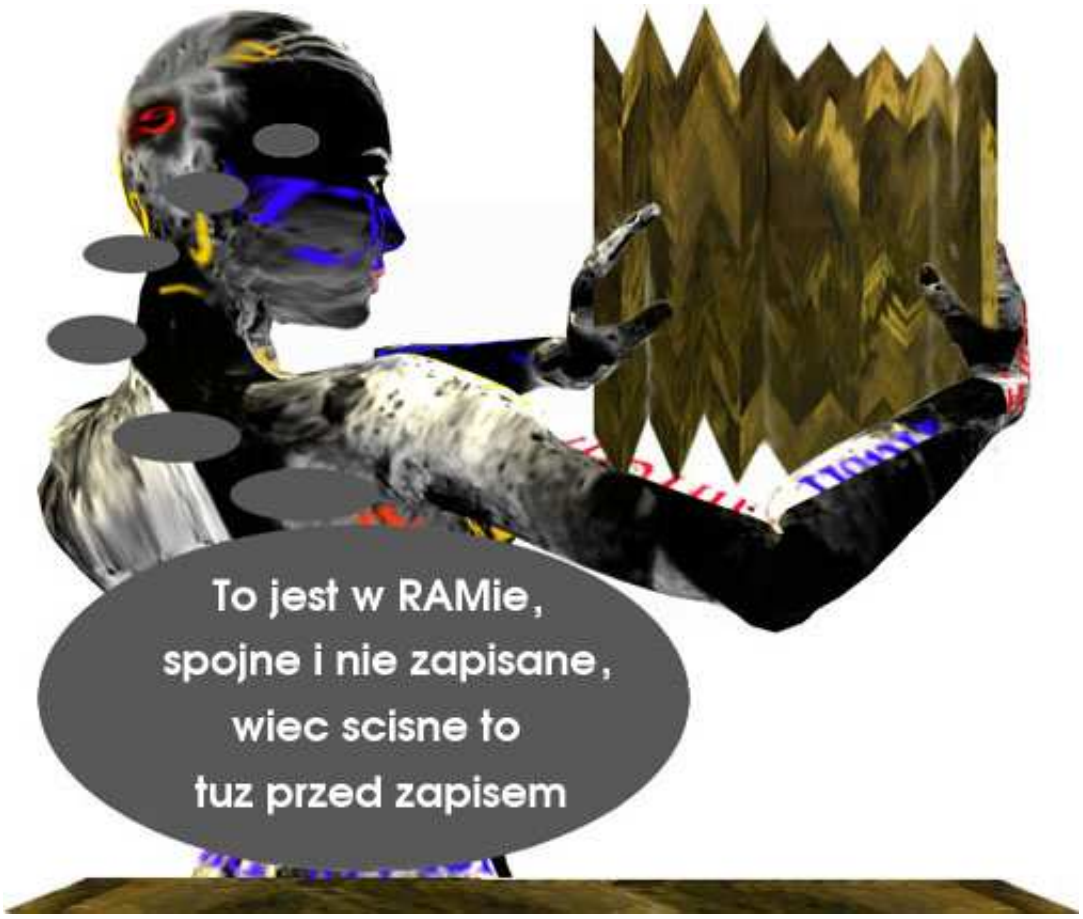
Reiser4 modyfikuje nieco tradycyjne B+drzewa zrównoważone, wprowadzając tzw. *tańczące drzewa, z ang. dancing trees*. Scalają one nie do końca wypełnione węzły, ale nie po każdej modyfikacji w drzewie:

- w reakcji na *flush*

- jako wynik zamknięcia transakcji (*commit*)

Po tychże zdarzeniach popycha się "kawałki mięska" jak najbardziej w lewo i zostaje dużo wolnego miejsca na "patyku". Daje to dużą oszczędność miejsca przy mniejszej liczbie przesunięć danych.

ReiserFS 3 przypisuje węzłom numery bloków w chwili ich utworzenia. Idąc za przykładem XFSa, programiści Reisera 4 postanowili jednak czekać z zapisywaniem danych do ostatniej chwili. Jak bardzo jest to ważne, widać na przykładzie sytuacji, gdy usuwamy plik, zanim trafi od na dysk. Istnienie takiego pliku powinno pozostać nie zauważone przez dysk, nie powinno zostawić żadnych, nawet chwilowych śladów.




## 14.7 Reiser4 jako atomowy system plików

Gdy coś złego stanie się z komputerem, podczas gdy w RAMie mamy dane, które nie dotarły na dysk, to mamy stratę. Ale nie zawsze opłaca nam się zachowywać to, co już zostało zapisane.

Wyobraźmy sobie transfer 10 zł z konta bankowego Michała na konto Pawła. Transfer składa się z dwóch operacji: potrącenia 10 zł z konta Michała i dodania 10 zł do konta Pawła. Wyobraźmy sobie, że jedna z tych elementarnych operacji została wykonana, a druga nie. Taka sytuacja nie jest dobra, mimo że poniekąd mamy częściowy zapis transakcji: komuś coś dodaliśmy albo komuś coś zabraliśmy.

Jeśli weźmiemy pod uwagę zbiór operacji, które albo muszą się wykonać wszystkie, albo nie może żadna z nich, to taki zbiór nazywamy atomem. Wszystkie *wywołania systemowe* są w Reiserze atomowe. Reiser pozwala definiować nowe atomowe operacje przy użyciu wtyczek. Reiser używa specjalnych algorytmów, które pozwalają uczynić operacje atomowymi przy niewielkim dodatkowym koszcie, podczas gdy inne systemy plików musiałyby płacić wysokie, wręcz niedopuszczalne ceny za coś takiego.

## 14.8 Przepychacz

80% danych na dysku pozostaje nie zmienianych przez długi okres czasu. Wydajnie by było, gdyby były dobrze rozmieszczone. Reiser4 oferuje specjalny program – *przepychacz, z ang. repacker*. Przechodzi on drzewo od lewej do prawej, spychając wszystko w lewo i od prawej do lewej, spychając w prawo. Defragmentuje drzewo i upycha dane.

## 14.9 Wtyczki

Lenistwo jest uznane wśród programistów jako szczyt formy, najwyższa sztuka. Twórcy Reisera szczycą się tym, że robią co mogą, aby jak najlepiej się wyrazić w tejże sztuce. Mówią, że architektura wtyczek to nic innego, ale cała masa lenistwa.

**plikowe** omówione wcześniej

**katalogowe** omówione wcześniej

**haszujące** Ponieważ nazwy plików nie mogą być kluczami w drzewie, nazwy są haszowane i w ten sposób otrzymujemy klucze.

Przy zakładaniu Reisera wybiera się algorytm do generowania kluczy. W Reiserze3 jeden z trzech dostępnych algorytmów powodował kolizje – mogło się zdarzyć, że jakiś plik znikał.

**bezpieczeństwa** dbają o ochronę danych

**obsługi rozdrabniania obiektów**

**przypisywania kluczy**

**wyszukiwania węzłów**

Wtyczki można tworzyć i dodawać. Przy dodawaniu trzeba przekompilować jądro i dodać nową wtyczkę na koniec listy. Przewiduje się, że w przyszłych wersjach Reisera dodane zostaną poprawki, które umożliwią dynamiczne ładowanie wtyczek (w trakcie działania programu).

# Rozdział 15

# Trochę praktyki

## 15.1 ext2 vs. reiserfs

Pokażemy teraz dwa krótkie testy, pokazujące praktyczne działanie systemów plików *ext2* i *reiserfs*. W celu ich przeprowadzenia należy utworzyć i zamontować system plików w danym formacie (polecenie *mke2fs* dla *ext2* i *mkreiserfs* dla *reiserfs*).

