Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Rafał Wijata
Nr albumu: 171743

Personalizacja operacji na plikach z poziomu użytkownika

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Grudzień 2001

Pracę przedkładam do oceny

Data Podpis autora pracy:

Praca jest gotowa do oceny przez recenzenta

Data Podpis kierującego pracą:

Streszczenie

Tematem tej pracy jest problem zmiany sposobu dostępu do plików w systemie operacyjnym. Realizacja założeń opisanego tu projektu pozwala na aktywne uczestnictwo użytkownika w definiowaniu sposobu realizacji funkcji dostępu do plików. Pozwala to na zmianę działania zakodowanych w interfejsie systemu operacyjnego funkcji realizujących dostęp do obiektów systemu plików. Specjalny moduł pozwala realizować te funkcje przez procesy użytkownika, a nie tylko poprzez uprzywilejowany tryb jądra. Pozwala to na personalizację obsługi plików przez użytkowników systemu, bez potrzeby ingerencji w kod systemu operacyjnego (a co za tym idzie potrzeby posiadania praw administratora). Przedstawiam tu różne sposoby podejścia do zagadnienia oraz wyniki mojej pracy programistycznej. Do pracy jest dołączona działająca implementacja dla systemu Linux 2.4.

Słowa kluczowe

Systemy plików, VFS, strażnicy plików, podmiana operacji na plikach, realizacja funkcji systemowych za pomocą procesów użytkownika

Klasyfikacja tematyczna

D. Software
D.4 Operating Systems
D.4.3 File System Management
Access Methods
User intervention in FileSystem implementation

Spis treści

Spis tablic

Rozdział 1

Wstęp

1.1. Wprowadzenie do problematyki pracy

W dzisiejszych czasach intensywnie rozwija się systemy plików. Prawie co dzień wchodzą do użytku nowe rozwiązania techniczne dla pamięci masowych, a rozmiary dysków twardych przekraczają wcześniejsze przewidywania. Okazało się, że stare implementacje systemów plików nie są wystarczające pod względem dzisiejszych wymagań co do funkcjonalności i szybkości dostępu do danych zapisanych na nośnikach pamięci masowych. Właściwie wszystkie nowe rozwiązania zawierają księgowanie (ang. *journalling* — za pomocą logu transakcyjnego) operacji, są 64-bitowe, oferują bardzo szybkie wyszukiwanie danych na dysku. Stosowane są nowe struktury danych do opisu rozmieszczenia plików na dysku, skracające czas wykonywanych na plikach operacji. Wszystkie te zabiegi powodują, iż systemy plików są coraz sprawniejsze i bardziej niezawodne. Księgowanie zapewnia szybkie doprowadzenie systemu do stanu, w którym dane na dysku są spójne — czyli nie ma w nich błędów. To niweluje do minimum czas przestoju systemu w razie awarii. Nowe rozmieszczenie danych na wolumenie gwarantuje szybkie wyszukiwanie danych, a także efektywne wykorzystanie powierzchni urządzenia (nawet przy rozmiarach osiągających setki gigabajtów).

Jednak niezależnie od tego jak nowoczesne są te nowe systemy plików, brakuje im pewnej właściwości — są tak samo sztywne, jak ich pierwowzory. Wszystkie jednakowo traktują pliki w obrębie partycji. Z każdym plikiem są związane te same operacje dostępu, zakodowane gdzieś wewnątrz jądra systemu operacyjnego, skutecznie ograniczając możliwości wykorzystania potencjału plików w inny sposób. To oznacza, że jeśli chcielibyśmy, aby plik nie był już tylko pasywnym obiektem systemu plików do przechowywania danych, lecz dopuszczał dynamiczną zmianę semantyki operacji dostępu, to nie będziemy w stanie tego zrealizować. Statyczność jest tak mocno zakodowana, że nie tylko dostęp, ale również zawartość plików, ich przestrzeń nazw, możliwe typy operacji są zakodowane gdzieś w środku systemu i nie ma możliwości zmiany tego zgodnie z własnymi potrzebami. Ktoś mógłby powiedzieć, że nawet w starym extended2[1] są różne typy plików i można je traktować na różne sposoby. Ale to nadal nie zmienia faktu, że jeden typ pliku jest traktowany przez system zawsze w taki sam sposób. Obowiązuje z góry zdefiniowana semantyka praw dostępu, otwierania, czytania, pisania itd.

Tak konserwatywne podejście spowodowało, że zadałem sobie pytanie: czy jest sposób, aby to zmienić? Czy jest możliwa dynamiczna zmiana semantyki operacji dostępu do obiektów systemu plików? Czy użytkownik może sam decydować, które operacje na pliku są dozwo-

[1] ang. *the second extended filesystem* (również extended2 i ext2). Jest to najczęściej wykorzystywany system plików w Linuksie. Teraz, na jego bazie, tworzony jest extended3, zawierający dodatkowo księgowanie operacji na plikach.

lone, które nie, a które należy zmodyfikować? Oczywiście wszystko zgodnie z potrzebami administratora lub użytkownika systemu. Po przejrzeniu skromnej literatury na ten temat doszedłem do wniosku, że jest to możliwe. W tej pracy zamierzam opisać sposoby rozwiązania problemu oraz implementację jednego z nich w systemie Linux.

1.2. Historia zagadnienia

Okazało się, że problem nie jest nowy, choć również niezbyt popularny. Prawdopodobnie jako pierwsi nad podobnym pomysłem zastanawiali się Brian N. Bershad i C. Brian Pinkerton w roku 1987. Wówczas obaj byli pracownikami Uniwersytetu w Waszyngtonie. Swój projekt Watchdogs opisali w pracy *"Watchdogs — Extending the UNIX File System"* [1]. Postanowili rozszerzyć funkcjonalność systemów plików za pomocą przekierowań funkcji systemowych do programów użytkownika. Programy te, nazwane strażnikami (ang. *watchdog*), realizują operacje na plikach. Do każdego pliku można przyporządkować strażnika. Jeśli jakiś plik jest powiązany ze strażnikiem, to system operacyjny przy próbie dostępu do pliku podczas realizacji właściwego wywołania systemowego przekazuje sterowanie do procesu strażnika. W ten sposób pomijamy kod systemu operacyjnego, w którym jest standardowa procedura obsługi.

Projekt Watchdogs został zaimplementowany w systemie 4.3BSD UNIX. W rozszerzonej wersji systemu znacznie zwiększyły się możliwości systemu plików. Otrzymane rozwiązanie okazało się proste, wystarczająco szybkie i elastyczne. Pozwala ono np. na realizację kompresji pliku w locie — plik jest kompresowany podczas zapisu na dysk i rozkompresowywany podczas odczytu. Niektóre z dzisiejszych systemów plików oferują podobne możliwości (jak np. kompresja w locie) bez potrzeby odwoływania się do strażników, ale uogólniają one rozwiązanie na cały wolumen (wszystkie pliki są kompresowane i to tym samym algorytmem). Dzięki rozwiązaniu Watchdogs użytkownik może również zdefiniować swoje własne prawa dostępu do pliku, modyfikując systemową funkcję otwarcia pliku. Skrzynki pocztowe nie muszą już być bierne. Można zdefiniować operacje, które mają zostać wykonane, gdy przychodzi do nas nowa poczta. Warto zauważyć, że rozwiązanie jest niezależne od systemu dostarczania poczty. Wreszcie strażnicy mogą generować zawartość plików i katalogów dynamicznie, gdy istnieje taka potrzeba. Zatem pliki w ogóle nie muszą istnieć w systemie plików zapisanym na dysku. Rozwiązanie to jest opisane bardziej szczegółowo w punkcie 3.1.

Kolejną próbą podjęcia tego tematu był projekt "*UNIX Guardians*" [2]. Był on prowadzony przez George I. Davida oraz Briana J. Matta, pracowników Uniwersytetu w Wisconsin-Milwaukee w roku 1989. W ramach projektu powstała niestety jedynie teoretyczna rozprawa na temat możliwości ingerencji użytkownika w ochronę danych w systemie operacyjnym. Ze względu na brak jakichkolwiek praktycznych propozycji rozwiązań nie ma sensu prezentować jej tu szerzej.

Jeszcze jeden podobny projekt jest realizowany na Uniwersytecie Notre Dame. Richard Kendal i Vincenta Frech swoje rozwiązanie nazwali MonA [4] (skrót od *Modify-on-Access filesystem*, nazywany również MonAFS). Projekt rozpoczął się w roku 1997. Oznacza to, że przez prawie dziesięć lat od powstania systemu Watchdogs nie pojawiały się prace dotyczące podobnej problematyki (przynajmniej ja nie znalazłem żadnych publikacji na ten temat). MonA jest nowym systemem plików mocno bazującym na extended2. Rozszerzenie polega na dodaniu możliwości modyfikacji danych w locie. Pozwala to na dowolne przetwarzanie danych otrzymanych od jądra systemu zanim dotrą one do użytkownika i analogicznie w drugą stronę — zanim dotrą do funkcji zapisu na dysk. Zmiany wprowadzono na poziomie realizacji systemu plików, nie zaś samego systemu operacyjnego. Rozwiązanie to opiszę szczegółowo w dalszej części pracy (punkt 3.2).

1.3. Cel pracy

Głównym celem projektu realizowanego w ramach tej pracy magisterskiej jest rozszerzenie funkcjonalności systemu plików poprzez umożliwienie użytkownikom zmiany semantyki operacji na plikach. Pozwoli to na pełną personalizację operacji na plikach, a poprzez umieszczenie kodu w procesach użytkownika na właściwie nieograniczoną możliwość modyfikacji obsługi obiektów systemu plików.

Takie podejście oczywiście będzie obarczone narzutem czasowym na wykonywanie operacji. Tak rozszerzony system plików na pewno będzie działał wolniej niż oryginalny. Trudno z góry oszacować stopień spadku wydajności. Nie wiadomo również czy wydłużony czas wykonania operacji na plikach będzie akceptowalny dla użytkownika. Znalezienie odpowiedzi na te pytania jest drugim ważnym celem pracy.

Nie można niestety przeprowadzić eksperymentów z systemami opisanymi w punkcie 1.2, gdyż brakuje działającej implementacji. System Watchdogs nigdy nie został udostępniony publicznie[2]. MonA ma być udostępniony jeszcze w tym roku (2001), ale realizuje jedynie niewielki podzbiór możliwości jakie posiada Watchdogs. Prawdopodobnie jego wydanie zbiegnie się w czasie z opublikowaniem mojego rozwiązania. Postanowiłem dostarczyć rozwiązanie, które byłoby w stanie zaspokoić wszystkie wymagania oraz wypełnić lukę, którą jest brak implementacji posiadającej wszystkie cechy projektu Watchdogs. Tak jak autorzy MonA, jako system operacyjny dla implementacji wybrałem Linux. Dzięki temu łatwiej będzie porównać zarówno projekty, jak i zrealizowane systemy.

1.4. Struktura pracy

Struktura pracy jest następująca:

Rozdział pierwszy, zawiera krótkie wprowadzenie w tematykę pracy. Jest w nim opisana istota problemu, który staram się rozwiązać oraz zarysowana jego historia (na podstawie danych dostępnych w Internecie).
Rozdział drugi to przede wszystkim wprowadzenie czytelnika w zagadnienia związane z systemami plików i sposobami ich realizacji.
W **rozdziale trzecim** opisuję dotychczas podejmowane próby rozwiązania problemu zmiany operacji na plikach. Przedstawiam tu inne sposoby podejścia do problemu. Na jednym z nich wzorowałem swoją pracę.
Rozdział czwarty jest głównym rozdziałem pracy. Opisuję tu mój projekt FileGuards. Jego implementacja jest kontynuacją systemu Watchdogs oraz alternatywą dla MonA. Przedstawiam tu wymagania stawiane systemowi, sposób jego zaimplementowania, przykłady użycia i wyniki pomiarów wydajności.
Rozdział piąty zawiera dyskusję nad poprawnością mojego projektu i jego realizacji. To próby odpowiedzi na pytania, dlaczego wybrałem takie a nie inne sposoby na zaimplementowanie projektu, ich zalety i wady. Tu również staram się porównać moje rozwiązanie z przedstawionymi w rozdziale trzecim.
W **rozdziale szóstym** staram się podsumować swoją pracę. Przedstawiam wnioski oraz wizję przyszłości takiego podejścia do systemu plików.
Dodatek A zawiera dokumentację techniczną, opis funkcji, struktur danych oraz scenariusze

[2]Watchdogs został zaimplementowany, ale nie był rozwijany wraz z systemem BSD. Wersja 4.3BSD jest już bardzo rzadko używana. W związku z tym Autorzy nie udostępnili implementacji publicznie.

przepływu sterowania.
Dodatek B zawiera informacje na temat dostępności kodów źródłowych oraz załączników do pracy w postaci elektronicznej.

Podziękowania

Na wstępie swojej pracy chciałbym wyrazić podziękowania osobom, które pomogły mi w jej realizacji:

- Pani dr Janinie Mincer-Daszkiewicz za prowadzenie tej pracy oraz ogromną pomoc w jej tworzeniu,
- Melody Kadenko-Ludwie z Uniwersytetu w Waszyngtonie, za udostępnienie materiałów, których zdobycie nie było możliwe w inny sposób, niż wydobycie ich z archiwów uniwersytetu.

Dodatkowo muszę tu zaznaczyć, że niektóre słowa i sformułowania użyte w tej pracy są znakami zastrzeżonymi lub towarowymi należącymi do odpowiednich firm lub osób.

Dokument stworzono z pomocą — LyX — TeX — LaTeX

Rozdział 2

Systemy plików

2.1. Wprowadzenie do systemów plików

Na wstępie spróbuję opisać kilka zagadnień istotnych dla zrozumienia dalszej części pracy. Często będę opisywał sposób, w jaki zrealizowano je w systemie Linux. Wybrałem właśnie ten system, ponieważ jest on dostępny wraz ze źródłami, co pozwala na dokładne opisanie jego struktur, jak i ich modyfikację. Jest to też powód, dla którego mój projekt zrealizowałem właśnie w tym systemie operacyjnym.

System plików jest wyodrębnionym modułem systemu operacyjnego, zajmującym się interpretacją danych na urządzeniu i prezentowaniem ich jako obiektów dostępnych w systemie. Fizycznym nośnikiem dla systemu plików jest najczęściej dysk twardy lub partycja na nim, będąca częścią jego powierzchni. Ale nic nie stoi na przeszkodzie, aby dane w postaci plików trzymać na innych nośnikach, takich jak taśmy magnetofonowe, pamięci typu FLASH czy dyski CD. Wszystkie one mają wspólną cechę, a mianowicie pamiętają zapisane na nich dane również po utracie zasilania. Ale nie jest to warunek podstawowy. Systemy plików można również umieszczać w ulotnej pamięci typu RAM. W takim przypadku są to tymczasowe, choć wyjątkowo szybkie, składnice danych. Inną, dość ważną cechą wszystkich tych urządzeń jest możliwość swobodnego wyszukiwania danych. Oznacza to, że w każdej chwili możemy zażądać dowolnego bajtu lub nawet bitu z urządzenia. Dzięki temu w każdej chwili jest możliwy dostęp do każdego obiektu w systemie plików.

Każdy współczesny system plików zawiera katalogi, pliki oraz tak zwane dane specjalne, pozwalające na odszukanie innych obiektów w ramach urządzenia. Te ostatnie będziemy nazywać metadanymi. Są to informacje, które opisują właściwości plików, takie jak: fizyczne rozmieszczenie zawartości na dysku, do kogo należy plik, jakie są prawa dostępu do niego itd. Istnieje wiele różnych sposobów rozmieszczenia tych danych na urządzeniu. System obsługi plików ma za zadanie odnajdywać je, gdy użytkownik żąda do nich dostępu (np. do zawartości*)*, jak również zapisywać je w taki sposób, aby spójność danych w obrębie wolumenu została zachowana.

Każdy system operacyjny używa systemu plików do przechowywania danych. W każdym zazwyczaj wyróżnia się podstawowy typ systemu plików. Dla Linuksa jest to extended2. Nie ma jednak żadnych przeciwwskazań, aby jeden system operacyjny wspierał wiele różnych systemów plików. Co więcej, systemy często projektuje się tak, aby było możliwe istnienie wielu systemów plików w jednym wspólnym drzewie katalogów.

