Akademia Górniczo-Hutnicza
im. Stanisława Staszica
w Krakowie

Praca magisterska

System plików działający poza jądrem systemu operacyjnego Linux

Jan Wróbel

Kierunek: Informatyka
Specjalność: Systemy rozproszone i sieci komputerowe

Promotor
dr hab. inż. Krzysztof Boryczko, prof. n. AGH

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

Kraków 2007

Spis treści

Rozdział 1. Wstęp 3
Rozdział 2. Cel, zakres i streszczenie pracy 4
Rozdział 3. Podstawowe pojęcia 5
3.1. Rzeczywisty system plików 5
3.2. Wirtualny system plików 5
Rozdział 4. Architektura systemu plików Linuksa 7
4.1. Obiekty systemu plików 7
4.1.1. I-węzeł 7
4.1.2. Plik 8
4.1.3. Superblok 8
4.1.4. Pamięć podręczna nazw 9
4.1.5. Pamięć podręczna i-węzłów 9
4.1.6. Pamięć podręczna buforów 9
4.1.7. Vfsmount 9
4.2. Komunikacja z systemem plików 9
Rozdział 5. Ewolucja systemów plików 12
5.1. System plików do obsługi konta email 13
5.2. Inne przykłady abstrakcyjnych systemów plików 13
Rozdział 6. Ograniczenia tradycyjnej architektury 15
Rozdział 7. Rzeczywisty system plików poziomu użytkownika 17
7.1. FUSE 17
Rozdział 8. System plików poziomu użytkownika 20
8.1. Wirtualizacja 20
8.2. User Mode Linux 21
8.3. Mechanizmy Linuksa umożliwiające implementacje systemu plików poziomu użytkownika 23
8.3.1. Wywołanie systemowe *ptrace()* 25
8.3.2. Ograniczenia funkcji ptrace() 26
8.3.3. Zamiana bibliotek ładowanych dynamicznie 26
8.3.4. Jawny wirtualny system plików poziomu użytkownika 27
Rozdział 9. Out of Kernel FileSystem 30

9.1. Zasada działania 30
9.2. Jądro OKFS 31
9.3. Wirtualny system plików 35
9.3.1. Delegacja wywołań 35
9.3.2. Zarządzanie obiektami systemu plików 38
9.3.3. Ograniczenie złożoności rzeczywistych systemów plików 40
9.3.4. Cachowanie informacji 40
9.4. Rzeczywiste systemy plików 41
9.4.1. LocalFS 43
9.4.2. SHFS 46
Rozdział 10. Efektywność OKFS 49
10.1.Metodologia testowania 50
10.2.Czas wykonania procesów obliczeniowych 50
10.3.Funkcje systemowe 51
10.4.Długotrwałe funkcje systemowe 53
10.4.1.Operacje zapisu 54
10.4.2.Zapis na sieciowym systemie plików 58
10.4.3.Operacje odczytu 59
10.4.4.Odczyt z sieciowego systemu plików 61
10.5.Wnioski 63
Rozdział 11. Ograniczenia OKFS 65
Rozdział 12. Stan prac nad OKFS 67
12.1.Dalszy rozwój systemu. 68
Rozdział 13. Podsumowanie 70
Bibliografia 71
Bibliografia 71

Rozdział 1

Wstęp

Systemy plików przeszły długą drogę ewolucji. W prostych systemach operacyjnych, takich jak DOS, każdy program mógł uzyskać bezpośredni dostęp do dysku i sam odpowiadał za zapis i odczyt danych w odpowiednim formacie [1]. UNIX wprowadził zupełnie inną architekturę systemu plików. Dostęp do plików odbywa się za pośrednictwem kodu systemu operacyjnego w taki sam sposób, bez względu na to gdzie fizycznie przechowywane są pliki. System operacyjny może zapisywać pliki na lokalnym dysku twardym posiadającym partycję, na której znajduje się jeden z wielu obsługiwanych systemów plików. Dostępne są także sieciowe systemy plików, składujące dane na serwerach. Charakteryzują się one różnym stopniem zaawansowania technologicznego oraz wynikającymi z niego poziomem bezpieczeństwa i wydajnością.

Ostatnio coraz większą popularność zyskują systemy plików, ukrywające bardzo złożone operacje za jednolitym interfejsem operacji na plikach. Trudności w implementacji takich złożonych operacji na poziomie jądra systemu operacyjnego wymusiły konieczność przeniesienia systemu plików w taki sposób, aby działał w całości lub przynajmniej częściowo poza jądrem systemu operacyjnego. Istotną zaletą takich systemów plików jest to, że zarządzać nimi może zwykły użytkownik, nie posiadający uprawnień administratora. Wpływa to na znaczne zwiększenie wygody pracy użytkowników. Ma także istotny, pozytywny wpływ na bezpieczeństwo systemu operacyjnego jako całości, gdyż ogranicza ilość kodu działającego w trybie uprzywilejowanym procesora. Dodatkowo, dzięki rosnącym możliwościom sprzętu, narzut czasowy na wykonywanie operacji z użyciem takiego systemu plików, najczęściej nie stanowi dzisiaj istotnego ograniczenia.

Rozdział 2

Cel, zakres i streszczenie pracy

Celem niniejszej pracy jest zaprojektowanie oraz implementacja systemu plików dla systemu operacyjnego Linux - „Out of Kernel FileSystem" (OKFS), działającego całkowicie w przestrzeni użytkownika.

W pracy przedstawione zostaną zalety takiej architektury, zastosowania, które pozwala zaimplementować w łatwy i elegancki sposób, a które nie są dostępne w architekturze tradycyjnej. Przedyskutowane zostaną wyniki testów wydajności stworzonego systemu, a także ograniczenia zaproponowanego rozwiązania i możliwości ich minimalizacji. Omówiony zostanie obecny stan prac nad OKFS oraz plany jego dalszego rozwoju.

Zanim jednak przedstawiony zostanie OKFS, wyjaśniona zostanie terminologia używana w pracy. Następnie omówione zostaną dostępne obecnie rozwiązania, umożliwiające operacje na plikach w Linuksie, i ich ogólna architektura. Wytłumaczenie idei OKFS wymaga także krótkiego omówienia metod wirtualizacji, ze szczególnym uwzględnieniem projektu User Mode Linux, czyli systemu operacyjnego Linux działającego jako proces pod kontrolą systemu operacyjnego Linux. Zostaną przedstawione i ocenione mechanizmy Linuksa, które umożliwiają implementację rozwiązań takich jak system plików w przestrzeni użytkownika.

Rozdział 3

Podstawowe pojęcia

W bardzo szybko rozwijającej się dyscyplinie naukowej, jaką jest informatyka, pojawiają się trudności ze „zgrabnym” tłumaczeniem pojęć i terminów. Dodatkowym utrudnieniem są niejednoznaczności. Przykładem może być tutaj termin *system plików*, który zarówno w polskiej, jak i angielskiej wersji, jest używany zamiennie do określenia różnych elementów. Zatem:

3.1. Rzeczywisty system plików

Rzeczywisty system plików to realizacja jednego, konkretnego systemu plików. Przykłady rzeczywistych systemów plików to:

- ext2, ReiserFS, XFS, JFS - systemy plików przechowujące dane na lokalnym dysku.
- NFS, SMBFS, Coda, AFS - systemy plików umożliwiające dostęp do danych poprzez sieć.
- TMPFS, RAMFS - systemy plików przechowujące dane w pamięci operacyjnej komputera.

Każdy rzeczywisty system plików składa się z kilku elementów. Pierwszym, dla lokalnego systemu plików, jest kod implementujący operacje tego systemu, który jest zazwyczaj częścią jądra systemu operacyjnego. Może zostać zaimplementowany jako kod wkompilowany lub moduł ładowany dynamicznie. Drugą część stanowią struktury danych, jakie system ten zapisuje na dysku. Sieciowe systemy plików definiują dodatkowo protokoły zdalnego dostępu do usług, jakie oferują.

3.2. Wirtualny system plików

Wirtualny system plików (VFS) [3] to warstwa abstrakcji, będąca częścią jądra systemu operacyjnego, umożliwiająca procesom użytkownika dostęp do wszystkich rzeczywistych systemów plików w jednolity sposób. Procesy użytkownika, w celu realizacji operacji na plikach, komunikują się tylko z wirtualnym systemem plików, który

deleguje te operacje do odpowiednich rzeczywistych systemów plików. Czasami pojęcia „wirtualny system plików" używa się do określenia systemów plików, które przechowują dane w pamięci RAM. W tej pracy termin ten nie będzie wykorzystywany w takim znaczeniu.

Pojęciem „system plików" określa się często całość kodu realizującą operacje na plikach, czyli wirtualny system plików i rzeczywiste systemy plików. Ilekroć w tej pracy nie będzie jednoznacznie wynikało z kontekstu, do czego odnosi się określenie „system plików", zostanie to doprecyzowane.

Rozdział 4

Architektura systemu plików Linuksa

W rozdziale tym zostanie omówiona architektura systemu plików Linuksa, której uproszczony schemat przedstawiono na rysunku 4.1. Opisane zostaną także podstawowe obiekty wykorzystywane przez system plików oraz wywołania służące do komunikacji.

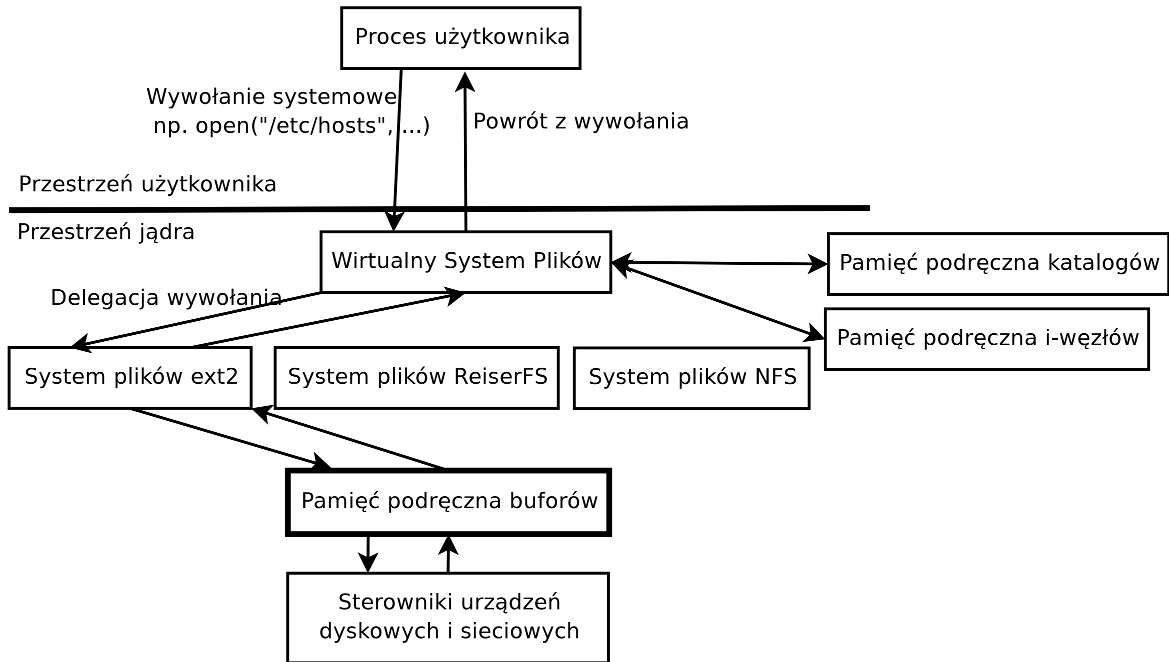
Przy tworzeniu tego rozdziału nieocenionym źródłem informacji była książka „LINUX systemy plików” autorstwa Moshe Bar [2]. Architektura systemu plików Linuksa jest w niej przedstawiona znacznie bardziej szczegółowo niż w tej pracy, zawierającej jedynie ogólną koncepcje. System plików Linuksa jest napisany w czystym języku C, pomimo tego kod ten wykorzystuje wiele paradygmatów programowania obiektowego, takich jak dziedziczenie czy funkcje wirtualne.

4.1. Obiekty systemu plików

Budowa projektowanego systemu plików w pewnym zakresie odpowiada budowie systemu plików Uniksa. Pojawią się w nim więc pewne obiekty dobrze znane z tego systemu. Przyjrzyjmy się najbardziej podstawowym z nich oraz funkcjom, jakie spełniają.

4.1.1. I-węzeł

Podstawowym obiektem systemu plików jest I-węzeł, który przechowuje informacje o plikach, katalogach, dowiązaniach symbolicznych, a także innych elementach systemu plików, jak na przykład plikach urządzeń czy potokach nazwanych. Każdy plik jest reprezentowany przez jeden i-węzeł. VFS nie rozróżnia różnych obiektów, które reprezentuje i-węzeł, jest to zadanie rzeczywistych systemów plików. Każdy i-węzeł posiada numer, który jest unikalny w obrębie systemu plików, do którego należy obiekt. reprezentowany przez ten i-węzeł. Plik jest więc jednoznacznie identyfikowany poprzez system plików, do którego należy, oraz przez numer i-węzła. W jądrze Linuksa i-węzły



Rysunek 4.1. Uproszczony schemat architektury systemu plików Linuksa.

są reprezentowane poprzez strukturę *struct inode*. Struktura ta zawiera listę operacji, które można wykonać na i-węźle (*struct indode_operations*).

4.1.2. Plik

Pliki to obiekty, z których można czytać i do których można zapisywać dane. W jądrze Linuksa reprezentowane są poprzez strukturę *file*. Nie należy utożsamiać obiektu pliku z fizycznym plikiem, np. takim, który znajduje się na dysku. Obiekt *file* jest bardziej podobny w swojej funkcji do deskryptora pliku, jaki definiuje biblioteka C. Jednemu plikowi na dysku, a więc i-węzłowi, który ten plik reprezentuje, może odpowiadać wiele struktur *file*. Każda z nich może być otwarta przez inny proces lub nawet przez ten sam, ale w innym celu (np. do zapisu w kilku różnych miejscach pliku). Plik, w odróżnieniu od i-węzła, posiada informację o stanie, takie jak tryb otwarcia pliku, uid procesu, który otworzył ten plik, pozycja następnego zapisu w pliku itd. Pliki mogą także odnosić się do obiektów nie posiadających i-węzłów, np. do potoków nienazwanych.

4.1.3. Superblok

Superblok reprezentuje zamontowany system plików. W jądrze struktura superbloku nazywa się *struct super_block*. Zamontowany system plików to zbiór i-węzłów, zawierający jeden węzeł główny (*root*). Odszukanie pliku w systemie plików zaczyna się od tego węzła głównego i przechodzi kolejno w dół struktury katalogu, aż do momentu dotarcia do szukanego pliku. W superbloku zapisane są informacje o systemie plików, który reprezentuje, takie jak na przykład znacznik określający, czy system plików jest zamontowany w trybie tylko do odczytu.

4.1.4. Pamięć podręczna nazw

Dostęp do i-węzłów systemu plików odbywa się za pomocą nazw. Przejście przez drzewo katalogów, od jego korzenia do poszukiwanego i-węzła, to proces czasochłonny. Wymaga przeczytania zawartości wszystkich katalogów znajdujących się po drodze, jak również podążenia za wszystkim napotkanymi dowiązaniami symbolicznymi. Przekłada się to na dużą ilość operacji, powodujących duże opóźnienie, zwłaszcza dla sieciowych systemów plików. W celu minimalizacji tego opóźnienia, wirtualny system plików przechowuje pamięć podręczną ostatnio wykorzystywanych nazw - *dcache*. Pamięć ta odwzorowuje nazwę na i-węzeł reprezentujący plik o podanej nazwie. Pamięć *dcache* odzwierciedla przedrostek całego drzewa katalogów. Oznacza to że, jeżeli i-węzeł danego systemu plików znajduje się w pamięci podręcznej, również wszyscy jego przodkowie, aż do węzła głównego tego systemu plików, muszą również znajdować się w tej pamięci. Informacja o każdym i-węźle, dla którego w danej chwili jest otwarty jakikolwiek plik, znajduje się zawsze w pamięci *dcache*.

4.1.5. Pamięć podręczna i-węzłów

W celu przyspieszenia dostępu do ostatnio używanych i-węzłów, wirtualny system plików przechowuje te i-węzły w pamięci podręcznej. Pamięć podręczna i-węzłów jest zaimplementowana jako tablica hashująca, w której do obliczenia wartości indeksu elementu jest używany numer i-węzła oraz identyfikator urządzenia, na którym znajduje się system plików, zawierający ten i-węzeł.

4.1.6. Pamięć podręczna buforów

Bufory służą do magazynowania zawartości fizycznych dysków w celu skrócenia czasu odczytu i zapisu danych. Dane, których żąda system plików, są odczytywane z dysku, tylko jeśli nie znajdują się w pamięci podręcznej buforów lub jeśli w pamięci znajduje się ich nieaktualna wersja. Po każdym odczycie z dysku zawartość odczytanych bloków danych jest dodawana do pamięci podręcznej. Za zarządzanie zwalnianiem pamięci podręcznej w Linuksie odpowiada demon działający na poziomie jądra - *bdflush* lub *pdflush*.

4.1.7. Vfsmount

Struktura *vfsmount* jest używana przez wirtualny system plików do przechowywania informacji o zamontowanych, rzeczywistych systemach plików. Zawiera m.in. takie pola jak urządzenie, na którym system jest zamontowany (np. */dev/hda1*), nazwa katalogu montowania czy wskaźnik do superbloku, reprezentującego ten system plików.

4.2. Komunikacja z systemem plików

Procesy poziomu użytkownika uzyskują dostęp do systemu plików poprzez wywołania systemowe, implementowane przez wirtualny system plików. Wirtualny system plików zapewnia więc wspólny interfejs dostępu do plików, niezależnie od tego, na jakim rzeczywistym systemie plików znajduje się dany plik.

Takie podejście bardzo upraszcza implementację programów użytkowych. Dla przykładu, program *cat* wywołuje takie same funkcje systemowe *open()* i *read()* do przeczytania pliku i nie musi martwić się, czy plik ten jest dostępny na lokalnej partycji (np. ext2) czy udostępniony na zdalnym systemie poprzez NFS.

Z drugiej strony, wirtualny system plików ułatwia także programowanie rzeczywistych systemów plików. Dla wielu wywołań systemowych, wirtualny system plików zapewnia domyślną implementację lub wykonuje część niezbędnej pracy, która jest taka sama, niezależnie od rzeczywistego systemu plików, do których odnosi się wywołanie. Dzięki temu rzeczywiste systemy plików muszą implementować tylko tę funkcjonalność, która jest różna dla różnych systemów plików.

Linux 2.6.19 definiuje 318 wywołań systemowych. Definicje te znajdują się w pliku źródłowym Linuksa *include/asm/unistd.h*. Jednak cześć tych wywołań nie została zaimplementowana. Spośród tych wywołań około 100 wykonuje operacje na plikach, katalogach, systemach plików, czyli takie, za które odpowiedzialny jest wirtualny system plików. To sporo, jednak duża część tych wywołań ma bardzo zbliżoną funkcjonalność.

Rekordzistą jest wywołanie *stat(const char *path, struct stat *buf)*, które zwraca status pliku wskazywanego przez *path*. Bardzo podobnie działa *lstat()*, z tą jednak różnicą, iż jeśli *path* jest dowiązaniem symbolicznym, to zwraca jego status, a nie pliku, na który dowiązanie wskazuje. Z kolei *fstat(int filedes, struct stat *buf)* zwraca status pliku wskazywanego przez deskryptor *filedes*. Dodatkowo Linux implementuje wywołania *oldstat()*, *oldfstat()*, *oldlstat()*, które działają dokładnie tak samo, tylko wypełniają przestarzałą wersję struktury *stat* i są zaimplementowane, aby umożliwić działanie starszym programom. Należy jeszcze wspomnieć o wywołaniach *stat64()*, *fstat64()*, *lstat64()*, które działają identycznie, ale umożliwiają pobieranie statusu dla plików większych niż 2GB. Jak widać, 9 wywołań jest odpowiedzialnych za w zasadzie taka samą czynność, co pozwala im współdzielić znaczną cześć implementacji.

Cześć wywołań systemowych jest dosyć egzotyczna i specyficzna dla Linuksa. Programy, które są przeznaczone dla wielu różnych platform lub dla starszych wersji Linuksa, które nie udostępniały tych wywołań, zwykle wykorzystują tylko standardowe funkcje systemu plików. Zostały one opisane w tabeli 4.1.

Tablica 4.1. Podstawowe wywołania systemowe Linuksa, odnoszące się do plików. Wywołania o podobnym zastosowaniu zostały zgrupowane.

read, write	czyta, zapisuje do pliku
open, creat	otwiera lub tworzy plik
close	zamyka plik
link	tworzy twarde dowiązanie
unlink	usuwa dowiązanie do pliku i plik, gdy nie ma więcej dowiązań
execve	wykonuje plik
chdir, fchdir	zmienia bieżący katalog procesu
mknod	tworzy pliki, także specjalne (urządzenia, potoki nazwane)
chmod, fchmod	zmienia prawa dostępu do pliku
chown, lchown	zmienia właściciela pliku
lseek	zmienia pozycje odczytu i zapisu w pliku
mount, umount	montuje, odmontowywuje system plików
access	sprawdza, czy proces ma prawa odczytu i zapisu do pliku
sync	zrzuca zawartość pamięci podręcznej i buforów na dysk
rename	zmienia nazwę pliku
mkdir, rmdir	tworzy, usuwa katalog
dup, dup2	tworzy kopie deskryptora pliku
ioctl	zmienia parametry urządzeń dostępnych poprzez pliki specjalne; komunikuje się ze sterownikami urządzeń
fcntl	wykonuje różne operacje na deskryptorze plików np. blokowanie
chroot	zmienia katalog główny procesu
ustat	zwraca informacje o zamontowanym systemie plików
select, poll, ppoll	monitoruje i informuje o zmianie stanu deskryptorów pliku
symlink	tworzy dowiązanie symboliczne
readlink	odczytuje, na co wskazuje dowiązanie symboliczne
readdir, getdents	odczytuje zawartość katalogu
mmap, mmap2, remap, munmap	mapuje, usuwa mapowanie pliku do pamięci
truncate, ftruncate	ustawia długość pliku
statfs, fstatfs	zwraca informacje o zamontowanym systemie plików
stat, lstat, fstat	zwraca informacje o podanym pliku
fsync, fdatasync	synchronizuje z dyskiem stan pliku w jądrze
sysfs	zwraca informacje o systemach plików obsługiwanych przez jądro
flock	zakłada blokadę na pliku
msync	synchronizuje z dyskiem plik zmapowany do pamięci procesu
readv, writev	czyta, zapisuje dane do wielu buforów
getcwd	pobiera bieżący katalog procesu

Rozdział 5

Ewolucja systemów plików

Jedną z głównych cech architektury Uniksa jest paradygmat „wszystko jest plikiem". Poprzez pliki w Uniksie można uzyskiwać dostęp do różnego typu urządzeń, kolejek komunikatów czy gniazd nazwanych. Aby wydrukować dokument tekstowy, wystarczy przekierować go do pliku reprezentującego drukarkę. Linux udostępnia system plików proc, który poprzez pliki reprezentuje i umożliwia manipulację wewnętrznym stanem jądra systemu operacyjnego. Zapis danych do tych plików nie powoduje zapisu na jakimkolwiek fizycznym urządzeniu. Powoduje natomiast modyfikację jakiejś właściwości jądra. Takie abstrakcyjne podejście do plików jest bardzo wygodne.

W przeszłości systemy plików służyły jedynie do organizacji danych na lokalnym dysku. Następnie koncepcja ta została rozszerzona. Stworzono sieciowe systemy plików, ukrywających przed użytkownikiem złożone operacje sieciowe, dając mu możliwość manipulowania zdalnymi plikami tak, jakby znajdowały się na lokalnym dysku. W przeszłości jednak głównym wymaganiem stawianym systemowi plików, była jego efektywność. Sieciowe systemy plików wykorzystywały specjalizowane protokoły, takie jak NFS, aby zoptymalizować czas operacji. Podejście to sprawdza się dobrze w przypadku dużych serwerów, i innych maszyn, dla których wydajność jest niezwykle istotna. Jednak, na co dzień, zwykli użytkownicy nie przesyłają tak ogromnej ilości danych. Dla nich najważniejsza jest prostota użytkowania i konfiguracji. Dodatkowo, prędkość połączeń sieciowych i komputerów wzrosła na tyle, że najczęściej wykorzystanie mniej efektywnych rozwiązań nie wprowadza opóźnień zauważalnych dla zwykłego użytkownika. Te czynniki przyczyniły się do ewolucji systemów plików, którą obserwujemy dzisiaj.

Ewolucja ta objawia się głównie w dalszym rozszerzaniu paradygmatu „wszystko jest plikiem". System plików staje się warstwą abstrakcji, która ukrywa złożone operacje, a także jest w stanie prezentować najróżniejsze zasoby oraz operacje na nich jako pliki i operacje na plikach.