Pierwszy test porównuje szybkość operacji *write()*:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int i, fd, wynik;
  char buf[1024];

  fd = open("duzyplik", O_CREAT | O_TRUNC | O_WRONLY, 0644);
  if (fd == -1) { printf("Błąd!\n"); exit(1); }

  for (i = 1; i <= 100000; i++) {
    printf("Zapisywanie do pliku nr %d...\n", i);
    write(fd, buf, 1024);
  }

  close(fd);
  return 0;
}
```

Po skompilowaniu kodu przykładu i przejściu do dowolnego katalogu w testowanym systemie plików, wystarczy uruchomić *time test1*. Oto otrzymane wyniki:

```
ext2:  0.45s user 2.49s system 20% cpu 14.518 total
reiserfs:  0.72s user 4.11s system 32% cpu 14.831 total
```

Jak widać, standardowy *ext2* uzyskał w łącznym czasie niezauważalną przewagę. Jednak *reiserfs* dużo lepiej radzi sobie z dużą liczbą małych plików, co teraz zobaczymy:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int i, fd, wynik;
  char nazwa[10];

  for (i = 1; i <= 10000; i++) {
    printf("Tworzenie pliku %d...\n", i);
    sprintf(nazwa, "%d", i);
    fd = open(nazwa, O_CREAT | O_TRUNC | O_WRONLY, 0644);
    if (fd == -1) { printf("Błąd!\n"); exit(1); }
    write(fd, "A", 1);
    close(fd);
  }

  return 0;
}
```

Postępujemy podobnie jak wyżej otrzymując tym razem rezultat:

```
ext2:  1.60s user 14.82s system 86% cpu 19.064 total
reiserfs:  1.59s user 3.87s system 68% cpu 8.012 total
```

Widać ponad dwukrotną przewagę *reiserfs*, zatem w zastosowaniach wymagających pracy z dużą liczbą plików będzie on znacznie bardziej wydajny.

## 15.2 unlink

Poniższy program otwiera plik, czeka na naciśnięcie klawisza, a następnie zamyka ten plik:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int fd;

  fd = open(argv[1], O_RDONLY);
  if (fd == -1) { printf("Błąd!\n"); exit(1); }
  scanf("%d");
  close(fd);

  return 0;
}
```

Tworzymy jakiś plik o rozmiarze zauważalnym w *df*, np. przez
`dd if=/dev/hda of=/tmp/hello count=10M`,
a następnie uruchamiamy powyższy program z nazwą tego pliku.

Jeżeli na innej konsoli skasujemy ten plik (poleceniem *rm*, które wywołuje funkcję *unlink* na danym pliku), to *ls* pokaże, że go już nie ma, ale *df* – że nadal zajmuje miejsce na dysku.

Faktycznie plik jest usuwany dopiero po zakończeniu naszego programu (naciśnięciem dowolnego klawisza), czyli po zamknięciu go.

## 15.3 libtrash

Biblioteka *libtrash* jest linuksową implementacją znanego z Windowsów „kosza", a dla nas stanowi również pouczający przykład. Pomysł polega na przechwyceniu funkcji *unlink()* i *rename()* z biblioteki *libc6* i przenoszeniu usuwanych plików do specjalnego katalogu.

Aby włączyć tę funkcjonalność wystarczy ustawić zmienną środowiskową *LD_PRELOAD* na ścieżkę do pliku z dynamicznie ładowaną biblioteką, np.
`export LD_PRELOAD=/usr/local/lib/libtrash.so`

Wtedy zawsze będzie ona ładowana przed załadowaniem dynamicznej biblioteki *libc6*. Następnie programy używające *unlink()* będą przenosiły usuwane pliki do naszego kosza.

Rozwiązanie to jest eleganckie, ponieważ nie ma potrzeby modyfikacji ani kodu jądra, ani samej biblioteki *libc6*. Ponadto biblioteka składa się tylko z jednego pliku .so oraz pliku konfiguracyjnego, w którym można ustawić jeszcze bardziej zaawansowaną funkcjonalność (związaną z przechwytywaniem także funkcji *open()* i *fopen()*).

To rozwiązanie działa, ponieważ programy użytkowe zwykle odwołują się do funkcji jądra poprzez funkcje biblioteczne z *libc6*. Jednak nie daje gwarancji, że każdy usunięty plik będziemy mogli odnaleźć w koszu.

Poniższy program jest przykładem „obejścia" powyższego mechanizmu, ponieważ wykonuje bezpośrednio funkcję systemową:

```
#include <stdio.h>
#include <errno.h>
#include <linux/unistd.h>

_syscall1(long,unlink, char *,plik)

int main(int argc, char *argv[])
{
  int wynik;