Ponieważ każdy system plików jest odrębnym modułem w ramach jądra systemu operacyjnego, zazwyczaj definiuje się standardowy sposób komunikacji między jądrem systemu operacyjnego a systemem obsługi plików. W ten sposób w jednym wspólnym środowisku może istnieć wiele różnych zarejestrowanych systemów plików. Ważne jest jednak, aby używały one tego samego interfejsu komunikacji z jądrem systemu, a co za tym idzie również z procesami użytkowników.

2.2. System plików w Linuksie

System plików w Linuksie jest zgodny z modelem przyjętym w systemach UNIX. Plik nie musi być obiektem trzymanym na jakimś wolumenie, ale dowolnym obiektem, którego zawartość można prezentować jako strumień danych wejściowych lub wyjściowych. Odpowiednio napisany program sterownika obsługi tych obiektów może je prezentować jako pliki w systemie operacyjnym. Linux obsługuje wszystkie te sytuacje ukrywając implementację poszczególnych systemów plików pod warstwą dodatkowego programowego wirtualnego systemu plików VFS [6] (ang. *Virtual File System*, czasem nazywany również *Virtual FileSystem Switch*).

VFS obejmuje dwie warstwy: zbiór definicji określających, jak powinien wyglądać plik oraz warstwę oprogramowania do działań na plikach. W VFS wyróżnia się trzy podstawowe typy obiektów: metryczka pliku (struktura *inode*), instancja otwarcia pliku (struktura *file*) oraz system plików (struktura *super_block*). Każdy z typów ma zdefiniowany zestaw operacji, które program obsługi systemu plików powinien realizować. Każdy obiekt w ramach VFS zawiera wskaźnik do tablicy funkcji realizujących działania na nim. Dzięki temu VFS nie musi wiedzieć w jakim systemie plików znajduje się plik, ale może wywołać odpowiednią funkcję zawartą w tablicy. VFS nie zajmuje się fizycznym nośnikiem obiektu (nie jest istotne, czy dane trzymane są na dysku, czy jest to zdalny system plików, czy może tymczasowy system plików w pamięci operacyjnej). Kiedy użytkownik wykonuje operację czytania, VFS znajduje odpowiedni wskaźnik w tablicy i wywołuje funkcję, nie wnikając w to, jak operacja czytania danych jest faktycznie realizowana. W przypadku braku obsługi dowolnej z funkcji przez system plików, VFS dostarcza generyczne realizacje wszystkich z nich (w najgorszym przypadku generuje odpowiedni błąd).

Plik jako obiekt na dysku jest reprezentowany przez metryczkę pliku. Istnieje dokładnie jedna metryczka (*inode*) w systemie dla jednego pliku na dysku. Reprezentuje ona plik jako całość. Przy każdej operacji otwarcia pliku jest tworzona nowa instancja otwarcia pliku (*file*). Jest ona dla procesu punktem dostępu (uchwyt pliku, deskryptor pliku) do pliku reprezentowanego przez metryczkę. O ile jeden proces może posiadać wiele instancji otwarcia wskazujących na tę samą metryczkę, o tyle metryczka jest obiektem wspólnym dla wszystkich procesów, które otworzyły ten plik.

Choć VFS implementuje wszystkie odwołania do plików, to bez wsparcia ze strony programu obsługi konkretnego systemu plików nie jest w stanie przechowywać danych. Jest zatem tylko interfejsem do wszystkich innych systemów obsługi plików. VFS pozwala na działanie wielu różnych systemów plików w jednym drzewie katalogów. Ponieważ VFS jest głównie pośrednikiem w wywołaniach systemowych, definiuje protokół komunikacji z programami obsługi systemów plików oraz programami użytkownika.

VFS udostępnia dwie grupy operacji na plikach. Są wśród nich operacje na metadanych (metryczkach), które nie wymagają otwierania pliku w systemie. Do takich operacji należą między innymi zmiana właściciela pliku lub praw dostępu. Zostały one zebrane w specjalnej strukturze nazwanej *inode_operations*, czyli operacje na metryczkach. Wśród nich są:

- *create* używana do utworzenia nowego pliku w katalogu, który jest parametrem wywołania. Nadaje nazwę za pomocą funkcji *link()* (patrz dalej),
- *lookup* używana do wyszukania żądanej metryczki pliku, zazwyczaj z nazwą jako kluczem,
- *link* używana do utworzenia nowej nazwy dla pliku. Taką nazwę określa się terminem dowiązanie. W Linuksie jeden plik może mieć wiele dowiązań. O ile *create* tworzy nowy plik nadając mu jedną nazwę, to dzięki temu wywołaniu możemy dodać dowolnie dużo innych nazw dla pliku. Nie każdy system plików musi implementować wiele dowiązań do jednego pliku,
- *unlink* jest odwrotnością funkcji *link*. Jeśli zostaje usunięte ostatnie dowiązanie do pliku, to plik zostaje fizycznie usunięty z systemu plików,
- *symlink* tworzy nowe dowiązanie symboliczne do pliku. Dowiązanie symboliczne różni się od zwykłego tym, że jest dowiązaniem do nazwy innego pliku, a nie do metryczki. W szczególności możliwa jest sytuacja, w której dowiązanie symboliczne wskazuje na nieistniejący plik,
- *mkdir* tworzy nowy wpis w katalogu, będący również katalogiem,
- *rmdir* usuwa pusty katalog, z aktualnego (lub podanego),
- *mknod* tworzy nowy wpis o specjalnym znaczeniu, np. pliki specjalne urządzeń lub kolejki *fifo*. Większość systemów plików go nie implementuje,
- *rename* zmienia nazwę obiektu w systemie plików,
- *readlink* znajduje metryczkę pliku, na którego nazwę wskazuje dowiązanie symboliczne,
- *follow_link* informuje system, że jest żądany dostęp do pliku, na który wskazuje dowiązanie symboliczne, nie zaś do niego samego,
- *truncate* zwalnia zajmowane nadmiarowo przez plik miejsce na dysku. Po wykonaniu tej operacji plik zajmuje tyle miejsca na dysku, na ile wskazuje jego wielkość. Jest to jedyny sposób na zmniejszenie rozmiaru pliku,
- *permission* sprawdza, czy żądany tryb dostępu do pliku jest możliwy dla aktualnego użytkownika,
- *revalidate* odświeża dane w pamięci, odczytując ponownie metryczkę pliku z dysku,
- *setattr* i *getattr* służą do odczytu i ustawiania dodatkowych atrybutów jakie może posiadać plik (takich jak właściciel pliki lub flaga tylko do odczytu).

Należy zauważyć, że w Linuksie katalog jest również plikiem dla systemu operacyjnego. Jest on jednak inaczej traktowany. Zawartość takiego pliku nigdy nie jest prezentowana jako surowe dane (są one najpierw interpretowane jako nazwane dowiązania do innych plików lub katalogów). Dodatkowo, dzięki funkcji *mknod* można tworzyć nowe typy plików. Ich zawartość nie musi być trzymana na wolumenie, ale automatycznie generowana przez VFS, znający ich właściwości (np. urządzenia systemowe).

Warto tu wspomnieć o funkcji *permission()*. W Linuksie prawa dostępu są dość ograniczone. Plik ma właściciela i należy do jakiejś grupy. Wszystko co można określić to prawa dla właściciela, dla grupy i dla wszystkich innych użytkowników. Jest to mało wygodne i często niewystarczające. Inne systemy operacyjne wprowadzają pojęcie ACL (ang. *access control list*). Są to listy kontroli dostępu do obiektów systemu operacyjnego, a w szczególności do plików. Linux ich nie implementuje[1]. Każdy wpis na liście ACL określa kto ma jakie prawa do obiektu. Prawa można przyznać konkretnemu użytkownikowi, grupie użytkowników lokalnych lub innej określającej poprawny zbiór użytkowników (np. grupy sieciowe lub NIS). Przykładowo możemy ustalić, że właściciel ma wszystkie prawa, a użytkownik *jasio* ma prawo zapisu i odczytu. Inni użytkownicy systemu nie mają żadnych praw do tego obiektu (czasem łącznie z administratorem).

Drugi zestaw operacji na plikach został zgrupowany w innej strukturze, nazwanej *file_operations*, czyli operacje na pliku. Są to operacje związane głównie z jego zawartością. Aby je wykonać, najpierw należy otworzyć docelowy plik. Funkcja otwarcia również znajduje się w tej strukturze. A oto one:

- *llseek* zmienia aktualną pozycję w pliku na podaną. Następne operacje odczytu lub zapisu będą się odbywać od tego miejsca (bajtu) w pliku,
- *read* i *write* odpowiednio czytają i zapisują dane z i do pliku,
- *readdir* służy do odczytywania zawartości katalogu. Działa wyłącznie na katalogach,
- *poll* jest funkcją sprawdzającą, czy dane w pliku są gotowe do odczytu lub czy w pliku znajduje się miejsce na dodatkowe dane,
- *ioctl* realizuje dodatkowe polecenia mające wpływ na dalsze korzystanie z pliku. Na przykład za jej pomocą można wyłączyć buforowanie danych przed zapisem na dysk,
- *mmap* powoduje odwzorowanie pliku do pamięci. Dalsze operacje zapisu i odczytu można wykonywać w pamięci operacyjnej, zamiast przy pomocy wywołań systemowych,
- *open* jest funkcją otwarcia pliku. Nie można jej wykonywać dla otwartego pliku,
- *flush* powoduje zapisanie danych z buforów systemowych na urządzenie,
- *release* jest funkcją zamykającą plik. Po jej wykonaniu nie jest możliwe wołanie żadnej z wymienionych tu funkcji. Plik należy ponownie otworzyć,
- *fsync* i *fasync* są funkcjami synchronizującymi dane na urządzeniu i w pamięci. Druga z nich służy wyłącznie do obsługi urządzeń znakowych (czyli strumieni danych, np. terminali),
- *lock* zakłada blokadę na plik. Blokada jest zdejmowana przy zamknięciu pliku.

System VFS dostarcza generyczne implementacje wszystkich tych funkcji. Jeśli tylko system plików odpowiednio wypełni struktury *file* i *inode* przy dostępie do pliku, to większość tych operacji automatycznie wykona się we właściwy sposób. Wspomniane struktury zawierają

[1] Najnowsza wersja wspomaga ACL, ale używane w Linuksie systemy plików nie.

odpowiednie dane dotyczące otwartego pliku oraz dane zawarte w metryczce pliku, czyli metadane pliku zapisane na dysku. Wszystkie odwołania do plików z procesów użytkownika przechodzą przez system VFS. Eksportowane przez jądro funkcje systemowe (np. *read*), używane w celu dostępu do plików, wywołują wymienione funkcje z VFS. Ten zaś albo wykonuje generyczne procedury zaimplementowane w nim, albo woła właściwe funkcje z implementacji docelowego systemu plików dla danego wolumenu. Nie ma możliwości wołania funkcji z konkretnych systemów plików w bezpośredni sposób, z pominięciem mechanizmu VFS.

Standardowym, dyskowym systemem plików stosowanym w Linuksie jest extended2. Ma on wiele wspólnego z systemami plików używanymi w systemach UNIX, a w szczególności w BSD (na przykład *Fast File System — ffs*). Coraz częściej zaczyna się stosować bardziej nowoczesne systemy plików posiadające większe od extended2 możliwości (takie jak wspieranie dużych plików, księgowanie, rozmieszczenie danych na nośniku za pomocą struktur drzewiastych itp.). Najpopularniejsze to reiserfs, xfs, jfs, extended3[2].

Dodatkowo Linux implementuje system plików *procfs*. Linux nie jest tu wyjątkiem. Prawie wszystkie systemy UNIX go implementują. Jest to system plików prezentujący wewnętrzne informacje jądra systemu poprzez pliki. Składa się z dwóch podstawowych części. Po pierwsze prezentuje informacje o wszystkich procesach w systemie, co pozwala na odczytywanie ich bez używania uprzywilejowanych wywołań systemowych. Po drugie udostępnia wartości danych systemowych takich jak zegar systemowy, zużycie pamięci, czy nawet cały obraz pamięci jądra systemu. Do niektórych plików można uzyskać prawo zapisu. Oznacza to, że pewne dane systemu operacyjnego mogą być modyfikowane podczas działania systemu (na przykład maksymalna liczba wątków w systemie lub wielkość tablicy przechowującej metryczki plików).

2.3. Urządzenia w Linuksie

Urządzenia są obsługiwane w systemie Linux tak samo jak pliki. Oznacza to, że korzysta się z nich za pomocą tego samego interfejsu, który został zdefiniowany dla plików. Do urządzeń odwołujemy się za pomocą dwóch numerów: numeru głównego — identyfikującego fizyczne urządzenie oraz numeru drugorzędnego — dla logicznych podurządzeń w ramach głównego. Na przykład pierwszy dysk typu IDE w systemie ma numer (3, 0). Jego kolejne partycje różnią się wartością drugiego numeru. Pełen spis numerów jest dostępny w dokumentacji Linuksa. Dla jądra systemu operacyjnego otwieranie urządzeń po numerach jest wystarczające, ale dla procesów nie. Dlatego wprowadzono pewnego rodzaju odwzorowanie urządzeń na pliki. Otóż wspomniana wcześniej funkcja *mknod()* potrafi stworzyć plik o specjalnym znaczeniu dla systemu. Taki plik nazywamy urządzeniem, ponieważ w rzeczywistości nie zawiera on danych, ale czytanie z tego pliku powoduje czytanie danych zapisanych na urządzeniu. To samo dotyczy wszystkich funkcji zawartych w *file_operations*. Na przykład, możemy stworzyć plik specjalny typu urządzenie, z numerem głównym 3 i drugorzędnym 0. Jeśli teraz proces otworzy taki plik, to dane, które będzie z niego odczytywał, będą surowymi danymi znajdującymi się na pierwszym dysku twardym typu IDE. Urządzenia niekoniecznie muszą mieć swoje fizyczne odpowiedniki. Podprogram obsługi urządzenia może produkować dane na zawołanie, bądź też przyjmować wszystkie dane i natychmiast je zapominać. Ten mechanizm może zostać wykorzystany do komunikacji między procesami i jądrem systemu. Aby napisać podprogram obsługi urządzenia, należy zaimplementować część funkcji (lub wszystkie) ze struktury *file_operations* i zarejestrować obsługę urządzenia w jądrze, podając jego numer główny [5]. Od tej pory VFS będzie wywoływał nasze funkcje zamiast właściwych dla systemu plików, na którym znajduje się specjalne dowiązanie do urządzenia.

[2]Reiserfs i extended3 są coraz częściej stosowanymi w Linuksie systemami plików.

Rozdział 3

Sposoby rozszerzania funkcjonalności systemów plików

W tym rozdziale prezentuję dwa wcześniejsze projekty dotyczące zagadnienia rozszerzania funkcjonalności systemów plików. Są to projekty Watchdogs i MonA. Pierwszy z nich posłużył mi jako wzór do opracowania mojego rozwiązania (patrz rozdział 4). Drugi nie jest pełną realizacją zagadnienia, ale na pewno zasługuje na szerszy opis.

3.1. Strażnicy plików w projekcie Watchdogs

3.1.1. Wstęp

Głównym celem tego projektu było umożliwienie użytkownikom definiowania operacji na plikach na własny sposób. Dzięki strażnikom jest możliwe napisanie własnych funkcji obsługi plików oraz wskazanie, które pliki mają być przez nie obsługiwane. Kod strażników jest zawarty w zwykłych programach użytkownika, natomiast interfejs umożliwiający ich działanie w systemie, znajduje się w jądrze.

Bershad i Pinkerton realizując projekt Watchdogs dążyli do uzyskania rozwiązania prostego, ogólnego i oczywiście szybkiego. Chcieli by realizowane za pomocą strażników operacje, wykonywały się szybciej niż podobne, zaimplementowane przez programy lub skrypty interpretatora poleceń. Na przykład, aby kompresja plików w locie za pomocą strażników była zarówno szybsza, jak i dużo łatwiejsza w użyciu dzięki temu, że system operacyjny robi to sam (za pomocą strażników). Należy tylko napisać w strażniku nowe funkcje obsługi, a korzystanie z nich staje się automatyczne. Strażnicy mogą też dodawać nowe własności do systemu plików. Na przykład podmieniając funkcję *open()* możemy do niej dodać realizację ACL dla otwieranego pliku.