5.1. System plików do obsługi konta email

Za przykład posłuży nam system plików, będący abstrakcyjnym interfejsem do webowej aplikacji emailowej. Aplikacje takie są ostatnio bardzo popularne i często udostępniają użytkownikom za darmo dosyć pokaźną przestrzeń dyskową. Tradycyjnie, użytkownik loguje się do takiej aplikacji poprzez stronę WWW i wykonuje wszystkie operacje, używając przeglądarki. Ukrycie tych operacji za abstrakcyjnym interfejsem systemu plików wprowadza nowe, bardzo ciekawe możliwości. Użytkownik może zamontować swoje konto email, np. *jimi@little.wing.art*, do lokalnego katalogu *email* w swoim katalogu domowym (czyli do *~/email*). System plików może udostępnić wszystkie nowe listy w *~/email/new*, nowe listy od Ani mogą być dostępne w *~/email/new/ania*, natomiast maile zakwalifikowane jako spam będą w *~/email/new/spam*. Zapis do pliku *email/sent* może spowodować wysłanie listu. *~/email/storage* może być katalogiem wykorzystywanym do przechowywania danych na zdalnym serwerze.

Co daje takie podejście? Korzyści jest bardzo wiele. Najważniejsza to wspólny interfejs dostępu dla wielu różnych aplikacji. Jeśli użytkownik chce wyszukać interesujące go listy, może wykorzystać standardowe programy *find*, *grep* lub jakiś inny, który lubi. Może tworzyć skrypty wykonujące automatycznie operacje na listach. Wykorzystywany język skryptowy może być dowolny, byleby tylko był dostępny na maszynie, na której użytkownik zamontował swoje konto email jako system plików. Może np. wykorzystać demona *at* do zlecenia wysłania listu o danej godzinie. Możliwości jest mnóstwo. Prosta aplikacja WWW może stać się całkiem potężnym i wygodnym narzędziem, bez wysiłku ze strony programistów tej aplikacji.

Czy aby takie montowanie konta email było możliwe, twórcy aplikacji WWW muszą udostępnić taką możliwość? Czy muszą zaimplementować jakiś protokuł dostępu umożliwiający te wszystkie operacje? Cała siła tej koncepcji ukryta jest w fakcie, iż nie jest to wymagane. System plików wykorzystuje, do wykonywania wszystkich operacji, standardowy protokuł dostępu do skrzynki poprzez HTTP. Załóżmy, że użytkownik Jimi chce wykorzystać swoje konto email jako sieciowy dysk do przechowywania danych, do których chce mieć dostęp z różnych maszyn, ale twórcy web aplikacji nie przewidzieli w ogóle takiej możliwości. Użytkownik zapisuje plik w katalogu *~/email/storage*. System plików, żeby dokonać zapisu, łączy się z aplikacją WWW do wysyłania emaili. Wykorzystuje standardowy protokół HTTP. Następnie zleca aplikacji wysłanie listu, w którym jako adres odbiorcy podaje adres właściciela konta, a jako załącznik plik zapisany przez użytkownika w katalogu *storage*. W ten sposób plik ten zostanie zapisany na zdalnym serwerze. Kiedy użytkownik będzie go chciał odczytać, system plików odczyta po prostu zawartość załącznika do odpowiedniego listu. Zrobi to jednak w sposób niewidoczny dla użytkownika, który będzie miał wrażenie, że otworzył taki sam plik, jak każdy inny.

5.2. Inne przykłady abstrakcyjnych systemów plików

Podany przykład nie jest tylko rozważaniem na temat możliwości ewolucji systemów plików w przyszłości. Takie rozwiązania już istnieją. Przykładem jest GmailFS [5], umożliwiający dostęp do darmowego konta pocztowego, oferowanego przez firmę Google. Poniżej znajduje się kilka innych, mniej lub bardziej egzotycznych przykła-

dów takich abstrakcyjnych systemów plików. Dokonują one mapowania najróżniejszych struktur danych oraz operacji na nich na pliki. Wszystkie te systemy plików zostały już zaimplementowane:

- CvsFS - dostęp do systemu kontroli wersji CVS,
- BitTorrent File System - współdzielenie plików przez protokuł BitTorrent,
- Bluetooth File System - reprezentowanie poprzez pliki urządzeń bluetooth będących w zasięgu i mapowanie operacji na tych plikach na operacje na urządzeniach,
- DBToy - dostęp do relacyjnych baz danych,
- playlistfs - prezentacja i zarządzanie listami z plikami audio,
- BloggerFS - dostęp do serwisu Blogger, tworzenia i zarządzania wpisami w blogu,
- FUSEPod - dostęp do iPoda,
- SieFS - dostęp do pamięci telefonu Siemens,
- EncFs - szyfrowanie danych zapisywanych na lokalnym dysku.

Jeff Dike w swojej książce [4] rozszerza jeszcze bardziej przedstawioną tu koncepcje, prezentując w jaki sposób aplikacje mogą eksportować swój wewnętrzny stan jako system plików.

Niestety, omówiona już tradycyjna architektura systemu plików, przedstawiana na rysunku 4.1, znacznie ogranicza możliwości tworzenia i wykorzystania siły najróżniejszych systemów plików. W kolejnym rozdziale wyjaśnione zostaną przyczyny takiego stanu rzeczy.

Rozdział 6

Ograniczenia tradycyjnej architektury

W tradycyjnej architekturze zarówno wirtualny system plików jak i wszystkie obsługiwane przez jądro rzeczywiste systemy plików są częścią jądra systemu. Udostępniania przez Linuksa, możliwość dynamicznego ładowania modułów obsługujących rzeczywiste systemy plików, zwiększa trochę elastyczność takiego rozwiązania. Architektura modułowa pozwala dodać i usunąć obsługę systemu pliku, bez konieczności rekompilacji całego jądra i ponownego uruchamiania systemu. Takie rozwiązanie ma jednak nadal kilka poważnych wad.

Pierwsze źródło ograniczeń wynika z konieczności zapewniania bezpieczeństwa całemu systemowi, co w przypadku tradycyjnej architektury systemu plików jest utrudnione. Każdy rzeczywisty system plików działa w trybie uprzywilejowanym procesora, czyli tak naprawdę nie ma żadnych ograniczeń na to, co taki kod może robić. System plików ma dostęp do wszystkich wewnętrznych struktur jądra. Może je dowolnie modyfikować, ma także dostęp do pamięci wszystkich procesów użytkownika, działających w systemie. Co za tym idzie, dodanie obsługi każdego nowego systemu plików wiąże się z potencjalnym zmniejszeniem poziomu bezpieczeństwa całego systemu. Im więcej kodu działa z uprawnieniami jądra, tym z punktu widzenia bezpieczeństwa, gorzej. Oczywistą konsekwencją takiej architektury jest to, że tylko użytkownik uprzywilejowany może dodawać obsługę nowych systemów plików. Znacznie ogranicza to dostęp zwykłych użytkowników do najróżniejszych, wygodnych systemów plików. Jeśli np. użytkownik chce wykorzystać NFS do dostępu do swoich plików na zdalnym serwerze, musi nakłonić administratora, aby dodał obsługę NFS do jądra i zamontował lub nadał mu uprawnienia do montowania interesujących go katalogów lokalnie. Oczywiste jest, że w przypadku dużych systemów, z wieloma użytkownikami, administrator ze względów bezpieczeństwa nie może spełnić oczekiwań wszystkich użytkowników odnośnie tego, jakie systemy plików powinny być udostępnione.

Kolejne źródło poważnych ograniczeń wynika z trudności programowania na poziomie jądra [6]. Kod poziomu jądra nie ma dostępu do żadnych zewnętrznych bibliotek,

nawet biblioteki standardowej. Cała wymagana funkcjonalność musi zostać zaimplementowana od zera, z wykorzystaniem jedynie bardzo ograniczonej biblioteki jądra. Poza tym programista musi znać wewnętrzne interfejsy i struktury danych jądra. Kod taki o wiele trudniej debugować i testować z wykorzystaniem standardowych narzędzi. Każdy błąd może powodować niestabilność pracy czy nawet zawieszenie systemu operacyjnego. Dlatego w tradycyjnej architekturze bardzo trudno byłoby napisać mniej tradycyjne systemy plików. Weźmy na przykład system plików, umożliwiający montowanie zdalnego katalogu z wykorzystaniem protokołu SSH. Jest to bardzo wygodne, ponieważ wiele serwerów zapewnia swoim użytkownikom możliwość logowania poprzez SSH. Jednak implementacja tego protokołu na poziomie jądra byłaby bardzo skomplikowana. Jednym z powodów jest to, że SSH wymaga odpowiedniej infrastruktury do zarządzania kluczami służącymi do autentykacji i szyfrowania. Klucze te przechowywane są w katalogu domowym użytkownika. Ale jądro działa na niższym poziomie abstrakcji, nie ma pojęcia o istnieniu i lokalizacji katalogów użytkownika. Uzyskanie dostępu do zdalnego systemu często wymaga podania hasła. Interaktywny dialog z użytkownikiem jednak też nie należy do funkcji, które powinny być i są łatwo realizowalne na poziomie jądra.

Rozdział 7

Rzeczywisty system plików poziomu użytkownika

W rozdziale tym przedstawiona zostanie koncepcja rzeczywistego systemu plików, działającego na poziomie użytkownika. Rozwiązanie takie pozbawione jest wad architektury tradycyjnej, omówionych w poprzednim rozdziale.

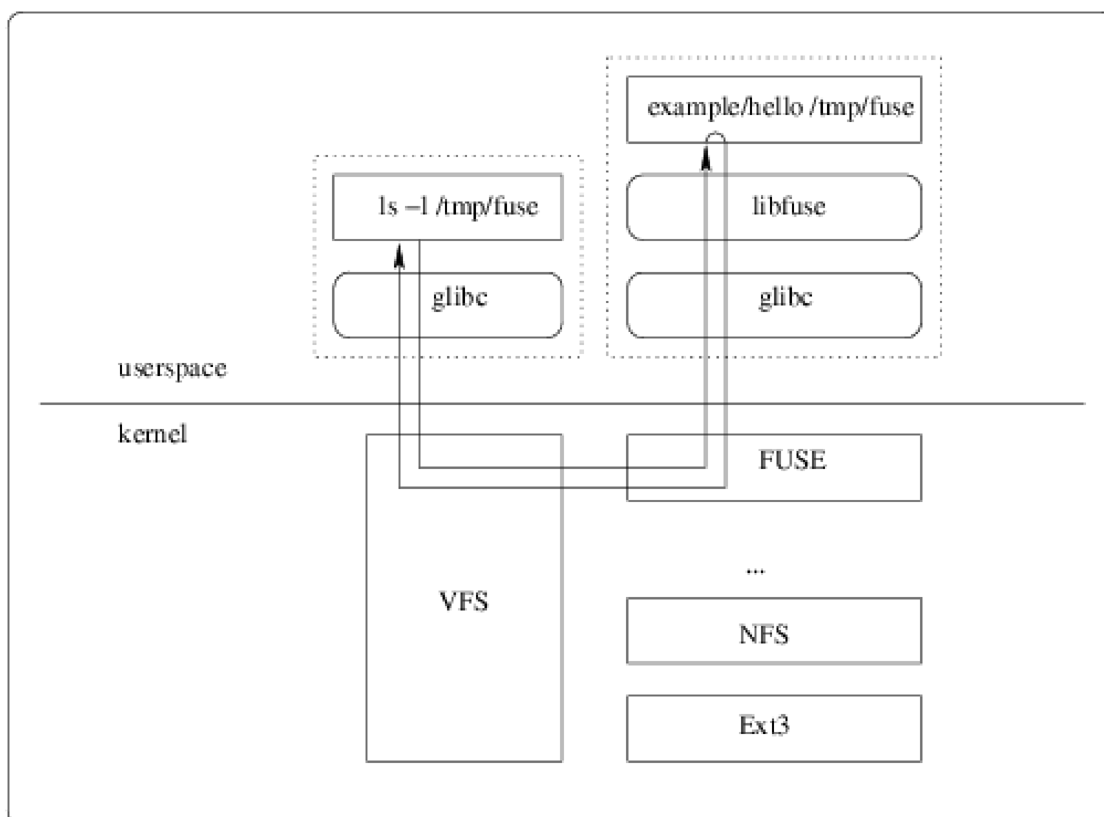
7.1. FUSE

FUSE (*Filesystem in Userspace*) [7] jest najbardziej popularnym projektem umożliwiającym implementowanie rzeczywistych systemów plików na poziomie użytkownika. Architekturę FUSE przedstawia rysunek 7.1. FUSE składa się z 3 podstawowych elementów:

1. modułu jądra, który jest odpowiedzialny za przekazywanie wywołań i ich argumentów do rzeczywistych systemów plików. Pobiera on także wyniki wywołań od systemów plików i przekazuje je do procesu wywołującego.
2. Rzeczywistego systemu plików. Każdy system plików jest procesem działającym w przestrzeni użytkownika i komunikującym się z jądrem za pomocą specjalnego urządzenia */dev/fuse*.
3. Biblioteki *libfuse*. Ukrywa ona przed programistą systemu plików złożoność procesu komunikacji z modułem jądra.

Podstawowe zalety architektury realizowanej przez FUSE są następujące:

- Rzeczywisty system plików nie musi działać z uprawnieniami administratora systemu. Daje to każdemu użytkownikowi możliwość montowania osobistych systemów plików, niewidocznych dla innych użytkowników, bez osłabiania poziomu zabezpieczeń komputera. System plików uruchomiony przez użytkownika działa z jego prawami, czyli może wykonywać tylko takie operacje, jak każdy inny proces uruchomiony przez tego użytkownika.



Rysunek 7.1. Architektura FUSE (ze strony projektu [7]). Przedstawiono ścieżkę przez jaką podąża wywołanie systemowe od procesu wywołującego do procesu implementującego rzeczywisty system plików i z powrotem.

- Rzeczywisty system plików, może wykorzystywać dowolne biblioteki i może być zrealizowany w wielu różnych językach programowania. Natywnym językiem dla FUSE jest C. Obecnie dostępne są interfejsy dla 12tu innych języków, w których można programować rzeczywiste systemy plików.
- Rzeczywisty system plików może być przenośny pomiędzy różnymi systemami operacyjnymi. FUSE jest obecnie dostępny dla Linuksa, FreeBSD i Mac OS X. „Dobrze" napisany rzeczywisty system plików będzie działał bez żadnych zmian w kodzie na wszystkich tych platformach. Jest to ogromna zaleta w porównaniu z tradycyjną architekturą, w której każdy system plików korzysta z interfejsów specyficznych dla jądra systemu operacyjnego, dla którego jest napisany. Kod taki jest praktycznie nieprzenaszalny i wymaga ponownej implementacji dla każdej platformy.
- Możliwość korzystania ze standardowych narzędzi programistycznych przy tworzeniu i debugowaniu rzeczywistego systemu plików.

Istotną wadą tej architektury jest jej mniejsza efektywność. Jak widać na rysunku 7.1, wywołanie systemowe musi przejść dłuższa drogę niż w przypadku standardowej architektury. Jednak, jak już wspomniano wcześniej, dla wielu zastosowań efektywność nie jest priorytetem. Takiej architektury nie można też oczywiście wykorzystać do implementacji systemów plików, które wymagają bezpośredniego dostępu do fizycznych urządzeń.

Rozdział 8

System plików poziomu użytkownika

W FUSE tylko rzeczywiste systemy plików były implementowane jako procesy przestrzeni użytkownika. W rozdziale tym przedstawiona zostanie architektura systemu plików działającego w całości na poziomie użytkownika. W praktyce oznacza to, że zarówno wirtualny system plików jak i rzeczywiste systemy plików działają poza jądrem systemu operacyjnego, zwiększając jeszcze bardziej elastyczność architektury zaproponowanej przez FUSE. Omówione także zostaną warunki, które muszą zostać spełnione, aby implementacja takiego systemu plików była możliwa.

8.1. Wirtualizacja

W celu wyjaśnienia architektury systemu plików działającego całkowicie poza jądrem systemu operacyjnego pomocne będzie pobieżne przyjrzenie się technologii wirtualizacji. Według Wikipedii „wirtualizacja to szerokie pojęcie odnoszące się do abstrakcji zasobów w różnych aspektach komputeryzacji” [8]. W pracy tej sporo miejsca poświecono koncepcji wirtualnego systemu plików, który stanowi warstwę abstrakcji dla operacji na plikach. W programowaniu obiektowym jednym z kluczowych pojęć są funkcje wirtualne, które stanowią warstwę abstrakcji dla konkretnej implementacji operacji, którą definiują.

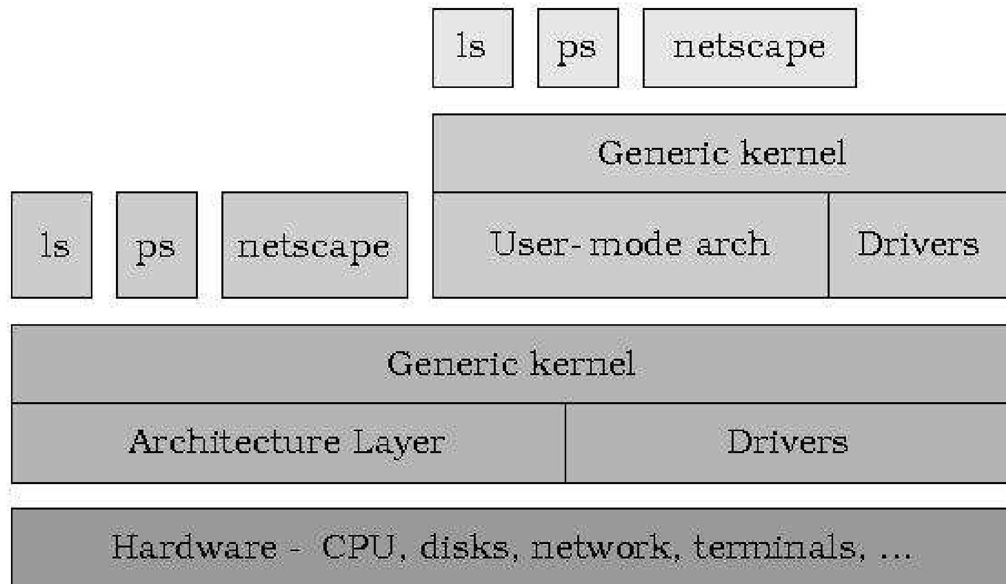
Wirtualizacja jest ukryciem sposobu, w jaki dana operacja jest rzeczywiście realizowana. Ma to często na celu ukrycie złożoności danej operacji, umożliwia działanie na wyższym poziomie abstrakcji. Często wirtualizacja służy symulowaniu jakiegoś środowiska, które jest wymagane do działania danej aplikacji. Symulować można jakąś konkretną maszynę w całości lub tylko pojedyncze zasoby maszyny. Poniżej przedstawiono pokrótce kilka metod wykorzystywanych przy tworzeniu wirtualnych maszyn. Listę tę stworzono na podstawie artykułu [8]. Następnie bardziej dokładnie omówiony zostanie projekt User Mode Linux (UML), gdyż bardzo wiele koncepcji jest wspólnych dla UML i OKFS. Oba te projekty opierają swoje działanie na tym samym mechanizmie jądra:

- emulacja - maszyna wirtualna symuluje warstwę sprzętową w całości, pozwalając na uruchomienie pod swoją kontrolą systemów operacyjnych przeznaczonych na inną architekturę niż ta, na której maszyna wirtualna jest uruchomiona. Takie podejście umożliwia np. testowanie programów przeznaczonych na nowy procesor, jeszcze zanim ten procesor zostanie stworzony. Przykłady takich systemów to Bochs [9] czy QEMU [10].
- Wirtualizacja natywna - maszyna wirtualna symuluje fragment warstwy sprzętowej, wystarczający do uruchomienia pod jej kontrolą systemu operacyjnego przeznaczonego na tą samą architekturę, na której uruchomiona jest maszyna wirtualna. Jest to podejście mniej elastyczne od emulacji, ale o wiele bardziej efektywne. Większość instrukcji procesora jest wykonywanych bezpośrednio przez procesor, a nie przez maszynę wirtualną. Przykłady takich maszyn to VMware [11] czy Parallels Desktop [12].
- Wirtualizacja częściowa - maszyna wirtualna symuluje część warstwy sprzętowej (np. przestrzeń adresową). Pozwala to na izolowanie procesów działających pod kontrolą maszyny wirtualnej, ale nie jest wystarczające do uruchomienia całego systemu operacyjnego pod kontrolą takiej maszyny.
- Wirtualizacja na poziomie systemu operacyjnego - system operacyjny pozwala na uruchamianie poszczególnych aplikacji w izolacji od siebie nawzajem. Każda odizolowana grupa aplikacji wydaje się działać pod kontrolą swojego własnego systemu operacyjnego i nie ma dostępu do zasobów innych odizolowanych grup. Przykłady takiego podejścia to Linux-VServer [13], Solaris Containers [14] czy też FreeBSD Jails [15].
- Wirtualizacja na poziomie aplikacji - tworzone jest środowisko do uruchomienia aplikacji przeznaczonych na daną maszynę wirtualną. Aplikacje takie działają tylko pod kontrolą odpowiedniej maszyny wirtualnej, nie są przeznaczone na żadną rzeczywistą platformę sprzętową. Maszyna wirtualna jest warstwą pomiędzy aplikacją a systemem operacyjnym, zapewniającą przenośność aplikacji. Przykłady to wirtualna maszyna Javy czy Common Language Runtime, będąca częścią Microsoft .NET Framework. Do tej grupy aplikacji niektórzy zaliczają także wszystkie języki interpretowane, gdyż interpreter można traktować jako rodzaj maszyny wirtualnej.

8.2. User Mode Linux

User Mode Linux (UML) [16] to wirtualny system operacyjny stworzony przez Jeffa Dike'a. UML pozwala na uruchomienie Linuksa pod kontrolą Linuksa, jako zwykłego procesu poziomu użytkownika. UML był początkowo rozwijany jako patch na jądro Linuksa, ale od wersji jądra 2.6.9 UML został włączony do oficjalnego drzewa Linuksa jako jedna z dostępnych architektur sprzętowych. W wyniku kompilacji jądra na architekturę UML, powstaje plik wykonywalny ELF. Uruchomienie tego pliku pod Linuksem inicjuje zwykły proces bootowania systemu operacyjnego, z tą różnicą, że system startuje jako zwykły proces pod kontrolą Linuksa-gospodarza.

Programy wykonywane pod kontrolą UMLa nie widzą żadnej różnicy pomiędzy tym środowiskiem, a zwykłym Linuksem. Architektura UML została przedstawiona na rysunku 8.1. UML udostępnia procesom użytkownika taki sam interfejs wywołań systemowych jak zwykły Linux. Różnica polega na tym, że UML nie ma bezpośrednio



Rysunek 8.1. Architektura UML (rysunek ze strony projektu [16])

dostępu do sprzętu, może za to zlecać zadania Linuksowi-gospodarzowi, pod którego kontrolą działa. Odbywa się to za pomocą zwykłych wywołań systemowych.

UML tworzy izolowane środowisko dla procesów uruchomionych pod jego kontrolą. Procesy te nie widzą procesów systemu gospodarza, ani procesów działających pod kontrolą innych instancji UMLa, działających na tej samej maszynie. Super użytkownik na danej instancji UMLa nie musi mieć uprawnień super użytkownika w systemie gospodarza. Super użytkownik ma co najwyżej takie prawa, jakie ma użytkownik, który uruchomił UMLa. Nie oznacza to wcale, że jeśli użytkownik uruchamiający UMLa nie ma praw np. do odczytu pliku */etc/passwd* w systemie gospodarza, to super użytkownik na tej instancji UMLa też nie będzie mógł odczytać pliku */etc/passwd* pod UMLem. Może mieć taką możliwość, gdyż UML tworzy wirtualne środowisko, w którym plik */etc/passwd* nie musi być tym samym plikiem co */etc/passwd* na systemie gospodarza. Może być to plik, do którego użytkownik uruchamiający UMLa ma prawa dostępu, a więc i super użytkownik może mieć do niego dostęp.

UML ma bardzo wiele zastosowań. Pozwala między innymi na konsolidację serwerów. Firmy hostingowe zamiast dostarczać oddzielny fizyczny serwer dla każdego klienta, mogą udostępnić każdemu klientowi oddzielną instancję UML na tym samym serwerze. UML jest także często stosowany do tworzenia honeypotów, czyli pułapek

na włamywaczy komputerowych. Używa się go także do bezpiecznego debugowania aplikacji, jądra Linuksa, czy testowania konfiguracji złożonych instalacji sieciowych.

UML jest systemem operacyjnym działającym całkowicie na poziomie użytkownika, a co za tym idzie, również wirtualny system plików UMLa działa na poziomie użytkownika. Pozwala to użytkownikowi UMLa na ładowanie modułów obsługujących najróżniejsze przydatne dla niego systemy plików, których jądro systemu gospodarza może nie obsługiwać.

Architekturę UMLa można więc uznać za nadzbiór architektury OKFS. OKFS wykorzystuje te same mechanizmy jądra Linuksa co UML. Wszystko na co pozwala OKFS, można wykonać także z użyciem UMLa. Jednak, ponieważ UML jest całym systemem operacyjnym, jest to rozwiązanie o wiele mniej efektywne, wymagające do działania o wiele więcej zasobów niż OKFS. Również konfiguracja UMLa nie jest procesem łatwym. Jest tak samo skomplikowana jak konfiguracja zwykłego systemu operacyjnego. Stosowanie UMLa przez użytkowników, którzy spośród wielu jego funkcji potrzebują jedynie możliwości dostępu do różnych systemów plików, jest bardzo niewygodne a często, ze względu na ograniczone zasoby, wręcz niemożliwe.