  wynik = unlink(argv[1]);
  if (wynik != 0) {
    return errno;
  }

  return 0;
}
```

Kompilując i uruchamiając go z parametrem pliku do usunięcia można się przekonać, że plik zostanie naprawdę usunięty. Można jednak założyć, że normalne programy korzystają z biblioteki *libc6*.

# Rozdział 16

# Podsumowanie lokalnych systemów plików

Unix oferuje bardzo wiele różnych systemów plików. Oto zbiorcze podsumowanie niektórych cech omawianych lokalnych systemów plików.

| **System plików** | **UFS** | **VxFS** | **Ext3** | **XFS** | **Reiser4** |
|---|---|---|---|---|---|
| **Maks. rozmiar systemu plików** | *) | 32TB | 4TB | 18 mln TB | 16TB |
| **Maks. rozmiar pliku** | *) | 2TB | 2GB | 9 mln TB | 16TB |
| **Rozmiar bloku** | 4KB, 8KB | 1KB, 2KB, 4KB, 8KB | 1KB, 2KB, 4KB | 512B - 64KB | 4KB |
| **Księgowanie** | Nie | Tak | Tak | Tak | Tak |
| **Dynamiczna alokacja i-węzłów** | Nie | Tak | Nie | Tak | Tak |
| **Ekstenty** | Nie | Tak | Nie | Tak | Tak |
| **Struktury do zarządzania wolnym miejscem** | Bitmapa | Bitmapa | Bitmapa | B+drzewa | B+drzewa |

*) W prezentacji omówiona została pierwotna wersja systemu UFS, od czasu której wiele się zmieniło - odsyłam do bibliografii.

Wyjaśnienie: 1TB = 1024 GB = $2^{40}$

Analizując różne testy na wydajność wydaje się, że najlepszymi systemami są XFS i ResierFS. UFS jest nadal rozwijany. ResierFS najlepszy jest do użytku domowego, gdy mamy dużo małych plików. Zaletą ext3 jest jego kompatybilność z bardzo popularnym i rozpowszechnionym ext2. Wadą systemu VxFS jest komercyjność i brak dostępnej dokumentacji.

Zapewne każdy z systemów ma swoją mocna stronę i zastosowanie, do którego najlepiej pasuje.

# Część IV

# Rozproszone systemy plików

# Rozdział 17

# Wprowadzenie

Lata siedemdziesiąte ubiegłego wieku, to era rozkwitu sieci komputerowych. Naturalną potrzebą stało się przesyłanie plików pomiędzy komputerami połączonymi w sieć. W końcu – jaki bez tego byłby pożytek z sieci komputerowej. Pierwsze rozwiązania, to programy `ftp` i `uucp` (Unix-to-Unix-copy). Dzięki nim można było ściągnąć na swój dysk lokalny cały plik, pracować na nim lokalnie, a następnie odesłać nową wersję, przesyłając na serwer cały plik. Jednak szybko okazało się, że taka forma pracy jest dla wielu użytkowników niewygodna, a części wręcz uniemożliwia pracę.

Rozwiązanie przyniosły lata osiemdziesiąte. Właśnie w tej dekadzie pojawiły się pierwsze projekty rozproszonych systemów plików i pierwsze ich implementacje.

W tej części dokumentu omówione zostaną sieciowe systemy plików. Ich zadaniem jest zapewnienie możliwie wygodnego dostępu wielu użytkowników do danych, znajdujących się zazwyczaj na jednym serwerze. Przedstawione zostaną najpopularniejsze (obecnie lub kiedyś) rozwiązania, które swoje korzenie mają w systemach Uniksowych.

# Rozdział 18

# Przegląd technologii pracy w środowiskach rozproszonych

Aby umożliwić pracę w sieci, należało stworzyć pewne standardy, definiujące sposób komunikacji pomiędzy komputerami.

## 18.1 Wywołania procedur zdalnych (RPC)

### 18.1.1 Wprowadzenie

Protokół wywołania procedur zdalnych określa format komunikacji między klientem a serwerem. Dzięki niemu programy klienckie, wykorzystujące procedury dostarczane przez serwer, mogą być pisane w bardzo podobny sposób do zwykłych programów. Protokół określa kwestie sposobu budowania komunikatów, autoryzacji oraz przesyłania przez sieć. RPC jest oczywiście niezależny zarówno od protokołu transmisji danych, jak i aplikacji go wykorzystujących.

W świecie komputerów dominują dwie najpopularniejsze wersje i zarazem implementacje protokołu RPC. Jedna z nich jest autorstwa firmy Sun Microsystems i została przygotowana wraz z pracami nad sieciowym systemem plików NFS. Druga natomiast należy do grupy OSF i jest wykorzystywana w środowisku Distributed Computing Environment.

Obydwie te wersje RPC są do siebie bardzo podobne. Różnią się tym, że DCE RPC oferuje dodatkowo możliwość asynchronicznych wywołań procedur RPC. Dzięki temu procesy klienckie nie są wstrzymywane w oczekiwaniu na odpowiedź serwera, ale mogą zająć się „w międzyczasie" swoimi sprawami. Z kolei Sun RPC oferuje jedynie synchroniczne wywołania RPC. Dzięki temu programowanie z użyciem procedur zdalnych niczym nie różni się od programowania z użyciem lokalnych procedur.

### 18.1.2 Sun RPC od środka

Protokół Sun RPC specyfikuje komunikaty wysyłane pomiędzy serwerem a klientem. Nagłówek komunikatu zawiera między innymi następujące pola:

- `xid` – identyfikator komunikatu, który pozwala serwerowi odróżnić dwukrotne wywołanie procedury od ponowienia wywołania procedury w przypadku błędu transmisji,
- `direction` – określa kierunek komunikatu (wywołanie lub odpowiedź),
- *informacja uwierzytelniająca* – zależna od rodzaju autoryzacji wymaganej przez serwer.

Obecność pola `xid` służy rozwiązaniu ważnego problemem – uniewrażliwienia systemu na błędy transmisji. To samo wywołanie nie powinno być wykonane dwa razy. Może się tak zdarzyć, jeśli klient, zniecierpliwiony długim czasem oczekiwania na komunikat potwierdzający wykonanie zlecenia wyśle je jeszcze raz, zakładając, że to pierwsze zginęło w sieci.

Wprawdzie w implementacjach RPC często wykorzystuje się `TCP/IP`, niemniej jednak sam protokół RPC miał z założenia być niezależny od protokołu transmisji i dlatego posiada wbudowaną w siebie obsługę błędów transmisji.

W komunikatach wywołania funkcji znajdują się również informacje o wersji używanego protokołu (`rpc_vers`), o programie wołającym (`prog`) i jego wersji (`vers`), identyfikatorze procedury (`proc`) oraz przekazywane są parametry dla wołanej procedury.

Przy odpowiedzi serwer dołącza do komunikatu jedynie stan wywołania i wyniki procedury.

### 18.1.3 Przykłady wywołań (NFS)

Standardowym wywołaniem, które musi obsłużyć serwer jest bezparametrowe zlecenie kontrolne (`NULL`). Służy ono sprawdzaniu, czy serwer prawidłowo funkcjonuje.

Poniższa tabelka prezentuje niektóre operacje RPC protokołu NFSv2 (obecne również w wersji 3 i 4 protokołu NFS):

| **Zlecenie** | **Argumenty wejściowe** | **Wyniki** |
|---|---|---|
| `GETATTR` | `fhandle` | `status, fattr` |
| `SETATTR` | `fhandle, setattr` | `status, fattr` |
| `LOOKUP` | `dirfh, name` | `status, fhandle, fattr` |
| `READ` | `fhandle, offset, count, totcount` | `status, fattr, data` |
| `STATFS` | `fhandle` | `status, file_stats` |

### 18.1.4 Generowanie środowiska RPC

Istnieje program, który potrafi generować środowisko RPC. Na podstawie jednego pliku, zawierającego specyfikację protokołu w języku RPC (RPC Language), potrafi on wygenerować oprogramowanie serwera oraz pliki nagłówkowe klienta. Przy odpowiednio skonstruowanym pliku ze specyfikacją, program wygeneruje również przykładowy program klienta i serwera.

Strony http://www.cc.gatech.edu/classes/cs4431_98_winter/rpc.html zawierają kompendium wiedzy zarówno na temat samego protokołu SunRPC, jak i środowiska rpcgen.