3.1.2. Operacje

Centralnym obiektem w tak rozszerzonym systemie jest strzeżony plik. Musi on istnieć na dysku, aby można było przypisać mu strażnika. Takie przypisanie odbywa się poprzez wpisanie nazwy strażnika do metryczki pliku. Strażnik wykonuje zmodyfikowaną funkcję obsługi za każdym razem, gdy jej oryginał (np. *open, read, ioctl*) jest wołany dla pliku. W ten sposób może przedefiniować wywołania systemowe, aby miały różną treść dla różnych plików.

Strażnicy są realizowani jako procesy działające w przestrzeni użytkownika, zatem inne procesy mogłyby komunikować się z nimi za pomocą mechanizmów komunikacji międzyprocesowej. Jednak procesy sięgające do strzeżonych plików i korzystające w ten sposób ze strażników mogą się komunikować ze strażnikami wyłącznie za pośrednictwem systemu operacyjnego (te procesy będę nazywał procesami klienckimi w stosunku do strażników). Takie rozwiązanie narzucił wymóg przezroczystości rozwiązania dla procesów korzystających ze strzeżonych plików. Procesy klienckie nie muszą (a nawet nie powinny) wiedzieć, czy aktualnie dostęp do pliku odbywa się za pomocą mechanizmu strażników czy nie. Taki sposób komunikacji jest również wymagany ze względów bezpieczeństwa. Procesy nie mogą otrzymać bezpośredniego dostępu do plików z pominięciem systemu operacyjnego (wyjątkami są procesy działające na całym systemie plików, takie jak proces wykonujący naprawę systemu plików *fsck*)!

Kiedy proces kliencki otwiera strzeżony plik, z funkcji otwarcia następuje przekierowanie do strażnika. Strażnik jest proszony przez jądro o wykonanie żądanej operacji w jego imieniu. W tym celu jądro specjalnym kanałem wysyła do strażnika komunikat zawierający potrzebne dane, argumenty dla wywołania funkcji systemowej oraz dane identyfikujące proces kliencki. Strażnik odpowiadając komunikatem do jądra instruuje je jak należy dalej realizować wywołanie systemowe.

Strażnik może pilnować wielu plików. Może również obsługiwać różne instancje otwarcia pliku na różne sposoby. Dzięki temu ten sam plik może mieć wiele różnych widoków. W odpowiedzi na żądanie jądra o wykonanie operacji na pliku, strażnik może:

- wykonać operację, korzystając z danych otrzymanych od jądra i przekazać do jądra wynik operacji. Na przykład dla funkcji *read()*, strażnik może samodzielnie wczytać dane, zmodyfikować je (jeśli to niezbędne) i wysłać do jądra. Aby zapobiec zapętleniu, strażnik może się odwoływać do pliku bezpośrednio (z pominięciem systemu strażników),
- zabronić wykonania operacji, wysyłając do jądra kod błędu,
- odnotować wywołanie (ang. accounting), a następnie poprosić system, aby wykonał operację w domyślny sposób, zakodowany w procedurach systemu plików.

W żadnym z tych przypadków proces wykonujący operację na pliku nie wie, czy jest ona wykonywana z mechanizmem strażników, czy nie.

Strażników można również przypisywać do katalogów, ponieważ w systemach UNIX katalogi są tak naprawdę również plikami. Różnice polegają na stosowaniu innych funkcji dostępu (np. *readdir*). Nie ma przeszkód, aby i te funkcje można było podmienić. Dzięki temu możemy zmieniać zawartość katalogu, chować pliki lub tworzyć pliki wirtualne. W ten sposób można także w przezroczysty sposób zrealizować dostęp do zdalnych plików, a nawet katalogów ftp. Polecenie *cd /ftp/ftp.icm.edu.pl/pub* może powodować, że automatycznie zostanie odczytana zawartość serwera ftp i zaprezentowana jako katalog lokalny.

3.1.3. Implementacja

Aby opisane rozwiązanie dało się zrealizować w systemie, muszą być spełnione pewne warunki. Muszą istnieć mechanizmy:

- tworzenia strażników i zarządzania nimi,
- przypisywania strażników do plików,
- przezroczystej komunikacji między procesem klienckim, który otwiera plik, i strażnikiem tego pliku.

Dodatkowo system musi być napisany tak, aby poradził sobie z błędami wynikającymi ze stosowania powyższych mechanizmów. Aby to osiągnąć, wprowadzono do systemu nowe komponenty:

- proces zarządcy strażników (jeden w systemie), odpowiedzialny za tworzenie, utrzymywanie i zabijanie procesów strażników,
- funkcję systemową do podłączania strażników do plików,
- mechanizm komunikacji strażnik-jądro oparty na komunikatach (mechanizm jądro-zwykły proces już istnieje).

Informacja o tym czy plik jest pilnowany oraz jaki strażnik jest odpowiedzialny za ten plik, jest metadaną, taką jak właściciel pliku czy też jego prawa dostępu. Oznacza to, że powinno się ją zapisać w metryczce pliku i dopuścić jej modyfikację wyłącznie poprzez bezpieczne wywołania systemowe. Ponieważ w metryczce zapisuje się nazwę strażnika, a nie ma tam tyle miejsca, aby wstawić całą ścieżkę dostępu, wszystkie programy strażników muszą znajdować się w katalogu */wdog*. W metryczce znajduje się tylko końcowa nazwa pliku z programem strażnika. Oczywiście wykorzystując mechanizm dowiązań (również symbolicznych) programy strażników można umieszczać w dowolnym miejscu drzewa katalogów. Rozwiązanie umożliwia przypisanie jednego strażnika do dowolnie wielu plików, jakkolwiek danego pliku może pilnować tylko jeden strażnik. Chociaż przypisywanie wielu strażników do jednego pliku nie gwarantuje istotnego rozszerzenia możliwości rozwiązania, to jednak w pewnych sytuacjach mogłoby być pomocne.

Do manipulacji informacją o przypisanym do pliku strażniku stworzono nową funkcję systemową *wdlink()*. Wykorzystano 20 nieużywanych bajtów w metryczce pliku, przeznaczając je na nazwę strażnika. Jeśli podana nazwa strażnika jest pusta (NULL), to plik zostaje oznaczony jako niestrzeżony.

Standardowa komunikacja procesów z jądrem jest zazwyczaj asymetryczna. Opiera się na wywołaniach systemowych oraz mechanizmie sygnałów wysyłanych do procesów. Strażnicy wymagają bardziej rozbudowanej komunikacji. Możliwość traktowania jądra i procesu jako stron w komunikacji opartej na komunikatach istnieje w innych systemach operacyjnych, ale nie w 4.3BSD. Są trzy możliwości rozwiązania problemu. Można zaimplementować ogólny mechanizm komunikacji system-proces, rozszerzyć istniejące mechanizmy albo napisać mechanizm zaprojektowany specjalnie dla systemu strażników. Wybrano trzecie rozwiązanie. Dodano nowy typ pliku oraz nowe wywołanie systemowe do tworzenia go. Mechanizm nazwano kanałem komunikacyjnym strażnika WMC (ang. *watchdog message channel*), zaś funkcję systemową *createwmc()*.

Każde otwarcie pilnowanego pliku powoduje stworzenie nowej sesji i zapamiętanie dla niej identyfikatora. Sesja kończy się, gdy plik zostaje zamknięty. Każdy wysyłany przez WMC komunikat zawiera identyfikator sesji i treść wiadomości. Dzięki temu, że identyfikator sesji jest unikatowy dla każdej instancji otwartego pliku, strażnik może pilnować wielu plików — a nawet tego samego — na różne sposoby. Wszystkie wywołania dla danej instancji zawierają ten sam identyfikator. Jądro systemu utrzymuje tablicę instancji otwarć strzeżonych plików oraz tablicę strażników.

System operacyjny pełni niewielką rolę w mechanizmie strażników. Musi być w stanie znaleźć odpowiedni kanał WMC na podstawie nazwy strażnika. Jest to możliwe dzięki zapamiętywaniu nazwy strażnika w momencie otwierania przez niego WMC. Te wiadomości są zapisywane w systemowej tablicy strażników. Następnie, gdy jest obsługiwany plik z przypisanym strażnikiem, należy tylko wyszukać odpowiedni wpis w tablicy strażników i za pomocą zapamiętanego tam WMC powiadomić strażnika o potrzebie obsługi żądania. Jeśli takiego nie ma, oznacza to, że strażnika nie ma w systemie i trzeba go wystartować. To zadanie jest zlecane zarządcy strażników. Jest to specjalny proces, który po otwarciu WMC informuje system o tym, że jest zarządcą. W systemie może być tylko jeden taki proces. Jeśli zarządca odpowie pozytywnie na żądanie uruchomienia strażnika, to pierwotna operacja jest kierowana do nowego kanału WMC, jeśli nie — jest generowany błąd.

Uruchamianie tego samego strażnika na nowo dla każdego otwarcia pliku byłoby skrajnie nieefektywne. W związku z tym strażnik ma za zadanie obsługiwać żądania od jądra tak długo, aż nie zostanie zatrzymany przez zewnętrzne okoliczności. To zadanie zazwyczaj zrealizuje zarządca. Dzięki takiej semantyce, proces zarządcy może utrzymywać w gotowości często wykorzystywanych strażników. Mimo, że to zarządca jest odpowiedzialny za pulę strażników, nic nie stoi na przeszkodzie, aby użytkownik sam uruchomił strażnika. Będzie on wówczas normalnie obsługiwał żądania od jądra systemu, jednakże zarządca nie będzie w stanie go zatrzymać (nie będzie posiadał na jego temat informacji). W związku z tym można uruchomiać strażnika pod kontrolą odpluskwiacza[1] (ang. *debugger*), co ułatwia pisanie i testowanie strażników.

3.1.4. Pisanie strażników

Kod strażnika jest bardzo prosty. Wystarczy, że proces strażnika otworzy kanał WMC, a następnie będzie nasłuchiwał komunikatów od jądra i realizował swoje zadania. Przykładowy szkielet kodu może wyglądać następująco:

```
struct wdmsg wdmsg;
int cc;
int wmc = createwmc();
for (;;) {
    cc = read(wmc, &wdmsg, sizeof(struct wdmsg));
    if (cc != sizeof(struct wdmsg))
        if (cc < 0)
            perror("read"), exit(-1);
        else
           abort(); //nie powinno się zdarzyć
    switch (wdmsg.type) {
    case WDMSG_OPENREQ: do_open(wmc, &wdmsg); break;
    case WDMSG_READREQ: do_read(wmc, &wdmsg); break;
    //pozostałe przypadki
    default:  abort(); //nie powinno się zdarzyć
    }; //switch
}; // for(;;)
```

Każda z funkcji *(do_open(), do_read* itd.) obsługuje odpowiednie żądanie na podstawie danych otrzymanych w komunikacie i wysyła odpowiedź (wynik operacji) do systemu za pomocą tego samego kanału WMC.

[1] Program pozwalający śledzić wykonywanie się innego procesu w systemie, podglądać wartości zmiennych, zatrzymywać wykonywanie śledzonego procesu w konkretnym miejscu itd. Używa się go głównie do szukania błędów niewidocznych w sposób oczywisty w tekście programu.

3.2. Transformacje w systemie plików MonAFS

3.2.1. Wstęp

Modify-on-Access (w skrócie MonA) jest nowym systemem plików zaimplementowany w Linuksie. Ciągle jeszcze jest on w fazie implementacji, choć prace nad nim trwają od 1997 roku [4]. Stanowi rozszerzenie popularnego systemu plików extended2. Dodatkowe mechanizmy obsługi plików można realizować zarówno w trybie jądra, jak i za pomocą programów użytkownika (porównaj punkt 3.2.5). MonA rozszerza możliwości systemu plików poprzez dynamiczną transformację danych, podczas ich przesyłania między jądrem i procesami użytkownika. Dzięki takiemu podejściu otrzymujemy dynamiczny system plików, w którym jądro kooperuje z procesami, aby ukończyć żądanie realizowane przez funkcję obsługi systemu plików. Taki system plików jest w stanie realizować zadania, które nie są możliwe przy użyciu extended2. Podczas operacji odczytu lub zapisu system sprawdza czy z plikiem nie jest związana jakaś transformacja. Jeśli tak, to zostaje ona zastosowana na danych będących wynikiem działania standardowej funkcji obsługi.

Transformacja może być dowolną operacją na strumieniu danych. W przypadku czytania z pliku transformacja ma miejsce tuż przed przekazaniem danych do procesu, przy zapisie — zanim dane zostaną zapisane na dysk. Dzięki takiemu podejściu w oczywisty sposób można zaimplementować dowolny mechanizm podmieniający dane w locie. Transformacje zostały zaprojektowane dla trybu jądra jako moduły[2]. Jednakże zarządzanie modułami (ładowanie i usuwanie) jest operacją uprzywilejowaną. Aby nie narażać bezpieczeństwa systemu opracowano mechanizm rejestrowania transformacji w trybie użytkownika. Mimo oczywistego spowolnienia działania, w większości przypadków szybkość działania jest wystarczająca. MonA daje wszystkim użytkownikom (nie tylko administratorowi) możliwość definiowania dowolnych transformacji na własnych plikach.

3.2.2. Transformacje

Transformacja jest niezależną operacją na strumieniu danych. Zwykle pobiera ona dane z wejścia, modyfikuje je i wysyła na wyjście — nie wykluczone, że do następnej transformacji. Taki model jest bardzo elastyczny, prosty i ogólny. Typ transformacji nie ma znaczenia dla użytkowników systemu (bądź programistów), ponieważ mechanizm transformacji jest dla nich przezroczysty. Sieć transformacji jest kombinacją wielu transformacji na jednym strumieniu danych. W ten sposób łatwo można stworzyć sieć prezentującą pliki *tar.gz* jako katalogi systemowe, za pomocą dwóch transformacji. Takie sieci przypominają model przetwarzania potokowego z użyciem łączy nienazwanych (ang. *pipe*. Chodzi tu o polecenia typu:
cat plik | grep jasio | tee inny_plik). Jednak rozwiązanie z siecią transformacji znacznie redukuje koszty związane z uruchamianiem procesów i przesyłaniem danych między nimi. Sieć transformacji można modyfikować podczas dostępu do pliku, otrzymując różne widoki pliku podczas jednej sesji otwarcia.

3.2.3. Implementacja

Implementacja MonA w systemie Linux opiera się na możliwości ładowania części kodu jądra jako modułu oraz pisania nowych systemów plików z wykorzystaniem dostarczanych przez VFS generycznych funkcji obsługi. MonA jest całkowicie wymienny z systemem extended2,

[2] Moduł jest częścią jądra systemu operacyjnego, która znajduje się w osobnym pliku wykonywalnym niż jądro. Moduł może być załadowany do systemu w dowolnej chwili jego działania. Działa w trybie jądra, i ma dostęp do jego struktur. Może również zostać usunięty, jeśli nie jest używany.

mimo że jest jego rozwinięciem. Partycję extended2 można zamontować jako MonA i odwrotnie, nie tracąc żadnych właściwości ani spójności systemu plików. VFS jest bardzo elastyczny i bardzo dobrze wspiera pisanie nowych systemów plików. Wiele różnych systemów może istnieć w jednym wspólnym drzewie katalogów. Dodatkowo VFS dostarcza większość domyślnych operacji na plikach oraz system buforów systemowych. MonA korzysta z tych właściwości VFS, definiując tylko siedem funkcji, tj. *open, read, write, close, llseek, symlink i ioctl*, przy czym tylko czytanie i pisanie mogą podlegać transformacjom i wszystkie wykonania operacji *read/write* im podlegają. Aby umożliwić prezentowanie surowych danych z pliku, domyślną dla wszystkich operacji jest identyczność. Jest to specjalna transformacja nie dokonująca żadnych zmian na strumieniu danych. Dodatkowo, ponieważ bufory systemowe są zaimplementowane na poziomie VFS, nie każda operacja czytania będzie obsłużona przez MonA. Część danych jest zapamiętana w buforach systemu VFS i powędruje wprost z pamięci operacyjnej[3].