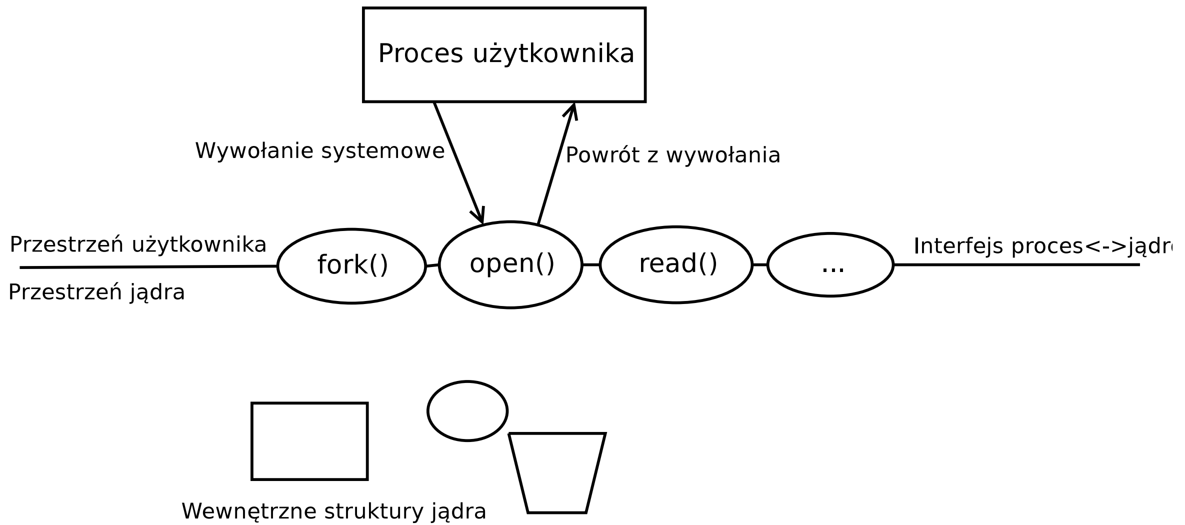
8.3. Mechanizmy Linuksa umożliwiające implementacje systemu plików poziomu użytkownika

Jakie warunki muszą zostać spełnione, ażeby umożliwić implementację takich systemów jak OKFS czy UML? Jedynym sposobem komunikacji procesu użytkownika z systemem operacyjnym są wywołania systemowe. Proces nie ma możliwości bezpośredniego dostępu do wnętrza systemu operacyjnego. Koncepcję tą prezentuje rysunek 8.2. Jeśli proces chce wykonać jakąś operację np. otwarcie pliku, wywołuje odpowiednią funkcje jądra i odbiera od niej wynik. Wirtualne środowisko w rodzaju OKFSa powinno więc mieć możliwość przejęcia wszystkich wywołań systemowych procesu, zanim zostaną one zrealizowane przez jądro Linuksa.

Aby wywołać jakąś funkcję Linuksa na architekturze x86, proces umieszcza jej numer w rejestrze EAX, a argumenty w kolejnych rejestrach (EBX, ECX, EDX, ESI, EDI). Jeśli wywołanie ma więcej niż 5 argumentów, do ich przekazania wykorzystywany jest stos. Następnie proces wywołuje przerwanie o numerze 0x80, co powoduje przekazanie sterowania do systemu operacyjnego. System operacyjny pobiera numer wywołania i jego argumenty z rejestrów, wykonuje odpowiednią akcję i umieszcza jej wynik w rejestrze EAX, a w końcu przekazuje sterowanie z powrotem do procesu.

Najczęściej programista wykorzystuje funkcje biblioteczne do wywoływania odpowiednich funkcji systemowych, ale można to także zrobić bezpośrednio. Poniższy kod pokazuje, w jaki sposób można wywołać funkcję *fork()* i pobrać jej wynik. Program umieszcza liczbę 2, czyli numer funkcji *fork()*, w rejestrze EAX, następnie wywołuje przerwanie 0x80 i pobiera jego wynik z rejestru EAX do zmiennej pid.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
```



Rysunek 8.2. Interfejs pomiędzy procesem a jądrem systemu. Proces nie ma możliwości dostępu do wewnętrznych struktur jądra. Jedynym sposobem komunikacji są wywołania systemowe. W Linuksie 2.6.19 jest ich ponad 300.

```
int main(void)
{
        pid_t pid;

        asm("\
        movl $2, %%eax \n\t\
        int $0x80 \n\t\
        mov %%eax, %0 \n\t"
            :
            :"m"(pid)
            : "eax");
        if (pid == 0){
            printf("Proces potomny\n");
        }
        else if (pid < 0)
            printf("Blad funkcji fork\n");
        else{
            printf("Proces macierzysty\n");
        }
        return 0;
}
```

W dalszej części tego rozdziału omówione zostaną mechanizmy, które udostępnia Linux, i które umożliwiają zaimplementowanie systemu plików poziomu użytkownika. Najpierw zaprezentowane zostanie wywołanie systemowe *ptrace()*, wykorzystywane zarówno przez UMLa, jak i OKFS. Następnie pokrótce przedstawione zostaną inne mechanizmy, które można alternatywnie wykorzystać do implementacji takiego systemu.

8.3.1. Wywołanie systemowe *ptrace()*

Ptrace to wywołanie systemowe standardowo udostępniane przez Linuksa. Zostało ono zaimplementowane w celu ułatwienia procesu debugowania aplikacji i jest wykorzystywane między innymi przez program *strace*.

Ptrace umożliwia śledzenie procesów. Śledzony proces zostaje zatrzymany w momencie wywołania jakiejkolwiek funkcji systemowej. Proces śledzący może sprawdzić, jaką funkcje wykonał proces przez niego śledzony. Może również sprawdzić i zmienić zawartość jego pamięci i rejestrów. Taką samą możliwość ma po powrocie z funkcji systemowej. Funkcje te wystarczają, aby przy ich pomocy zaimplementować system plików, czy nawet cały system operacyjny poziomu użytkownika, taki jak UML.

Ponieważ jest to podstawowa funkcja, na której opiera się implementacja OKFS, warto przyjrzeć się jej dokładniej [17]:

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

Proces może zacząć śledzić inny proces na 2 sposoby. Jeśli proces, który ma być śledzony, już istnieje, należy wywołać *ptrace()* z argumentem *request* o wartości *PTRACE_ATTACH*. Jeśli proces śledzący sam tworzy proces, który będzie chciał śledzić, wywołuje funkcję *fork()*, a następnie w procesie potomnym wywołuje *ptrace()* z argumentem *request* o wartości *PTRACE_TRACEME*. Kolejnym krokiem jest wywołanie przez proces potomny funkcji *exec()*, w celu rozpoczęcia wykonywania konkretnego programu. Proces śledzący wywołuje funkcję *wait()*. Zostanie ona przerwana przez system operacyjny za każdym razem, gdy proces śledzony wywoła jakąś funkcję systemu operacyjnego, funkcja systemowa wywołana przez proces śledzony powróci lub do procesu śledzonego zostanie dostarczony jakiś sygnał.

Inne, ważne dla systemu plików poziomu użytkownika wartości, jakie może przyjmować argument *request* funkcji *ptrace()* to

- *PTRACE_PEEKDATA* - odczyt zawartości pamięci procesu śledzonego spod wskazanego adresu.
- *PTRACE_POKEDATA* - zapis do pamięci procesu śledzonego pod wskazanym adresem.
- *PTRACE_GETREGS* - odczyt zawartości rejestrów procesu śledzonego.
- *PTRACE_SETREGS* - ustawienie rejestrów procesu śledzonego.
- *PTRACE_CONT* - wznowienie działania zatrzymanego, procesu śledzonego. Umożliwia także dostarczenie sygnałów do wznawianego procesu.
- *PTRACE_SYSCALL* - działa tak samo jak PTRACE_CONT, ale powoduje ponowne zatrzymanie śledzonego procesu przy najbliższym wywołaniu lub powrocie z funkcji systemowej.

Ptrace obsługuje jeszcze kilka innych opcji, nie są one jednak wykorzystywane przez OKFS.

8.3.2. Ograniczenia funkcji ptrace()

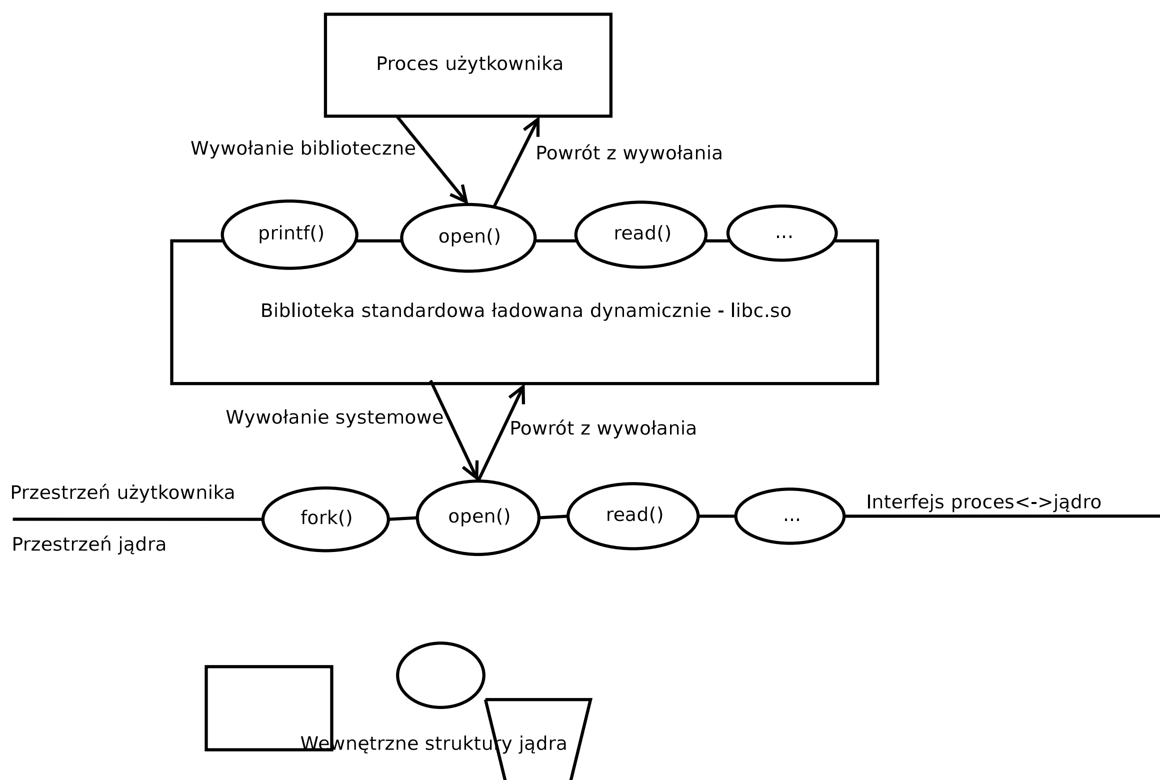
Niestety funkcja systemowa *ptrace()* nie jest pozbawiona wad, które utrudniają implementacje wirtualnego systemu plików poziomu użytkownika. Część niekorzystnych konsekwencji, wynikających z tych wad, można wyeliminować. Ale i tak są one źródłem znacznych komplikacji całego projektu. Do podstawowych wad funkcji *ptrace()* należą:

- *ptrace()* zmienia semantykę niektórych wywołań systemowych, a także niektórych sygnałów. W wyniku tego, proces śledzony zachowuje się w pewnych sytuacjach inaczej niż zwykły proces. Istnieje jednak możliwość symulowania rzeczywistego środowiska i odwrócenie niekorzystnych zmian. Przykładem takiego zachowania jest brak możliwości zatrzymania śledzonego procesu przy pomocy sygnału *SIGSTOP*. Proces śledzony otrzymuje sygnał *SIGSTOP*, jednak jądro natychmiast wznawia jego pracę. Łatwo to sprawdzić, próbując zatrzymać proces śledzony przy pomocy programu *strace*. Nie jest to możliwe. Jest to niedopuszczalne, gdyż m.in. uniemożliwia sterowanie zadaniami. Jednak możliwa jest emulacja poprawnego zachowania. Należy w momencie dostarczenia do procesu sygnału *SIGSTOP* zamienić następną instrukcję w jego pamięci (do której można uzyskać dostęp odczytując rejestr IP procesu) na wywołanie funkcji systemowej *pause()*. Po otrzymaniu *SIGCONT*, przywracana jest pierwotna zawartość pamięci i rejestr IP ustawiany jest na pierwotną wartość. Dzięki takiej operacji, proces zachowuje się po otrzymaniu *SIGSTOP* dokładnie tak, jak zachowywałby się, gdyby nie był śledzony.
- W starszych wersjach jądra, *ptrace()* nie daje możliwości rozpoczęcia śledzenia nowego procesu, utworzonego przez proces już śledzony, natychmiast po jego utworzeniu. Nowy proces może wykonać pewną ilość instrukcji, zanim rozpocznie się jego śledzenie. To ograniczenie da się ominąć. Wystarczy zmienić kod procesu, który wywołał funkcję *fork()* tak, aby po powrocie z tej funkcji została wywołana funkcja *pause()*. Dzięki temu, pomimo tego, że może minąć pewien okres czasu przed rozpoczęciem śledzenia potomka, potomek i tak w tym czasie będzie zatrzymany w funkcji *pause()* i nie wykona żadnych istotnych operacji. Po rozpoczęciu śledzenia potomka, proces śledzący przywraca pierwotną zawartość pamięci potomka i ustawia jego rejestr IP wskazujący kolejną instrukcję do wykonania na miejsce w pamięci zaraz za wywołaniem funkcji *fork()*.
- Ze względów na bezpieczeństwo nie można śledzić przy pomocy *ptrace'a* procesów z ustawionym bitem setuid, a co za tym idzie, dla takich procesów system plików poziomu użytkownika będzie niewidoczny.
- Jest to mechanizm specyficzny dla Linuksa, a przez to nieprzenośny na inne systemy operacyjne. Nawet zapewnienie przenośności pomiędzy różnymi architekturami sprzętowymi, na których działa Linux, wymaga dodatkowego wysiłku.

Dyskusję wad wywołania *ptrace()* rozwinięto w rozdziale opisującym ograniczenia OKFS.

8.3.3. Zamiana bibliotek ładowanych dynamicznie

Innym mechanizmem, który umożliwia stworzenie systemu plików poziomu użytkownika, jest możliwość podmiany bibliotek dynamicznych, których używa program poprzez ustawienie zmiennej systemowej *LD_PRELOAD*. Zmienna ta pozwala podać



Rysunek 8.3. Schemat interakcji pomiędzy procesem, biblioteką standardową ładowaną dynamicznie i jądrem systemu.

ścieżkę do biblioteki dynamicznej, która zostanie załadowana zamiast biblioteki ze standardowej lokalizacji.

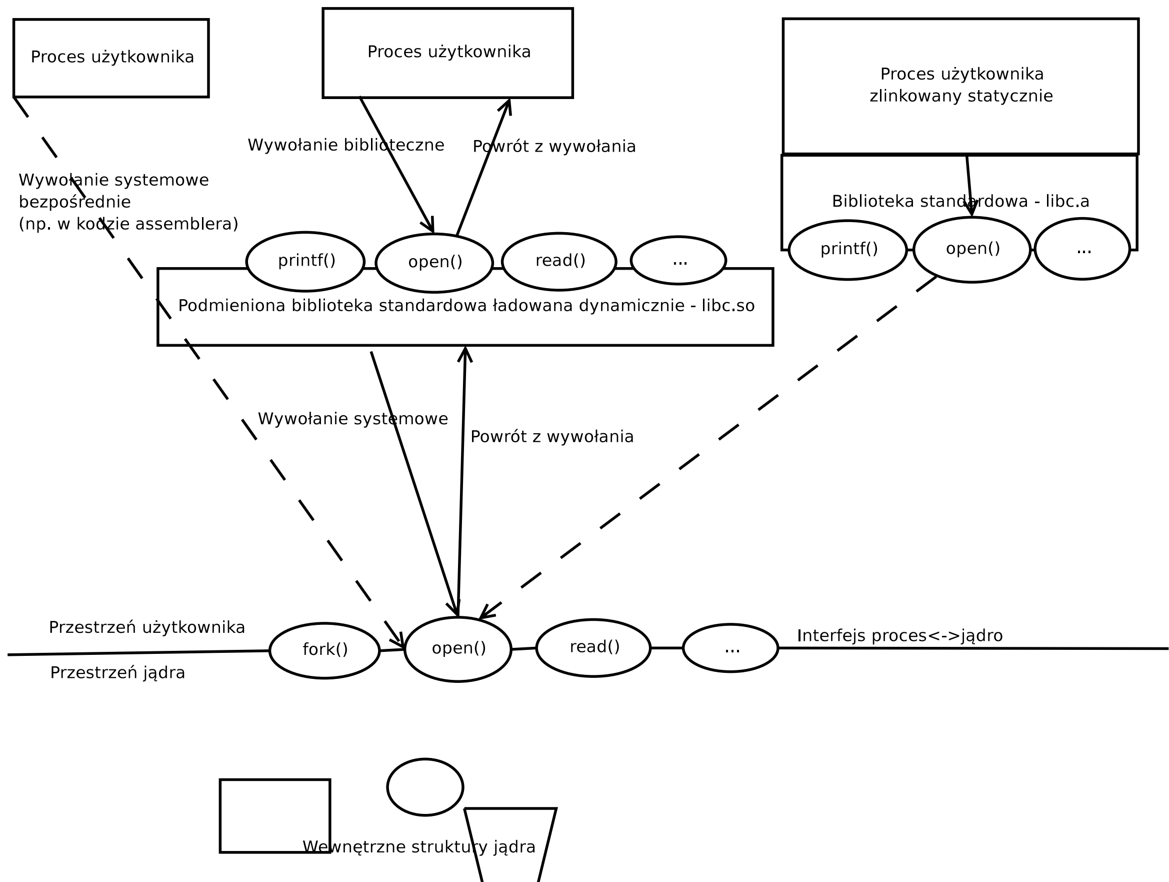
W znakomitej większości przypadków programy użytkownika nie wywołują bezpośrednio funkcji systemu operacyjnego, ale używają w tym celu standardowej biblioteki (Rysunek 8.3). Standardowa biblioteka jest więc warstwą pomiędzy procesem a systemem operacyjnym. Podmiana standardowej biblioteki pozwala więc na przechwycenie wywołań systemowych zanim zostaną one przekazane do systemu operacyjnego.

Głównym ograniczeniem tego mechanizmu, którego był pozbawiony mechanizm *ptrace*, jest to, że działa on tylko z programami linkowanymi dynamicznie, które nie wywołują funkcji systemowych bezpośrednio, ale zawsze stosują bibliotekę standardową. Najczęściej założenia te są spełnione, ale niestety nie zawsze. Problem ten ilustruje rysunek 8.4.

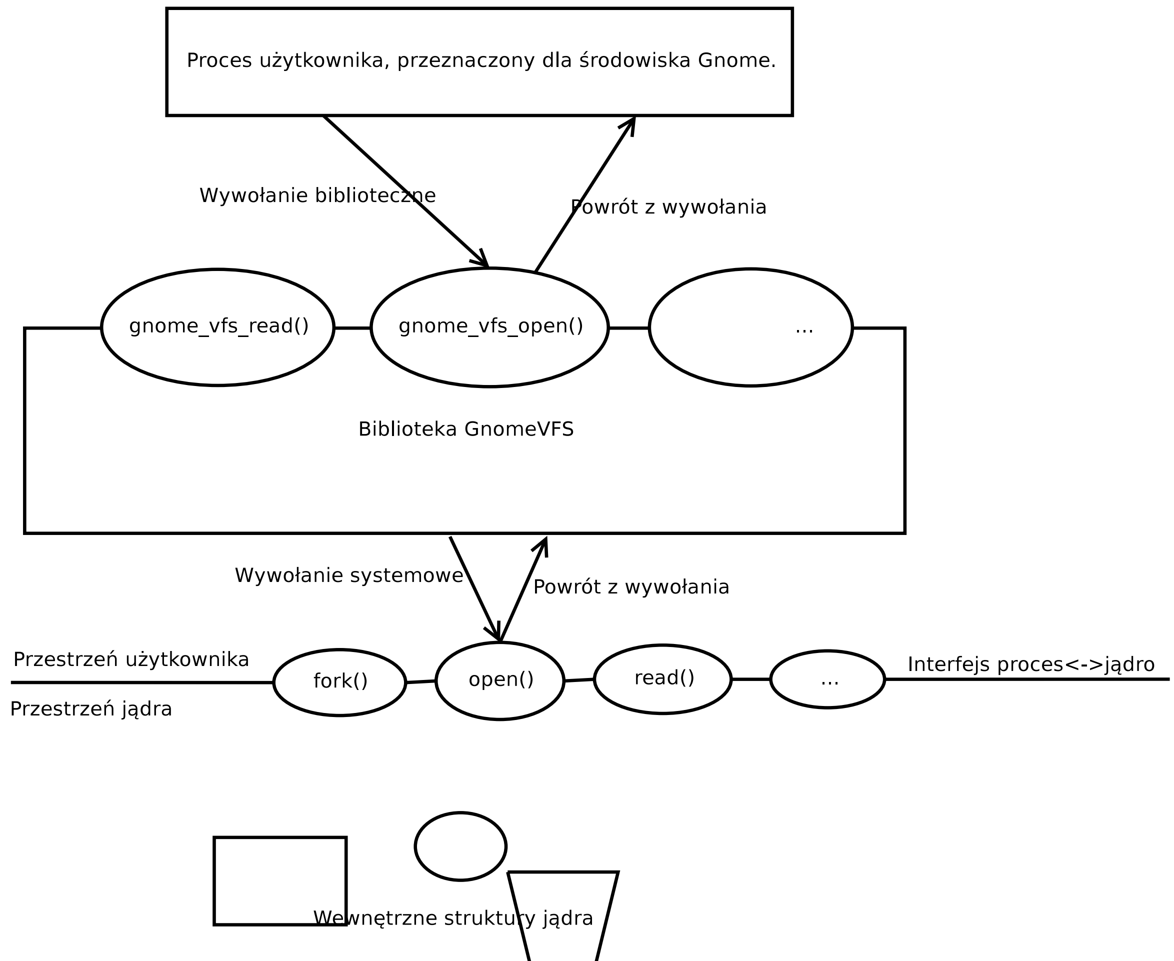
8.3.4. Jawny wirtualny system plików poziomu użytkownika

Cechą mechanizmów *ptrace* oraz podmiany bibliotek dynamicznych jest ich przezroczystość. Mechanizmy te nie wymagają żadnych zmian w kodzie źródłowym programów, rekompilacji, ani ponownego linkowania programów. Działają w sposób zupełnie niewidoczny dla procesu, który ma wrażenie, iż jest wykonywany w standardowym środowisku.

Inne podejście do problemu wirtualnego systemu plików poziomu użytkownika pro-



Rysunek 8.4. Ograniczenia mechanizmu podmiany biblioteki dynamicznej, wykorzystywanego w celu przechwytywania wywołań systemowych procesu. Możliwości ominięcia tego mechanizmu.



Rysunek 8.5. Działanie GnomeVFS - wirtualnego systemu plików, wykorzystywanego przez aplikacje dedykowane dla środowiska Gnome.

ponują systemy, których istnienia aplikacja je wykorzystująca musi być świadoma. Systemy takie wymagają od aplikacji jawnego odwoływania do nich przy użyciu odpowiedniej biblioteki. Aplikacja, wykorzystująca jeden z takich systemów, nie używa standardowych funkcji bibliotecznych czy systemowych do wykonywania operacji na plikach, lecz operacji udostępnianych przez bibliotekę danego wirtualnego systemu plików poziomu użytkownika. Architekturę taką przedstawia rysunek 8.5.

W porównaniu z poprzednimi, zaletą takiego rozwiązania jest stosunkowo prosta implementacja. Wadą natomiast konieczność zmian w kodzie źródłowym aplikacji i tworzenia aplikacji, przystosowanych do działania z jednym konkretnym wirtualnym systemem plików poziomu użytkownika. Utrudnia to znacznie przenośność aplikacji.

Przykładem takiego rozwiązania jest GnomeVFS, udostępniany przez środowisko Gnome [18], czy też KIO środowiska KDE [19].

Rozdział 9

Out of Kernel FileSystem

W poprzednich rozdziałach przedstawiono dostępne obecnie w Linuksie systemy plików, a także koncepcje i mechanizmy, które umożliwiają zaimplementowanie wirtualnego systemu plików na poziomie użytkownika. Dalsza część pracy zawiera dokładne omówienie projektu OKFS - architekturę programu, istotne szczegóły implementacyjne, napotkane problemy i możliwości dalszego rozwoju projektu. Przedstawiono także wyniki testów wydajności OKFS sprawdzające narzut, z jakim wiąże się korzystanie z tego typu systemu.

9.1. Zasada działania

Podstawowym mechanizmem, na którym opiera się implementacja OKFS, jest omówiona wcześniej funkcja systemowa *ptrace()*. Poniżej znajduje się przykład użycia OKFS, który pomaga zrozumieć ogólną zasadę działania programu i pokazuje, w jaki sposób wykorzystywana jest funkcja *ptrace()*. Zatem:

- użytkownik uruchamia OKFS na komputerze pracującym pod kontrolą systemu operacynego Linux. Poprzez plik w formacie *fstab* [20] przekazywane jest do OKFS polecenie zamontowania zdalnego katalogu:
  jimi@little.wing.art:/home/jimi do lokalnego katalogu */net* z użyciem rzeczywistego, sieciowego systemu plików SHFS.
- OKFS uruchamia shella jako swojego „niewolnika”, którego będzie śledził z użyciem funkcji *ptrace()* wraz ze wszystkimi jego potomkami (czyli programami uruchomionymi z linii poleceń shella).
- Użytkownik wykonuje polecenie *ls /net/foo.txt*.
- OKFS przechwytuje wywołania systemowe *fork()* i *execve()* odpowiedzialne za uruchomienie nowego procesu *ls* i rozpoczyna śledzenie tego nowego procesu.

- Proces *ls* wywołuje funkcję *lstat()* na pliku */net/foo.txt*. Poza środowiskiem OKFS funkcja ta zakończyłaby się błędem, gdyż z punktu widzenia systemu operacyjnego plik */net/foo.txt* nie istnieje.
- OKFS przechwytuje wywołanie *lstat()* i używa *PTRACE_PEEKDATA* do pobrania argumentów tego wywołania - czyli ścieżki do pliku, dla którego proces *ls* chce uzyskać statystyki.
- OKFS rozpoznaje, że za pliki w katalogu */net* odpowiedzialny jest SHFS. Każe więc swojemu podsystemowi implementującemu SHFS wykonać operację *lstat* na pliku *jimi@little.wing.art:/home/jimi/foo.txt*
- OKFS używa opcji *PTRACE_POKEDATA* do zapisu wyniku operacji *lstat()* na zdalnym pliku do pamięci procesu *ls*.
- Dla procesu *ls* wynik wykonania funkcji *lstat()* niczym nie różni się od wyników przekazywanych mu bezpośrednio przez jądro systemu. Wynik ten jest przez *ls* wypisywany, dając użytkownikowi informacje o zdalnym pliku tak, jakby plik ten znajdował się lokalnie.

OKFS działa w bardzo podobny sposób dla innych niż *lstat()* wywołań systemowych. W opisie tym pominięto wiele szczegółów implementacyjnych tak, aby w prosty sposób wyjaśnić ogólny mechanizm. Jak widać, podstawowa funkcjonalność opiera się na możliwości śledzenia wywołań systemowych, a także zapisu i odczytu pamięci procesów, jaką daje wywołanie systemowe *ptrace()*. Jak już wspomniano, wywołanie to jest standardowo dostępne w Linuksie dla zwykłych użytkowników. Do działania OKFS nie jest więc wymagana modyfikacja jądra ani posiadanie przywilejów nadzorcy systemu.

W architekturze OKFS można wyróżnić trzy części:

1. jądro OKFS,
2. wirtualny system plików,
3. rzeczywiste systemy plików.

9.2. Jądro OKFS

Istnieje wiele podobieństw pomiędzy jądrem OKFS a jądrem systemu operacyjnego. Jak już wspomniano, funkcjonalność, jaką udostępnia OKFS, jest podzbiorem funkcjonalności udostępnianej przez system User Mode Linux, który jest w pełni kompletną wersją Linuksa działającą na poziomie użytkownika. Jądro OKFS implementuje jedynie funkcjonalność niezbędną do stworzenia systemu plików poziomu użytkownika. Jednak, poprzez rozszerzenie jądra, może zostać stworzony cały wirtualny system operacyjny, taki jak UML.

Głównym zadaniem jądra OKFS jest śledzenie procesów, implementowanie i delegowanie wywołań systemowych, a także emulowanie standardowego środowiska udostępnianego przez Linuksa. Jądro posiada listę śledzonych procesów. Przy starcie OKFS rozpoczyna śledzenie procesu shella lub innego procesu, podanego jako argument z linii poleceń. Proces śledzony może tworzyć nowe procesy, jądro odpowiada za rozpoczęcie śledzenia każdego z nich.

Zadaniem jądra jest również obsługa wywołań systemowych, wykonywanych przez śledzone przez niego procesy. Większość wywołań systemowych jest przekazywana do Linuksa. Wywołania dotyczące systemu plików są przez jądro OKFS przekazywane

do opisanego dalej wirtualnego systemu plików OKFS. Niektóre wywołania, np. *fork()* czy *execve()*, wymagają ze strony jądra dodatkowych działań, takich jak rozpoczęcie śledzenia nowego procesu i stworzenie odpowiednich struktur danych z informacjami o tym procesie.

Jądro posiada tablicę wywołań systemowych. Każdy element tej tablicy jest typu *struct call_entry*. Struktura ta posiada następujące pola:

```
struct call_entry{
  /*this function is called before Linux gets control and
    executes system call specified by this entry*/
  void (*call_before)(void);

  /*this function is called after system call specified
    by this entry returns*/
  void (*call_after)(void);

  /*name of a system call, useful for debugging*/
  char* name;

  /*some system calls use memory block to pass
    arguments instead of registers*/
  int has_stack_args;
};
```

Jak już wspomniano, proces śledzący inny proces z użyciem wywołania *ptrace()*, uzyskuje sterowanie za każdym razem, gdy proces śledzony wywoła jakąś funkcję systemową lub funkcja ta powróci. Struktura *call_entry* pozwala dla każdego wywołania zdefiniować funkcję, która ma zostać wykonana, zanim to wywołanie zostanie przekazane do Linuksa. Pozwala także na zdefiniowanie funkcji, która zostanie wykonana zanim wynik zakończonego wywołania systemowego zostanie przekazany do procesu wywołującego i ponownie uzyska on sterowanie. Jądro rozpoznaje, jaką funkcję systemową proces wywołał, odczytując jej numer z rejestru *EAX* procesu. Numer tej funkcji staje się indeksem do tablicy wywołań systemowych. Poniższy listing pokazuje i wyjaśnia wybrane fragmenty kodu, inicjalizujące tablicę wywołań systemowych. Tablica ta zawiera tyle wpisów, ile jest funkcji systemowych.

```
/*Ten plik nagłówkowy jądra Linuksa definiuje numery funkcji
  systemowych np. __NR_fork. Definiuje także liczbę funkcji
  systemowych wspieranych przez daną wersję jądra
  Linuksa - NR_syscalls */
#include <linux/unistd.h>
/*...*/

/*Tablica wszystkich wywołań systemowych*/
struct call_entry *call;

/*
```

```
  Funkcja ułatwiająca definiowanie akcji dla pojedynczego
  wywołania systemowego.
*/
static void
add_call(int nr, void (*before)(), void (*after)(),
         const char *name, int stack_args)
{
  ASSERT(nr < NR_syscalls);
  call[nr].call_before = before;
  call[nr].call_after = after;
  call[nr].name = Strdup(name);
  call[nr].has_stack_args = stack_args;
}

/*Funkcja wywoływana na poczatku działania jądra OKFS*/
void init_call_table(void)
{
  /*...*/
  /*Alokacja tablicy wywołań systemowych*/
  call = Calloc(NR_syscalls, sizeof(struct call_entry));

  /*...*/

  /*Zdefiniowanie funkcji, które będą wykonywane przed i
    po wywołaniu fork*/
  add_call(__NR_fork, before_fork, after_fork, "fork", 0);

  /*Zdefiniowanie funkcji, za której obsługe odpowiada wirtualny
    system plików OKFS*/
  add_call(__NR_read, okfs_fd_call, okfs_read, "read", 0);
  add_call(__NR_unlink, dummy_call, okfs_unlink, "unlink", 0);
  /*...*/

  /*Nie dla wszystkich funkcji OKFS musi implementować
    własne akcje. Większość może zostać po prostu przekazane
    Linuksowi*/
  add_call(__NR_setuid, NULL, NULL, "setuid", 0);
  /*...*/

  /*Przykład funkcji mającej więcej niż 5 argumentów, do których
  przekazania wykorzystywany jest stos zamiast rejestrów*/
  add_call(__NR_mmap, okfs_before_mmap, NULL, "mmap", 1);
  /*...*/

  /*Przykład funkcji, dla której jądro musi zapewniać prawidłowe
    działanie, gdyż jest ono zmieniane w środowisku ptrace'a*/
```

```
  add_call(__NR_sigaction, NULL, after_sigaction, "sigaction",0);
  /*...*/

  /*Niektóre funkcje nie są dostępne w środowisku OKFS, jest
    dla nich zwracany błąd ENOSYS. Przykład to ptrace(). Procesy
    działające pod kontrolą OKFS nie mogą używać tej funkcji*/
  add_call(__NR_ptrace, unimplemented_call, NULL, "ptrace", 0);
  /*...*/
}
```

Dla wielu wywołań systemowych OKFS wykonuje takie same akcje. Przykładowo funkcja *okfs_fd_call()* jest wykonywana przez wirtualny system plików przed każdym wywołaniem, który odnosi się do deskryptora plików. Funkcja *dummy_call()* jest wywoływana przed każdym wywołaniem, które nie ma zostać przekazane do jądra Linuksa. *Ptrace()* nie umożliwia anulowania wywołania systemowego, ale umożliwia jego zamianę na inne. *dummy_call()* zamienia wywołanie systemowe, którego dotyczy, na wywołanie neutralnej funkcji *getpid()*.

Wcześniej wymienione zostały wady wywołania *ptrace()*, które powodują różnice w działaniu procesu śledzonego. Przedstawiono możliwości eliminacji tych różnic, co również jest zadaniem jądra OKFS. Dlatego jądro, poza delegacją wywołań do Linuksa czy też wirtualnego systemu plików, musi także implementować kod, zapewniający prawidłowe działanie niektórych wywołań systemowych, takich jak np. *sigaction()*.

Wyraźne, logiczne oddzielenie jądra OKFS, pozwala wykorzystać jego dosyć skomplikowany kod do zadań innych niż tylko stworzenie systemu plików. Przykładem jest „udawanie" środowiska nadzorcy systemu. Programy rozpoznają, czy użytkownik ma prawa nadzorcy systemu, poprzez sprawdzenie jego identyfikatora użytkownika i grupy. Robią to przy pomocy wywołań systemowych *getuid*, *getgid*, *geteuid()* i *getegid()*. OKFS pozwala w łatwy sposób emulować środowisko, w którym te wywołania zawsze zwracają 0, niezależnie od tego czy użytkownik rzeczywiście ma prawa administratora, czy też nie. Wystarczy dodać odpowiednie funkcje do tablicy wywołań systemowych OKFS:

```
static void rootid(void)
{
  /*Zapisanie do rejestru EAX śledzonego procesu wartości 0.
    (W rejestrze EAX przekazywany jest przez jądro wynik
     wykonania funkcji systemowej)*/
  set_slave_return(0);
}

void fake_root_init(void)
{
  /*Zdefiniowanie funkcji wykonywanej po powrocie
    z wywołań get**id(). Zadaniem tej funkcji jest
    zastąpienie wartości UID, GID, EUID lub EGID przekazanej przez
    jądro systemu wartościa 0.*/
  call[__NR_getuid].call_after = rootid;
```

```
  call[__NR_getgid].call_after = rootid;
  call[__NR_geteuid].call_after = rootid;
  call[__NR_getegid].call_after = rootid;
  call[__NR_getuid32].call_after = rootid;
  call[__NR_getgid32].call_after = rootid;
  call[__NR_geteuid32].call_after = rootid;
  call[__NR_getegid32].call_after = rootid;
}
```

Pozwala to „oszukać” niektóre programy, a także użytkownika, że ma on prawa użytkownika root. Oczywiście żadna operacja, która tych praw rzeczywiście wymaga, nie powiedzie się.

Bardziej skomplikowanym przykładem jest możliwość wykorzystania jądra OKFS do implementacji wysokopoziomowego debuggera pozbawionego wad programu *strace*. *Strace* nie emuluje rzeczywistego środowiska Linuksa, co często ogranicza jego przydatność.

9.3. Wirtualny system plików

Wszystkie wywołania systemowe, odnoszące się do plików, są przekazywane do wirtualnego systemu plików OKFS. Najważniejsze zadania tego systemu to:

- delegacja wywołań systemowych do rzeczywistych systemów plików,
- zarządzanie obiektami systemu plików,
- ograniczenie złożoności rzeczywistych systemów plików, obsługa operacji wspólnych dla wszystkich rzeczywistych systemów plików,
- cachowanie informacji.

Omówmy dokładnie wymienione powyżej funkcje.

9.3.1. Delegacja wywołań

Wirtualny system plików rozpoznaje, czy dane wywołanie systemowe odnosi się do jednego z rzeczywistych systemów plików OKFS, czy do systemu plików Linuksa. Następnie przekazuje polecenie wykonania właściwej operacji do opowiedniego podsystemu. Wywołania systemowe wykonujące operacje na plikach identyfikują plik, na którym mają wykonać operacje na jeden z dwóch sposobów:

1. za pomocą ścieżki do pliku,
2. za pomocą deskryptora pliku.

Ścieżka do pliku lub deskryptor są przekazywane do wywołania jako jeden z jego argumentów. Struktura *okfs_superblock* pozwala na identyfikację, do którego systemu plików powinno zostać przekazane wywołanie systemowe, przyjmujące jako argument ścieżkę do pliku. Struktura ta jest odpowiednikiem struktury *superblock* w jądrze Linuksa. Jest ona zdefiniowana następująco:

```
/*Infromation about mounted filesystems*/
struct okfs_superblock{
  /*a pointer to filesystem specific structure, used to store
   *information needed by this filesystem*/
  void *fsdata;
```

```
  /*next mounted filesystem*/
   struct okfs_superblock *next_fs;
  char *mnt_point;

  /*first field from fstab*/
  char *fs_spec;

  /*options field from fstab*/
  char *mnt_opts;

  int mnt_point_len, fs_spec_len;

  /*Hash tables with pointers to inodes (hashed using one of the
    paths to inode)*/
  struct ipath_lhead* ipath_cache[MAX_HASH];
  struct okfs_dentry_lhead* dcache[MAX_HASH];

  /*Hash table with pointers to inodes
    (hashed by inode nr and dev_nr).*/
  struct inr_lhead* inr_cache[MAX_HASH];

  /*table of calls used by the filesystem
   *specified by this mount_point:*/
  struct fs_entry_point *entry;
};
extern struct okfs_superblock *mounted_fs;
```

Każdemu zamontowanemu systemowymi plików OKFS odpowiada jedna struktura *okfs_superblock*. Struktury te są połączone w listę. Zmienna *mounted_fs* wskazuje na pierwszy element tej listy. Pole *mnt_point* zawiera ścieżkę do katalogu, w którym dany system został zamontowany. W przypadku gdy kilka systemów plików zostało zamontowanych w tym samym katalogu, wywołania są przekazywane do zamontowanego najpóźniej.

Rozpoznanie, do jakiego systemu plików odnosi się wywołanie, nie jest prostym zadaniem. Wynika to ze skomplikowanych form, jakie może przyjmować ścieżka do pliku, przekazywana jako argument wywołania przez program użytkownika. Oto kilka przykładów:

- */etc/passwd* - najprostszy przypadek - pełna ścieżka od katalogu głównego w postaci kanonicznej.
- *passwd* - ścieżka względna w odniesieniu do bieżącego katalogu procesu. OKFS musi śledzić, jaki jest bieżący katalog każdego procesu. Każdy nowy proces dziedziczy ten katalog od swojego procesu macierzystego, a następnie może go zmieniać poprzez wywołania systemowe *chdir()* i *fchdir()*.
- *../../etc/.././etc/passwd* - bardziej skomplikowany przykład ścieżki względnej.

Pomijając sensowność używania ścieżek w takiej postaci, są one całkowicie poprawne i wirtualny system plików musi umieć je obsługiwać.

Dodatkowym utrudnieniem jest to, że każdy element ścieżki może być dowiązaniem symbolicznym wskazującym inny katalog. Wirtualny system plików przed przekazaniem wywołania systemowego do rzeczywistego systemu plików, musi wykonać operację kanonikalizacji ścieżki. Ścieżka w postaci kanonicznej jest ścieżką od katalogu głównego, nie zawierającą żadnych dowiązań symbolicznych ani katalogów *'.'* i *'..'*. Dodatkowo wirtualny system plików sprawdza, czy każdy element ścieżki istnieje. Nie dotyczy to ostatniego elementu, który w przypadku niektórych wywołań, takich jak np. *mkdir*, nie musi istnieć, gdyż zostanie utworzony. Wymaga to wywołania funkcji *lstat()* dla każdego elementu ścieżki. Jeśli *lstat()* zwróci informacje, że plik jest dowiązaniem symbolicznym, musi zostać wywołana funkcja systemowa *readlink()* w celu rozpoznania, gdzie to dowiązanie wskazuje. Należy zwrócić uwagę, że kolejne wywołania *lstat()* i *readlink()* mogą odnosić się do plików należących do systemu plików Linuksa lub do różnych rzeczywistych systemów plików OKFS. Wirtualny system plików musi kierować te wywołania do odpowiedniego podsystemu.

Najlepiej zilustruje to przykład. Załóżmy, że użytkownik używa OKFS do zamontowania zdalnego katalogu poprzez SHFS w lokalnym katalogu */net*. Następnie wykonuje polecenie: *cat /net/doc/../../etc/passwd*. Na pierwszy rzut oka wydaje się ono równoważne wykonaniu polecenia *cat /etc/passwd*, jednak nie do końca tak jest. Pierwsze polecenie zakończy się sukcesem, tylko jeśli katalog */net/doc* istnieje. Dodatkowo zarówno pliki */net*, jak i */net/doc*, mogą być dowiązaniami symbolicznymi, które spowodują, że wywołanie nie będzie się odnosiło do pliku */etc/passwd*. Wirtualny system plików OKFS musi wykonać funkcję *lstat()* na plikach *net*, *doc* i *etc* w celu dokonania kanonikalizacji ścieżki. Pomimo tego, że plik */etc/passwd* znajduje się lokalnie, wywołanie *lstat()* na pliku *doc* musi zostać przekazane do zdalnego systemu z użyciem SHFS.

Jak widać operacja sprowadzenia ścieżki do postaci kanonicznej jest skomplikowana, nie tylko ze względu na trudności implementacyjne. Wprowadza ona także znaczny narzut czasowy na wykonywane operacje na plikach. W dalszej części tej pracy zostanie przedstawiony mechanizm cachingu, który znacznie ogranicza ten narzut. Wirtualny system plików przekazuje do rzeczywistych systemów plików ścieżkę w postaci kanonicznej od punktu montowania do pliku. Rzeczywiste systemy plików nie muszą więc wykonywać skomplikowanych operacji przekształcania ścieżki.

Część wywołań systemowych odwołuje się do wcześniej otwartego pliku, podając jako argument liczbę, będącą deskryptorem pliku. Dla każdego śledzonego procesu OKFS przechowuje tablicę używanych przez niego deskryptorów plików. W tablicy tej przechowywana jest informacja, czy dany deskryptor odnosi się do pliku znajdującego się na jednym z rzeczywistych systemów plików OKFS, a jeśli tak, to na którym.

Kiedy proces śledzony wywołuje funkcję tworzącą nowy deskryptor pliku, np. *open()*, która odnosi się do pliku będącego pod kontrolą OKFS, OKFS obsługuje to wywołanie i odpowiada za alokację i przekazanie do procesu śledzonego nowego deskryptora. Problemem jest to, że Linux nic nie wie o nowo otwartym pliku i jego deskryptorze. Następne wywołanie *open()* dla pliku, znajdującego się po kontrolą systemu plików Linuksa, może zwrócić taki sam numer deskryptora, jak poprzednie wywołanie, które obsłużył OKFS. Zastosowano prosty zabieg, mający na celu uniknięcie skomplikowanej obsługi mapowania numerów deskryptorów procesu, widzianych przez ten proces, na

te widziane przez jądro Linuksa. Każde wywołanie systemowe, otwierające plik pod kontrolą OKFS, jest przekazywane do jądra ze zmienioną ścieżką. Nowa ścieżka wskazuje na plik w lokalnym katalogu tymczasowym. Linux alokuje więc nowy deskryptor dla tego pliku pomimo tego, że operacje tak naprawdę będą wykonywane na zupełnie innym pliku, o którego istnieniu Linux nie wie. Dzięki temu proces z punktu widzenia Linuksa ma tyle samo otwartych plików, co z punktu widzenia OKFS. Numery deskryptorów nie wymagają zamiany przy przekazywaniu wywołań do jądra systemu. Koncepcję tą ilustruje rysunek 9.3.1.

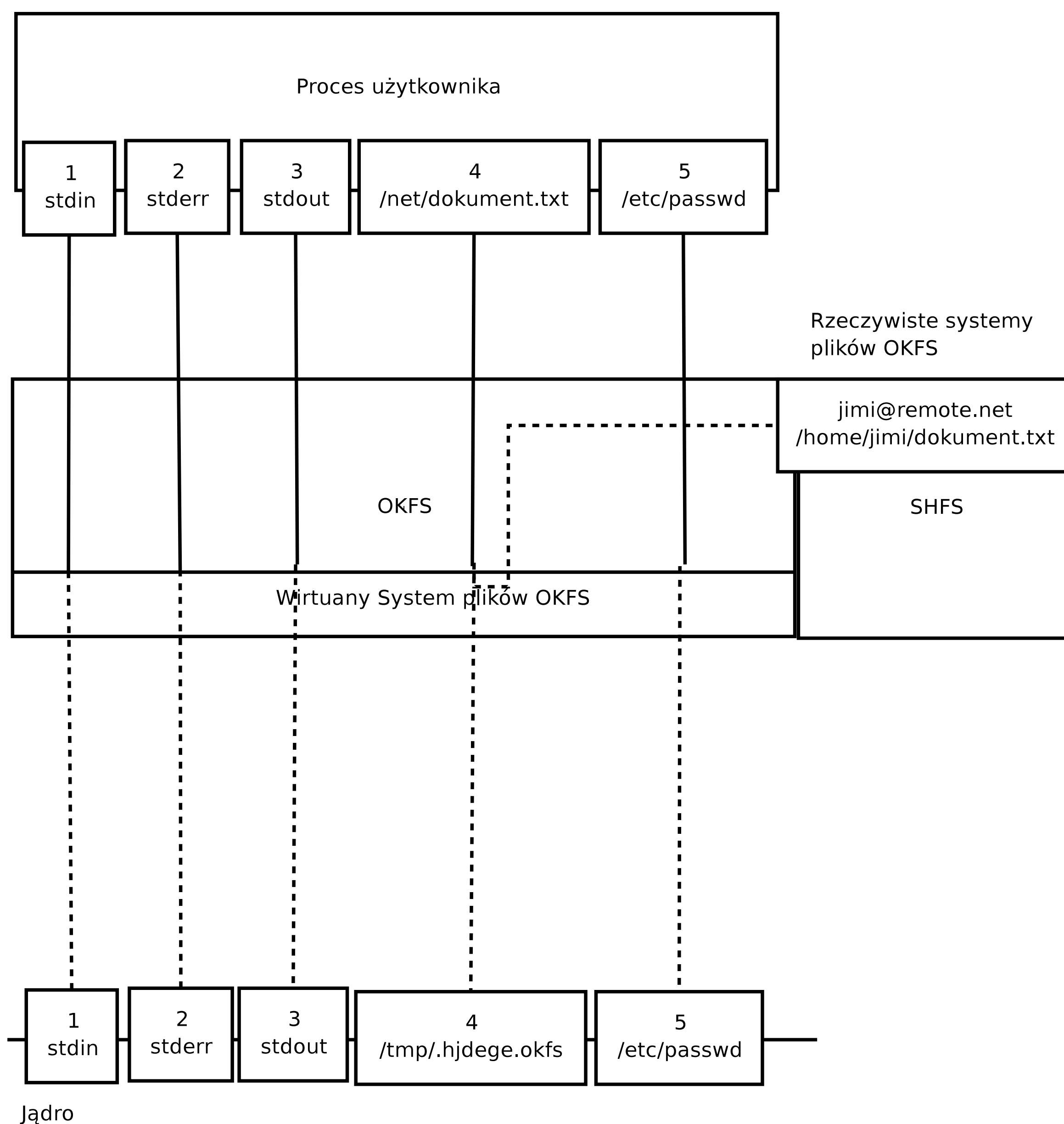
9.3.2. Zarządzanie obiektami systemu plików

OKFS używa podobnych obiektów, co system plików Linuksa. Najważniejsze z nich to opisany już *okfs_superblock*, reprezentujący otwarty plik *okfs_file* i reprezentujący obiekt systemu plików *okfs_inode*.

Szczególnej uwagi wymaga obsługa obiektów *file*, gdyż mogą być one współdzielone przez wiele procesów. Także pojedynczy proces może posiadać kilka deskryptorów wskazujących na ten sam obiekt *file*. Struktura *file* zdefiniowana jest następująco:

```
struct okfs_file{

  /*path to file pointed by file descriptor(if any)*/
  char *path;

  /*path to this file from mount point of filesystem
    responsible for this file*/
  char *rel_path;
  struct okfs_inode *ino; /*inode associated with this file*/

  /*pointer to the filesystem responsible for this file
  NULL means that it is managed by OS*/
  struct okfs_superblock *fsp;

  /*read/write position*/
  loff_t f_pos;

  /*how many descriptors are pointing to this file*/
  int nshared;
};
```

OKFS musi śledzić, ile deskryptorów plików wskazuje dany plik i zwolnić go, kiedy liczba ta osiągnie 0. Podobnie jak w Linuksie, wywołanie *close()* jest przekazywane do rzeczywistego systemu plików, tylko kiedy odnosi się do ostatniego aktywnego deskryptora plików. Deskryptor może zostać zduplikowany poprzez wywołania systemowe *dup()*, *dup2()* i *fcntl()*. OKFS musi śledzić proces tworzenia duplikatów oraz zamykania deskryptorów i odpowiednio aktualizować licznik odwołań do danego pliku. Każdy nowo utworzony proces współdzieli wszystkie otwarte deskryptory plików procesu macierzystego, również dla tych deskryptorów zwiększany jest licznik odwołań.



Rysunek 9.1. Mapowanie deskryptorów na linii proces użytkownika ↔ OKFS ↔ jądro Linuksa. Każdemu deskryptorowi odnoszącemu się do pliku obsługiwanego przez OKFS, odpowiada w jądrze Linuksa deskryptor odnoszący się do tymczasowego pliku.

9.3.3. Ograniczenie złożoności rzeczywistych systemów plików

Jak już wcześniej wspomniano, podając jako przykład wywołanie systemowe *stat()*, wiele wywołań systemowych, wykonywanych przez system plików jest bardzo do siebie podobnych. Wiele z nich istnieje dla wygody i da się je zaimplementować przy pomocy innych. Wirtualny system plików, kiedy tylko jest to możliwe, wykorzystuje jedne wywołania systemowe do implementacji innych. Również wszystkie wywołania systemowe dotyczące zarządzania deskryptorami plików nie są w ogóle przekazywane do rzeczywistych systemów plików. Dzięki temu spośród 49 operacji na plikach i deskryptorach plików, obsługiwanych obecnie przez OKFS, każdy rzeczywisty system plików powinien zaimplementować jedynie 18. Dodatkowe 3 wywołania (*close()*, *lseek()* i *access()*) może zaimplementować opcjonalnie. Nie musi tego robić, jeśli domyślna implementacja, jaką zapewnia wirtualny system plików, jest wystarczająca.

9.3.4. Cachowanie informacji

Cachowanie informacji w systemie plików jest niezwykle istotne ze względu na efektywność operacji. Jak już objaśniono wcześniej, każde wywołanie systemowe przyjmujące jako argument ścieżkę do pliku, musi wykonać tyle operacji *lstat()*, ile jest katalogów na tej ścieżce. Nawet kiedy operacje te należy wykonać na lokalnych katalogach, wprowadzają one spory narzut czasowy. Kiedy katalogi znajdują się w zdalnym systemie i każde wywołanie *lstat()* musi zostać przesłane przez sieć, narzut czasowy jest nieakceptowalnie duży.

Następujący przykład obrazuje skalę problemu. Użytkownik wykonuje polecenie *ls -al /net/dokumenty/studia*, katalog */net/dokumenty/studia/* zawiera 10 plików. Program *ls* wykona najpierw funkcję *open("/net/dokumenty/studia'",...)*, następnie za pomocą funkcji *fstat64()* sprawdzi, że plik ten jest katalogiem i wywoła *getdents64()* w celu pobrania zawartości tego katalogu. Ponieważ *ls -al* wypisuje nie tylko pliki w podanym katalogu, ale również atrybuty tych plików, takie jak rozmiar i prawa dostępu, *ls* wywołuje *lstat64()* dla każdego pliku w tym katalogu. Dodatkowo, jeśli system plików obsługuje rozszerzone atrybuty plików, dla każdego pliku są one pobierane z użyciem wywołania *getxattr()*. Jest to jednak wywołanie opcjonalne, które nie musi być zaimplementowane przez każdy system plików. Bez uwzględnienia *getxattr()*, *ls* wykonało 11 funkcji systemowych przyjmujących jako argument ścieżkę do pliku. Dodatkowo problem potęguje fakt, że każde z tych wywołań wymaga od OKFS wykonania *lstat()* na każdym elemencie ścieżki. Czyli wywołanie *open("/net/dokumenty/studia", ...)* pociąga za sobą wywołania *lstat("/net", ...)*, *lstat("/net/dokumenty", ...)* i *lstat("/net/dokumenty/studia", ...)*. Operacje te zostaną powtórzone przy każdym wywołaniu *lstat()*, odnoszącym się do plików w katalogu *studia*. Z tego wynika, że funkcja *lstat()* zostanie dodatkowo wywołana 43 razy. Jeśli ścieżka */net* wskazuje na katalog na zdalnym serwerze, większość tych wywołań będzie odnosiła się do zdalnych plików. Każde z nich będzie trwało minimum tyle, ile trwa podróż pakietu do serwera i z powrotem (RTT).

Cachowanie pozwala znacznie ograniczyć zarówno ilość wykonywanych operacji, jak i ich czas wykonania. Zapobiega wielokrotnemu wykonywaniu tych samych operacji. OKFS cachuje i-węzły oraz zawartość katalogów (obiekty dirent). Obecnie nie jest cachowana zawartość zwykłych plików. Tabela 9.1 pokazuje, o ile udało się ograniczyć

Tablica 9.1. Ilość funkcji systemowych wywoływanych po wykonaniu polecenia *ls -al /net/dokumenty/studia*. Do katalogu *net* zamontowany jest zdalny system plików, w katalogu */net/dokumenty/studia* znajduje się 10 plików.

Bez cachowania		Cachowanie pierwsze wykonanie (cache zapełniany)		Cachowanie drugie wykonanie (cache pełny)	
suma	zdalne	suma	zdalne	suma	zdalne
48	37	28	17	13	2

ilość operacji dzięki wprowadzeniu cachowania. OKFS cachuje jedynie informacje o plikach znajdujących się na jego rzeczywistych systemach plików. Cachowanie informacji o plikach pod kontrolą Linuksa nie miałoby większego sensu. Linux posiada własny system cachowania, pozwalający odczytywać informacje bez wielokrotnego wykonywania operacji na fizycznych urządzeniach lub operacji sieciowych.

Do implementacji pamięci podręcznej wirtualny system plików wykorzystuje tablice hashujące. Każdy rzeczywisty system plików posiada własne tablice, zawierające informacje o plikach zarządzanych przez ten system. Tablice te znajdują się w strukturze *okfs_superblock*. Wyszukanie i-węzła wymaga obliczenia wartości funkcji hashującej na podstawie ścieżki do pliku i sprawdzenie w tablicy *ipath_cache* czy taki i-węzeł jest w pamięci podręcznej. Jeśli i-węzeł znajduje się w pamięci podręcznej, odczytywane są z niego wszystkie konieczne informacje bez przekazywania wywołania do rzeczywistego systemu plików. Jeśli i-węzła nie ma w pamięci podręcznej, wywoływanie jest przekazywane do rzeczywistego systemu plików. Pojedynczy i-węzeł może być wskazywany przez więcej niż jedną ścieżkę. Jest to możliwe dzięki dowiązaniom twardym. Zanim wirtualny system plików stworzy obiekt reprezentujący nowy i-węzeł i doda go do pamięci podręcznej *ipath_cache*, musi upewnić się, czy dany i-węzeł nie znajduje się już w pamięci podręcznej pod inną ścieżką. Wykorzystywana jest w tym celu pamięć *inr_cache*. I-węzły są wyszukiwane w tej pamięci na podstawie ich unikalnych numerów identyfikacyjnych (numer urządzenia i numer i-węzła na danym urządzeniu). Jeśli i-węzeł o podanym numerze znajduje się w tej pamięci podręcznej, do pamięci *ipath-cache* dodawany jest jedynie wskaźnik do tego i-węzła. W przeciwnym wypadku, możemy mieć pewność, że obiekt typu *struct inode* dla danego i-węzła nie został jeszcze utworzony, można go więc utworzyć i dodać do pamięci *inr_cache* oraz *ipath_cache*.

9.4. Rzeczywiste systemy plików

Rzeczywiste systemy plików są odpowiedzialne za ostateczne wykonanie operacji na plikach. Każdy z nich może wykonywać te operacje w inny sposób i może mieć inne zastosowanie. W punkcie, opisującym FUSE, podano przykłady rzeczywistych systemów plików i opisano, jakie czynności mogą one upraszczać.

Przy projektowaniu OKFS duży wysiłek włożono w to, aby dodanie nowego rzeczywistego systemu plików było jak najprostsze. Starano się tak zaprojektować interfejs pomiędzy wirtualnym systemem plików a rzeczywistym system plików, aby rzeczywi-

sty system plików musiał implementować stosunkowo niewiele podstawowych wywołań systemowych.

Każdy rzeczywisty system plików musi zainicjalizować strukturę *fs_entry_point*, która zdefiniowana jest następująco:

```
struct fs_entry_point{
  /*name of the filesystem*/
  char *fs_type;
  int (*mount)(struct okfs_superblock*);
  int (*lstat)(struct okfs_superblock*, const char*,
               struct stat*);
  int (*readlink)(struct okfs_superblock*, const char*,
                  char*, size_t);
  int (*open)(struct okfs_superblock*, const char*,
              int, mode_t);
  int (*readdir)(struct okfs_file*, int,
                 struct okfs_dirent **);
  int (*close)(struct okfs_file*,int);
  off_t (*lseek)(struct okfs_file*, int, off_t offset,
                 int whence);
  int (*read)(struct okfs_file*, int, void*, size_t);
  int (*write)(struct okfs_file*, int, void*, size_t);
  int (*link)(struct okfs_superblock*, const char *,
              const char *);
  int (*unlink)(struct okfs_superblock*, const char *);
  int (*access)(struct okfs_superblock*,
                const char *, int);
  int (*rename)(struct okfs_superblock*, const char *,
                const char *);
  int (*rmdir)(struct okfs_superblock*, const char *);
  int (*mkdir)(struct okfs_superblock*, const char *,
               mode_t);
  int (*symlink)(struct okfs_superblock*, const char *,
                 const char *);
  int (*utime)(struct okfs_superblock*, const char *,
               const struct utimbuf *);
  int (*chmod)(struct okfs_superblock*, const char *,
               mode_t);
  int (*truncate)(struct okfs_superblock*, const char *,
                  off_t);
  int (*statfs)(struct okfs_superblock*, const char *,
                struct statfs *);
  int (*lchown)(struct okfs_superblock*, char *, uid_t,
                gid_t);
};
```

Struktura ta definiuje nazwę systemu plików, poprzez którą system plików jest rozpo-

znawany w trakcie montowania. Zawiera także wskaźniki do funkcji implementujących wywołania systemowe, odnoszące się do danego systemu plików. Jak widać, każdy z rzeczywistych systemów plików OKFS powinien zaimplementować najwyżej 21 wywołań systemowych. Za wszystkie pozostałe odpowiedzialny jest wirtualny system plików. Można stworzyć system plików o ograniczonej funkcjonalności, np. taki, który będzie pozwalał jedynie na odczyt, a nie będzie zapisywał i tworzył żadnych plików. Wtedy wystarczy zaimplementować tylko niezbędną część wywołań. Dla wszystkich niezaimplementowanych wywołań wirtualny system plików zwróci odpowiedni kod błędu, sygnalizujący, że dane wywołanie nie jest obsługiwane.

OKFS przy uruchomieniu przyjmuje jako argument ścieżkę do pliku w formacie fstab [20], z którego odczytuje informacje konieczne do zamontowania rzeczywistych systemów plików. Obecnie nie jest obsługiwane montowanie nowych systemów plików w trakcie działania OKFS. Należy dokonać edycji pliku fstab i ponownie uruchomić OKFS.

Wirtualny system plików przechowuje listę, zawierającą po jednej strukturze *fs_entry_point* dla każdego obsługiwanego typu systemu plików. W momencie montowania nowego systemu plików lista ta jest przeglądana i znajdowany jest system plików o odpowiedniej nazwie. Następnie wirtualny system plików tworzy i inicjalizuje nową strukturę *okfs_superblock*. W szczególności wypełnia pola *fs_spec* i *mnt_opts* odpowiednimi wartościami, odczytanymi z pliku fstab. Dalej wywoływana jest funkcja *mount()* odpowiedniego rzeczywistego systemu plików. Jako argument przekazywana jest do niej nowo utworzona struktura *okfs_superblock*. Zadaniem tej funkcji jest zainicjalizowanie nowego systemu plików. Operacje jakie są do tego wymagane, zależą od systemu plików. Dla przykładu, sieciowe systemy plików mogą w tej funkcji zainicjalizować połączenie do serwera. Każdy rzeczywisty system plików może także wypełnić pole *fsdata* struktury *okfs_superblock*. Zawiera ono wskaźnik do danych specyficznych dla danego systemu plików. Np. sieciowe systemy plików mogą wykorzystać to pole do przechowywania adresu serwera i loginu na wypadek, gdyby połączenie zostało zamknięte i była wymagana jego ponowna inicjalizacja.

OKFS implementuje obecnie dwa rzeczywiste systemy plików - Localfs i SHFS. Opis ich działania znajduje się w kolejnych podpunktach.

9.4.1. LocalFS

LocalFS był pierwszym systemem plików, obsługiwanym przez OKFS. Jest on wzorowany na podobnym systemie plików, implementowanym przez LUFS [21] (Linux Userland FileSystem to system plików, działający częściowo w przestrzeni użytkownika, na bardzo podobnej zasadzie jak FUSE). Jego funkcjonalność jest bardzo prosta, służy on głównie celom demonstracyjnym i testowym, istnieje jednak kilka przydatnych zastosowań tego systemu. LocalFS pozwala na zamontowanie lokalnego katalogu w innym miejscu w drzewie katalogów. Dla wpisu w pliku, fstab odnoszącego się do LocalFS, wartość pola fs_spec odpowiada ścieżce do katalogu, który ma zostać odzwierciedlony w punkcie montowania.

Implementacja wszystkich 21 wywołań systemowych, jakie może implementować rzeczywisty system plików, zajęła w przypadku LocalFS niecałe 300 linii kodu źródłowego. Każde wywołanie systemowe, przyjmujące jako argument ścieżkę do plików,

jest przekazywane do systemu operacyjnego. Ścieżka jest jednak wcześniej zamieniana w taki sposób, aby wskazywała odpowiedni plik, w katalogu odzwierciedlanym przez LocalFS. Działanie LocalFS przedstawia rysunek 9.2.

Poniżej przedstawiona jest podstawowa funkcja LocalFS, wykonująca zamianę, ścieżki przekazanej mu przez wirtualny system plików, tak aby wskazywała ona na katalog, który jest odzwierciedlany w punkcie montowania.

```
static char *
create_path(struct okfs_superblock *fsi, const char *path)
{
  char *retval;
  int len;
  len = strlen(path) + fsi->fs_spec_len;
  retval = Malloc(len + 1);
  strcpy(retval, fsi->fs_spec);
  strcat(retval, path);
  return retval;
}
```

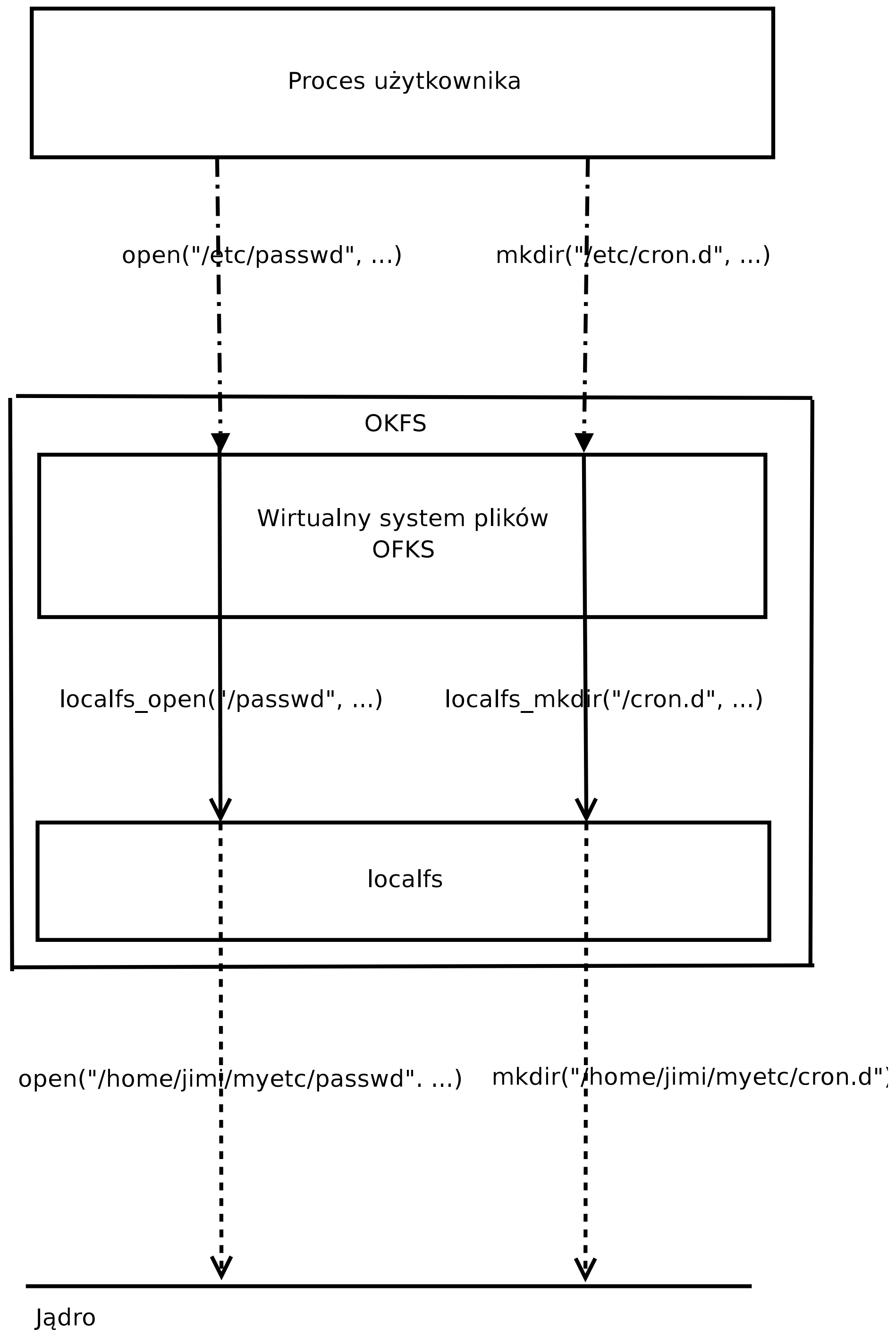
Jak widać, ścieżka do odzwierciedlanego katalogu pobierana jest z pola *fs_spec* struktury *okfs_superblock*, odpowiadającej danej instancji LocalFS.

Z użyciem tej funkcji implementacja poszczególnych wywołań systemowych jest bardzo prosta. Dla przykładu, wywołanie *mkdir()* wygląda następująco:

```
int
localfs_mkdir(struct okfs_superblock *fsi,
              const char *path, mode_t mode)
{
  char *newpath;
  int ret;
  newpath = create_path(fsi, path);
  ret = mkdir(newpath, mode);
  free(newpath);
  return (ret < 0) ? -errno : ret;
}
```

Po modyfikacji ścieżki wywołanie jest przekazywane do systemu operacyjnego, a jego kod powrotu zwracany do wirtualnego systemu plików OKFS. Wirtualny system plików i jądro OKFS odpowiadają za przekazanie tego kodu powrotu do programu który wywołał funkcję *mkdir()*.

LocalFS może być użyteczny np. wtedy, gdy użytkownik chce używać aplikacji, która zakłada, że jej pliki konfiguracyjne są umieszczone w katalogu */etc* i nie udostępnia możliwości zmiany tego ustawienia. Jeśli ta aplikacja nie wykonuje żadnych operacji, wymagających praw użytkownika root, LocalFS pozwala użytkownikowi na jej używanie, nawet jeśli nie ma on możliwości modyfikacji katalogu */etc*. Wystarczy uruchomić aplikację pod kontrolą OKFS, zlecając mu zamontowanie katalogu, w którym użytkownik ma prawo zapisu, do katalogu */etc*. Następnie użytkownik może używać i konfigurować aplikację bez żadnych ograniczeń.



Rysunek 9.2. Translacja ścieżek przez LocalFS po zamontowaniu katalogu */home/jimi/myetc* do katalogu */etc*. Wirtualny system plików przekazuje do rzeczywistych systemów plików bezwzględne ścieżki do plików, zaczynające się od punktu montowania.

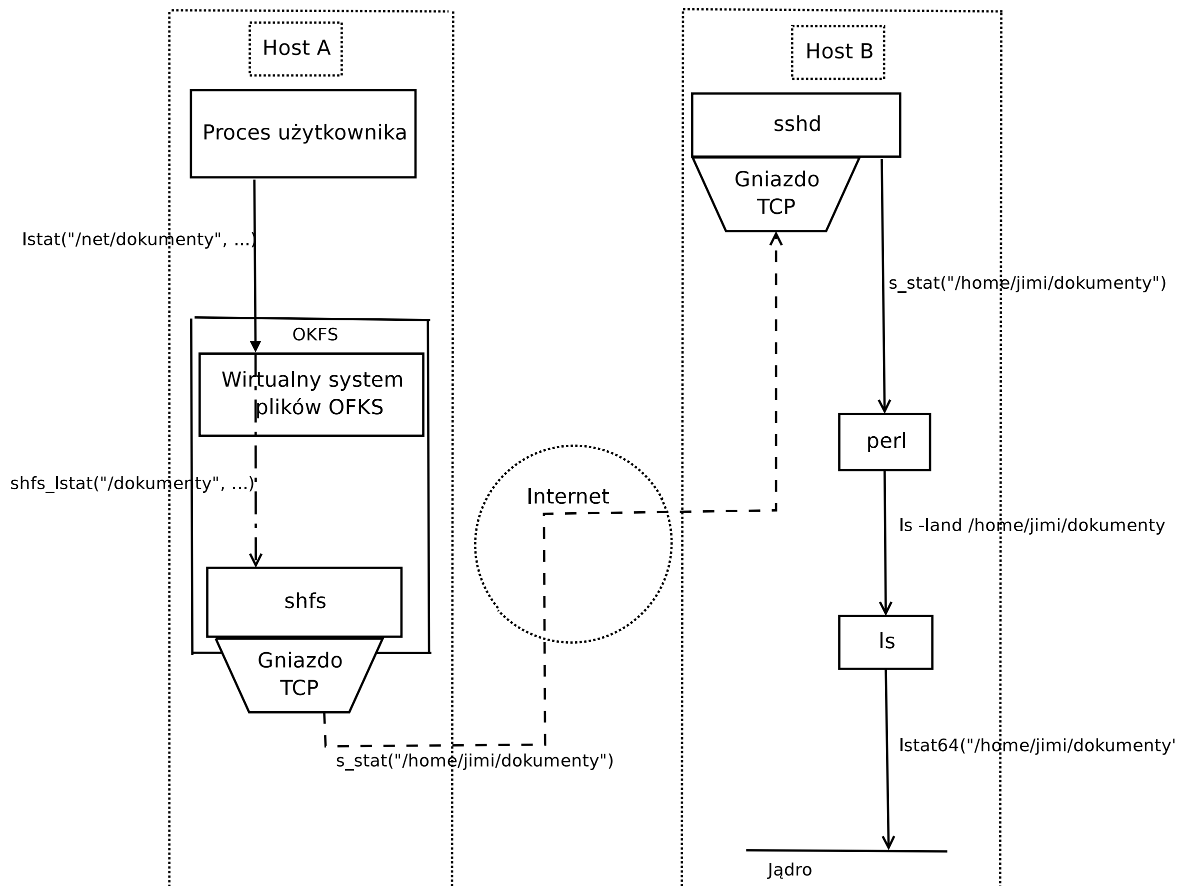
Tablica 9.2. Polecenia powłoki wykonywane przez SHFS po stronie serwera w celu realizacji odpowiednich funkcji systemowych. Jeśli serwer udostępnia interpreter Perla, większość tych poleceń jest zastępowanych przez znacznie szybsze wywołania funkcji Perla.

Funkcja systemowa	Polecenia powłoki
lstat	ls
readlink	readlink
open	>; chmod; dd
readdir	ls
read	dd
write	dd
link	ln
unlink	unlink
rename	rename
rmdir	rmdir
mkdir	mkdir
symlink	ln
utime	utime
chmod	chmod
truncate	>; dd
statfs	df
lchown	chown; chgrp

9.4.2. SHFS

(Secure) SHell FileSystem jest systemem plików o wiele bardziej złożonym i użytecznym niż LocalFS. Jego implementacja bazuje na kodzie istniejącego modułu jądra Linuksa o tej samej nazwie [22]. SHFS jest sieciowym systemem plików, wykorzystującym protokół SSH do połączenia ze zdalnym systemem, do którego plików użytkownik chce uzyskać dostęp. Taki system plików jest niezwykle przydatny ze względu na bardzo dużą popularność protokołu SSH. Wiele serwerów umożliwia użytkownikom logowanie z użyciem tego protokołu. SHFS daje użytkownikom możliwość dostępu na jednym komputerze do plików z wielu takich serwerów tak, jakby znajdowały się one lokalnie, co znacznie ułatwia pracę.

Połączenie SSH jest wykorzystywane do przekazywania do serwera poleceń, które następnie są wykonywane z użyciem funkcji interpretera języka Perl lub standardowych programów Uniksowych, dostępnych na większości serwerów. Jeśli po stronie serwera dostępny jest interpreter Perla, skrypt w Perlu jest wykorzystywany do interpretacji poleceń od klienta i wykonywania większości z nich. Jeśli interpreter Perla nie jest dostępny, używany jest mniej efektywny skrypt powłoki. Dostępność interpretera Perla o odpowiedniej funkcjonalności jest sprawdzana w momencie montowania systemu plików. Tabela 9.2 przedstawia, jakie programy są wykorzystywane po stronie serwera do realizacji poszczególnych operacji implementowanych przez rzeczywisty system plików na serwerze, na którym interpreter Perla nie jest dostępny. Przebieg komunikacji pomiędzy SHFS a zdalnym systemem posiadającym interpreter Perla przedstawia rysunek 9.3.



Rysunek 9.3. Uproszczony schemat przebiegu komunikacji z wykorzystaniem SHFS. Pokazano jedynie logiczne połączenie TCP pomiędzy OKFS, a zdalnym demonem SSHD. W rzeczywistości komunikacja przebiega poprzez jądra systemów A i B.

Jak już wspomniano, SHFS jest systemem plików o wiele bardziej złożonym od LocalFS. Implementacja składa się z prawie 3000 linii kodu w języku C, 570 linii w Perlu i 362 linii skryptu shella.

Rozdział 10

Efektywność OKFS

W tym rozdziale zostaną przedstawione i omówione wyniki testów wydajności systemu plików OKFS. Stworzony system stanowi dodatkową warstwę pomiędzy procesami użytkownika a systemem operacyjnym. Warstwa ta wprowadza narzut na czas wykonania operacji przez wszystkie procesy użytkownika, działające pod kontrolą OKFS, nie tylko te wykorzystujące jego systemy plików. Szczególny nacisk położono na przetestowanie jak duży jest to narzut i jaki ma wpływ na obszar możliwych zastosowań OKFS.

OKFS może zostać wykorzystany do implementacji różnych rzeczywistych systemów plików. Celem poniższych testów nie jest porównanie efektywności tych systemów. Obecnie dostępnych jest wiele prac porównujących efektywność różnych implementacji wielu rodzajów systemów plików. Przykładowo, porównanie efektywności operacji zapisu i odczytu systemu NFSv4, NFSv3 i SAMBA można znaleźć w pracy „NFSv4 Test Project” [23].

Należy zwrócić uwagę na to, iż porównanie samej tylko efektywności często może prowadzić do mylących wniosków na temat wyższości jednego systemu nad innym. Wynika to z faktu, że różne rzeczywiste systemy plików mogą służyć zupełnie różnym celom. Przykładem niech będzie porównywanie efektywności sieciowego systemu plików NFS i omówionego w tej pracy SHFS. NFS wymienia dane z użyciem efektywnego, dedykowanego protokołu, wykorzystującego najczęściej UDP. Kod serwera, w celu zwiększenia efektywności, jest często częścią jądra systemu. Dla kontrastu SHFS wymienia dane z użyciem wprowadzającego duży narzut, szyfrowanego protokołu, wykorzystującego protokół TCP. Po stronie serwera polecenia są wykonywane przy pomocy skryptu powłoki lub Perla. Do wykonania niektórych rodzajów poleceń wymagane jest utworzenie oddzielnego procesu. Biorąc pod uwagę samą tylko efektywność, NFS jest o wiele lepszym systemem, a stosowanie SHFS mija się z celem. Z drugiej strony, SHFS jest o wiele bardziej elastyczny od NFS. Umożliwia użytkownikowi montowanie katalogów z prawie każdego serwera Uniksowego, do którego ma dostęp. Wielu użytkowników

potrzebuje wykorzystać sieciowy system plików jedynie do transferu niewielkiej ilości małych plików ze swojego katalogu domowego. Przy tego typu operacjach różnica w wydajności pomiędzy SHFS a NFS jest zupełnie nieodczuwalna. A wygoda, jaką daje SHFS, i możliwość użycia go w sytuacjach, kiedy NFS jest niedostępny, powoduje, że SHFS często jest systemem lepszym.

Tak jak implementacja SHFS została dostosowana do działania z OKFS, tak samo można dostosować jedną z istniejących implementacji NFS. Testy mają na celu sprawdzenie, jaki jest koszt wykorzystania OKFS do implementacji rzeczywistego systemu plików w porównaniu z implementacją działającą na poziomie jądra systemu.

10.1. Metodologia testowania

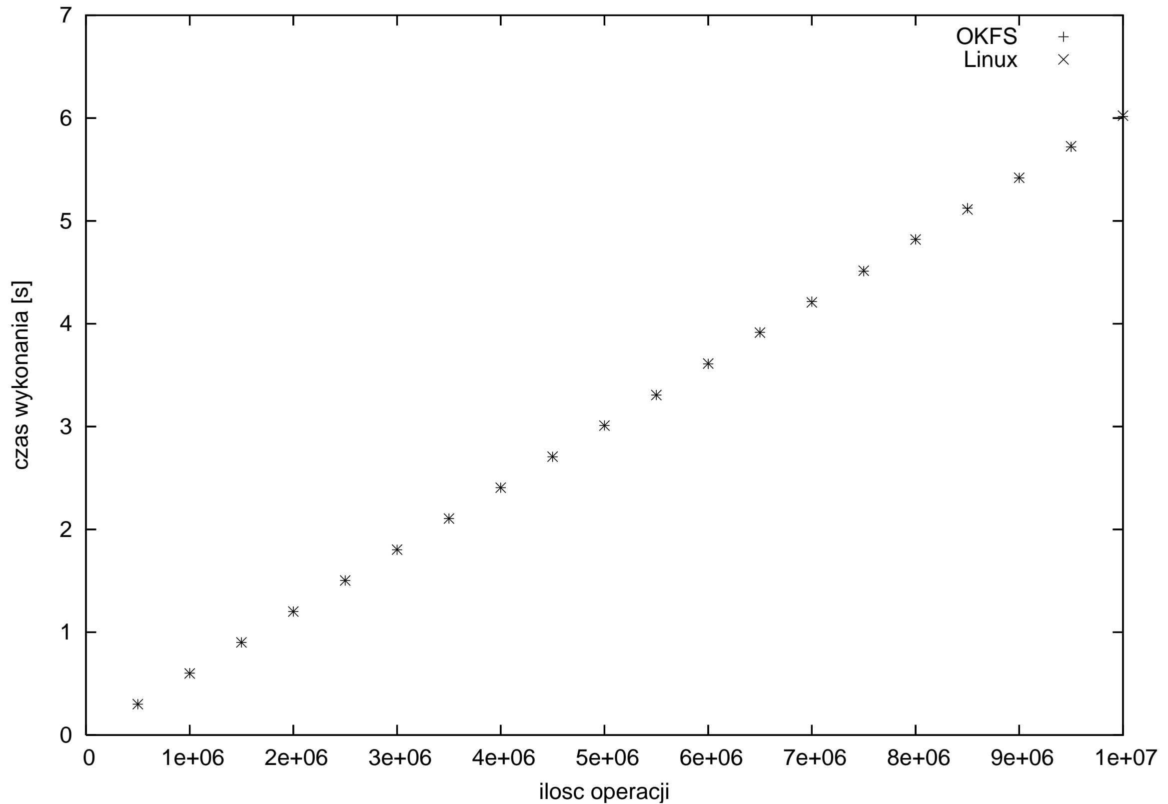
Wszystkie przeprowadzone testy polegały na zmierzeniu czasu wykonania pewnych działań pod kontrolą OKFS i odpowiadających im działań pod kontrolą samego tylko Linuksa. Zmierzony został rzeczywisty czas wykonania działania. Ponieważ na ten czas mogą mieć wpływ losowe czynniki, takie jak np. przerwania odbierane przez system operacyjny w trakcie testu, każdy test powtórzono dziesięciokrotnie. Wynik testu to wartość średnia, obliczona na podstawie tych 10 wywołań, po odrzuceniu dwóch minimalnych i dwóch maksymalnych wartości. Testy wykonano na nieobciążonym komputerze z procesorem AMD Athlon XP 1700+, pod kontrolą systemu operacyjnego Linux z jądrem w wersji 2.6.19.

10.2. Czas wykonania procesów obliczeniowych

OKFS tworzy wirtualne środowisko poprzez modyfikację wywołań systemowych, będących interfejsem pomiędzy procesem użytkownika a jądrem systemu operacyjnego. W odróżnieniu od kilku innych metod wirtualizacji, omówionych w tej pracy, OKFS nie interpretuje instrukcji wykonywanych przez proces. Dlatego też nie powinien mieć negatywnego wpływu na czas wykonania jakichkolwiek instrukcji procesora, poza instrukcją odpowiadającą za wywołanie funkcji systemowej (instrukcja *int 0x80* na procesorach Intel). W celu sprawdzenia tej właściwości przetestowano czas wykonania następującej pętli obliczeniowej:

```
int i;
double r = 2.7;
for(i = 0; i < cnt; i++){
  r *= sin(sin(log(cos(sin((double) i))))) / sin(1234123);
}
```

Testy wykonano dla wartości *cnt* zmieniającej się od 500000 do 10000000, co 500000. Zmierzone czasy wykonania pętli, wykonywanej zarówno przez proces działający pod kontrolą OKFS, jak i samego tylko Linuksa, przedstawia wykres 10.1. Wykres 10.2 przedstawia różnicę tych dwóch czasów wykonania dla odpowiednich wartości zmiennej *cnt*. Jak widać, różnica ta jest bardzo niewielka w porównaniu z czasem wykonania pętli i raz wskazuje na korzyść OKFS, a raz Linuksa. Różnica ta wynika najprawdopodobniej z losowych czynników, mających wpływ na pomiar czasu w wielozadaniowym systemie operacyjnym, a nie z szybszego wykonania pętli przez jeden z procesów. Test ten



Rysunek 10.1. Czas wykonania pętli obliczeniowej przez proces działający pod kontrolą OKFS i Linuksa.

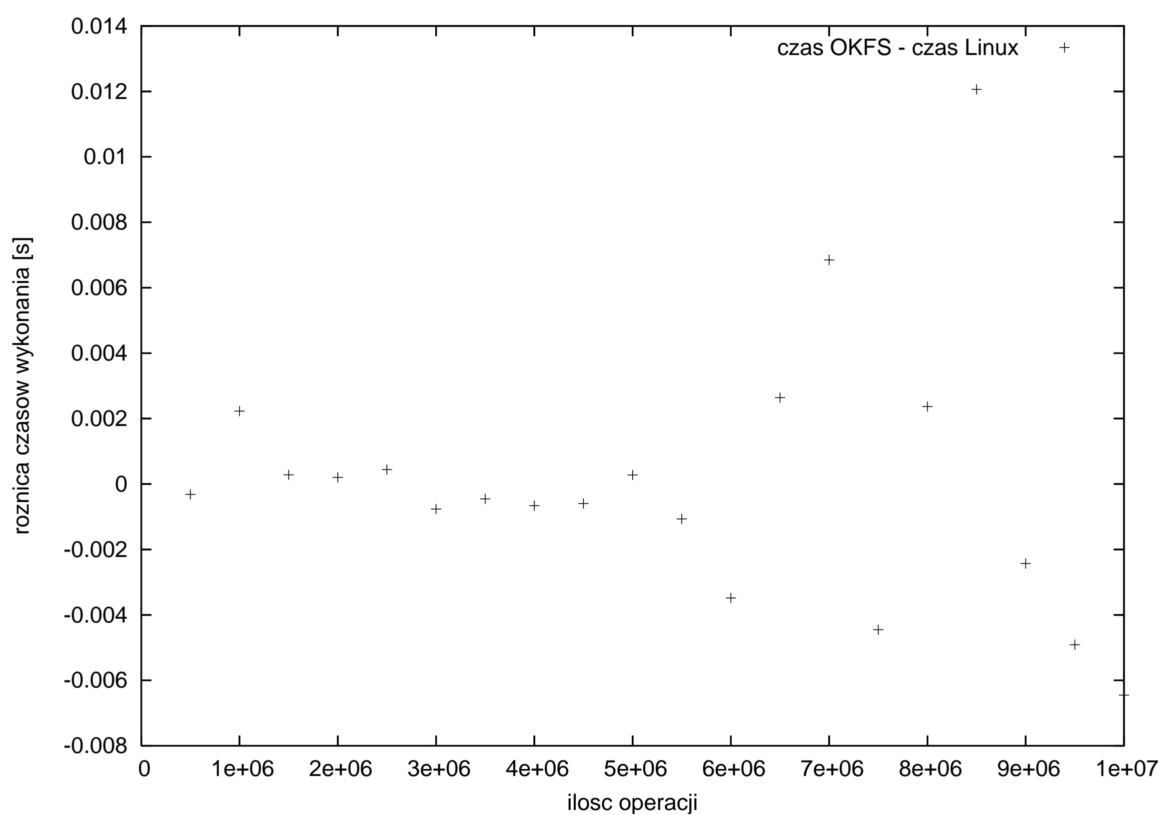
potwierdza tezę, że OKFS nie ma żadnego negatywnego wpływu na czas wykonania operacji innych niż wywołania systemowe.

10.3. Funkcje systemowe

Linuks przekazuje sterowanie do OKFS za każdym razem, gdy proces śledzony wywoła jakąkolwiek funkcję systemową, a także kiedy funkcja ta wykona instrukcję *return*. Funkcja *ptrace()* wykorzystywana przez OKFS, nie pozwala niestety na wybiórcze śledzenie tylko tych wywołań systemowych, które są istotne dla implementacji systemu plików. Możliwe jest jedynie śledzenia wszystkich wywołań. Wprowadza to dodatkowy narzut na czas wykonania wszystkich funkcji systemowych. Wynika on głównie z konieczności dodatkowego, czterokrotnego przełączenia kontekstu pomiędzy procesem śledzonym a OKFS.

Kiedy proces śledzony wywoła funkcję systemową, Linux budzi proces OKFS i rozpoczyna jego wykonanie. Po wykonaniu niezbędnych operacji przez OKFS, Linux przekazuje sterowanie do procesu śledzonego i funkcja systemowa jest wykonywana w kontekście tego procesu. Takie same działania są wykonywane, kiedy funkcja systemowa powraca.

Sam proces rozpoznania przez OKFS, jakie wywołanie systemowe wykonał proces



Rysunek 10.2. Różnica czasów wykonania pętli obliczeniowej przez proces działający pod kontrolą OKFS i Linuksa.

Tablica 10.1. Porównanie czasu wykonania kilku wybranych wywołań systemowych pod kontrolą OKFS i poza nim.

Funkcja systemowa	ilość wywołań (w tysiącach)	Średni czas [s]		Procentowy narzut czasowy OFKS
		Linux	OKFS	
gettimeofday	1000	0.291	8.996	2986%
sigaction	1000	0.65	11.2	1622%
getcwd	1000	1.12	26.02	2213%
chdir	1000	1.97	25.47	1195%
fork i exit	8	1.114	23.119	1975%

potomny i jakie działania należy w związku z tym podjąć, także wprowadza dodatkowy narzut czasowy.

Tabela 10.1 porównuje czasy wykonania kilku wybranych funkcji systemowych. Jak widać, narzut wprowadzany przez OKFS jest tym większy, im prostsza i krótsza jest funkcja systemowa. Czas wykonania najkrótszych funkcji systemowych może zostać wydłużony prawie trzydziestokrotnie. Dla przykładu, wywołanie *gettimeofday()* jest realizowane poprzez skopiowanie dwóch wartości liczbowych, definiujących bieżący czas (sekundy i mikrosekundy), do pamięci procesu wywołującego. Czas potrzebny na taką operację jest bardzo krótki w porównaniu z czasem potrzebnym na czterokrotne przełączenie kontekstu, jakiego wymaga OKFS. W przypadku bardziej skomplikowanych wywołań, takich jak *fork()* czy *chdir()*, narzut czasowy jest mniejszy, ale nadal spory.

Narzut czasowy na wywołanie funkcji systemowej wydaje się być duży. W praktyce nie stanowi on jednak większego problemu. Funkcje systemowe nie są tak często wywoływane, a znaczna część wywołań to wywołania długotrwałe, oczekujące na operacje wejścia-wyjścia. Dodatkowy narzut dla takich funkcji jest bardzo niewielki w porównaniu z czasem ich wywołania. Wpływ OKFS na czas wykonania długotrwałych funkcji przetestowano w następnym punkcie. W testach każdą testowaną funkcję (za wyjątkiem *fork()*) wywołano milion razy. Tabela 10.2 pozwala na porównanie tej liczby wywołań z ilością funkcji systemowych, wywoływanych przez kilka popularnych programów. Jak widać, milion wywołań to naprawdę dużo. W typowych zastosowaniach funkcje systemowe nie są wywoływane tak często, a narzut wprowadzany przez OKFS jest praktycznie niezauważalny.

10.4. Długotrwałe funkcje systemowe

Istnieje wiele funkcji systemowych, których wykonanie może zająć sporo czasu. Proces wywołujący taką funkcję może zostać uśpiony do momentu jej zakończenia. Do funkcji tych zaliczają się wszystkie te, które wymagają komunikacji ze sprzętem, która najczęściej jest długotrwała. Funkcje, wykonujące operacje na plikach, odwołują się do urządzeń, które umożliwiają dostęp do tych plików (dyski twarde, interfejsy sieciowe). Czas wykonania takich operacji jest spory w stosunku do prędkości działania procesora.

Znaczna część programów spędza większość czasu w oczekiwaniu na zakończenie operacji wejścia-wyjścia. Wystarczy przyjrzeć się wyjściu polecenia *ps aux*, które dla

Tablica 10.2. Ilość wywołań systemowych wykonywanych przez popularne programów w typowych zastosowaniach.

Operacja	Ilość wywołań systemowych
Odegranie trzyminutowego pliku mp3 z użyciem mplayera.	65688
Otwarcie Firefoxa, wejście na stronę www.slashdot.org, zamknięcie Firefoxa.	55703
Połączenie ze zdalnym serwerem z użyciem SSH i rozłączenie.	875
Uruchomienie edytora emacs, otwarcie 3 plików, zamknięcie edytora.	26866
Otwarcie sześćdziesięciostronnicowego dokumentu z użyciem xdvi.	18722

większości procesów pokazuje, że znajdują się one w stanie uśpienia (stan S), czyli że oczekują na jakieś zewnętrzne zdarzenie. Komenda *top* najczęściej pokazuje wykorzystanie procesora w okolicach 0%, pomimo wielu działających procesów.

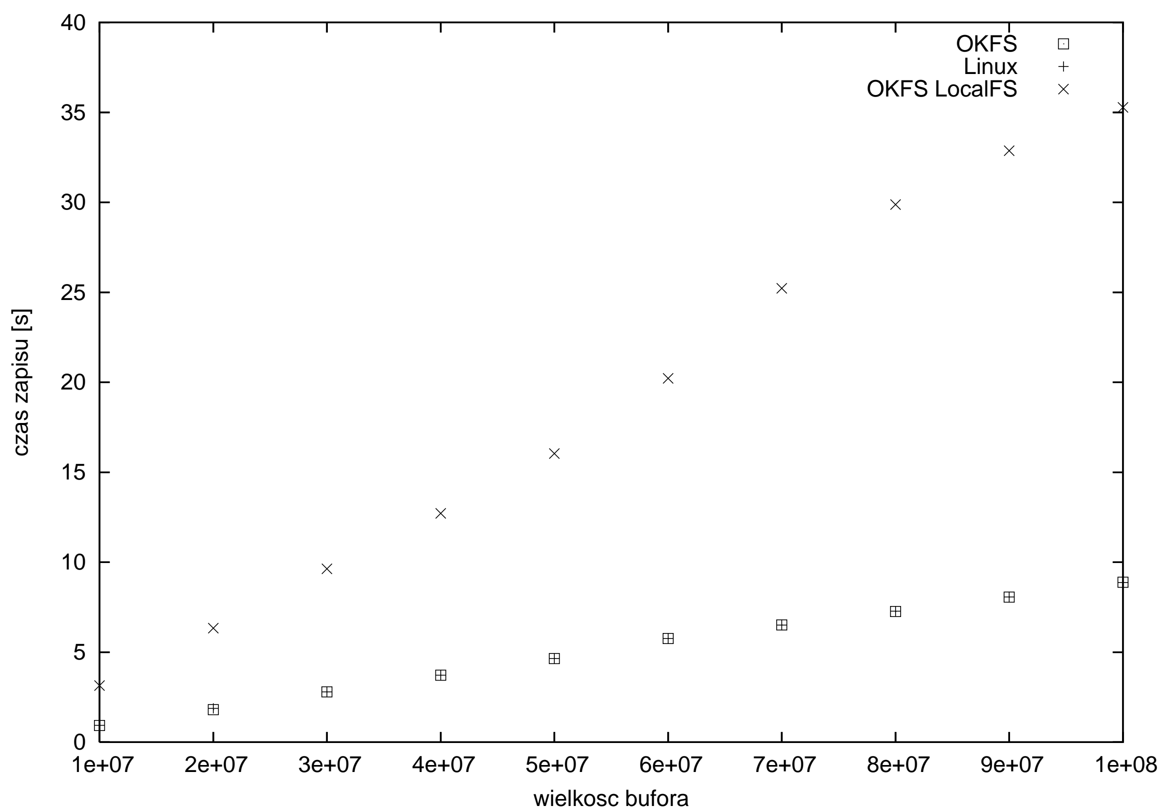
OKFS nie powinien mieć znaczącego wpływu na czas wykonania długotrwałych wywołań systemowych. Narzut czasowy, którego wymagają dodatkowe operacje wykonywane przez OKFS w momencie wywołania i powrotu z funkcji, powinien być niewielki w stosunku do czasu wykonania samej funkcji. W celu sprawdzenia tej tezy przeprowadzono testy wydajności operacji odczytu i zapisu w środowisku OKFS.

10.4.1. Operacje zapisu

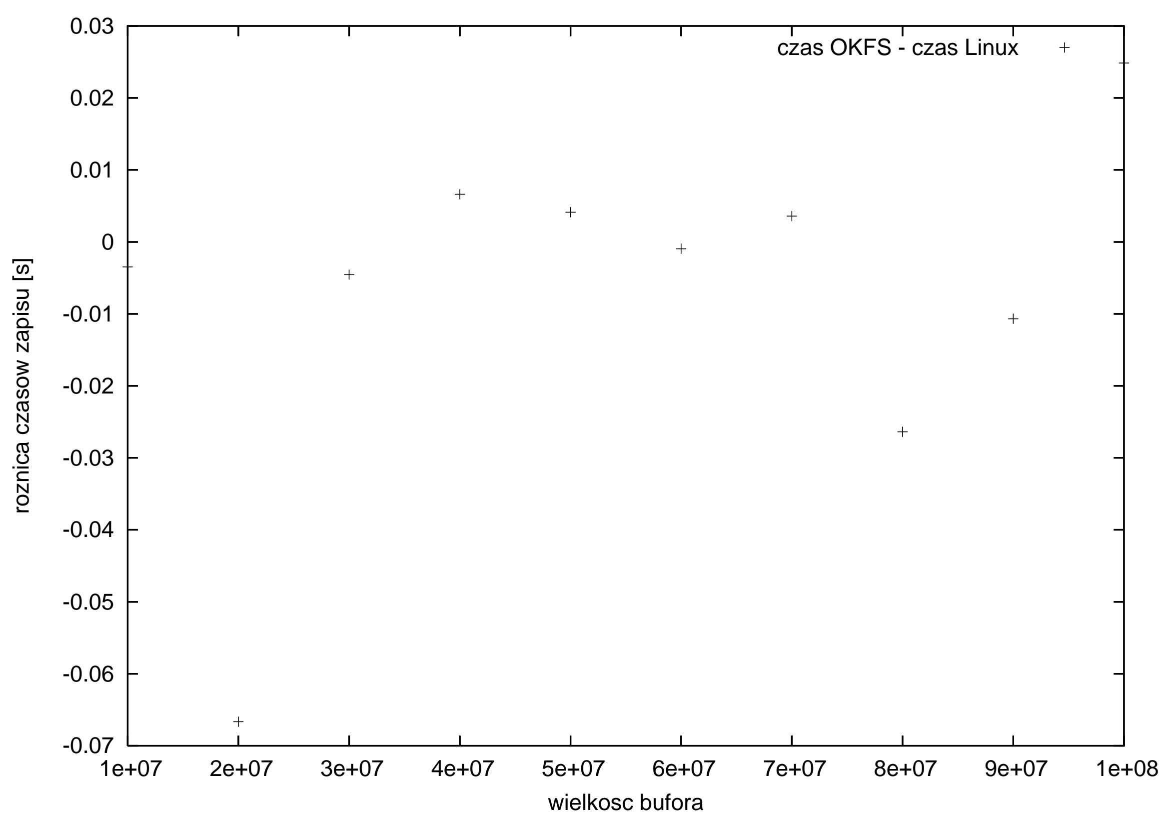
Wydajność operacji zapisu sprawdzono mierząc czas wykonania funkcji systemowych *write()* i *sync()*. Rozmiar danych zapisywanych przez funkcję *write()* dla kolejnych testów wynosił od 10MB do 100MB, zmieniając się co 10MB. Funkcję *sync()* wywoływano w celu wymuszenia zapisu na urządzeniu, a nie tylko do buforów pamięci podręcznej.

Przetestowano trzy przypadki. Zapis danych na Linuksowy system plików poza OKFS, pod kontrolą OKFS, a także zapis danych na rzeczywisty system plików OKFS (LocalFS). Wyniki przedstawiono na rysunku 10.3. Jak widać, czas zapisu na Linuksowy system plików, jest bardzo podobny, kiedy zapis jest realizowany przez proces, wykonywany w środowisku OKFS, oraz poza nim. Dodatkowo ilustruje to wykres 10.4, przedstawiający różnice tych dwóch czasów. Różnica ta jest niewielka w stosunku do czasu wykonania operacji zapisu, wynosi maksymalnie 0,07s. Tak jak w przypadku testu pętli obliczeniowej, różnica ta wskazuje raz na korzyść OKFS, a raz samego Linuksa, co pozwala przypuszczać, że wynika ona jedynie z niedokładności pomiaru. Potwierdza to tezę, że narzut, jaki wprowadza OKFS na czas wykonania długotrwałych funkcji systemowych, jest tak niewielki w porównaniu z czasem wykonania tych funkcji, że można go zaniedbać.

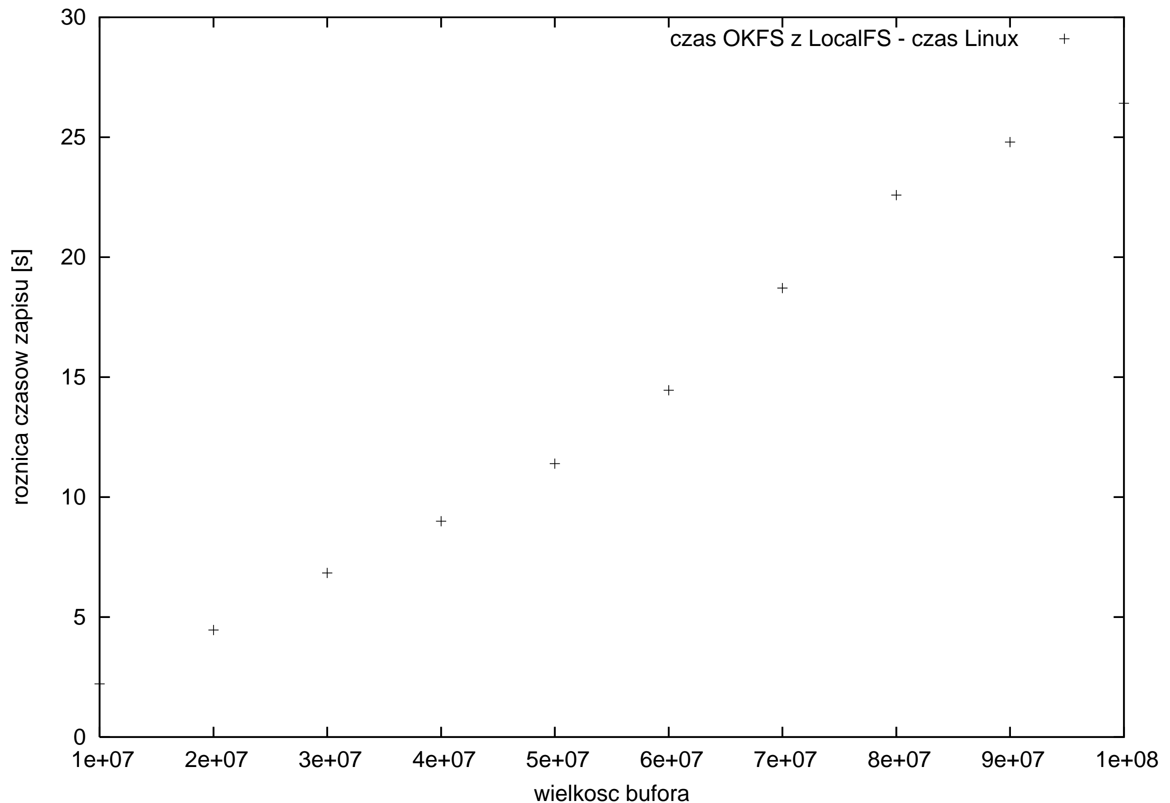
Kiedy zapis dotyczy pliku leżącego na LocalFS, wydajność tej operacji jest około 4 razy niższa. Wykres 10.5 ilustruje różnicę czasu zapisu na LocalFS i system plików



Rysunek 10.3. Porównanie wydajności operacji zapisu, realizowanych przez proces działający pod kotrolą OKFS i Linuksa. W przypadku OKFS przetestowano zarówno zapis do pliku, znajdującego się na systemie plików Linuksa, jak i na rzeczywistym systemie plików OKFS (LocalFS).



Rysunek 10.4. Różnica czasów zapisu do pliku będącego pod kontrolą Linuksa przez proces działający w środowisku OKFS i poza nim.

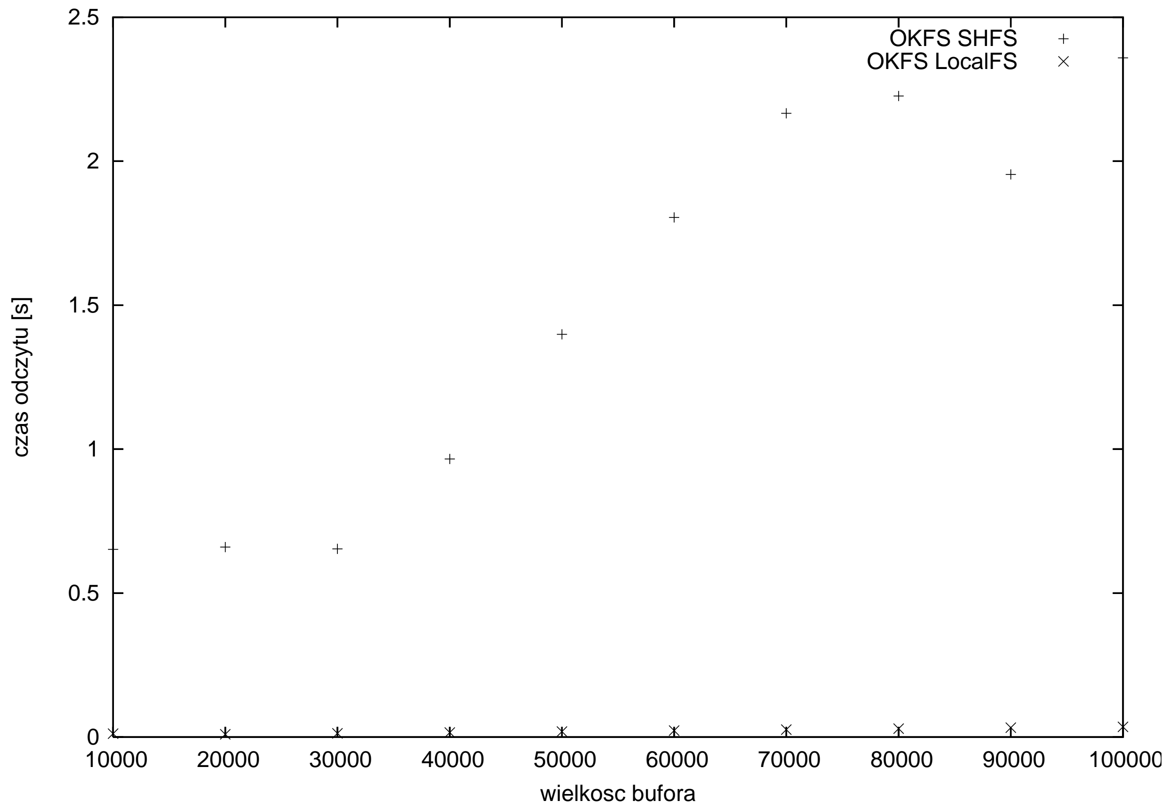


Rysunek 10.5. Różnica czasów zapisu do pliku znajdującego się na LocalFS i czasu zapisu do pliku znajdującego się na systemie plików Linuksa poza OKFS.

Linuksa poza OKFS. Dla 100MB pliku różnica ta wynosi prawie 25s. Warto przypomnieć, że LocalFS pozwala na montowanie lokalnego katalogu w innym miejscu. Wszystkie operacje zapisu są przekazywane do Linuksa. Poniżej znajduje się implementacja operacji zapisu:

```
int
localfs_write(struct okfs_file *fp, int fd, void *buf, size_t count)
{
  int ret;
  ret = write(fd, buf, count);
  return (ret < 0) ? -errno : ret;
}
```

Wywołanie funkcji *write()* przekazuje Linuksowi polecenie zapisu do pliku, którego deskryptor został utworzony przez LocalFS w momencie wywołania funkcji *open()*. Dlaczego w takim razie narzut czasowy na operacje zapisu jest taki duży, skoro, tak jak w dwóch pozostałych testowanych przypadkach, zapis jest ostatecznie wykonywany przez Linuksa? Wynika to z faktu, iż wirtualny system plików OKFS musi przekazać do rzeczywistych systemów plików zawartość bufora, który ma zostać zapisany. Bufor ten musi zostać odczytany z pamięci procesu śledzonego i to ta operacja jest źródłem dużego narzutu czasowego. OKFS wykorzystuje do odczytu danych z pamięci pro-



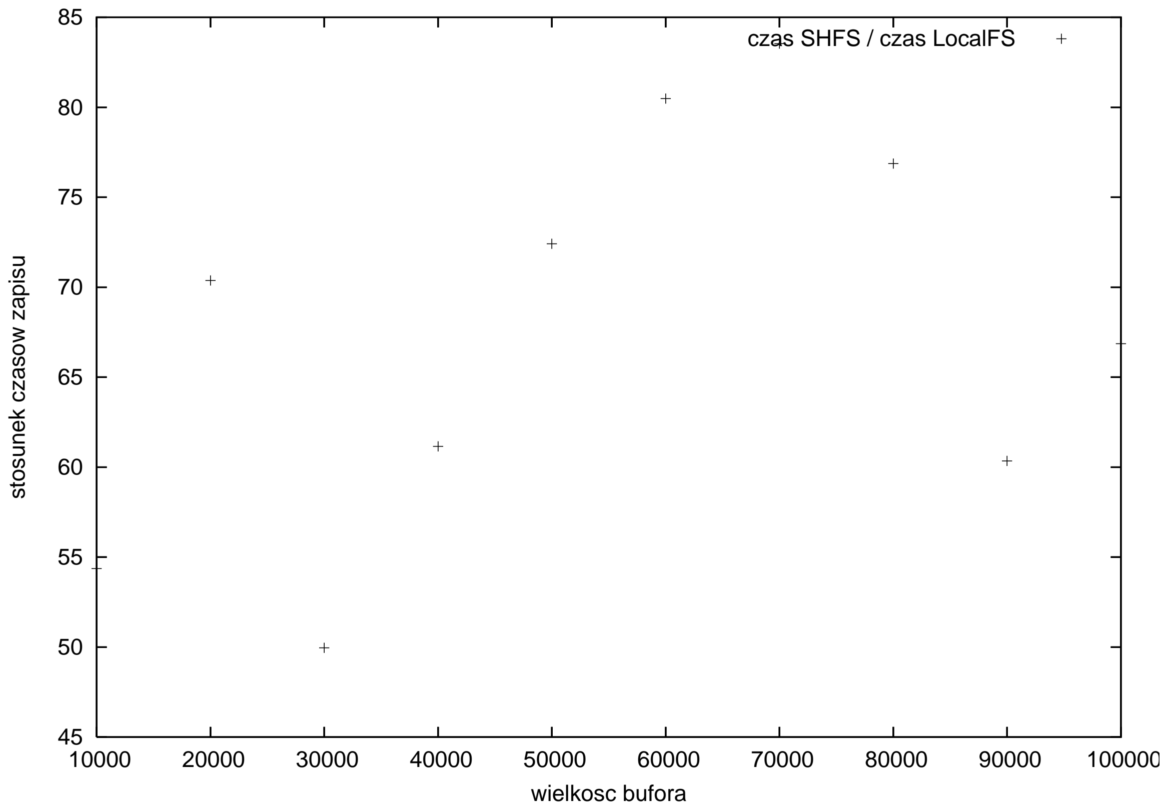
Rysunek 10.6. Czas zapisu do pliku znajdującego się w zdalnym katalogu, zamontowanym z użyciem SHFS, w porównaniu z czasem zapisu do pliku znajdującego się w lokalnym katalogu, zamontowanym z użyciem LocalFS.

cesu śledzonego funkcję *ptrace()* z opisaną już opcją *PTRACE_PEEKDATA*. Niestety, obecnie jedno wywołanie *ptrace()* umożliwia odczyt tylko jednego słowa z pamięci procesu śledzonego. Odczyt 100MB danych na 32 bitowym komputerze wymaga więc 25 milionów wywołań *ptrace()*.

10.4.2. Zapis na sieciowym systemie plików

Głównym zastosowaniem OKFS jest umożliwienie dostępu do sieciowych systemów plików. Poprzedni test pokazał, że zapis na rzeczywisty system plików OKFS, który umożliwia dostęp do lokalnych plików (LocalFS), jest obarczony sporym narzutem w porównaniu z analogicznym zapisem na system plików Linuksa poza OKFS. Jednak operacje sieciowe wprowadzają jeszcze większy narzut. W celu sprawdzenia, jak duży jest to narzut, przeprowadzono analogiczne testy zapisu na system plików SHFS. Z użyciem tego systemu zamontowano zdalny katalog z serwera, z którym komputer testowy był połączony łączem 40KB/s. Testowano prędkość zapisu mniejszych plików niż w przypadku zapisu lokalnego. Ich rozmiar był z zakresu od 10kB do 100kB i zmieniał się co 10kB. Wyniki przedstawia wykres 10.6.

Zgodnie z przewidywaniami czas zapisu do zdalnego katalogu jest znacząco dłuższy od czasu zapisu do katalogu lokalnego. Stosunek tych dwóch czasów przedstawia

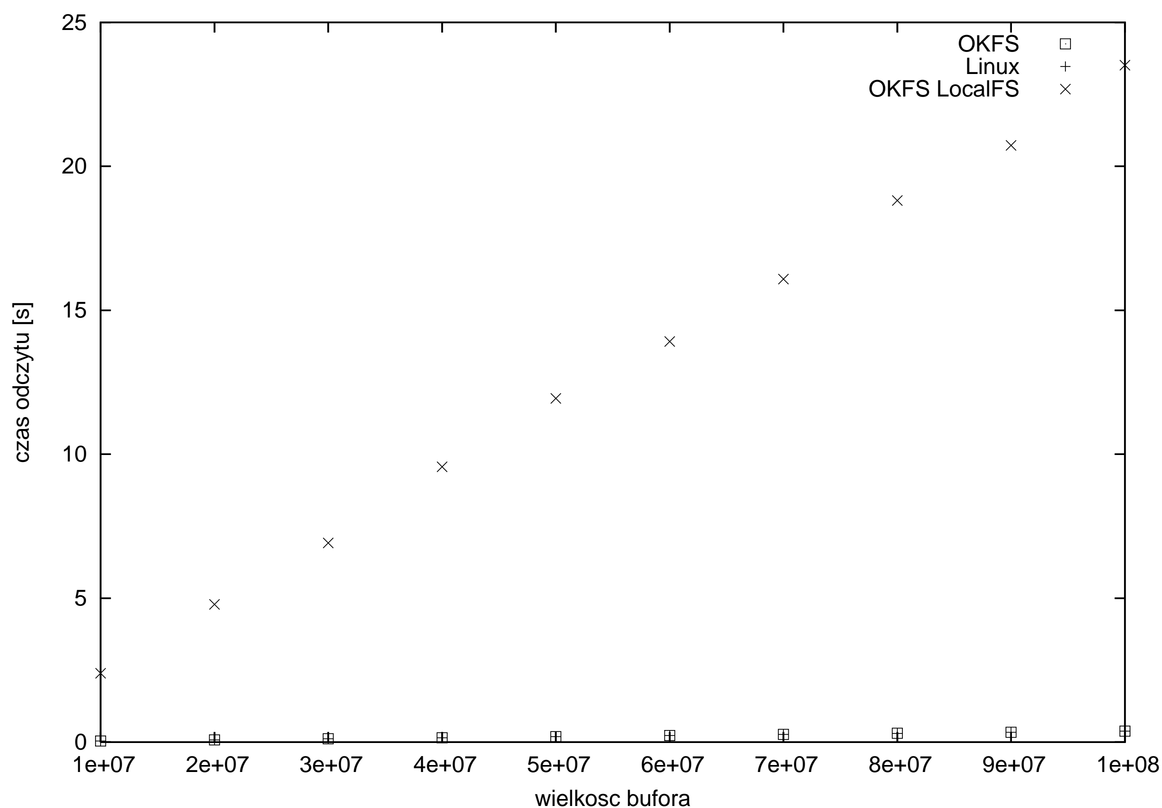


Rysunek 10.7. Czasu zapisu do pliku znajdującego się w zdalnym katalogu, zamontowanym z użyciem SHFS, w stosunku do czasu zapisu do pliku znajdującego się w lokalnym katalogu, zamontowanym z użyciem LocalFS.

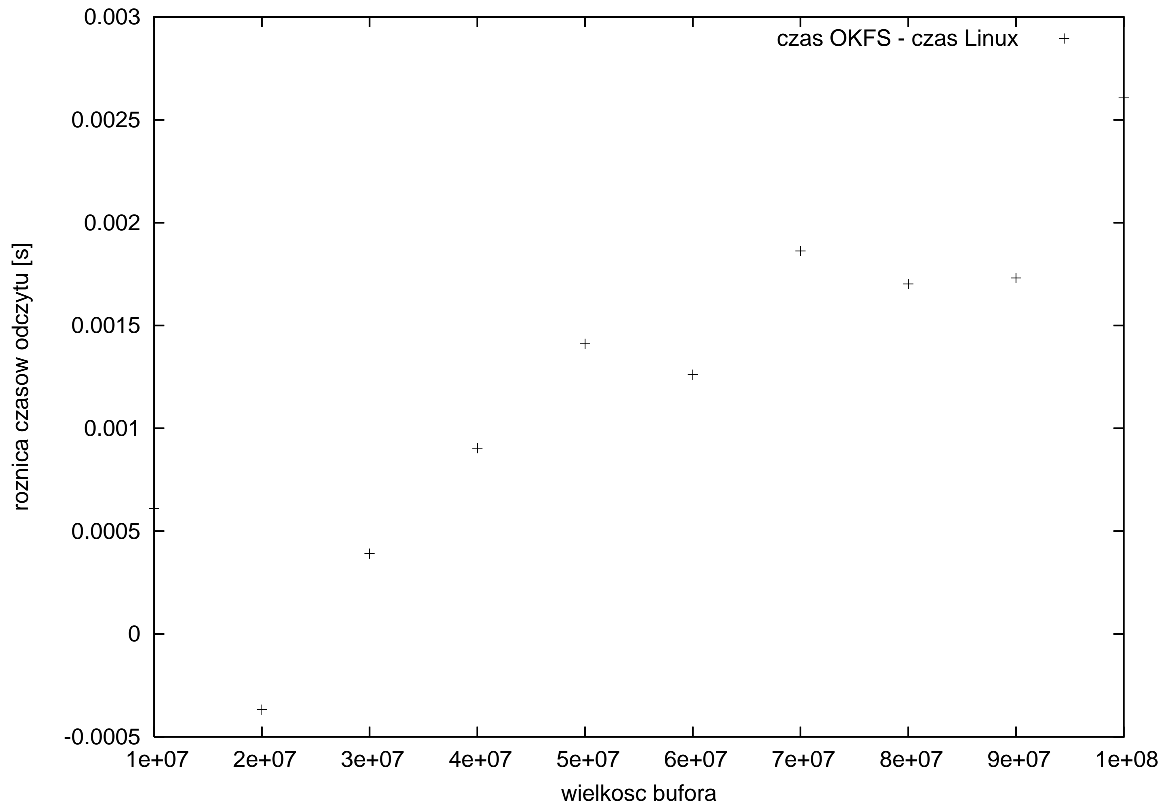
wykres 10.7. Wynika z niego, że operacje zdalne trwały w najlepszym przypadku aż 50 razy dłużej niż lokalne. Całkowity czas zapisu na LocalFS stanowi górne ograniczenie na narzut jaki wprowadzają opisane w poprzednim punkcie operacje odczytu z pamięci procesu zawartości zapisywanego bufora. Widać, że narzut ten jest tak mały w przypadku operacji na sieciowych systemach plików, że można go zaniedbać. Oczywiście narzut zależy od prędkości łącza, a także od rodzaju użytego rzeczywistego systemu plików. Wydaje się jednak, że czas, jakiego potrzebuje OKFS na dodatkowe kopiowanie danych do pamięci procesu, nie ma, w typowych zastosowaniach, istotnego wpływu na efektywność operacji zapisu.

10.4.3. Operacje odczytu

Dla operacji odczytu przeprowadzono testy analogiczne to testów efektywności operacji zapisu. Uzyskane wyniki przedstawia wykres 10.8. Operacje odczytu są znacznie szybsze od operacji zapisu. Wynika to z tego, że w przypadku funkcji *write()* możliwe było wymuszenie zapisu na fizycznym urządzeniu poprzez wywołanie *sync()*. W przypadku funkcji *read()* nie istnieje metoda wymuszająca odczyt z fizycznego urządzenia. Nawet jeśli bufory zostaną zapisane na dysk, i tak nadal są używane przy odczycie danych. Wielokrotne powtarzanie każdego testu w celu obliczenia średniego



Rysunek 10.8. Porównanie wydajności operacji odczytu, realizowanych przez proces działający pod kotrolą OKFS i Linuksa. W przypadku OKFS przetestowano zarówno odczyt z pliku znajdującego sie na systemie plików Linuksa, jak i na rzeczywistym systemie plików OKFS (LocalFS).



Rysunek 10.9. Różnica czasów odczytu z pliku, będącego pod kontrolą Linuksa, przez proces działający w środowisku OKFS i poza nim.

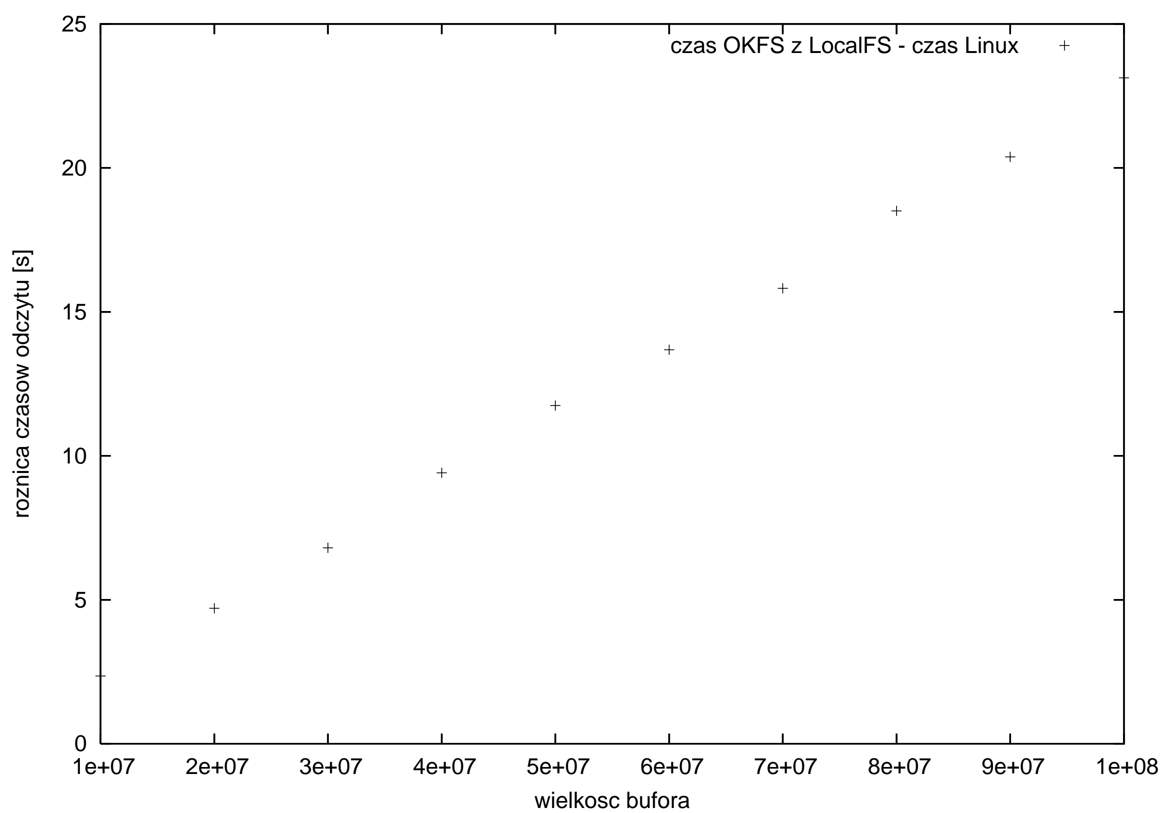
czasu operacji spowodowało, że odczytywane dane zawsze znajdowały się w pamięci podręcznej.

Tak jak w przypadku zapisu, różnice czasów odczytu z pliku, znajdującego się na systemie plików Linuksa pod kontrolą OKFS i poza nim, są bardzo niewielkie (maksymalnie niecałe 0,002s). Przedstawiono je na wykresie 10.9. W tym przypadku jednak wszystkie testy poza jednym wskazują na niekorzyść OKFS. Prawdopodobnie, ze względu na niewielki czas operacji odczytu, zaburzenia pomiaru czasu były na tyle niewielkie, że udało się wykryć narzut czasowy, jaki wprowadza OKFS.

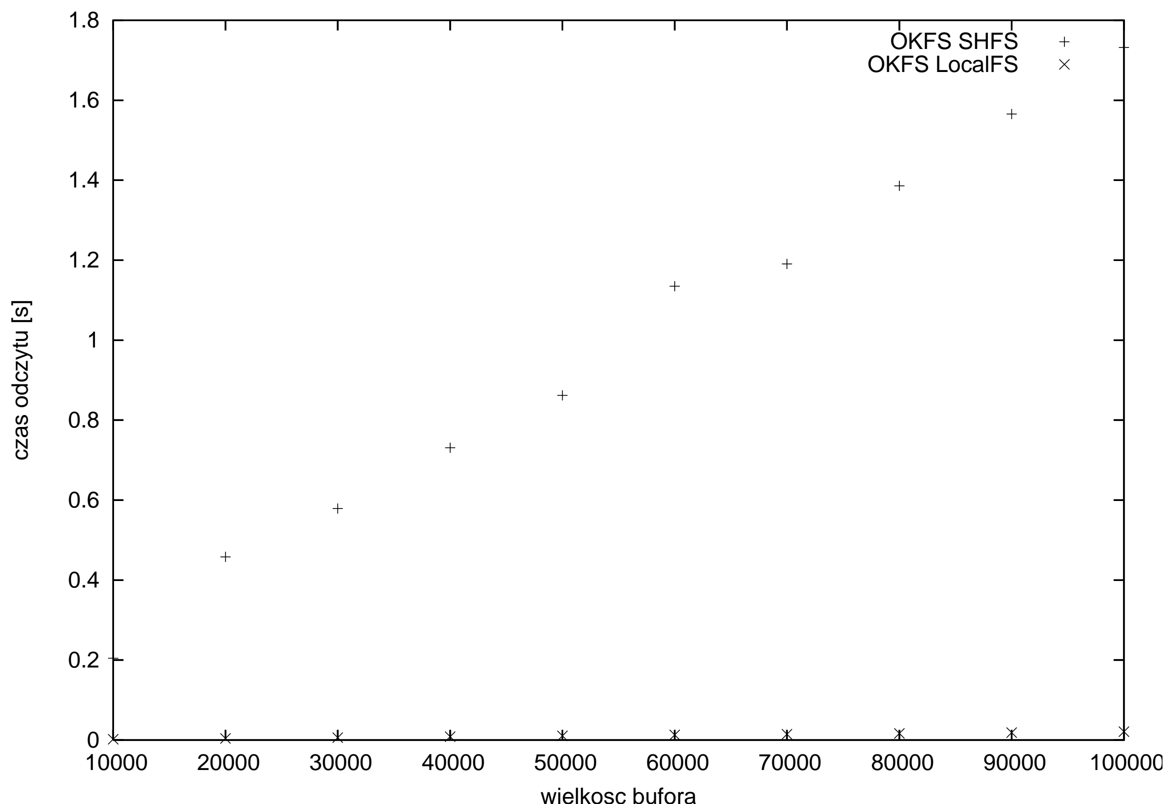
Odczyt z systemu plików LocalFS, podobnie do zapisu, wiąże się ze sporym narzutem. Przedstawia to wykres 10.10. Odczyt z rzeczywistego systemu plików OKFS wymaga zapisu odczytanych danych do pamięci procesy śledzonego. OKFS używa w tym celu funkcji *ptrace()* z opcją opcją *POKE_DATA*. Niestety, tak jak w przypadku opcji *PEEK_DATA*, istnieje możliwość zapisu tylko jednego słowa pamięci w jednym wywołaniu *ptrace()*.

10.4.4. Odczyt z sieciowego systemu plików

Również dla operacji odczytu z sieciowego systemu plików przeprowadzono testy, analogiczne jak dla operacji zapisu na taki system. Wyniki przedstawiają wykresy 10.11 oraz 10.12. Także w tym przypadku narzut, wprowadzany przez operacje sie-



Rysunek 10.10. Różnica czasów odczytu z pliku znajdującego się na LocalFS, i czasu odczytu z pliku, znajdującego się na systemie plików Linuksa poza OKFS.



Rysunek 10.11. Czas odczytu z pliku znajdującego się w zdalnym katalogu zamontowanym z użyciem SHFS, w porównaniu z czasem odczytu z pliku znajdującego się w lokalnym katalogu zamontowanym z użyciem LocalFS.

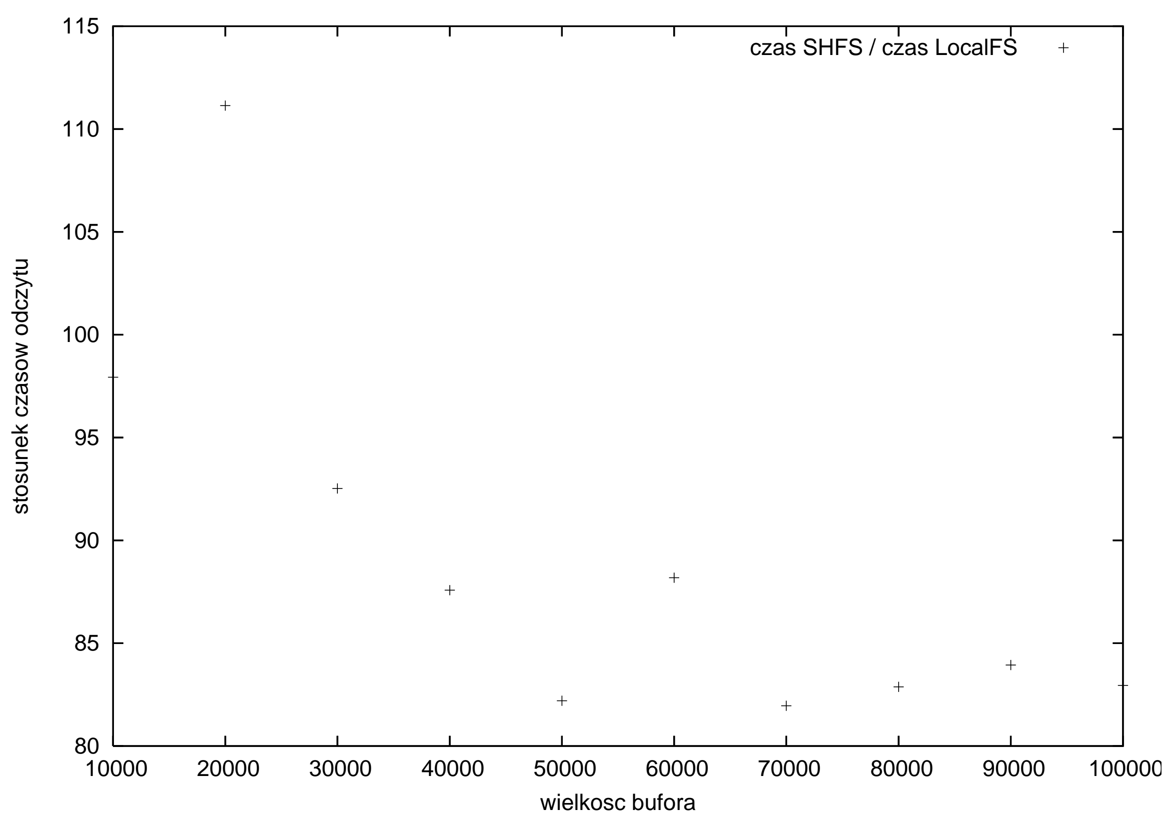
ciowe, okazał się znacznie większy od narzutu, wprowadzanego przez mechanizmy OKFS. Operacje sieciowe w najlepszym z testowych przypadków trwały 30 razy dłużej niż operacje lokalne.

10.5. Wnioski

W rozdziale tym przedstawiono testy efektywności OKFS. Pokazały one następujące cechy wirtualnego środowiska, opartego na funkcji *ptrace()*:

- brak narzutu na czas wykonania operacji innych niż wywołania systemowe,
- duży narzut na czas wykonania krótkotrwałych wywołań systemowych,
- niewielki w stosunku do czasu trwania funkcji, narzut na czas wykonania długotrwałych wywołań systemowych. W szczególności bardzo mały narzut na czas wykonania operacji sieciowych.

Na współczesnych, szybkich komputerach, wirtualizacja na poziomie wywołań systemowych nie wprowadza zauważalnego spadku wydajności w typowych zastosowaniach. Spadek ten może być jednak znaczący z punktu widzenia zastosowań wymagających maksymalnej wydajności.



Rysunek 10.12. Czas odczytu z pliku, znajdującego się w zdalnym katalogu, zamontowanym z użyciem SHFS, w stosunku do czasu odczytu z pliku, znajdującego się w lokalnym katalogu, zamontowanym z użyciem LocalFS.

Rozdział 11

Ograniczenia OKFS

Większość ograniczeń systemu OKFS ma swoje źródło w omówionych już cechach i niedoskonałościach mechanizmu *ptrace*. Wzrost popularności opisanego w tej pracy projektu UML i włączenie go do kodu źródłowego Linuksa daje nadzieję, że sporo istniejących niedoskonałości *ptrace'a* zostanie w przyszłych wersjach jądra wyeliminowanych. W punkcie tym omówione zostaną tylko wady, wynikające z architektury OKFS i mechanizmów przez niego wykorzystywanych. Wady, będące efektem niedoskonałości obecnej implementacji, zostaną omówione w następnym punkcie.

Poniżej zamieszczono listę słabych stron OKFS oraz przeprowadzono dyskusję możliwości ich wyeliminowania. I tak:

- *ptrace* jest mechanizmem specyficznym dla Linuksa, posiadającym różny interfejs dla różnych architektur, na których działa Linux. Czyni to OKFS projektem przeznaczonym jedynie dla Linuksa. Zapewnienie wsparcia dla architektur innych niż x86 wymaga przepisania części kodu OKFS. W obecnej wersji OKFS działa jedynie na procesorach z rodziny x86. Inne systemy Uniksowe posiadają mechanizmy analogiczne do *ptrace'a*, ale często o odmiennej semantyce. Dla przykładu Solaris pozwala na śledzenie procesów z użyciem systemu plików */proc*. W oparciu o ten interfejs został stworzony system plików poziomu użytkownika dla systemu Solaris, działający na podobnej zasadzie co OKFS [24].
  Można wyobrazić sobie zaprojektowanie OKFS w taki sposób, aby część kodu specyficznego dla Linuksa była dobrze oddzielona od reszty kodu. Umożliwiłoby to podmienienie tej części i dodanie wsparcia dla innych niż Linux systemów operacyjnych. Wydaje się to jednak być zadaniem bardzo trudnym w realizacji i znacznie komplikującym projekt.
- OKFS komunikuje się bezpośrednio z jądrem systemu z pominięciem biblioteki standardowej. Przez to OKFS musi używać interfejsu wywołań systemowych jądra, przekazywać do wywołań argumenty zgodne z tymi, których używa wersja jądra systemu, na której on działa. Dodatkowo nowe wersje jądra mogą udostępniać nowe

wywołania systemowe, których istnienia OKFS powinien być świadomy. OKFS przy kompilacji musi więc używać plików nagłówków jądra systemu, a nie tych będących częścią biblioteki standardowej.
Dla przykładu, struktura *stat* przekazywana w wywołaniu *lstat()* różni się nieznacznie w jądrach z serii 2.4 i 2.6. Zwykłe programy użytkowe nie muszą przejmować się tą różnicą, gdyż biblioteka standardowa wykonuje za nie translację pomiędzy różnymi wersjami tej struktury.
Te czynniki powodują, że OKFS powinien być skompilowany dla wersji jądra, na której będzie działał. Utrudnia to instalację systemu. Dodatkowo rozwój OKFS wymaga śledzenia zmian w jądrze i zapewniania współpracy z nowymi jego wersjami. OKFS jest pod tym względem bardzo podobny do modułów jądra, które także powinny być kompilowane dla wersji jądra, na której będą działać, i także muszą być świadome zmian w jądrze, które mogą wpływać na ich działanie [6].
Problem ten pomaga zminimalizować testowanie OKFS przy pomocy zestawu automatycznych testów. Odpowiadają one za sprawdzenie, czy funkcje OKFS działają poprawnie po zmianach w kodzie źródłowym, ale także po kompilacji na nową wersję jądra. Pozwala to na szybkie i łatwe wykrycie wielu problemów, a także zapobiega powrotowi raz wykrytych i naprawionych błędów.

- *ptrace()* jest funkcją bardzo specyficzną i słabo udokumentowaną. Często zrozumienie jakiegoś jej aspektu wymaga studiowania kodu źródłowego jądra. Niestety, niektóre wersje jądra zmieniają nieznacznie zachowanie tej funkcji. Zmiany te mają wpływ na działanie takich systemów jak OKFS czy UML. Jest to źródłem podobnych problemów, jak w przypadku zmian w interfejsach funkcji systemowych, omówionych w poprzednim punkcie. Również w tym przypadku testy automatyczne pomagają minimalizować negatywne skutki takich zmian.
- Ze względów bezpieczeństwa *ptrace()* nie pozwala na śledzenie programów, działających z innymi uprawnieniami niż posiada użytkownik, który je uruchamia (programy z ustawionym bitem setuid). W szczególności wszystkie programy, uruchamiane z prawami roota, nie mogą być śledzone przez użytkowników innych niż root. Dlatego też dla takich programów systemy plików, udostępniane przez OKFS, są niewidoczne. Jednak, przy typowym wykorzystaniu systemu UNIX, programy takie nie są często używane i mają bardzo konkretne, proste zastosowania. Brak dostępu takich programów do systemu plików OKFS nie wydaje się być istotnym ograniczeniem.
- Istnieją łaty na jądro Linuksa, ograniczające dostęp do funkcji *ptrace()* i przez to ograniczające możliwość używania OKFS.

Rozdział 12

Stan prac nad OKFS

OKFS jest projektem dostępnym na zasadach licencji GNU GPL. Jego kod źródłowy można pobrać pod adresem http://blues.ath.cx/okfs.tar.gz.

Projekt był jednym z 20 nagrodzonych w międzynarodowym konkursie IBM Linux Scholar Challenge [25]. Konkurs, do którego zgłoszono prawie 2000 prac, polegał na przedstawieniu innowacyjnego pomysłu rozszerzającego możliwości systemu Linux.

Poniżej znajduje się struktura katalogów kodu źródłowego OKFS wraz z krótkim opisem ich zawartości.