Zadanie dla ambitnych: do najnowszej wersji specyfikacji protokołu NFSv4 jest stworzony plik, z którego można wygenerować środowisko RPC dla najnowszej wersji NFS i rozpocząć implementowanie algorytmów ;-).

### 18.1.5 Wykorzystanie

RPC jest bardzo wygodnym w użyciu protokołem, który szalenie ułatwia pisanie programów w architekturze klient-serwer. Z tego względu stosuje się go nie tylko w sieciowych systemach plików, ale również na przykład w NIS (*Network Information Server*).

Stworzono również środowisko XML-RPC i zaimplementowano je w Javie i Pythonie (wykorzystuje się protokół `http` do transportu i XML do kodowania danych).

## 18.2 Rozszerzona reprezentacja danych (XDR)

### 18.2.1 Wprowadzenie

Wysyłanie argumentów wraz z funkcjami wywołanymi poprzez protokół RPC nie sprawia żadnych problemów, dopóki po obu stronach łącza znajdują się komputery oparte na takim procesorze. Co jednak, gdy serwerem jest komputer pracujący na procesorze Pentium, a stacjami roboczymi komputery Suna z Motorolami?

Problem polega na tym, że różne procesory mogą różnie kodować liczby. Np. w Motorolach 680x0 bajt 0 jest najbardziej znaczący. Jednak w procesorach firmy Intel, ten jest najmniej znaczący. Oczywiście nie uda się nawiązać komunikacji, gdy komputery będą różnie interpretowały wysyłane komunikaty.

Rozwiązanie polega po prostu na wprowadzeniu jakiegoś standardu. Żaden z powyższych systemów nie jest oczywiście ani lepszy ani gorszy. Brakuje jedynie standardu. Taki zaproponowała firma Sun Microsystems. Zaskakująco, był on zbliżony do systemu, w jakim pracują procesory Motorola 680x0, wykorzystywane w stacjach roboczych Sun-2 i Sun-3.

### 18.2.2 Przykłady

Z punktu widzenia XDR rozróżnia się dwa typy danych. Pierwszy z nich to dane ze strukturą, które podlegają konwersji do formatu XDR. Drugie to dane bez struktury, które przekazywane są bez ingerencji w ich format.

**Liczba całkowita** jest wielkością 32-bitową. Bajt skrajnie lewy jest najbardziej znaczący.

**Dane bez struktury** reprezentuje się jako pole, opisujące 32-bitową liczbą długość bloku danych oraz blok danych, uzupełniony znakami `NULL` do wielokrotności 4 bajtów.

**Napisy** opisuje się za pomocą pola długości, będącego liczbą całkowitą oraz ciągu znaków ASCII, uzupełnionego ewentualnie znakami `NULL` do wielokrotności 4 bajtów.

**Tablice elementów jednorodnych** koduje się za pomocą pola rozmiaru (będącego liczbą całkowitą) oraz serii danych. Każdy element musi być tego samego typu, ale oczywiście – jak w przypadku napisów – elementy nie muszą być jednakowej długości.

### 18.2.3 Zastosowanie w RPC

Standard XDR może zostać wykorzystany do kodowania komunikatów przesyłanych protokołem RPC. Dzięki temu pozbywamy się ograniczenia, że wszystkie komputery muszą posiadać tę samą architekturę.

Zatem przed wywołaniem konkretnej procedury RPC, klient powinien dostosować jej argumenty, aplikując standard XDR. Nie jest to oczywiście możliwe, w przypadku przesyłania bloku danych (takich, jak na przykład blok danych do zapisania dla procedury `write`), gdyż na poziomie samego RPC nie jesteśmy w stanie wniknąć w strukturę danych. O ile zatem plik liczb całkowitych może być dzięki RPC swobodnie przesyłany między różnymi platformami, o tyle będzie on prawidłowo przeczytany tylko na tej, która go stworzyła.

### 18.2.4 Problemy

Zastosowanie XDR zazwyczaj ułatwia komunikację. Niestety, może niekiedy ją utrudnić. Jeśli bowiem zdecydujemy się na stosowanie XDR, wtedy część danych, przed ich wysłaniem, będziemy musieli przekonwertować. Z kolei serwer, przed ich zinterpretowaniem, będzie je musiał znowu przekonwertować. W przypadku, gdy zarówno klient jak i serwer operują na tej samej architekturze procesora, operacje te są zupełnie zbyteczne.

W niektórych zastosowaniach, jak na przykład Distributed Computing Environment, próbuje się obejść ten problem, uzgadniając na początku komunikacji, czy XDR jest wymagany. W konsekwencji tego, zarówno klient jak i serwer, muszą zapamiętywać pomiędzy transmisjami, w jakim formacie należy się ze sobą porozumiewać. W przypadku systemów z założenia bezstanowych (jak NFSv3), jest to niemożliwe.

# Rozdział 19

# RFS

## 19.1 Projekt

Autorem systemu plików RFS (Remote File Sharing) jest firma AT&T. Pierwsza wersja RFS została przygotowana dla Uniksa w wersji System V w wydaniu 3. Powstała mniej więcej w tym samym czasie, co NFSv2, czyli w połowie lat osiemdziesiątych.

Jako że firma AT&T jest mocno związana z Systemem V, nie dbała o projektowanie RFS tak, aby był łatwo przenośny na inne środowiska, w szczególności sama nie przygotowała innych implementacji. Celem projektowym było dostarczenie środowiska swobodnej pracy sieciowej dla Systemu V. Co więcej, RFS stał się częścią Uniksa, dostarczanego przez AT&T, więc korzystanie z niego wymagało odpowiedniej licencji.

To właśnie najprawdopodobniej polityka AT&T sprawiła, że RFS, mimo swoich niewątpliwych zalet, które zostały opisane w następnym rozdziale, nie miał tak zawrotnej kariery, jak NFS.

## 19.2 Architektura RFS

**Założenia projektowe.** Projektowanie systemu RFS rozpoczęto od fundamentalnego założenia: należy zachować Uniksową semantykę plików otwartych. Wymusiło to zastosowanie architektury stanowej, co stanowi podstawową różnicę pomiędzy systemami RFS i NFS.

**Organizacja sieci.** Ważną cechą systemu RFS jest istnienie wyznaczonego serwera, który zajmuje się zarządzaniem zasobami. Serwer nazw przechowuje informację, na jakiej fizycznej maszynie znajduje się dany zasób.

Koncepcja RFS obejmuje podział całej sieci (potencjalnie bardzo dużej) na tzw. domeny. Każda domena ma swój serwer nazw, odpowiedzialny za tłumaczenie nazw zasobów należących do domeny, którą kontroluje.

**Komunikacja.** Komunikacja pomiędzy klientem a serwerem opiera się na tzw. obwodach

wirtualnych. Dla każdej pary klient-serwer istnieje co najwyżej jeden taki obwód, ustanawiany w momencie pierwszej operacji `mount`.

System RFS korzysta z protokołu RPC.

**Środowisko klienta na serwerze.** Z założenia utrzymania Uniksowej semantyki plików wyniknęła konieczność pamiętania na serwerze środowiska klienta, związanego z otwartymi plikami. Serwer pamięta, jacy klienci mają podmontowane jego zasoby oraz jakie otwarte pliki w jakim trybie posiadają.

**Buforowanie danych.** Klienci buforują odczytywane i zapisywane dane. Serwer dba o to, żeby klient nie mógł użyć nieaktualnych danych ze swojego bufora.

## 19.3 Przebieg wywołania `mount`

Na początku pracy, serwer eksportujący pewien zasób, powinien go zarejestrować w serwerze nazw. Nawiązuje on w tym celu połączenie z serwerem nazw i woła zdalnie odpowiednią funkcję (`adv` od *advertise* – ogłaszać) przekazując nazwę zasobu. Serwer nazw rejestruje zasób zapamiętując, na którym serwerze jest on umieszczony.