3.2.4. Interfejs użytkownika

Użytkownik może wykorzystywać możliwości oferowane przez MonA na dwa sposoby: tymczasowy i stały. Jeśli skorzysta się z drugiego sposobu, można później używać pliku w standardowy sposób, co zapewnia przezroczystość z punktu widzenia programów.

Tymczasowe widoki plików otrzymuje się poprzez modyfikowanie sieci transformacji podczas otwarcia pliku. Służy do tego funkcja *ioctl()*, dzięki której można na strumień nakładać transformacje i je zdejmować. Te zmiany nie są widoczne dla innych instancji otwarcia tego pliku i znikają przy zamknięciu sesji.

Stałe sieci transformacji można otrzymać poprzez specjalne symboliczne dowiązania transformacyjne. Jest to rozszerzona wersja dowiązania symbolicznego, zawierająca oprócz ścieżki docelowej sieć transformacji na wskazywanym pliku. Podczas otwierania takiego dowiązania, system automatycznie buduje dla tej instancji sieć transformacji na podstawie danych zawartych w dowiązaniu. Jednak nadal jest możliwe odczytanie pliku z surowymi danymi poprzez podążenie za dowiązaniem (*follow_link*), zamiast otwierania go. Oprócz specjalnego znaczenia dla funkcji *open()* i *close()*, dowiązanie transformacyjne jest zwyczajnym symbolicznym dowiązaniem do dowolnej nazwy w drzewie katalogów, w tym do innego dowiązania transformacyjnego lub pliku specjalnego. Do tworzenia dowiązań transformacyjnych stworzono specjalny program *lnx*, który korzysta z rozszerzonych możliwości funkcji *symlink()*.

Transformacje są zaimplementowane nie jako procesy, ale jako funkcje. Każda sieć transformacji jest reprezentowana jako lista funkcji. Strumień danych jest reprezentowany jako bufor przesyłany do funkcji poprzez parametr. Dzięki temu redukujemy koszty związane z uruchamianiem procesów i przesyłaniem między nimi danych. Transformacje nie muszą zachowywać rozmiaru danych, co pozwala na implementację np. kompresji w locie.

3.2.5. Transformacje wykonywane w trybie użytkownika

Transformacje wykonywane w trybie jądra są realizowane za pomocą wywołań funkcji transformacji. Aby móc wykonać transformacje w trybie użytkownika, trzeba zrezygnować z uruchamiania ich poprzez wywołania funkcji. Jądro nie może uruchamiać w trybie systemowym

[3] Tak twierdzą autorzy pracy, choć w rzeczywistości jest inaczej. To nie VFS zajmuje się buforowaniem danych, ale podsystem obsługi pamięci. Właściwie wszystkie systemy plików korzystają z funkcji *generic_file_read/write*, które wykonują operacje na buforach systemowych. MonA również korzysta z tego mechanizmu. Być może autorzy mieli na myśli, że w ogóle korzystają z mechanizmu buforów.

kodu znajdującego się w programie użytkownika! Dlatego transformacje wykonywane w trybie użytkownika są realizowane za pomocą procesów i przetwarzania potokowego.

Specjalną transformację o nazwie *export* napisano po to, aby móc wynieść kod transformacji poza jądro systemu. Sama transformacja *export* jest zaimplementowana jako moduł systemu operacyjnego, zatem obsługa transformacji w trybie użytkownika nie zmienia zachowania jądra systemu MonA. Dopiero transformacja *export* realizuje mechanizmy obsługi transformacji działających jako procesy.

Taka implementacja jest potrzebna z kilku powodów. Po pierwsze kod systemu jest uprzywilejowany, zatem nie można uruchamiać w takim trybie programów użytkownika. Po drugie, kod jądra nie jest wywłaszczalny[4], co przy czasochłonnych transformacjach oznacza duże opóźnienia w reakcjach systemu na inne zdarzenia.

W systemie MonA użyto specjalnego demona do obsługi transformacji w przestrzeni użytkownika. Kiedy MonA inicjuje transformację *export*, demon uruchamia proces potomny, który ją obsługuje. Do komunikacji między jądrem i procesami użyto funkcji *ioctl()*. Demon wykonuje funkcję *ioctl*, w której jest usypiany aż do momentu, kiedy odpowiednie dane dotyczące nowej transformacji są gotowe. Wtedy tworzy proces potomny, który również za pomocą wywołań *ioctl* odbiera od jądra żądania i tą samą drogą odsyła odpowiedzi. Wszystkie wywołania *ioctl* są blokujące, co oszczędza czas procesora. Jako parametry podawane są między innymi wskaźniki do obszarów pamięci procesu, za pomocą której odbywa się wymiana danych.

MonA implementuje dwie kolejki do komunikacji z demonem. Po pierwsze kolejkę inicjacji, w której znajdują się transformacje oczekujące na uruchomienie w przestrzeni użytkownika. Rozszerzona funkcjonalność wywołania *ioctl* daje dostęp do tych danych. Kiedy dane są dostępne, następuje powrót z wywołania systemowego, demon odczytuje dane z kolejki, tworzy proces potomny, ustawia mu prawa dostępu na właściciela transformacji i ładuje odpowiednią transformację z bibliotek dzielonych. Następnie czeka na następne żądanie stworzenia transformacji. Druga kolejka jest kolejką wykonania. Są to zainicjowane transformacje czekające na wykonanie. Za pomocą unikatowego klucza transformacji, proces potomny wykonując funkcję *ioctl*, otrzymuje dostęp do danych z tej kolejki. Po wykonaniu transformacji, również za pomocą wywołania *ioctl*, wkłada do kolejki dane wynikowe. Powtarza to aż do momentu otrzymania informacji, że plik został zamknięty. Wtedy kończy swoje działanie.

Proces potomny realizujący transformację musi wykonywać się z jakimiś uprawnieniami w systemie. Uprawnienia jakie otrzymuje proces są takie same jak uprawnienia użytkownika, który stworzył transformację. Inne możliwości, czyli prawa użytkownika otwierającego plik lub właściciela pliku, prowadzą do możliwości nadużyć w systemie. Demon, ponieważ pracuje jako superużytkownik ustawia je w taki sposób, aby zachować bezpieczeństwo systemu.

Dane przesyłane między procesem a jądrem są ograniczonej wielkości i są obcinane do maksymalnej, jeśli proces próbuje przesłać więcej danych (aktualnie 4KB). Dodatkowo dane otrzymane od procesów nigdy nie są traktowane jako wykonywalny kod (nie mają wpływu na działanie systemu). Dostęp do danych jest kontrolowany poprzez klucze sesji. Jakkolwiek jest możliwe, aby inny proces odgadł klucz i podszył się pod transformację, to jednak jest to bardzo mało prawdopodobne.

Podstawowe transformacje przestrzeni użytkownika są umieszczone w plikach jako biblioteki dzielone (ang. *shared object*). Ale mogą to być również dowolne programy, w tym skrypty. Dane z nimi są wymieniane za pomocą standardowego wejścia-wyjścia, co powoduje, że jako transformację można wykonać nawet takie polecenie jak: *grep jasio*.

[4]Nie można przerwać jego wykonania, tak jak to można zrobić z procesami. Taka własność systemu operacyjnego ma być wprowadzona do nowej wersji Linuksa (2.5).

Rozdział 4

FileGuards — nowe możliwości systemów plików

4.1. Informacje wstępne

W tym rozdziale opiszę projekt i implementację systemu FileGuards, który powstał w ramach mojej pracy magisterskiej.

Moje rozwiązanie ma wiele elementów wspólnych z opisanym w punkcie 3.1 systemem Watchdogs, który był pierwowzorem dla mojego systemu. Używałem tam słowa strażnik jako polski odpowiednik *watchdog*. Ponieważ to słowo pasuje również do mojego rozwiązania, używam go jako tłumaczenie *fileguard*.

Celem tego projektu jest umożliwienie personalizacji sposobu prezentacji obiektów systemu plików przez użytkowników. Można to osiągnąć poprzez umożliwienie zmiany sposobu realizacji funkcji, za pomocą których dostęp do pliku jest realizowany. Oczywiście tę operację można wykonać dla każdego pliku osobno, otrzymując w ten sposób różne semantyki dla różnych plików w ramach tego samego systemu plików. Jest to ten sam pomysł, który został zrealizowany w projekcie Watchdogs. Ja również na nim oprę swój projekt. Podobne podejście, choć bardziej ograniczone, prezentował opisany przez mnie w punkcie 3.2 projekt MonA.

Mimo, że podobne rozwiązanie zostało już zrealizowane, warto zająć się tym zagadnieniem od nowa. Poprzedni projekt ma już ponad dziesięć lat. Wykonana implementacja nie wyszła poza fazę testową dla systemu 4.3BSD i nie była dalej rozwijana. Nikt nie podjął się kontynuowania projektu i przeniesienia go do systemu 4.4BSD. To sprawia, że właściwie nie ma działającego rozwiązania pozwalającego na pełną modyfikację semantyki działania plików. Implementacja MonA pozwala tylko na modyfikację danych w ramach funkcji pisania i czytania z pliku. W moim projekcie pozwolę na podmianę tych funkcji systemowych a także wszystkich innych dotyczących obsługi plików. Jest to zatem rozwiązanie bardziej ogólne, dające większe możliwości ingerowania w działanie systemu plików. Dodatkowo mój projekt zakłada realizację strażników za pomocą procesów w trybie użytkownika (tak jak Watchdogs) oraz jako funkcji ładowanych z modułów systemowych (podobnie jak w MonA).

Projekt dopuszcza modyfikację potrzebnych do jego realizacji wywołań systemowych tak, aby zamiast nich można było wykonać dowolny inny kod. Oznacza to, że dowolną funkcję operującą na pliku (np. czytanie z pliku) użytkownik może napisać na swój własny sposób. Co więcej, może napisać dowolnie dużo różnych wersji tej samej funkcji. Do każdego pliku w systemie może teraz przypisać jedną z tych funkcji (dokładniej zestaw funkcji dla wszystkich operacji), która od tej pory będzie wykonywana zamiast oryginalnej. Dodatkowo użytkownicy mogą te funkcje implementować w zwykłych programach, wykonujących się na tych samych prawach, co inne procesy w trybie użytkownika. Rozwiązanie zapewni przekierowanie sterowania z jądra systemu do tych programów, które obsłużą wywołania. To oznacza, że dowolny użytkownik systemu może zupełnie zmienić zachowanie swoich plików w (prawie) nieograniczony sposób.

Rozwiązanie zostało zaprojektowanie specjalnie z myślą o implementacji w systemie Linux. W opisie rozwiązania odwołuję się do specyficznych struktur i pojęć istniejących tylko w tym systemie (takich jak VFS).

4.2. Wymagania projektu

Dobre rozwiązanie musi przede wszystkim dostarczać mechanizm pozwalający podmieniać funkcje systemowe wykonywane na plikach w czasie działania systemu. Jest to podstawowy wymóg, aby była możliwa dynamiczna zmiana semantyki operacji na plikach.

W Linuksie funkcje obsługi plików są zgrupowane w jednej strukturze o nazwie *file_operations*. Są tam prawie wszystkie funkcje pozwalające całkowicie kontrolować dostęp do pliku. Nie są to jednak wszystkie możliwości. Do obsługi pliku, a dokładniej jego metryczki zawierającej metadane, można jeszcze użyć funkcji znajdujących się w drugiej strukturze, o nazwie *inode_operations*. Spośród funkcji dostępnych w obu strukturach wybrałem te, które według mnie pozwalają na wystarczającą kontrolę nad plikiem, jaka została założona w projekcie. Są to wszystkie funkcje dostępne w *file_operations* czyli *llseek, read, write, readdir, poll, ioctl, mmap, open, flush, release, fsync, fasync i lock*. Właściwie ten zestaw by wystarczył, gdyby nie pewne operacje znajdujące się w *inode_operations*, które pozwalają na modyfikację danych wewnątrz pliku. Takim przykładem jest funkcja *truncate()*. Nie musi ona otwierać pliku, aby usunąć jego zawartość. To samo stwierdzenie o braku potrzeby otwierania pliku stosuje się do wszystkich innych funkcji zawartych w *inode_operations*. Mają one jedną wspólną cechę. Zanim zostaną wykonane jakiekolwiek operacje na pliku, sprawdzane są prawa dostępu za pomocą funkcji *permission()*. W związku z tym stwierdziłem, że wystarczy dodanie możliwości modyfikacji funkcji *permission()*, nie zaś całego zestawu *inode_operations*. Co prawda nie wiadomo, z której z pozostałych funkcji wywołano *permission()*, ale to nie będzie potrzebne dla poprawnego działania systemu. Do zestawu trzynastu wymienionych wyżej funkcji z *file_operations* dodałem tylko jedną z operacji na metryczkach, czyli *permission()*. Jeśli będziemy mogli zdefiniować ten zestaw na nowo, to osiągniemy cel, czyli (prawie) całkowitą zmianę semantyki operacji na pliku.

Oczywiście strażnik nie musi definiować wszystkich funkcji na swój sposób, co oznacza możliwość selektywnego wyboru zmian operacji na pliku. Na przykład strażnik może definiować tylko funkcję *open()*, notując odwołania do pliku bądź implementując ACL. Reszta funkcji powinna się wykonywać w standardowy sposób i nie powinno mieć to wpływu na prędkość ich działania, czyli nie powinno to dodawać narzutu czasowego związanego z obsługą strażnika. Możliwe akcje do wykonania przez podmienioną funkcję obsługi żądania w strażniku to:

- wykonanie operacji i przekazanie wyniku bądź dowolnego kodu błędu,
- wykonanie oryginalnej funkcji w celu otrzymania wyników, które następnie można zmienić (również na kod błędu),
- wykonanie dowolnych operacji, a następnie wysłanie prośby do jądra o realizację operacji w standardowy sposób.

Aby zlecanie operacji strażnikowi było możliwe musi istnieć dobrze zdefiniowany interfejs komunikacji ze strażnikami. Ponieważ strażnicy są częścią systemu operacyjnego, można próbować przeprojektować struktury systemu VFS tak, aby była możliwość wkomponowania ich w działanie systemu operacyjnego. Jednak optymalnym rozwiązaniem wydaje się realizacja tego poprzez wywołania funkcji ze strażnika. Strażnik zgłasza do systemu swoje funkcje, które zasłaniają funkcje oryginalnie wywoływane przez VFS.

Następnie musi istnieć możliwość pilnowania przez strażnika wielu plików, jak również wielu instancji otwarcia jednego pliku na różne sposoby. To pociąga za sobą potrzebę wiązania strażnika do pliku, a nie odwrotnie. Teoretycznie można sobie wyobrazić, że przy strażniku mamy listę plików, które on obsługuje, ale oczywiście byłoby to mniej efektywne rozwiązanie. Takie powiązanie musi być stałe i informacja o nim musi przetrwać restart bądź krytyczne zatrzymanie systemu. Identyfikator strażnika powinien być zatem zapisany na dysku, razem z innymi atrybutami pliku, takimi jak właściciel czy prawa dostępu. A zatem ta informacja powinna znaleźć się w metryczce pliku. Wydaje się to być najbardziej sensownym miejscem, ograniczy bowiem możliwości manipulowania dostępem do pliku za pomocą dowiązań do pliku. Prawo modyfikacji powiązania pliku ze strażnikiem powinien mieć tylko właściciel (oraz superużytkownik). Ponieważ w metryczce pliku zawsze jest ograniczona ilość miejsca trzeba zrezygnować z możliwości pilnowania jednego pliku przez kilku strażników. Mogłoby to zwiększyć ogólność rozwiązania, ale większość potrzeb da się zaimplementować za pomocą jednego strażnika. W ostateczności można się pokusić o napisanie strażnika wołającego innych strażników. Wymóg rozróżniania instancji otwarcia danego pliku może być rozwiązany poprzez nadawanie unikatowego identyfikatora sesji przy otwieraniu pliku. Dzięki temu strażnik będzie w stanie na różne sposoby obsłużyć różne instancje otwarcia tego samego pliku. Na przykład prezentować odmienną zawartość dla różnych użytkowników.