```
.
|-- doc
|   `-- [dokumentacja]
`-- src
    |-- [Kod jądra OKFS]
    |-- filesystems
    |   |-- [Wirtualny system plików OKFS]
    |   |-- localfs
    |   |   `-- [Kod LocalFS]
    |   `-- shfs
    |   |   `-- [Kod SHFS]
    |-- lib
    |   `-- [Ogólne funkcje wykorzystywane w wielu miejscach kodu]
    `-- tests
        `-- [Automatyczne testy]
```

Obecnie kod źródłowy składa się z 73 plików w języku C, których całkowita objętość wynosi ponad 10 tyś linii.

OKFS przyjmuje następujące argumenty linii poleceń:

```
Usage: ./okfs [options] [progname]
```

```
Options supported:
   --fstab file or -f file    Mount filesystems specified in file
   --fakeroot or -r           Fake root environment
   --tmpdir file or -t file   Specify temporary directory to use
   --help or -h               Print this help screen

[progname]     Specify program with arguments to run, default bash
```

Poprzez argument linii poleceń można podać ścieżkę do pliku w formacie fstab, zawierającą informacje o systemach plików, które OKFS powinien zamontować. Poniżej znajduje się przykład pliku fstab, przekazywanego do OKFS:

```
#Zamontowanie katalogu /usr/bin do /usr/sbin.
/usr/bin  /usr/sbin  localfs

#Przykład użycia SHFS.
#Zamontowanie zdalnego katalogu /home/jimi z serwera
#little.wing.art, do którego OKFS zaloguje się jako
#użytkownik jimi, do lokalnego katalogu /home/wood/net/.
jimi@little.wing.art:/home/jimi  /home/wood  shfs

#Można także łączyć się z serwerem SSH działającym na
#niestandardowym porcie.
jimi@little.wing.art:/home/jimi  /home/wood  shfs  port=8765
```

12.1. Dalszy rozwój systemu.

Obecna wersja OKFS jest wersją demonstracyjną. Bez problemów działają pod jej kontrolą standardowe narzędzia Uniksowe. Brakuje jednak jeszcze kilku rzadziej używanych funkcji, wykorzystywanych przez niektóre bardziej zaawansowane programy. Wspomniane funkcje to:

- obsługa wątków. Wątki, w odróżnieniu od procesów, współdzielą pomiędzy sobą całą pamięć z wyjątkiem stosu. Niektóre funkcje OKFS wymagają modyfikacji pamięci procesów śledzonych. Dodanie obsługi wątków wymaga zadbania o to, aby wszystkie funkcje modyfikujące pamięć procesu operowały jedynie na stosie, lub aby inne wątki zostały zatrzymane do momentu przywrócenia pierwotnego stanu pamięci.
- Mapowanie plików do pamięci z użyciem funkcji systemowej *mmap()*. Funkcja *mmap()*[26] pozwala na mapowanie pliku do pamięci procesu. Wszystkie operacje zapisu do tej pamięci zostają ostatecznie odzwierciedlone w pliku, który został zmapowany.
- Blokowanie deskryptorów plików . Funkcja *fcntl()* umożliwia blokowanie plików, w celu zapewnienia synchronizacji dostępu do nich dla wielu procesów. Większość sieciowych systemów plików nie wspiera blokowania globalnego. Oznacza to, iż blokada, założona na plik na jednym komputerze montującym sieciowy system plików, jest widoczna tylko dla procesów działających na tym komputerze. Pozostałe kom-

puterу, montujące ten sam sieciowy system, plików nie widzą tej blokady. Istnieją jednak sieciowe systemy plików, pozwalające na globalne blokowanie plików, np. najnowsza wersja NFS. OKFS nie wspiera obecnie ani globalnego, ani i lokalnego blokowania plików, znajdujących się na jego rzeczywistych systemach plików.
- Montowanie i odmontowywanie systemów plików w czasie działania OKFS. Obecnie wspierane jest jedynie montowanie systemów plików, wymienionych w pliku fstab, podawanym przy starcie OKFS.
- Funkcje wykonywane przez OKFS nie mogą być obecnie przerywane. Jeśli proces wywoła funkcję, operującą na zdalnym pliku, której wykonanie trwa długo ze względu na obciążenie sieci, żadne sygnały nie będą dostarczane do tego procesu w czasie wykonywania tej funkcji.

Dużym ulepszeniem architektury OKFS, znacznie zwiększającym możliwości tego programu, byłoby zintegrowanie go z opisanym w tej pracy systemem FUSE. OKFS definiuje obecnie własny interfejs pomiędzy swoim wirtualnym systemem plików, a rzeczywistymi systemami plików. Interfejs ten projektowano tak, aby był jak najprostszy w użyciu, pomimo tego i tak konieczne jest dostosowywanie różnych rzeczywistych systemów plików do współpracy z OKFS. Zmiana tego interfejsu w taki sposób, aby był kompatybilny z interfejsem, definiowanym przez FUSE, pozwoliłaby na używanie w OKFS sporej liczby rzeczywistych systemów plików FUSE. Nie wymagałoby to żadnych zmian w kodzie źródłowym tych systemów plików i pozwoliłoby w prosty sposób znacznie powiększyć obszar możliwych zastosowań OKFS.

Rozdział 13

Podsumowanie

W pracy tej została przedstawiona koncepcja, architektura i implementacja systemu plików, działającego całkowicie poza jądrem systemu operacyjnego. Pokazano, że to nowe rozwiązanie posiada, w porównaniu z dostępnymi obecnie systemami plików, wiele zalet. Może być ono szczególnie użyteczne dla właścicieli kont na wieloużytkownikowych serwerach Linuksowych. Nie mają oni często kontroli i wpływu na to, jakie moduły jądra są załadowane. Nie mogę więc korzystać z systemów takich jak FUSE, działających częściowo na poziomie jądra. Funkcja *ptrace()* jest jednak standardowo dostępna na większości systemów Linuksowych. Pozwala ona na znaczne zwiększenie wygody pracy zwykłych użytkowników serwerów, poprzez dostęp do takich systemów, jak UML i OKFS.

Zaproponowana w tej pracy architektura systemu plików jest także rozwiązaniem bezpieczniejszym od innych, obecnie dostępnych. Cały kod systemu plików działa na poziomie użytkownika i nie wymaga żadnych dodatkowych uprawnień. Dzięki temu znalezienie błędu w kodzie systemu plików umożliwia przejęcie kontroli jedynie nad kontem jednego użytkownika, a nie nad całą maszyną.

W pracy tej przedstawiono również wyniki testów, które pozwoliły ocenić, z jak dużym narzutem wiąże się korzystanie z OKFS. Wyniki te są także przydatne do oceny narzutu, wprowadzanego przez UMLa. Liczne testy oraz praca z wykorzystaniem systemu OKFS pozwoliły na jednoznaczne określenie jego słabych stron. Stały się one podstawą do określenie kierunków jego dalszego rozwoju, zarówno od strony poprawy efektywności zaproponowanego rozwiązania, jak również pod względem zwiększenia funkcjonalności. Osobną kwestią jest zapewnienie jego przenośności.

Bibliografia

[1] Maurice J. Bach: Budowa systemu operacyjnego UNIX, Wydawnictwo Naukowo-Techniczne, Warszawa, 1995

[2] Moshe Bar: LINUX systemy plików, Wydawnictwo RM, 2002

[3] Richard Gooch: Overview of the Virtual File System, Kod źródłowy Linuksa, plik Documentation/filesystems/vfs.txt

[4] Jeff Dike: User Mode Linux, Prentice Hall, 2006

[5] Richard Jones: Gmail Filesystem,
http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html

[6] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman: Linux Device Drivers, Third Edition, O'Reilly,,2005

[7] Filesystem in Userspace,
http://fuse.sourceforge.net/

[8] http://en.wikipedia.org/wiki/Virtualization oraz
http://pl.wikipedia.org/wiki/Wirtualizacja

[9] the Bochs IA-32 Emulator Project,
http://bochs.sourceforge.net/

[10] QEMU a generic and open source machine emulator and virtualizer,
http://fabrice.bellard.free.fr/qemu/

[11] VMware Desktop Virtualization,
http://www.vmware.com/

[12] Parallels Desktop for Mac,
http://www.parallels.com/products/desktop/

[13] the Linux-VServer Home Page,
http://linux-vserver.org/

[14] Solaris Containers-Resource Management and Solaris Zones,
http://docs.sun.com/app/docs/doc/817-1592

[15] Poul-Henning Kamp, Robert N. M. Watson: Jails: Confining the omnipotent root,
http://docs.freebsd.org/44doc/papers/jail/jail.html

[16] The User-mode Linux Kernel Home Page,
http://user-mode-linux.sourceforge.net/

[17] Linux man pages, ptrace() manual,
http://www.die.net/doc/linux/man/man2/ptrace.2.html

[18] Seth Nickell: GnomeVFS - Filesystem Abstraction library,
http://developer.gnome.org/doc/API/gnome-vfs/

[19] Bernd Gehrmann: KDE's IO architecture,
http://developer.kde.org/documentation/library/kdeqt/kde3arch/nettransparency.html

[20] Linux man pages, fstab manual,
http://www.die.net/doc/linux/man/man5/fstab.5.html

[21] Linux Userland FileSystem,
http://lufs.sourceforge.net/

[22] (Secure) SHell FileSystem Linux kernel module,
http://shfs.sourceforge.net/

[23] Aurelien Charbon et. al.: NFSv4 Test Project,
http://nfsv4.bullopensource.org/doc/OLS06/NFSv4_test_project.pdf

[24] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, Chris J. Scheiman: Extending the Operating System at the User Level: the Ufo Global File System, http://www.cs.ucsb.edu/research/ufo

[25] The Linux Scholars Challenge,
http://www-304.ibm.com/jct09002c/university/students/contests/linux/

[26] Linux man pages, mmap manual,
http://www.die.net/doc/linux/man/man3/mmap.3.html