Gdy na komputerze klienta wywoływane jest polecenie `mount`, wykonują się następujące kroki:

1. Klient odpytuje serwer nazw, gdzie można odnaleźć podany przez użytkownika zasób.
2. Serwer nazw przesyła do klienta nazwę serwera oferującego podany zasób.
3. Klient komunikuje się z odnalezionym serwerem danych. Jeśli pomiędzy nim a danym serwerem nie istnieje jeszcze obwód wirtualny, to go ustanawia.
4. Klient wykonuje funkcję systemową `mount`, zapamiętując w danych specyficznych dla tego punktu montowania wskaźnik do wirtualnego obwodu oraz nazwę zasobu.

## 19.4 Przebieg wywołania `open`

1. Program użytkownika wykonuje `open`, podając nazwę pliku.
2. Komputer klienta dokonuje analizy nazwy ścieżkowej. Gdy dojdzie do pierwszego punktu montowania, sprawdza, czy przy dalszej analizie będą przekraczane jakieś dalsze punkty montownia.
    - Jeśli tak, to analiza wykonywana jest krok po kroku: do serwera wysyłana jest nazwa kolejnego katalogu, on ją weryfikuje (sprawdza czy istnieje i czy użytkownik posiada wystarczające uprawnienia) i wynik zwraca od razu do klienta. Ten bierze kolejną składową nazwy i – o ile nie jest punktem montowania – wysyła ją serwerowi do przeanalizowania.

- Jeśli nie, to analiza wykonywana jest w całości na serwerze: jest tam przesyłana cała nazwa, serwer analizuje ją u siebie krok po kroku i odsyła wynik całej procedury.

3. Serwer przyznaje otwartemu plikowi nowy deskryptor i zapamiętuje go, kojarząc z odpowiednim klientem.
4. Do klienta przesyłany jest deskryptor oraz numer wersji pliku.
5. Sprawdzana jest pamięć podręczna u klienta: jeśli w pamięci podręcznej klienta znajdują się już jakieś fragmenty otwieranego pliku i zapamiętany numer wersji pliku nie zgadza się z przesłanym z serwera, to bufor dla tego pliku jest opróżniany.

## 19.5 Przebieg wywołania `write`

1. Użytkownik wykorzystuje zapamiętany deskryptor pliku do wykonania operacji `write` na pliku z systemu RFS.
2. System operacyjny wywołuje odpowiednią procedurę obsługi zapisu do pliku, właściwą dla RFS.
3. Oprogramowanie klienckie RFS wywołuje na serwerze zdalną procedurę zapisu na pliku, podając jego deskryptor oraz pakiet danych. Wykorzystuje w tym celu obwód wirtualny.
4. Serwer odnajduje plik, którego wywołanie dotyczy (na podstawie przekazanego deskryptora oraz informacji od kogo on pochodził).
5. Serwer sprawdza, czy dany plik nie jest otwarty przez innych użytkowników. Jeśli tak, to wysyła do nich informację o unieważnieniu danych w ich pamięci podręcznej i czeka na potwierdzenie.
6. Serwer zapisuje dane (korzystając z bufora zapisu).
7. Serwer wysyła wynik operacji do klienta.
8. Funkcja obsługi `write` dla RFS zwraca kod wynikowy operacji do programu użytkownika.

## 19.6 Reakcja na awarie

Awaria sieci lub jakiegoś komputera objawia się przerwaniem obwodu wirtualnego. Jeśli klient stwierdzi, że przerwana została komunikacja z serwerem:

1. Jeśli jest wykonywana jakaś funkcja systemowa na systemie RFS (`open`, `read`, `write`, itp.), to jest przerywana i zwraca błąd `ENOLINK`.

2. Wszystkim i-węzłom związanym z plikami na serwerze RFS ustawiana jest specjalna flaga, która powoduje, że następne odwołania do tych i-węzłów kończą się błędem.

Jeśli z kolei to serwer wykryje, że utracił kontakt z którymś ze swoich klientów, to musi po prostu po nim posprzątać, tak jak system operacyjny sprząta po zabitym procesie:

1. Modyfikowane są liczby odwołań do i-węzłów, których używał dany klient.

2. Następuje zwolnienie wszelkiego rodzaju blokad, założonych przez klienta.

# Rozdział 20

# NFSv3

## 20.1 Co odróżnia NFS od innych

Tym, co odróżnia system NFS od innych (zwłaszcza RFS) jest sposób prowadzenia projektu. Firma Sun zajmuje się jedynie opracowywaniem specyfikacji protokołu NFS (dotyczy to tylko NFSv2 i NFSv3 – p. 21). Implementacją może zająć się każdy.

Projektanci mieli przyjęte za cel stworzenie protokołu, który mógłby być łatwo zaimplementowany na wielu platformach sprzętowych i programowych. Bez wątpienia to się udało. Wprawdzie NFS uważany jest za standard w systemach Uniksowych, to jednak doczekał się implementacji na komputery Mach oraz systemy Novell Netware i MSDOS.

Protokół w tej wersji został omówiony po krótce. Po szczegóły odsyłamy na stronę wykładu.

## 20.2 Przebieg wywołania `mount`

1. Użytkownik wywołuje polecenie `mount`, podając nazwę zasobu oraz serwera.

2. Jądro wywołuje funkcję `nfs_mount()`.

3. Funkcja `nfs_mount()` wysyła polecenie RPC do odpowiedniego serwera.

4. Demon `mountd` działający na serwerze przyjmuje zlecenie, sprawdza czy podana nazwa jest nazwą eksportowanego katalogu. Jeśli tak, to przesyła potwierdzenie oraz uchwyt do katalogu.

5. Klient odbiera komunikat i zapamiętuje w strukturze vfs adres sieciowy serwera oraz uchwyt do katalogu.

## 20.3 Kwestia `open`

W systemie NFS nie występuje pojęcie pliku otwartego. Serwer jest z założenia bezstanowy, w związku z czym nie może pamiętać, który klient, jakie ma otwarte pliki. Wszystkie operacje na plikach wykonywane są na podstawie unikatowego (w skali systemu) uchwytu do pliku, otrzymywanego w wyniku operacji `LOOKUP`. W związku z tym operacja `open` redukuje się do sprawdzenia praw dostępu do pliku.

1. Program użytkownika wykonuje `open`, podając nazwę pliku.

2. Komputer klienta dokonuje analizy nazwy ścieżkowej, katalog po katalogu. Jeśli jakiś katalog „po drodze" jest katalogiem zdalnym, to system wysyła polecenie `LOOKUP` do odpowiedniego serwera.

3. Serwer tworzy nowy uchwyt dla pliku i zapamiętuje z jakim plikiem jest on związany. Nie jest to jednak informacja o otwarciu pliku, a jedynie umożliwienie skróconego dostępu do pliku. Ten sam uchwyt może zostać przekazany wielu klientom.

4. Do klienta przesyłany jest uchwyt do pliku.

## 20.4 Przebieg wywołania `write`

1. Użytkownik wykorzystuje zapamiętany deskryptor pliku do wykonania operacji `write` na pliku z systemu NFS.

2. System operacyjny wywołuje odpowiednią procedurę obsługi zapisu do pliku, właściwą dla NFS.

3. Oprogramowanie klienckie NFS wywołuje na serwerze zdalną procedurę zapisu na pliku, podając jego deskryptor oraz pakiet danych.

4. Serwer odnajduje plik, którego wywołanie dotyczy (na podstawie unikatowego deskryptora).

5. Serwer otwiera plik, sprawdzając, czy użytkownik posiada wystarczające do tego uprawnienia.

6. Serwer zapisuje dane.

7. Serwer wysyła wynik operacji do klienta.

8. Funkcja obsługi `write` dla NFS zwraca kod wynikowy operacji do programu użytkownika.

## 20.5 Reakcja na awarie

Jeśli serwer ulegnie awarii, lecz w miarę szybko zostanie przywrócony, to może się zdarzyć, że klienci nie zauważą jego nieobecności – przy kolejnym odwołaniu klienta do jakiegoś pliku na serwerze, ten zorientuje się, że skoro jest odwołanie, to musiało kiedyś nastąpić montowanie i założy, że wszystko jest w porządku.