Kolejną właściwością rozwiązania musi być możliwość zdefiniowania operacji w przypadku braku odpowiedniego strażnika. Najwłaściwsze wydają się dwie możliwości: wykonać operację w standardowy sposób bądź zabronić jej wykonania. Ponieważ nie ma możliwości, aby strażnik definiował to zachowanie, trzeba zastosować rozwiązanie ogólnosystemowe.

Następną wymaganą właściwością strażników jest przezroczystość. Mechanizm musi być w taki sposób wbudowany w jądro, aby istniejące programy mogły korzystać z odmiennego dostępu do plików, nie wiedząc nawet o jego istnieniu. Co więcej, jest nawet niepożądane, aby procesy wiedziały czy i w jaki sposób plik jest pilnowany. Można by to wykorzystać do otrzymania bezpośredniego dostępu do strażnika (z pominięciem systemu operacyjnego), co byłoby potencjalną luką w bezpieczeństwie. W związku z tym interfejs dostępu do plików nie może się zmienić. Cała sesja otwarcia pliku musi być obsługiwana przez VFS, który to wykona odpowiednie operacje dla pilnowanych plików.

Strażnik musi mieć możliwość wykonywania się jako proces w przestrzeni użytkownika. Jest to wymóg istotny, jeśli weźmie się pod uwagę potrzebę prostoty używania i tworzenia strażników, jak również potrzebę wywoływania strażników przez zwykłych użytkowników systemu. Uruchamianie strażników w trybie użytkownika ma wszystkie zalety pracy ze zwykłymi programami (np. odpluskiwanie), a jednocześnie pozwala na realizację przez nie wywołań systemowych. Uruchamianie pod kontrolą odpluskwiacza pozwoli na łatwe sprawdzanie działania i szukanie błędów. Kodu wykonującego się w trybie jądra nie można uruchomić w takim środowisku. Ponieważ procesy podlegają mechanizmowi szeregowania i mogą być wywłaszczane (system nie), funkcje strażników wykonywane w trybie użytkownika mogą zająć dużo więcej czasu procesora niż kod w trybie systemowym. Długie wykonywanie kodu w trybie jądra powoduje duże opóźnienia w reaktywności systemu.

Jednak nie możemy również pozwolić na blokowanie systemu poprzez nie odpowiadającego na żądania strażnika. W tym celu należy zaimplementować metodę wykrywania takich nieprzyjaznych strażników i eliminować ich z systemu. Oczywiście strażnicy pracujący w trybie jądra mogą być bardzo pomocni lub nawet niezbędni w przypadku krótkich i sprawdzonych procedur obsługi. Spowoduje to niezaprzeczalny wzrost wydajności poprzez wyeliminowanie obsługi procesów użytkownika oraz wymiany danych z nimi.

Jest oczywiste, że procesy strażników nie byłyby w stanie realizować swych zadań w systemie operacyjnym bez wsparcia ze strony jądra. Zatem część rozwiązania musi znajdować się w jądrze systemu operacyjnego. Ponieważ strażnicy będą się wykonywać jako procesy użytkowe, musi zostać opracowany mechanizm komunikacji między jądrem i nimi. Ma to być swego rodzaju interfejs dostępu do systemowej części obsługi strażników z poziomu procesu użytkownika. Mechanizm funkcji systemowych i sygnałów jest asymetryczny i nie będzie wystarczający. Wygodnym będzie mechanizm pozwalający na traktowanie stron jako rozmówców w procesie porozumiewania się opartym na komunikatach. Jednym z takich sposobów jest wykorzystanie urządzenia *netlink*, ale można również rozpatrzeć takie mechanizmy jak system plików i urządzeń, podsystem *proc* albo nawet wymiana danych poprzez pamięć procesu. Na pewno jednak nie może to być rozwiązanie w jakikolwiek sposób blokujące system (np. odpytywanie z dużą częstotliwością). Cechy, jakie powinien posiadać wybrany mechanizm, to: bezpieczeństwo, dobrze zdefiniowany protokół komunikacji i prawidłowa reakcja na błędy.

Pożądane byłoby, aby strażnicy działający w trybie użytkownika używali tego samego interfejsu co ci załadowani z modułów. Chodzi o to, aby nie tworzyć dwóch różnych mechanizmów dla obsługi osobnych przypadków: strażnik jako proces użytkownika i strażnik systemowy. Sensowne wydaje się zbudowanie mechanizmu komunikacji procesów z jądrem tak, aby móc wykorzystać interfejs zbudowany na potrzeby rejestracji modułów. Można to uzyskać tworząc dodatkowego pośrednika, który zajmuje się komunikacją z procesami użytkownika i przekazywaniem informacji do jądra zgodnie z interfejsem zdefiniowanym dla modułów systemowych.

W końcu należy opracować metodę przypisywanie strażników do plików. Aby manipulowanie strażnikami było możliwe w czasie pracy systemu, trzeba zdefiniować interfejs użytkownika pozwalający na odczyt, kasowanie i zmianę strażnika dla pliku. Odpowiednim rozwiązaniem wydaje się dodatkowa funkcja systemowa, specjalnie dedykowana do tego celu, albo nowe polecenie dla funkcji *ioctl()*.

Do wszystkich powyższych punktów należy dołożyć wymóg bezpieczeństwa i odporności na błędy. Projekt musi być tak zaimplementowany, aby zapewniać bezpieczeństwo zarówno danych, jak i systemu operacyjnego. Dodatkowo reakcje na błędy muszą być dobrze zdefiniowane, jak również implementacja musi wychwytywać wszystkie możliwe sytuacje awaryjne, tak aby nie była możliwa blokada systemu operacyjnego.

4.3. Implementacja

Jednym z celów pracy była taka implementacja rozwiązania, która będzie wymagać jak najmniejszej ingerencji w źródła Linuksa. Sądzę, że cel został osiągnięty. Zmodyfikowano dziewięć plików źródłowych, nie wliczając w to dokumentacji i konfiguracji. Zmiany polegały na dodaniu osiemnastu wierszy kodu. Resztę źródeł systemu strażników zgrupowałem w dodatkowym katalogu, tak samo jak dokumentację oraz przykłady zastosowania. Zarówno dokumentacja techniczna, jak również źródła, są dołączone do tej pracy jako załączniki. Podawane w dalszej części pracy nazwy plików są nazwami absolutnymi (jeśli zaczynają się od ukośnika np. */proc/fs/fg_allow*) lub nazwami względnymi, dla których punktem odniesienia jest katalog zawierający źródła Linuksa (standardowo */usr/src/linux*). W kolejnych podrozdziałach przedstawię sposób realizacji zadania, które sobie postawiłem.

4.3.1. Strażnicy w trybie jądra

Jednym z wymagań było, aby strażnicy mogli działać w trybie uprzywilejowanym, czyli jądra. Dość naturalne wydaje się, aby każdy strażnik mógł być ładowany do systemu jako moduł (moduł może zawierać więcej niż jednego strażnika). W tym celu należało stworzyć jakiś mechanizm rejestracji strażników w systemie. Oczywiście nie miałoby to sensu, gdyby brakowało mechanizmu przejmowania kontroli nad opisanymi wcześniej funkcjami dotyczącymi obsługi plików. Te dwa mechanizmy postaram się teraz opisać. Całość jest zaimplementowana w jednym dodatkowym pliku w źródłach systemu Linux. Kod rozwiązania znajduje się w pliku *fs/fg/fg_main.c*. Wykorzystywany jest też plik z nagłówkami *include/linux/fg.h*.

Podmianę funkcji zaimplementowłem w dość prosty sposób. Otóż przy otwieraniu pliku, każda instancja otwarcia otrzymuje swój wskaźnik na strukturę *file_operations*, która zawiera funkcje obsługi odpowiednie dla tego pliku. Postanowiłem wykorzystać ten fakt i przy otwieraniu pliku oznaczonego jako pilnowany podmieniać ten wskaźnik tak, aby wskazywał na moją własną zdefiniowaną strukturę *file_operations*. Tę strukturę oraz wszystkie inne potrzebne funkcje zdefiniowałem w pliku *fg_main.c*. Niestety nie mogę modyfikować ogólnej systemowej struktury. Opóźniłoby to wykonywanie operacji na plikach, które nie są pilnowane. Tak by się stało, ponieważ system strażników obsługiwałby wszystkie wykonania funkcji na wszystkich plikach. Taki narzut czasowy jest niepożądany. Dzięki podmianie struktury wszystkie wywołania funkcji na otwartym i pilnowanym pliku są przekierowywane do mojego kodu. Wyjątkiem są dwie z nich, które nie wymagają otwierania pliku. Są to *open()* i *permission()*. Obie te funkcje wymagały modyfikacji systemowych źródeł na poziomie VFS. Pierwszą z modyfikacji umieściłem w funkcji *dentry_open()* z pliku *fs/open.c*. Sprawdzam tam, czy otwierany plik jest pilnowany i jeśli tak, to zamiast wołania domyślnej funkcji otwarcia pliku dla danego systemu plików, wołam mój zamiennik *fg_open()*. Tu właśnie odbywa się wołanie funkcji ze strażnika i w przypadku sukcesu (strażnik przekazał zero, a nie kod błędu), następuje podmiana struktury *file_operations* na moją. Dalsze wywołania operacji na otwartym pliku trafiają już do moich funkcji. Drugą funkcją potrzebującą modyfikacji w jej oryginalnej implementacji jest *permission()*. Modyfikując jej kod w pliku *fs/namei.c* zdefiniowałem przekierowanie do własnej funkcji *fg_permission()*. Sposób realizacji jest taki sam jak dla *open()*. Parametry wywołań funkcji są zawsze zachowywane. Pozwala to dowolną modyfikację funkcji domyślnych z takimi samymi uprawnieniami i możliwościami. Dla przykładu pokażę wywołanie *fg_permission()*:

```
int permission(struct inode *inode, int mask) {
    if (inode->fg) { /* czy plik pilnowany?  */
        int ret = fg_permission(inode, mask);
        if (ret != -ENOSYS) return ret;
    };
/* dalej oryginalny kod funkcji */
```

Oczywiście od strażnika wymaga się również dostarczenia funkcji (np. *permission()*) o takim samym prototypie jak oryginalna. Mój kod nie robi nic ponad sprawdzenie, czy strażnik obsługuje tę funkcję i woła ją dalej z takimi samymi parametrami. Warto zaznaczyć, że sprawdzenie pola *inode->fg* jest jedynym narzutem czasowym w przypadku obsługi niepilnowanego pliku (w porównaniu do systemu bez strażników). Tam gdzie było to możliwe, moje funkcje oznaczyłem jako rozwijane w miejscu wywołania (ang. *inline*)[1] dla optymalizacji kodu. Na

[1] Oznacza to, że kod funkcji zostanie wstawiony w miejsce wywołania funkcji. Przyspiesza to wykonanie programu redukując koszty wywołania funkcji.

przykład:

```
inline int fg_permission(struct inode *inode, int mask){
    /* czy strażnik zarejestrowany i obsługuje fg_permission() ?  */
    return ( strażnik->permission
        ?  strażnik->permission(inode, mask)
        :  -ENOSYS);
};
```

Trzeba tutaj zauważyć, że do swoich celów przeciążyłem kod błędu *ENOSYS*. Informacją niesioną przez ten kod błędu jest brak implementacji żądanej operacji. Można zatem ponownie próbować wywołać funkcję systemową. Jeśli funkcja strażnika daje wynik *ENOSYS*, oznacza to, że należy wykonać oryginalną funkcję. Nie prowadzi to do żadnych nieprzewidzianych bądź niezdefiniowanych zachowań systemu plików, choć nakłada pewne ograniczenie na strażników. Nie ma możliwości przekazania jako wyniku funkcji systemowej błędu o kodzie *ENOSYS*. Uczyniłem to świadomie. Ten kod błędu jest zarezerwowany dla przypadku braku funkcji a nie błędnego działania. Jedynym przypadkiem powodującym potencjalny konflikt jest sytuacja, w której strażnik sam woła oryginalną funkcję i jej wykonanie powoduje błąd o kodzie *ENOSYS*. Sądzę jednak, że nie stanowi to problemu krytycznego. Moje funkcje najwyżej wykonają oryginalną funkcję jeszcze raz. Strażnik może również sam reagować na błąd wołanej przez siebie funkcji i zmieniać go na inny. Nie jest to konieczne, ponieważ ponowne wywołanie funkcji wygeneruje taki sam wynik — funkcja nie jest zaimplementowana. Aby strażnik mógł wykonać oryginalną funkcję systemową dla pliku, poprzednia struktura *file_ operations* jest zapamiętana na dodatkowej zmiennej w strukturze *file*. Dzięki temu strażnicy mają dostęp do oryginalnych funkcji odpowiednich dla danego systemu plików.

Każda z funkcji opakowujących wywołania funkcji strażników jest napisana według tego samego schematu. Dodatkowo funkcja *release()* przywraca wskaźnik na oryginalną strukturę *file_ operations*. Schemat jest następujący (na przykładzie *llseek()*):

```
ret = -ENOSYS;
strażnik = znajdź_odpowiedniego_strażnika();
if (strażnik && strażnik_obsługuje_llseek)
    ret = strażnik->fgops->llseek(file, offset, origin);
if (ret == -ENOSYS)
    ret = (system_plików_obsługuje_llseek
        ?  oryginalny_llseek(file, offset, origin)
        :  generyczna_funkcja_VFS);
return ret;
```

Dzięki takim zabiegom, strażnicy działający w trybie jądra mają całkowitą kontrolę nad sposobem wykonania operacji na pliku. Można się zastanawiać czy jest bezpieczne dawać im tak duże uprawnienia. Ten temat poruszam w następnym rozdziale. Tutaj ograniczę się do stwierdzenia, że moduły są częścią systemu operacyjnego, do którego z założenia ma się zaufanie.

Pozostaje jeszcze problem rejestracji strażników w systemie. Inaczej niż w projektach opisanych w poprzednim rozdziale, system nie bierze czynnego udziału w uruchamianiu strażników. Aby strażnik zaczął działać w systemie, administrator systemu musi załadować moduł ze strażnikiem. Od tej chwili strażnik jest zarejestrowany w systemie i udostępnia swoje usługi. Wyjątkiem jest sytuacja, gdy w systemie operacyjny istnieje wsparcie do automatycznego ładowania modułów przez jądro (w systemie Linux jest to opcjonalne, choć zazwyczaj używane). Wtedy przy dostępie do pilnowanego pliku, gdy strażnik nie jest zarejestrowany, następuje próba załadowania odpowiedniego modułu do jądra. Nazwa jest budowana na podstawie nazwy strażnika. W przypadku braku strażnika jądro musi zdecydować co zrobić z próbą dostępu do pliku. Sensowne wydają się dwie możliwości. Po pierwsze można pominąć mechanizm strażników i zrealizować dostęp do pliku w standardowy sposób. Po drugie można zakazać dostępu, jako że nie są dostępne funkcje pozwalające interpretować plik w odpowiedni sposób. To zachowanie można zmieniać podczas działania systemu za pomocą pliku */proc/fs/fg_allow*. Domyślne zachowanie ustala się podczas konfiguracji jądra systemu przed jego kompilacją. Procesy superużytkownika otrzymają dostęp do pliku w każdym przypadku.

Sama rejestracja strażników odbywa się poprzez dwie eksportowane funkcje. Są to odpowiednio *register_fgfunc* oraz *unregister_fgfunc*. Strażnik powinien zawołać pierwszą z nich gdy rozpoczyna pracę, zaś drugą gdy kończy. Obydwie funkcje przyjmują te same argumenty. Są to nazwa strażnika oraz wskaźnik do struktury *fgf_operations*, w której znajdują się wskaźniki do obsługiwanych przez strażnika funkcji. Jeśli w strażniku nie ma implementacji którejś z funkcji, to na jej miejscu w *fgf_operations* powinna znaleźć się wartość NULL. Strażnicy są wpisywani do struktury *fg_fo_list*, w której pamięta się wszystkich działających w systemie strażników. Wygląda ona następująco:

```
struct fg_fo_list {
    int name; /* nazwa strażnika zmieniona na liczbę */
    struct fgf_operations *fg_ops; /* wskaźniki do zestawu funkcji */
    atomic_t count; /* liczba otwartych plików */
    struct fg_fo_list *next; /* implementacja listy */
};
```

W tej realizacji projektu jest to lista prosta, ale nie ma przeciwwskazań, aby zaimplementować tę strukturę jako tablicę z funkcją mieszającą lub drzewo. Komentarza wymaga pole zawierające nazwę strażnika. W celu uogólnienia rozwiązania powinno ono być typu ciąg znaków (char *), jednak ograniczenie wynikające z użycia systemu plików extended2 (również extended3 i reiserfs) pozwala na realizację nazw zawierających nie więcej niż cztery znaki. Stąd pomysł odwzorowania nazwy w liczbę. Liczba całkowita jest zapisana na czterech bajtach — to tyle samo co ciąg długości czterech znaków. Taki zabieg istotnie przyspiesza wyszukiwania strażników po nazwach. Warto nadmienić, że interfejs użytkownika traktuje nazwy strażników jako ciągi znaków. Nic więc nie stoi na przeszkodzie, aby w przyszłości, kiedy będzie możliwość nadawania strażnikom nazw dłuższych niż cztery znaki, zmienić to pole na typ znakowy.