Klienci jednak zakładają, że serwer pracuje cały czas. Nie ma mechanizmów, które pozwalałyby na stwierdzenie, że serwer jest niedostępny i wykonanie odpowiedniej akcji. Oprogramowanie, jeśli serwer nie odpowie przez dłuższy czas, zakłada, że nastąpił błąd transmisji i/lub awaria serwera i ponawia żądanie.

# Rozdział 21

# NFSv4

## 21.1 Wstęp

Od połowy lat 90. trwają prace nad przygotowaniem protokołu NFSv4. Sun, widząc rosnącą popularność trzeciej wersji protokołu, zdecydował się oddać projekt w ręce Internet Engineering Task Force (IETF) – grupy zrzeszającej wielu fachowców, w tym również wywodzących się z Sun Microsystems.

Za najważniejsze cele dla wersji 4. protokołu NFS przyjęto:

- poprawienie szybkości działania w sieciach rozległych,
- zwiększenie bezpieczeństwa operacji,
- stworzenie specyfikacji łatwo rozbudowywalnej i modyfikowalnej,
- zachowanie kompatybilności wstecznej oraz kompatybilności z popularnymi systemami operacyjnymi.

NFSv4 jest cały czas w fazie tworzenia. Pierwsze implementacje dopiero się pojawiają, nie zawierają jeszcze wszystkich funkcji przewidzianych w specyfikacji. Kilka idei przewidzianych we wstępnych wersjach protokołu nie znalazło jeszcze rozwiązań teoretycznych (nie opracowano algorytmów). Najnowszą wersją protokołu jest RFC3530 – opublikowana w kwietniu 2003r.

## 21.2 Nowe koncepcje

**Stanowość serwera.** Serwer pamięta informacje o swoich klientach – ich otwartych plikach, założonych blokadach. Jest to najpoważniejsza zmiana koncepcyjna względem poprzedniej wersji protokołu.

**Operacje złożone.** Większość standardowych akcji na pliku wymaga wykonania kilku operacji pod rząd. W sieciach rozległych potęguje to czas wykonania pojedynczej (funkcjonalnie) operacji. W NFSv4 opracowano wywołania operacji złożonych

(tzw. compound) – w jednym wywołaniu zakaspułkowane są wywołania potencjalnie kilku procedur protokołu. Dzięki temu znacznie spada liczba przesyłanych przez sieć komunikatów.

**Przeniesienie operacji `LOOKUP` na serwer.** Tłumaczenie nazw ścieżkowych odbywa się na serwerze. Spada dzięki temu liczba wymienianych przez sieć komunikatów.

**Zrezygnowanie z operacji `mount`.** Przed skorzystaniem z zasobu nie trzeba już wykonywać operacji `mount`. Operacja ta powodowała dotychczas specyficzny problem z portem na serwerze – port, pod którym nawiązywane było połączenie z serwerem NFS wyznaczany był dynamicznie, co uniemożliwiało korzystanie z serwerów NFS stojących za zaporą ogniową (*firewall*).

**Delegowanie.** Jeśli użytkownik jest jedynym, który korzysta z danego pliku, może otrzymać od serwera delegację danego pliku, tj. wyłączność na jego edycję. Wszystkie operacje na nim może wówczas wykonywać lokalnie, a na serwer odsyła plik dopiero po operacji `close`. Należy się również spodziewać, że dzięki stanowości serwera NFSv4 będzie możliwe zapewnienie aktualności danych znajdujących się w buforach klientów. Szczegóły nie są jeszcze znane.

**Bezpieczeństwo.** Do metod autoryzacji obsługiwanych przez poprzednie wersje protokołu (za pomocą `uid` i `gid`, klucza 192-bitowego i systemu Kerberos4) dołączono standard
RPCSEC_GSS (Generic Security Services). Jest to moduł, pod który przezroczyście dla NFS może być podpięty dowolny system zabezpieczający. Planuje się stworzyć co najmniej implementacje Kerberos5, PGP, RSA.

**Blokady.** Dzięki stanowemu serwerowi, możliwe stało się zastosowanie blokad. W protokole przewiduje się wszelkiego rodzaju blokady – na pliki, rekordy, itp.

**Przezroczystość migracji i replikacji.** Jest to tajemnicza właściwość, o której do tej pory zostało powiedziane bardzo mało. Planuje się, aby pewne dane mogły być swobodnie migrowane (mogły podążać za użytkownikiem) i replikowane (dotyczy tylko danych pozostających w trybie tylko do odczytu) pomiędzy serwerami. Nie są znane szczegóły.

## 21.3 NFSv4 a Internet

NFSv4 wyraźnie chce stać się standardem Internetowym. Chwilowo wyraźnie brakuje takiego. Dzięki spodziewanemu wzrostowi wydajności i bezpieczeństwa, NFSv4 będzie zapewne mógł być stosowany w następujących obszarach:

1. ściąganie plików – jako zastępca protokołu `ftp`, zapewniający przezroczyste dla użytkownika wznawianie przerwanej transmisji, łatwe uwierzytelnianie i bezpieczeństwo przesyłanych danych (czego wyraźnie brak w `ftp`).

2. kopie zapasowe – jako system plików ułatwiający oferowanie usług tworzenia kopii zapasowych swoich danych na serwerach klastrowych w Internecie.

3. dyski internetowe – jako system znakomicie ułatwiający korzystanie z Internetowych „przestrzeni roboczych".

# Rozdział 22

# Rodzina AFS

Zarówno NFS, jak i RFS, opisane w poprzednich rozdziałach, sprawdzają się jedynie w małych sieciach (LAN), takich jak jeden budynek, czy kilka pomieszczeń. Istnieje jednak wiele środowisk większych (zarówno pod względem zajmowanej przestrzeni, jak i ilości komputerów) rozmiarów (tzw. WAN, np. osiedla, sieci ogólnouniwersyteckie).

Potrzeba rozwiązania problemu wygodnej pracy w środowisku dużej siedzi sprawiła, iż, powołane do życia w roku 1982 przez Carnegie-Mellon University we współpracy z IBM, Information Technology Center (ITC) rozpoczęło pracę nad projektem AFS (Andrew's File System).

Projekt AFS zaowocował kilkoma wersjami tego protokołu. Z wersji drugiej wyodrębniły się dwie gałęzie rozwoju: z jednej powstała Coda (25) a z drugiej wersja AFS-3 a z tej DFS (24).

# Rozdział 23

# AFS

## 23.1 Założenia projektowe

System AFS z założenia miał być zdolny do obsłużenia wielu tysięcy użytkowników. Przy projekcie założono, że pliki są zazwyczaj przypisane do konkretnych użytkowników. Właściciel pliku będzie zapewne z niego korzystać ciągle przy tym samym komputerze (w domu) lub przy grupie komputerów (uczelniane laboratorium komputerowe).

## 23.2 Architektura

Powyższe założenia doprowadziły do opracowania następującej architektury:

**Dedykowane serwery.** W systemie AFS występuje wyraźny podział na serwery i klientów. Jeden komputer nie może łączyć w sobie obydwu tych funkcji. Co więcej, serwery wykorzystują oprogramowanie niejako zintegrowane z obsługą systemu plików. Konsekwencją tego jest fakt, że na serwerach stosuje się specjalny system plików, który potrafi obsługiwać jedynie oprogramowanie serwerowe AFS.

**Klastry.** Sieć podzielona jest na klastry. Składają się one z serwera oraz pewnej liczby komputerów klienckich. Klastry są ze sobą połączone. Użytkownicy z jednego klastra mają dostęp zarówno do plików na serwerze ze swojego klastra, jak również do plików na wszystkich innych serwerach. Oczywiście dostęp do zasobów serwera z własnego klastra jest najszybszy.

**Woluminy.** Dane zorganizowane są w woluminy. W ich skład wchodzi pewna ilość danych, zazwyczaj powiązana logicznie – np. dane jednego użytkownika lub grupy. Wolumin jest logiczną (niezależną od dysków i partycji) jednostką przechowywania danych. Są podstawową jednostką danych przemieszczaną pomiędzy serwerami w sieci.

**Swobodna migracja danych.** Woluminy mogą migrować pomiędzy serwerami. Dzieje się to automatycznie - jeśli jakiś serwer stwierdzi, że odwołania do konkretnego woluminu pochodzą zazwyczaj z innego klastra niż jego własny, może zainicjować operację

migracji. Dokonuje się ona przezroczyście dla użytkownika. Jedyny obserwowalny objaw takiej operacji, to skrócenie czasu dostępu do danych.

**Baza danych położenia woluminów.** Informacje o położeniu woluminu są pamiętane na każdym serwerze. W związku z możliwością migracji woluminów, w momencie podłączania się do zasobu, system operacyjny użytkownika musi się dowiedzieć, na którym komputerze w sieci należy szukać żądanych informacji. Temu celowi służą *Bazy danych położenia woluminów*, które są przechowywane na każdym z serwerów i zawierają informację, na którym serwerze znajduje się obecnie dany wolumin.

**Przeglądanie zawartości sieci.** Użytkownicy mogą przeglądać zawartość sieci. Na swoim lokalnym systemie plików mają w katalogu `/afs` podłączony obraz *Bazy danych położenia woluminów*, po których mogą poruszać się standardowymi poleceniami (`cd`, `ls`), przechodząc pomiędzy klastrami, woluminami, folderami i plikami.

**Obietnice powiadomienia.** Jeśli dane, na które została złożona obietnica powiadomienia, ulegną zmianie, serwer wyśle odpowiednią informację do komputera użytkownika.

**Duży bufor lokalny.** Użytkownicy posiadają na swoich komputerach duży bufor lokalny. Przechowywane są w nich ściągnięte pliki (a właściwie 64kB-bloki danych, otrzymanych od serwera) i przebywają tam tak długo jak mogą być potrzebne, o ile serwer udostępniający plik nie powiadomi o jego zmianie.

**Semantyka sesji.** Zastosowano semantykę sesji, co oznacza, że inni użytkownicy nie zobaczą zmian, które dokonuje w pliku jakiś użytkownik, póki ten nie zamknie sesji pracy z plikiem (`close`). Dopiero wtedy dane są wysyłane z komputera użytkownika i umieszczane na właściwym serwerze.

## 23.3 Zastosowania

System AFS udało się zaimplementować i uruchomić. Na matczynym uniwersytecie (Carnegie-Mellon University) zaczął działać w roku 1985, a w 1989 obsługiwał już 9000 kont użytkowników, 30 serwerów i 1000 komputerów klientów, gromadząc 45 gigabajtów danych.

Podjęto również próbę uruchomienia systemu AFS w sieci Internet. Wiosną 1992 r. działało 67 klastrów dostępnych bez ograniczeń dla użytkowników Internetu.

Projekt AFS został jednak przejęty przez IBM i w tym momencie jego przyszłość jest – mówiąc delikatnie – niepewna.

# Rozdział 24

# DFS

## 24.1 Początek

W końcu lat osiemdziesiątych Open Software Foundation, w czasie prac nad DCE (Distributed Computing Environment), ogłosił *Request for Technology* na sieciowy system plików. Na wezwanie skutecznie odpowiedzieli autorzy systemu AFS, co doprowadziło do powstania systemu DFS, będącym nowym wcieleniem AFS.

Pod względem koncepcji system DFS jest identyczny ze swoim systemem macierzystym. Różnią się jedynie paroma kwestiami implementacyjnymi.

## 24.2 Różnice względem AFS

1. Zastosowanie VFS.

   DFS zintegrowano z VFS, dzięki czemu stało się możliwe wykorzystywanie na serwerze systemów plików innych niż dedykowany.

2. Semantyka Uniksowa.

   Dzięki zastosowaniu żetonów, uzyskano semantykę Uniksową – na poziomie operacji `read` i `write`. Serwer, wraz z blokiem danych, przydziela klientowi żeton, odpowiedni do operacji, którą chce wykonać. Serwer może również odebrać klientowi żeton. Posiadanie żetonu gwarantuje, że znajdujące się u użytkownika dane są aktualne, a wykonywane operacje nie kolidują z innymi wykonywanymi w sieci.

3. Wyodrębnienie serwerów dla pewnych usług.

   W DFS wyodrębniono następujące serwery (niekoniecznie pracujące na osobnych maszynach):

   - Serwer zestawu plików, implementujący obsługę na zestawach plików (odpowiednik woluminów z AFS), np. ich migrację.

- Serwer uwierzytelniający, realizujący uwierzytelnianie w oparciu o system Kerberos.
- Serwer zwielokrotnienia, dbający o replikację na różnych serwerach tego samego zestawu plików – tworzone są kopie w wersji tylko do odczytu, aby skrócić czas dostępu do popularnych w sieci danych.

## 24.3 Podsumowanie

System DFS okazał się być bardzo skomplikowanym, dalece bardziej niż AFS. Wprowadzono szereg rozwinięć, które dołożyły dużo obowiązków do pracy serwera DFS (np. zarządcę żetonów). Przez to system DFS nie doczekał się wsparcia w wielu systemach operacyjnych. Co więcej, projekt DCE (którego częścią jest DFS), został zamknięty na początku lat dziewięćdziesiątych. Należy się spodziewać, że pamięć po systemie DFS zupełnie zaginie.

# Rozdział 25

# Coda

## 25.1 Projekt Coda

Projekt Coda został zapoczątkowany na Carnegie Mellon University, uczelni macierzystej systemu AFS (p. 23). Coda jest zaawansowanym sieciowym systemem plików o wielu interesujących właściwościach. Oryginalnie zaprojektowany na Mach 2.6, został później przeniesiony na platformy NetBSD, Linux, FreeBSD, Windows 95.

Dużą zaletą systemu jest bezpłatny dostęp do jego kodów źródłowych (na licencji GPL) – najnowsza wersja została opublikowana 17.10.2003.

## 25.2 Ważne założenia i ciekawe konsekwencje

Przy tworzeniu systemu Coda podjęto wiele istotnych założeń projektowych. Podstawowym celem, przyświecającym twórcom projektu było zapewnienie stałego dostępu do danych.

### 25.2.1 Założenia projektowe

- Nad jednym plikiem w jednym momencie pracuje zazwyczaj co najwyżej jeden użytkownik.
- Współdzielone pliki są zazwyczaj stosunkowo niewielkie.
- Serwery sieciowe są podatne na awarie.
- Użytkownik chce pracować niezależnie od awarii serwera.

### 25.2.2 Konsekwencje decyzji projektowych

- Awarie serwerów są (zazwyczaj) przezroczyste dla użytkowników.

- Użytkownicy mają możliwość pracy off-line (np. po odłączeniu od sieci laptopa czy palmtopa, albo po zakończeniu połączenia telefonicznego z siecią Coda).

- Konflikty wersji plików mogą wymagać interwencji użytkownika, jeśli kilku jednocześnie pracowało nad tym samym pliku w trybie off-line, to po odzyskaniu dostępu do serwera muszą jakoś uzgodnić wspólną wersję.

## 25.3 Architektura Cody

**Koncepcja sieci.** Sieć Coda jest zbudowana w oparciu o dowolną liczbę współpracujących serwerów i dowolną liczbę korzystających z nich komputerów użytkowników. Komputery tę tworzą pojedynczą komórkę. Bazą danych informacji o woluminach, dostępnych w komórce, zarządza SCM (System Control Machine).

Zasoby dostępne w komórce są replikowane na kilka serwerów tak, aby w przypadku awarii jednego z nich, użytkownik nadal miał dostęp do żądanych danych. Wymaga to rozsyłania zmian w plikach do wszystkich serwerów, posiadających dany plik.

Podobnie jak w AFS, plik grupowane są w woluminy, które są podstawową logiczną jednostką danych, którymi operują serwery między sobą.