Ważne jest również pole *count*. Jest ono zwiększane przy każdej udanej operacji otwarcia pliku i zmniejszane przy zamknięciu. Dodatkowo jeśli przy zwiększaniu pole zmienia wartość na jeden, to w imieniu modułu wykonywana jest makrodefinicja zwiększająca liczbę odwołań systemowych do modułu. Jeśli *count* osiągnie wartość zero, wykonywana jest operacja odwrotna. To skutecznie zapobiega sytuacji, w której strażnik chce się wyrejestrować z systemu operacyjnego mając otwarte pliki. Taki przypadek spowodowałby odwołania do pamięci, która została już zwolniona, a w konsekwencji nawet do zatrzymania systemu. Dopóki jednak nie może on zwolnić zajmowanej przez siebie pamięci, dopóty sytuacja jest bezpieczna. Powyższe rozwiązanie wydaje się być zbyt restrykcyjne. Łatwo wyobrazić sobie scenariusz, w którym strażnik nigdy nie będzie miał szansy, aby zakończyć działanie. W ten sposób unika się dwuznaczności związanej z obsługą pliku. Załóżmy, że strażnik może się wyrejestrować podczas gdy w systemie istnieją otwarte instancje pilnowanych przez niego plików. Otóż dalsza obsługa tych plików nie jest jednoznaczna. Ponieważ od tej pory nie możemy korzystać z funkcji strażnika, nie jest do końca jasne jak mają być obsługiwane otwarte pliki. W przypadku strażników działających w uprzywilejowanej przestrzeni jądra nie ma to większego znaczenia, ponieważ operacje powinny być wykonywane zgodnie z wartością flagi *fg_ allow*. Jeśli jej wartością jest prawda, plik będzie obsługiwany w standardowy sposób, co może prowadzić do błędnych sytuacji (na przykład pierwsza połowa danych z pliku zakodowana a druga nie). Z tych rozważań wynika, że zabezpieczenie jest uzasadnione i poprawne.

Na koniec należy dodać, że zostały zaimplementowane dwa pliki w systemie *procfs*:

- */proc/fs/fg_ allow*, którego wartość determinuje zachowanie jądra w przypadku braku strażnika dla pilnowanego pliku,
- */proc/fs/fg_ stat*, który zawiera różne statystyki dla zarejestrowanych strażników. W tej chwili jest to po prostu lista aktywnych w systemie strażników.

4.3.2. Strażnicy w trybie użytkownika

Z punktu widzenia implementacji strażnicy działający w trybie użytkownika niczym się nie różnią od tych działających jako moduły. Całość kodu odpowiedzialnego za możliwość pilnowania plików przez procesy użytkownika znajduje się w osobnym module, który działa w systemie strażników zgodnie z mechanizmem opisanym w poprzednim punkcie. Za każdym razem, gdy proces użytkownika rejestruje się jako strażnik, robi to za pomocą dostarczonego przeze mnie modułu *fg_ user*. Moduł ten rejestruje się w systemie operacyjnym jako strażnik o tej samej nazwie (wykonuje funkcję *register_fgfunc* — proces nie może tego uczynić[2]). W ten sposób moduł jest tylko pośrednikiem (ang. *proxy*) między procesami użytkownika i jądrem. Dzięki takiemu podejściu, jeśli ktoś zechce napisać taki moduł pośredniczący na inny sposób, będzie to mógł zrobić. Nie ma powodów, aby w systemie nie mogło istnieć wiele takich modułów-pośredników. Istotne jest, że jeśli w systemie nie jest obecny moduł *fg_ user*, procesy nie mogą działać jako strażnicy, gdyż nie są w stanie się zarejestrować. Procesy mogą same próbować załadować moduł do systemu. Wymagane są do tego prawa superużytkownika.

Interfejs używany między modułem *fg_ user* i jądrem został już zaprezentowany w punkcie 4.3.1. Trzeba jeszcze opisać w jaki sposób procesy użytkownika komunikują się z modułem. Do tego celu postanowiłem wykorzystać mechanizm już istniejący w systemie i na jego bazie zaimplementować tę komunikację. Tak samo jak we wcześniejszych rozwiązaniach (opisanych w rozdziale 3), jest ona oparta na wymianie komunikatów. Mechanizm ten to interfejs systemu plików dla urządzeń specjalnych, zrealizowany w VFS. I tak z punktu widzenia procesu jest to plik, a dla modułu specjalne urządzenie, obsługiwane przez niego.

Dla swoich potrzeb wybrałem jeden główny numer urządzenia, zarezerwowany w Linuksie na potrzeby programowania i testowania. Moduł *fg_ user* rozpoczyna działanie od zainstalowania procedury obsługi tego urządzenia. Następnie czeka, aż jakiś proces skorzysta z urządzenia i poprosi o rejestrację. Rejestracja polega na otworzeniu urządzenia z numerem pomocniczym równym zero i odczytaniu z niego komunikatu. Komunikaty mają ustaloną strukturę, którą opisuję w dalszej części pracy. W odczytanym komunikacie jest zawarta informacja o numerze kanału komunikacji zarezerwowanym dla procesu strażnika. Następnie

[2]Używanie eksportowanych symboli jądra (w tym *register_fgfunc*) jest operacją uprzywilejowaną, niedostępną dla procesów użytkownika.

należy otworzyć urządzenie z takim numerem podrzędnym, jaki odczytano z komunikatu. Poprzez nowy kanał należy wysłać komunikat zawierający nazwę strażnika oraz tablicę funkcji obsługiwanych przez strażnika. Implementacja zabrania jakiegokolwiek zapisu do urządzenia głównego. W tym momencie można już je zamknąć, jako bezużyteczne. Moduł zapewnia, że urządzenie o numerze, który został wysłany w komunikacie, będzie mógł otworzyć tylko proces, który odebrał ten komunikat. Rozpoznawanie odpowiedniego procesu jest realizowane na podstawie unikatowego identyfikatora PID (ang. *process id*). Zatem ani proces potomny, ani żaden inny, nie będzie w stanie otworzyć zarezerwowanego kanału komunikacji. Jest to zabezpieczenie na wypadek próby podszycia się innego procesu pod strażnika. Tablica przekazywana do modułu w czasie rejestracji zawiera kody błędów jakie należy przekazać procesowi próbującemu otrzymać dostęp do danej funkcji. Tablica jest strukturą *fgu_ operations* zawierającą wyłącznie liczby całkowite, oznaczające kody błędów dla odpowiednich funkcji: *llseek, read, write, readdir, poll, ioctl, mmap, open, flush, release, fsync, fasync, lock i permission.* Wartości są interpretowane w następujący sposób:

- FGU_MYSELF (0) oznacza, że strażnik obsługuje funkcję,
- FGU_NORMAL (-ENOSYS) oznacza, że strażnik nie implementuje funkcji i prosi jądro o wykonanie funkcji standardowej,
- FGU_REJECT (-EPERM) oznacza, że wykonywanie funkcji jest niedozwolone,
- dowolna wartość mniejsza od zera oznacza kod błędu jaki zostanie przekazany w miejsce wywołania,
- FGU_WDATA (1) oznacza, że strażnik chce, aby jądro wykonało operację i dostarczyło wyniki do weryfikacji i ewentualnej zmiany. Obecnie jest to możliwe tylko dla funkcji *open()*.

Od momentu rejestracji komunikacja odbywa się na zasadzie: żądanie od jądra do strażnika, odpowiedź w drugą stronę. Strażnik nie ma prawa wysyłać do jądra komunikatów, które nie są odpowiedzią na jakieś żądanie. Każdy komunikat ma ustalony format i składa się z dwóch części: nagłówka i dodatkowych danych. Nagłówek ma zawsze taką samą strukturę:

```
struct fgu_mesg {
    int magic;
    int cmd;
    int no;
    int sid;
    pid_t pid;
    int len;
};
```

Pole magic powinno być w każdym komunikacie takie samo i mieć wartość FGU_MAGIC. Jeśli jądro otrzyma komunikat zawierający inną wartość, to oznacza to, że albo wysłano do niego błędne informacje, albo w jakiś sposób nastąpiło przesunięcie danych. W obu przypadkach zostanie wygenerowany błąd.

Kolejne pole (cmd) to kod operacji, której dotyczy komunikat. Dzięki niemu strony wiedzą, o którą funkcję chodzi (np. *open*). Dotyczy to przede wszystkim żądania wysyłanego do procesu.

Następne pole to numer komunikatu (no). Jeśli z jądra zostanie wysłane żądanie do procesu, ten musi odpowiedzieć komunikatem z tym samym numerem. Numer jest wartością losową, dzięki czemu system ma dodatkową pewność, że nikt nie podszywa się pod strażnika. Jeśli komunikat zawiera inny numer, to jest sygnalizowany błąd. Obecnie kolejne komunikaty mają numer równy trzydziestu dwóm młodszym bitom licznika TSC (ang. *time stamp counter*) procesora lub wartość *jiffies* dla maszyn bez wsparcia dla TSC[3]. Sądzę, że rozdzielczość licznika TSC jest wystarczająco duża, aby praktycznie niemożliwe było odgadnięcie wygenerowanego numeru komunikatu. W razie obawy, że to za mało, można zaimplementować w to miejsce liczby pseudolosowe. Liczba ta jest też używana do odnalezienia żądania, na które odpowiedzią jest wiadomość przysłana przez strażnika.

Kolejna wartość to identyfikator sesji (sid). Każde otwarcie pliku powoduje rozpoczęcie nowej sesji i przydzielenie nowego identyfikatora. Wszystkie komunikaty dotyczące tej samej instancji otwarcia pliku będą miały ten sam identyfikator. Wszystkie otwarte w danym momencie sesje mają różne identyfikatory. Dzięki temu proces może pilnować wielu instancji otwarcia pliku, zawsze wiedząc, której z nich dotyczy komunikat od jądra systemu. Wartość tego pola to po prostu wartość wskaźnika na strukturę *file*, a dla wywołań *permission()* adres struktury *inode*. Powoduje to, że strażnik nie może jednoznacznie określić, do której sesji należy wywołanie *permission()*, ale może się dowiedzieć, na którym pliku jest wykonywana ta operacja. To pole może być traktowane jako dodatkowy identyfikator komunikatu, dlatego proces nie może zmienić jego wartości podczas odpowiedzi na żądanie.

Pole pid jest identyfikatorem procesu wykonującego funkcję systemową (klienta). Dzięki niemu strażnik, korzystając z systemu *procfs*, może określić właściciela procesu klienta i szereg innych informacji potrzebnych mu do realizacji zadania.

Ostatnie pole (len) jest długością następujących po strukturze danych dodatkowych wyrażoną w bajtach. Te dane również posiadają własną strukturę, w zależności od typu polecenia, którego dotyczy komunikat. Zawierają takie informacje jak parametry wywołania funkcji i dodatkowe dane potrzebne do wykonania operacji albo wyniki dostarczane przez strażnika. Dla przykładu żądanie wykonania wykonania funkcji *write()* zawiera dane przeznaczone do zapisu, zaś dla wywołania *open()* — ścieżkę dostępu do obsługiwanego pliku i ewentualnie kod, jaki otrzymano z wywołania oryginalnej funkcji. Jest to często stosowana metoda, ponieważ dzięki przekazanej nazwie strażnik może otworzyć plik dla własnych celów, będąc pewnym, że otwarcie pliku dla klienta się powiodło. Trzeba tu nadmienić, że otwieranie przez strażnika pilnowanego przez siebie pliku może prowadzić do zapętlenia odwołań do pliku. Jeśli strażnik ma ciągle ten sam identyfikator PID, pod jakim występował rejestrując się, to może swobodnie wykonywać taką operację. System wykrywa tę sytuację i przyznaje dostęp do pliku z pominięciem mechanizmu strażników. W przeciwnym przypadku proces sam musi wykrywać taką sytuację i nie obsługiwać wywołań od siebie samego. Niestety nie znalazłem innej możliwości rozwiązania tego problemu, poza bardzo czasochłonnym sprawdzaniem, czy proces wysyłający żądanie nie ma przypadkiem otwartego odpowiedniego urządzenia komunikacji. Jest to jedyny zawsze wspólny fakt dla takich procesów, ponieważ otwarte urządzenie mogło być tylko odziedziczone.

Po szczegółowe dane dotyczące protokołu wymiany komunikatów odsyłam do załączonych do pracy źródeł implementacji i komentarzy w nich zawartych. Nadmienię jednak, że jakiekolwiek naruszenie protokołu, który został określony dość szczegółowo i restrykcyjnie, powoduje zerwanie komunikacji ze strażnikiem.

[3]Jest to co najmniej sześćdziesięciocztero bitowa liczba zwiększana w każdym cyklu procesora. Zatem dla jednostki taktowanej 1MHz wartość TSC zmienia się milion razy na sekundę. Wartość *jiffies* zmienia sie sto razy na sekundę niezależnie od prędkości jednostki centralnej.

Strażnicy realizowani przez procesy nie mają możliwości obsługi wszystkich funkcji. Dla niektórych z nich mogą najwyżej określić kod błędu. Są to:

- *readdir()*, ponieważ nie ma możliwości przekazania do procesu funkcji wypełniającej katalog (parametr *filldir_t*),
- *poll()*, ponieważ nie ma możliwości modyfikowania przez proces tablic systemowych *(poll_table_struct)*,
- *mmap()*, ponieważ proces nie może modyfikować systemowych struktur pamięci,
- *lock()*, ponieważ proces nie może modyfikować struktury jądra *file_lock*, potrzebnej przy realizacji blokowania obowiązkowego (ang. *mandatory/exclusive lock*), przy którym system sam weryfikuje blokady na plikach.

Moduł *fg_user* implementuje wszystkie funkcje, które są zawarte w strukturze *fgf_operations*. To one są wywoływane przez system, gdy jest wykonywana operacja na strzeżonym pliku. Są one niezbędne, ponieważ proces w każdej chwili może zniknąć z systemu, a zanim moduł wyrejestruje strażnika, musi w jakiś sposób reagować na nadchodzące żądania. Jest to realizowane w tych funkcjach. Brak strażnika jest wykrywany w bardzo prosty sposób. System przy kończeniu procesu (nie istotne z jakich powodów) zamyka wszystkie otwarte przez niego deskryptory. Wśród nich jest urządzenie komunikacji. Jest to też standardowa procedura wyrejestrowania strażnika. Nie został zaprojektowany żaden komunikat zawierający polecenie wyrejestrowania (pamiętajmy, że proces nie może zgłaszać żadnych żądań do jądra).

Mechanizm wywołania strażnika jest następujący. Na początku system wykonuje funkcję modułu *fg_user* z odpowiednimi parametrami. Tworzony jest odpowiedni komunikat, który zostaje wysłany do strażnika przy pomocy urządzenia komunikacji. Proces wołający funkcję obsługi pliku (klient) zostaje uśpiony do czasu otrzymania od strażnika odpowiedzi na zlecenie. Kiedy strażnik wykona operację i przyśle odpowiedź, wykonanie procesu-klienta zostanie wznowione, a na podstawie komunikatu odpowiedzi zostaje wygenerowana odpowiedź dla systemu operacyjnego, a w efekcie dla procesu próbującego skorzystać z pliku.