**Serwery.** Konkretny wolumin jest przechowywany w sieci przez pewną grupę serwerów (VSG – Volume Storage Group). W danym momencie użytkownik może „widzieć" w sieci wszystkie lub (z przyczyn technicznych) jedynie pewien podzbiór tych serwerów. Grupę dostępnych serwerów nazywa się AVSG (Available VSG).

**Klient.** Na komputerze klienckim pamiętane są kopie używanych plików sieciowych. Jeśli użytkownik odwołuje się do pliku, którego nie ma w swojej pamięci podręcznej, kontaktuje się z którymś z serwerów AVSG dla danego woluminu i otrzymuje żądane informacje wraz z obietnicą powiadomienia w przypadku zmian w pliku.

Część systemu Coda, znajdująca się po stronie klienta, nosi nazwę Venus. Kontaktuje się ona z pracującym na serwerze procesem Vice.

**Wektory wersji.** Wektor wersji (CVV – Coda Version Vector) jest zbiorem liczb, które określają którą wersję danego pliku posiada dany serwer z grupy VSG odpowiedniego woluminu. Dzięki temu zbiorowi danych możliwe jest ustalanie, który z serwerów posiada aktualną wersję danych oraz wykrywanie konfliktów.

Problem z konfliktami wersji pojawia się wówczas, gdy użytkownicy modyfikujący konkretny plik, posiadają dla niego rozłączne zbiory AVSG. Każdy z nich jest w stanie wysłać nową wersję pliku jedynie do komputerów, do których ma dostęp. Po zakończeniu awarii, gdy serwery uzyskają łączność pomiędzy sobą, nie będą one w stanie uzgodnić nowej wersji pliku bez interwencji użytkowników. Rozpatrzmy przebieg tego procesu na następującym przykładzie:

1. Plik P jeden jest przechowywany na serwerach S1 oraz S2.
2. Użytkownicy U1 i U2 pobierają najnowszą wersję P i rozpoczynają równoległą pracę nad nim.
3. W wyniku awarii sieci, przerwana zostaje komunikacja pomiędzy podsiecią z U1 i S1 a podsiecią z U2 i S2.
4. Użytkownik U1 kończy pracę nad plikiem P i wysyła go do jedynego serwera, który widzi (S1). Analogicznie U2 wysyła swoją (inną) wersję do S2.
5. Awaria zostaje usunięta.
6. Serwery S1 i S2 komunikują się ze sobą i z analizy wektora wersji dla P „dowiadują się", że zarówno S1 jak i S2 posiadają różne wersje, z których żadna nie jest „bardziej aktualna".

Oczywiście część konfliktów może zostać rozwiązana automatycznie (np. pliki tekstowe mogą być analizowane tak, jak to ma miejsce w przypadku środowiska CVS). W większości przypadków wymagana jest jednak interwencja użytkownika.

**Praca odłączona.** W systemie Coda możliwe jest wykonywanie pracy nad plikiem bez podłączenia do sieci. Wszystkie operacje są wówczas buforowane na komputerze użytkownika i natychmiast po podłączeniu z siecią, ogłaszane w niej.

**Przeglądanie zawartości.** Fizyczna lokalizacja pliku w sieci jest niewidoczna dla użytkownika. Na swoim komputerze ma on katalog `/coda`, w którym znajdują się podłączone woluminy, katalogi i pliki, dostępne w całej sieci (niezależnie od tego, czy jakikolwiek serwer z konkretnym woluminem jest obecnie dostępny w sieci).

## 25.4 Wydajność

System Coda ma bardzo poważne problemy wydajnościowe. Dołączanie kolejnych użytkowników do sieci powoduje drastyczny spadek wydajności, dużo większy niż w przypadku AFS.

Według pewnych testów, po zwiększeniu liczby użytkowników z 5 do 50, przy instalacji z 3 zwielokrotnieniami, wydajność Cody spadła o 70% a AFS o 16%.

W przypadku Cody bez zwielokrotnień można ją porównywać z działaniem NFS. Testy autorów systemu wykazały, że obciążenie serwera Cody może być nawet dwa razy mniejsze niż serwera NFS. Należy jednak pamietać, że związane jest to ze specyficznymi właściwościami Cody, z którymi związane są znaczne różnice koncepcyjne pomiędzy tymi systemami.

# Rozdział 26

# Porównanie RFS-(NFSv3-NFSv4)-Coda

| | **RFS** | **NFSv3** | **NFSv4** | **Coda** |
|---|---|---|---|---|
| **Licencja** | zamknięta | otwarta | otwarta | otwarta |
| **Projekt prowadzi** | AT&T | Sun Microsystems | IETF | Uniwersytet w Carnegie Mellon |
| **Implementacje** | AT&T | otwarte | otwarte | Uniwersytet w Carnegie Mellon |
| **Serwer czy zasób?** | zasób | serwer | serwer | zasób |
| **Semantyka** | Uniksowa | nie Uniksowa | Uniksowa (?) | bardzo nie Uniksowa |
| **Blokady** | tak | nie | tak | nie |
| **Praca jednoczesna** | tak, bezproblemowo | tak, potencjalnie problematyczne | tak, bezproblemowo (?) | nie |
| **Przezroczyste migracje** | nie | nie | tak (?) | tak |
| **Przenośność** | nie (tylko System V) | tak | tak | tak (raczej tylko środowiska Uniksowe) |

# Rozdział 27

# Przyszłość

Nie ulega wątpliwości, że standardem w zakresie sieciowych systemów plików jest NFS. Wygrywa on dzięki następującym cechom:

1. Licencjonowanie.

   Firma Sun Microsystems publikuje pełne specyfikacje kolejnych wersji protokołu NFS. Zachęca to do korzystania z tego właśnie protokołu.

2. Prostota implementacji.

   W protokole NFS uzyskano dużą przejrzystość całego systemu; opiera się on o określone standardy, co szalenie ułatwia jego implementowanie.

3. Przenośna architektura.

   NFS tworzony był z założeniem, że musi on móc być wykorzystany w dowolnym systemie operacyjnym i w dowolnej architekturze komputera.

Nie da się jednak ukryć pewnych jego wad (nie jest skalowalny, jego semantyka nie jest zgodna z Uniksową, ma niską wydajność przy dużych systemach). Mimo wszystko jego zalety przeważają, a przyszłość (NFSv4) wygląda bardzo różowo.

Wydaje się, że najbardziej aktualnymi wymaganiami względem sieciowego systemu plików są obecnie:

- możliwość zastosowania w dużych sieciach (WAN, Internet),
- bezpieczeństwo,
- szybkość.

Projektanci NFSv4 obiecują spełnić te wymagania. Miejmy nadzieję, iż nastąpi to szybko.

# Część V

# Bibliografia i ciekawe linki

**UFS**

- http://uw713doc.sco.com/en/FS_admin/CONTENTS.html
- S.D. Pate, *UNIX Filesystems: Evolution, Design, and Implementation*

**Veritas**

- S.D. Pate, *UNIX Filesystems: Evolution, Design, and Implementation*
- http://uw713doc.sco.com/en/FS_admin/CONTENTS.html

**Ext2 i Ext3**

- olstrans.sourceforge.net/release/ OLS2000-ext3/OLS2000-ext3.html
- e2fsprogs.sourceforge.net/ext2.html

**XFS**

- http://oss.sgi.com/projects/xfs/

**ReiserFS i Reiser4**

- http://www.namesys.com/

**NFSv4**

- http://www.nfsv4.org
- http://www.nluug.nl/events/sane2000/papers/pawlowski.pdf

**Coda**

- http://www.coda.cs.cmu.edu/
- http://www-2.cs.cmu.edu/afs/cs/project/coda/Web/docs-coda.html

**Środowisko rpcgen**

- http://www.cc.gatech.edu/classes/cs4431_98_winter/rpc/intro.ps
- http://www.cc.gatech.edu/classes/cs4431_98_winter/rpc.html
- http://www.cisco.com/univercd/cc/td/doc/product/software/ioss390/ios390rp/rprpcgen.htm
- http://netbula.com/oncrpc/rpcgen.html

**RPC**

- www.xmlrpc.com

**Bibliografia do tematu**


- U. Vahalia, *Jądro systemu UNIX*
- M.J. Bach, *Budowa systemu operacyjnego UNIX*
- B. Goodheart, J. Cox, *Sekrety magicznego ogrodu*
- S.D. Pate, *UNIX Filesystems: Evolution, Design, and Implementation*