Kilka kwestii wymaga dokładniejszego omówienia. Po pierwsze, strażnik ma określony czas na obsługę żądania. Domyślna wartość to pięć sekund, ale administrator może ją dowolnie definiować w zakresie od 1 do 60 sekund. Wartość tę można ustawiać za pomocą pliku */proc/fs/fgu_timeout*. Z niego można również odczytać aktualną wartość. Moduł czeka na odpowiedź od strażnika maksymalnie *fgu_timeout* sekund. Jeśli odpowiedź nie przyjdzie w ustalonym czasie, wiadomość jest anulowana, a proces otrzymuje kod błędu informujący o potrzebie ponownego wywołania funkcji systemowej (ERESTART). Jeśli strażnik ma zbyt dużo pracy i realizacja zadań przekracza ustalony czas, można stworzyć dla niego procesy potomne wykonujące te same zadania. Trzeba jednak wziąć pod uwagę fakt, że komunikaty dotyczące tej samej sesji mogą trafić do różnych procesów.

Strażnik nie musi obsługiwać wielu żądań równolegle. Implikuje to potrzebę stworzenia kolejki komunikatów wysyłanych do strażnika. Każdy strażnik ma dwie takie kolejki. Jedna to kolejka komunikatów czekających na wysłanie do strażnika. Są to żądania, które zostały już przygotowane do wysłania, ale strażnik nie odebrał jeszcze wiadomości. Po wysłaniu wiadomości do strażnika, nagłówek komunikatu trafia do drugiej kolejki — odbiorczej. Z niej moduł pobiera odpowiedzi do dalszej pracy. Jest istotne, że czas *fgu_timeout* liczy się od momentu wstawienia komunikatu z żądaniem do kolejki wiadomości do wysłania, nie zaś od momentu przekazania wiadomości do strażnika.

Wyjaśnienia wymaga jeszcze jeden fakt. Otóż z uwagi na sposób rejestracji, każdy proces — nawet nie uprzywilejowany — może zostać strażnikiem. W związku z tym, domyślnie, system zabrania rejestracji procesów nie działających z prawami superużytkownika. Aby to zmienić można skorzystać z pliku */proc/fs/fgu_non_root*. Jeśli znajduje się w nim wartość 0, to oznacza, że tylko procesy uprzywilejowane mogą rejestrować się jako strażnicy. Dowolna inna wartość pozwala dowolnym procesom działać jako strażnik w systemie. Należy zwrócić uwagę, że opisana flaga ma wpływ tylko na proces rejestracji i nie jest sprawdzana w czasie wymiany komunikatów ze strażnikiem. Oznacza to, że zmiana jej na 0 nie spowoduje zakończenia działania strażników z identyfikatorem użytkownika różnym od superużytkownika, które zarejestrowały się wcześniej (tj. gdy wartość flagi była różna niż 0).

Fakt, że zwykłe procesy użytkownika mogą działać jako strażnicy, w żaden sposób nie czyni ich bardziej uprzywilejowanymi. Jeśli użytkownik nie ma prawa do wykonania jakiejś operacji, to strażnik działający z jego uprawnieniami również nie będzie mógł jej wykonać (np. dostęp do pliku */etc/shadow*). To zapewnia, że system strażników w żaden sposób nie zmniejszy bezpieczeństwa systemu, nadając komuś większe przywileje, niż wynikałoby to z uprawnień w systemie bez strażników. Generalną zasadą jest, że strażnik może jedynie ograniczyć prawa dostępu, jakie sam posiada.

Oto pseudokod obsługi żądania przez moduł fg_user:

```
funkcja_systemowa_np_read(file) {
    if (strażnik nie jest zarejestrowany)
        return -EBADF;
    if (pid strażnika == pid klienta)
        return obsługa_wywołania_bez_strażników;
    if (strażnik chce przekazać błąd)
        return wskazany_błąd;
    skonstruuj_odpowiedni_komunikat_dla_strażnika;
    wyślij_komunikat_send_and_wait;
    if (brak odpowiedzi)
        return -ERESTART;
    if (odpowiedź nie jest poprawna)
        return -EBADF;
    skopiuj_odpowiedź_do_klienta;
    return wynik_otrzymany_od_strażnika;
};
```

4.3.3. Przypisywanie strażników do plików

Przypisanie strażnika do pliku musi mieć podstawową cechę, jaką jest trwałość. Przypisanie nie może zniknąć (lub ulec zmianie) podczas restartu systemu. W związku z tym informacja o strażniku znajduje się w metryczce pliku, a dostęp do tej informacji jest uprzywilejowany. Zaimplementowałem zatem nową funkcję systemową pozwalającą na modyfikację tych danych. Funkcja nazywa się *sys_fglink()*, a jej kod znajduje się w pliku *fs/fg/fg_main.c*. W przypadku kompilacji jądra bez wsparcia dla systemu strażników, używana jest funkcja z pliku *fs/fg/fg_main_none.c*, której zadaniem jest poinformowanie wołający program o braku implementacji funkcji systemowej (*ENOSYS*). Funkcja *sys_fglink()* pozwala na ustawienie strażnika dla pliku, jego odczyt oraz skasowanie. Typ operacji jest uzależniony od wartości drugiego parametru, którym jest nazwa strażnika. Są możliwe następujące akcje:

- jeśli podano nazwę, to ustawia się nowego strażnika dla danego pliku (konkretnie pierwsze cztery znaki),
- jeśli nazwa jest pusta, to nazwa aktualnie przypisanego strażnika jest kopiowana do użytkownika, czyli jest to operacja odczytu przypisanego strażnika,
- jeśli parametr ma wartość NULL, to kasuje się strażnika dla danego pliku.

Pierwszy parametr to nazwa pliku, na którym ma być wykonana operacja. Oto przykładowe użycie funkcji:

```
#include <stdio.h>
#include <linux/unistd.h>
/* funkcji nie ma w gnu libc - deklarujemy */
_syscall2(int, fglink, const char *, filename, char *, fg)

if (fglink("/etc/passwd", NULL))
    perror("/etc/passwd");
```

O fakcie, czy plik jest pilnowany, informuje dodatkowe pole w strukturze *inode* o nazwie *i_fg*. Zmiany dotyczą kodu VFS, a zatem rozwiązanie będzie działać ze wszystkimi systemami plików działającymi pod Linuksem. Oczywiście ta informacja (o strażniku) musi zostać zapisana na dysku. Tym razem wsparcie ze strony systemu plików jest konieczne, gdyż VFS nie wie o sposobie przechowywania danych na dysku. Obecnie stosowanie strażników jest możliwe tylko dla systemów extended2 i extended3. Wszystkie informacje dotyczące extended2 mają zastosowanie do systemu extended3. W metryczce pliku tych systemów znajduje się czterobajtowe pole, które nie jest używane do żadnych celów. Tam została umieszczona nazwa strażnika — maksymalnie cztery litery. Jeśli inne systemy plików również posiadają miejsce w metryczce, w którym można trzymać informację o strażniku, to po minimalnych zmianach są gotowe, aby wspierać system strażników. W aktualnie obsługiwanych systemach modyfikacje sprowadzają się do dwóch wierszy kodu zawierających odpowiednie przepisania na zmienną (pola *i_fg* ze struktury *inode* do struktury metryczki na dysku i odwrotnie). Można próbować przypisywać strażników do plików znajdujących się w innych systemach plików, ale zmiana nie jest stała. Znika, jeśli tylko struktura *inode* dla tego pliku zostanie ponownie załadowana z dysku. Do tego momentu plik jest pilnowany. Jest to uboczny skutek buforowania danych przez VFS.

Wraz z kodem strażników, dostarczany jest program *fglink*. Służy on do zarządzania strażnikami dla plików. Jest to opakowanie wywołania funkcji *sys_fglink()* interfejsem użytkownika. Jego źródła znajdują się w *fs/fg/user_sample/fglink.c*.

4.4. Przykłady strażników

4.4.1. Tworzenie strażników

Pisanie strażników jest bardzo proste. Aby stworzyć strażnika dla trybu jądra wystarczy dostarczyć nowe funkcje obsługi, następnie zgrupować wskaźniki do nich w strukturze *fgf_operations* i zarejestrować je pod jakąś nazwą w systemie strażników. Należy pamiętać o wypełnieniu pierwszego pola tej struktury. Jest to wskaźnik do modułu rejestrującego strukturę. Do tego celu została stworzona makrodefinicja *SET_MODOWNER(fgf_operations)*, która wykonuje opisaną czynność. Trudność w tworzeniu strażników działających jako moduły leży w implementacji poszczególnych funkcji. Może to wyglądać mniej więcej tak:

```
ssize_t _read(...)  { implementacja read(); };
struct fgf_operations _fops ={ NULL, NULL, _read, ...  };
int init_module(void) {
    SET_MODOWNER(&_fops);
    register_fgfunc(”nazwa”, &_fops);
};
int cleanup_module(void) { unregister_fgfunc(”nazwa”, &_fgops); };
```

Przykładowy kod jest załączony do tej pracy wraz z kodami źródłowymi rozwiązania.

Strażnicy realizowani przez procesy użytkownika wydają się być bardziej skomplikowani. Ale ponieważ wszyscy działają według tego samego schematu, trudności można się jedynie spodziewać przy realizacji funkcji obsługi żądań. Rejestrując strażnika należy trzymać się szczegółowo zdefiniowanego protokołu komunikacji (opisanego w punkcie 4.3.2 i w dokumentacji technicznej). Można również skopiować ten kod z jednego z przykładów, ponieważ zadziała on w każdym strażniku (nazwa strażnika i struktura *fgu_operations* są parametrami). Następnie należy już tylko odczytywać żądania od systemu, obsługiwać je i odsyłać informację zwrotną. Przykładowy kod jest załączony do tej pracy wraz z kodami źródłowymi rozwiązania.

4.4.2. Zaimplementowane przykłady strażników

W ramach pracy zaimplementowałem kilku przykładowych strażników. Są to proste przykłady, lecz mimo to dobrze pokazują możliwości mojego rozwiązania. Wśród strażników trybu jądra są między innymi:

- *fg_jiff*, który pokazuje aktualną wartość licznika *jiffies* oraz czas działania systemu (ang. *uptime*). Realizuje on tylko funkcję czytania;
- *fg_deny*, który zabrania dostępu do pilnowanego przez siebie pliku. Dotyczy albo tylko operacji zapisu, albo wszystkich form dostępu (przy parametrze *all* równym 1). Przy ładowaniu modułów do systemu można im przekazywać parametry (patrz polecenie *insmod*);
- *fg_date*, który pokazuje aktualną datę systemową (tak samo jak polecenie *date*);
- *fg_cryp* szyfrujący dane w pliku za pomocą klucza, który jest przekazywany do modułu jako parametr o nazwie *seed*. Podczas odczytu dane są deszyfrowane.

Oto krótki opis przykładowych strażników, działających w trybie użytkownika:

- *fg_date* jest odpowiednikiem strażnika z modułu o takiej samej nazwie. Służy do porównania sposobu tworzenia strażników, jak i wydajności dwóch zrealizowanych typów rozwiązań;
- *fg_cryp* tak jak poprzednio, jest taką samą (jak w module) implementacją szyfrowania danych z kluczem;

- *fg_note* notuje do dzienników systemowych (ang. *log files*) wszystkie próby otwarcia pliku, którego pilnuje. Zapisywane wiadomości mają format:
data hostname fg_note: /ścieżka/do/pliku access by uid:gid res=wynik_operacji_open()
dla przykładu:
Jul 17 21:12:59 mój_host fg_note: /etc/passwd access by 0:0 res=0

- *fg_hour* otwiera zamiast żądanego przez użytkownika pliku, inny o nazwie *nazwa_pliku.godzina*. Na przykład, jeśli aktualnie jest godzina 4:30, proces otwierający plik */etc/passwd*, otworzy tak naprawdę plik o nazwie */etc/passwd.04*. Jeśli docelowego pliku nie ma, prezentowana zawartość pliku jest pusta, a zapis jest zabroniony.

- *fg_acl* jest implementacją list kontroli dostępu ACL dla systemu plików. Ponieważ jest to realizowane w funkcji *permission()*, która operuje tylko na metryczkach (nie ma potrzeby otwierania pliku), więc definiowanie list jest kłopotliwe. Jest to spowodowane faktem, że plik reprezentowany na dysku przez metryczkę nie posiada nazwy. Zatem nie jest możliwe określenie list kontroli dostępu dla plików za pomocą nazw. Dokładniej rzecz ujmując, w funkcji *permission()* nie ma możliwości określenia nazwy pliku, na którym wykonuje się operacja. Stąd nietypowy sposób określania pliku za pomocą pary liczb: numeru urządzenia i numeru metryczki na tym urządzeniu. W ten sposób jednoznacznie określamy obiekt w systemie plików. Następnie w katalogu */etc/acl/* tworzymy pliki o nazwach konstruowanych w sposób: *numer_urządzenia.numer_metryczki*. Przykładem jest nazwa */etc/acl/70A.25* (liczby w postaci szesnastkowej). Do tego pliku zapisujemy listy dostępu. Składnia ACL jest następująca: *[u / g / a]:numer:rwx*. Wpis *g:100:101* oznacza, że grupa o numerze 100 ma prawo do czytania i wykonywania pliku, ale nie ma prawa do zapisu. Szczegóły konstruowania ACL znajdują w pliku źródłowym (na dołączonej do pracy dyskietce). Pomijając niewygodę w zarządzaniu listami dostępu, przykład pokazuje, w jak prosty sposób można uzyskać własności systemu plików, na brak których tak wiele osób narzeka. Dodając narzędzie pomagające definiować ACL można znieść trudności zarządzania listami kontroli dostępu dla plików.

4.4.3. Możliwości tworzenia innych strażników

Mechanizm strażników pozwala w prosty sposób dodawać nowe możliwości do systemów plików, których bez niego nie można w żaden sposób osiągnąć. Wkładając niewiele wysiłku, można stworzyć strażnika, który będzie prezentował w systemie pliki typu *gzip* jako rozkompresowane. Wystarczy tylko zmienić funkcje czytania i pisania na pliku tak, aby w locie kompresowały lub dekompresowały dane. Dzięki temu możemy oszczędzić dużo miejsca na dysku, szczególnie dla danych tekstowych (takich jak dzienniki systemowe). W implementacji można się spodziewać problemów z zakodowaniem funkcji *llseek()*, ponieważ nie wiadomo z góry gdzie znajduje się żądana pozycja w pliku. Pewnym rozwiązaniem tego problemu jest brutalne potraktowanie użytkowników i zabronienie wykonywania tej operacji na pliku. To zmusi wszystkich do korzystania z pliku wyłącznie w sposób strumieniowy. Jest to rozwiązanie akceptowalne, gdyż sama kompresja *gzip* jest operacją strumieniową. Innym podejściem do problemu jest zastosowanie alternatywnego algorytmu kompresji, opartego o blokowe kodowanie danych. Wtedy w szybki sposób można określić, w którym bloku pliku znajduje się docelowa pozycja.

Inny przykład to traktowanie plików *tar* jako katalogów w systemie. Ponieważ są to nieskompresowane archiwa plików, naturalnym wydaje się traktowanie ich jako katalogów. Kod strażnika byłby bardzo prosty, tak jak funkcje *tar()* i *untar()* nie są skomplikowane.

To rozwiązanie bardzo łatwo można połączyć z poprzednim uzyskując skompresowany folder plików trzymany w jednym obiekcie systemu plików.

Kolejny przykład to automatyczne wersjonowanie plików. Posiadając zainstalowany (lokalnie, bądź na zdalnej maszynie) serwer CVS[4], można stworzyć strażnika, który spowoduje, że kolejne wersje plików będą automatycznie zapisywane na serwerze CVS. Wystarczy tylko podmienić funkcję zapisu do pliku. Za każdym razem, gdy aplikacja zapisuje plik na dysk, zmieniona funkcja oprócz faktycznej operacji wysyła do serwera CVS nową wersję pliku. Oczywiście, aby była możliwość powrotu do wcześniejszej wersji, należy jeszcze dodać nowe polecenia do funkcji *ioctl()*, które będą w stanie przywrócić lokalną kopię do żądanej wersji. To również wymaga osobnego narzędzia znającego nowe polecenia funkcji *ioctl()* i pozwalającego na zarządzanie wersjami pliku. Dzięki takiemu rozwiązaniu jest możliwy dostęp do historii zmian w pliku, zaś wersja zapisana lokalnie na dysku jest zawsze najnowszą kopią. Całość jest zupełnie przezroczysta dla użytkownika (z wyjątkiem odzyskiwania poprzednich wersji). Przy tym rozwiązaniu możemy być spokojni o dane zawarte w pliku, w szczególności gdy serwer CVS znajduje się na odległej maszynie. Ani awaria systemu, ani przypadkowe nadpisanie pliku nie spowodują utraty danych.

Przykłady można mnożyć. Sądzę, że każdą operację możliwą do wykonania na pliku można zautomatyzować za pomocą tego mechanizmu, tworząc odpowiedniego strażnika.

4.5. Wydajność strażników

4.5.1. Sposób wykonania testów

Testy zostały zaprojektowane głównie z myślą o pomiarze opóźnień generowanych poprzez system strażników. Opóźnienia powstają głównie w wyniku wykonywania funkcji opakowujących wywołania strażników, wyszukiwania strażników w strukturach i komunikacji z procesami. Znając wyniki tych pomiarów będzie można odpowiedzieć na pytanie, jak bardzo pogorszyło się działanie systemu poprzez wprowadzenie strażników oraz czy powstałe opóźnienia są akceptowalne, a w efekcie czy tworzenie strażników jest uzasadnione.

Wyniki testów są podane w taktach procesora i mikrosekundach. Prędkość procesora, na którym zostały wykonane testy była równa 572,476,000Hz. Maszyna nie była obciążona dodatkowymi czynnościami. W systemie było zarejestrowanych trzech strażników.

Do testów użyłem procedury odczytującej licznik taktów procesora od momentu włączenia go (TSC). Jest to wartość długości 64 bitów dostępna dla większości architektur. Dla platformy Intela wymagany jest co najmniej procesor klasy Pentium. Sama funkcja nazywa się *get_cycles()* i jest zadeklarowana w pliku *include/asm/timex.h*. Czas wykonania (liczba cykli procesora) był zapisywany do pliku za pomocą mechanizmu dzienników systemowych (demon *klogd*). Następnie po zadanej liczbie powtórzeń operacji na pliku, dziennik systemowy był analizowany pod kątem interesujących nas danych. Odpowiednie wpisy w dzienniku były rozpoznawane po numerze metryczki, który był zapisywany razem z wynikiem.

Jeśli wyniki prezentowane w tabelkach zostały podane w formacie *liczba1/liczba2*, to bardziej miarodajna jest *liczba2*, ponieważ jest ona wartością uzyskaną po zignorowaniu dziesięciu najlepszych bądź najgorszych wyników. Jeśli wynik podano w nawiasach, to jednostką są mikrosekundy (10^{-6} sekundy).

[4]Serwer, za pomocą którego można pamiętać różne wersje plików lub nawet całych projektów. Oprócz wersjonowania oferuje również rozgałęzienia projektów i jednoczesne tworzenie ich przez wiele osób.

4.5.2. Wyniki testów

Aby oszacować opóźnienia generowane podczas otwierania pliku, w funkcji *dentry_open()* umieściłem kawałek kodu sprawdzający czas trwania wywołania funkcji otwarcia pliku specyficznej dla obsługiwanego pliku — w przypadku pilnowanego pliku tej z podmienionej struktury *file_operations* o nazwie *fg_open()*. Oto ten kawałek kodu:

```
/* plik fs/open.c, funkcja dentry_open() */
_tick = get_cycles();
if (inode->fg) { /* jeśli plik pilnowany */
    if ( ( error = fg_open(inode, f)) )
        goto cleanup_all;
} else { /* oryginalny kod */
    if (f->f_op && f->f_op->open)
        if (error = f->f_op->open(inode, f))
            goto cleanup_all;
};
_tick = _tick - get_cycles();
```

Należy tu zauważyć, że przypadek, w którym plik nie ma przypisanego strażnika można traktować tak samo jak przypadek systemu plików bez modyfikacji wprowadzających strażników. Jedyną różnicą i jedynym narzutem czasowym jest pojedyncze sprawdzenie warunku, co można zignorować (zajmuje to do dwóch cykli procesora). Inne przypadki są obsługiwane wewnątrz funkcji *fg_open()*. To ona generuje narzuty czasowe. Jej kod wygląda tak:

```
ret = -ENOSYS;
fg = znajdź_strażnika(inode->fg);
if (fg) { //strażnik jest w systemie
    if (fg->fgops->open) //implementuje open()
        ret = fg->fgops->open();
    if (ret == -ENOSYS)
        ret = oryginalne_open();
    if (ret == 0) //plik otwarty
        podmień_strukturę_f->f_op;
    return ret
};
if ((!fg_allow) && current->fsuid) return -EPERM;
return oryginalne_open();
```

Wyniki otrzymane przy teście strażników działających jako moduły są zamieszczone w tabeli 1.

	bez strażnika	bez funkcji	z pustą funkcja	strażnik nie istnieje
minimalny	83/85	280/288	290/296	227/230
maksymalny	232/114	693/338	11980/374	716/247
średni	92(0.16)	319(0.55)	319(0.55)	228(0.39)

Tab. 1: Opóźnienia funkcji *open()* dla strażników-modułów
Funkcja *open()*, nazwa absolutna, 10000 prób,
bez strażnika — brak przypisanego strażnika do pliku; narzut jednego sprawdzenia warunku w stosunku do systemu bez strażników, porównywalny z systemem bez strażników,
bez funkcji — strażnik nie implementuje funkcji *open()*,
z pustą funkcja — funkcja *open()* nie robi nic i od razu przekazuje *-ENOSYS*,
strażnik nie istnieje — w systemie nie ma takiego strażnika — wykonywana jest oryginalna funkcja.

Z tabeli 1 wynika fakt, że ładowanie modułu do pamięci przy dostępie do pliku nie ma większego znaczenia, szczególnie jeśli był on już wcześniej ładowany podczas pracy systemu i nie musi być wczytywany z dysku (kod znajduje się w buforach systemu). Można to dobrze zaobserwować ponawiając wykonanie testu i obserwując wyniki pierwszego rezultatu, który zawiera czas ładowania modułu (jeśli modułu nie było jeszcze w pamięci). Przyrost czasu wykonywania funkcji ze strażnika jest około trzykrotny. Być może to dużo, ale gdy się porówna czasy, które w obu przypadkach są poniżej jednej mikrosekundy, ten narzut czasowy wydaje się jak najbardziej akceptowalny. Jest to tak naprawdę narzut wywołania dodatkowej funkcji, wyszukania odpowiednich struktur dla aktualnego strażnika i kilku sprawdzeń warunku. Zmiana reprezentacji struktury, w której pamiętamy strażników z listy na drzewo powinna poprawić ten wynik.

	bez funkcji	z pustą funkcją	funkcja z zakładaniem sesji
minimalny	2350/2360	18800/19000	3000?/42000
maksymalny	3100/2500	74000/42300	5782000/128000
średni	2450(4.2)	21100(36.8)	47000(82)

Tab 2: Opóźnienia funkcji *open()* dla strażników-procesów użytkownika
Funkcja *open()*, nazwa absolutna, 5000 prób,
funkcja z zakładaniem sesji — strażnik również otwiera plik do własnych celów (np. aby uzyskać dostęp do danych w nim zawartych).

Z tabeli 2 wynika, że dla strażników realizowanych za pomocą procesów użytkownika narzut na otwarcie pliku jest duży. Prawdopodobnie dla zastosowań, w których liczy się czas obsługi plików, jest nawet za duży. Jednak jest to normalne następstwo wymiany danych między procesem i jądrem. Każde otwarcie pliku wymaga przesłania co najmniej dwóch komunikatów (żądanie — odpowiedź), a przesyłanie danych przez granicę jądro-użytkownik jest kosztowne. Kosztuje również to, że aby całość operacji mogła dojść do skutku, należy kilka razy przełączyć kontekst i procesy muszą być zawieszane w kolejkach oczekiwania. W tym przypadku trzeba wykonać o wiele więcej czynności niż proste wywołanie funkcji i realizacja kilku wierszy kodu (jak to jest w przypadku modułów). Tym razem do zadania zostają włączone takie części jądra jak szeregowanie procesów, obsługa pamięci, system wejścia-wyjścia i urządzeń. Angażowanie tak dużych środków jest niezbędne i sądzę, że trudno będzie znaleźć wydajniejszy sposób (jeśli w ogóle istnieje). Gdy ktoś faktycznie potrzebuje strażników w trybie użytkownika, to te opóźnienia są akceptowalne. Oczywiście z punktu widzenia użytkownika pracującego na pliku przy konsoli, te opóźnienia mogą być niezauważalne. Dlatego do zastosowań na plikach, które nie są zbyt często używane, te wyniki wydają się być do zaakceptowania. Jeśli jednak zadanie można wykonać w trybie jądra, to polecałbym zakodowanie go w module.

Dla lepszego poparcia tezy o koszcie wywołania strażnika-procesu wykonałem testy czytania *(read())* tego samego pliku, raz pilnowanego przez moduł, a raz przez proces. Sposób wykonania pomiarów był podobny jak dla funkcji *open()* — był mierzony czas wykonania funkcji ze struktury *file_operations*. Strażnik, który został użyty do testów to *fg_date*, który jako wynik funkcji *read()* generował aktualny czas systemowy (tak samo jak polecenie date).

	moduł	proces
minimalny	1400/3100	31000/34000
maksymalny	17000/5100	34000
średni	3900(6.8)	34000(59.3)

Tab 3: Porównanie opóźnień czytania strażników różnych typów
Funkcja *read()*, 5000 prób,
funkcja realizuje pobranie czasu z *sys_time()* i konwersję do ciągu znaków.

Wyniki zebrane w tabeli 3 różnią się między sobą o rząd wielkości. Jeśli do tego samego zadania użyje się modułu jądra, można dużo zyskać znacznie odciążając system. Oczywiście nie wszystko da się umieścić w modułach. Nie zapominajmy, że system nie jest wywłaszczalny, więc takie funkcje nie mogą zabierać zbyt wiele czasu systemowego.

Kolejnym wykonanym testem jest test wydajności modułu *fg_cryp*. Jest to proste szyfrowanie danych za pomocą mechanizmu XOR[5]. Każdy bajt pliku zanim zostanie przekazany użytkownikowi jest modyfikowany za pomocą operacji XOR z wybraną liczbą, generowaną na podstawie dostarczonego klucza. To samo dzieje się przy zapisie danych. Sama funkcja szyfrująca jest więc bardzo prosta i szybka. Strażnik jest napisany zarówno jako moduł systemowy, jak i program użytkownika. Program testujący odczytywał plik porcjami po 4 kilobajty. Wyniki uwzględniają tylko te wywołania funkcji *read()*, które przekazywały 4 kilobajty danych.

	bez strażnika	moduł	proces
minimalny	3090/3180	19600/20600	6300/12000
maksymalny	22000/10800	45500/37800	30mln/5.7mln(9956!)
średni	5560(9.7)	23200(40.5)	55000(96)

Tab 4: Porównanie opóźnień czytania danych ze strażników
Funkcja *read()*, 5000 prób,
liczone były tylko operacje przekazujące 4KB danych.

Wyniku testu zgrupowane w tabeli 4 dowodzą jak nieprzewidywalnie duży może być czas oczekiwania na odpowiedź w przypadku procesu (od 6 tysięcy cykli do 30 milionów — ostatnia kolumna tabeli 4). Jest to normalne następstwo mechanizmu szeregowania procesów. Proces strażnika ma taki sam priorytet, jak każdy inny proces w systemie (superużytkownik może

[5] Operacja logiczna na bitach (ang. *exclusive or*).

zwiększyć priorytet procesu, ale zwykły użytkownik nie). Natomiast fakt, że czytanie danych z pliku bez strażnika jest szybsze (pokazuje to pierwsza kolumna tebeli 4) od pozostałych przypadków nic nie wnosi do wnioskowania, ponieważ opóźnienia są generowane głównie przez samą funkcję *read()* i szyfrowanie danych. Narzut mechanizmu strażników jest taki sam jak przy funkcji *open()* (tabela 1) — jest to taki sam mechanizm wywołania funkcji. Fakt, że tym razem różnica wyników średnich (moduł i proces) jest mniejsza niż pokazana w tabeli 3, można wytłumaczyć tym, że w poprzednim teście obliczenie daty systemowej jest bardzo szybkie, więc większość czasu wykonania funkcji jest poświęcana na komunikację moduł — proces. Świadczą o tym niewiele większe czasy dla średniego wykonania funkcji *read()* przez proces, i stosunkowo wyraźnie większe dla modułu jądra (porównanie tabeli 4 z tabelą 3). W obu tabelach narzut na czytanie za pomocą strażnika-procesu jest tej samej wielkości, około 30000 taktów zegara, czyli jakieś 50 mikrosekund.

Wykonywane testy z wieloma zarejestrowanymi strażnikami, wskazują na niewielką, choć zauważalną różnicę spowodowaną wyszukiwaniem strażników na liście prostej. W zależności od tego czy strażnik znajdował się na pierwszym miejscu listy, czy też na ostatnim (około 20 strażników na liście) różnice oscylowały w okolicy 100 taktów procesora. To dowodzi, że sposób pamiętania strażników w strukturach jądra (lista prosta) ma wpływ na wydajność mechanizmu (choć nie jest ona znacząca).

4.5.3. Wnioski

Dla małych funkcji (zabierających mało czasu procesora) nie opłaca się umieszczać strażnika w przestrzeni użytkownika (bardzo duże narzuty na obsługę komunikacji z procesem strażnika). Dla dużych funkcji może być korzystne wyniesienie funkcjonalności poza jądro systemu i uzyskanie możliwości jakie daje pisanie programów dla przestrzeni użytkownika. Narzut czasowy związany z przejsciem od implementacji w trybie jądra do implementacji w trybie użytkownika dla funkcji wykonującej jedną operację logiczną (XOR) na każdym bajcie danych (strażnik fg_cryp) jest dwukrotny. Dla bardziej skomplikowanych funkcji takie porównanie wypadnie jeszcze lepiej na korzyść procesów użytkownika. Jednak licząc czasy bezwzględne, czyli prostą różnicę czasów wykonania, będzie to około 50 mikrosekund (dla testowej maszyny). Wynika to z faktu, że narzut na wywołanie funkcji z procesu-strażnika kosztuje właściwie zawsze tyle samo (od 20000 do 30000 taktów procesora).

Mimo że narzut generowany przez procedury obsługi strażników ma mniejsze znaczenie dla większych funkcji strażnika, istnieje zagrożenie, że przy bardzo czasochłonnych funkcjach, może przestać być opłacalne wynoszenie strażników poza jądro. Powodem jest to, że kod jądra wykonuje się szybciej (brak opóźnień związanych z szeregowaniem procesów). Pojawia się tu dylemat do rozstrzygnięcia: czy blokowanie systemu operacyjnego na tak długi czas (w czasie wykonywania funkcji strażnika z modułu, proces, w imieniu którego była wywołana funkcja, nie może zostać wywłaszczony do jej zakończenia), nie spowoduje ogólnego pogorszenia stabilności i funkcjonalności systemu.

Drugi ważny wniosek jest następujący: strażnicy z przestrzeni użytkownika nie powinni być stosowani do plików, które są bardzo często i długo używane. Zastosowanie strażnika szyfrującego dane dla pliku */etc/passwd* spowodowałoby zmniejszenie reaktywności systemu. Plik jest bardzo często odczytywany. Oczywiście niewielki kod obsługujący wymieniony plik w trybie jądra nie miałby takiego znaczenia. Z drugiej strony, jeśli chcemy do naszego pliku *.forward* przypiąć strażnika — nic nie stoi na przeszkodzie, aby był on napisany przez nas samych i działał w przestrzeni procesów użytkownika. Tym razem opóźnienia nie będą dla nikogo zauważalne. Taka sama sytuacja dotyczy wszystkich plików, których używa się od czasu do czasu. Nikt nie zauważy opóźnień związanych z systemem strażników jeśli będzie po prostu odczytywał, modyfikował, bądź używał pliku od czasu do czasu (np. edytował plik). Następstwem tego jest fakt, że edytując pilnowany przez strażnika fg_cryp plik, właściwie nie zauważamy różnicy w porównaniu z pracą na pliku niepilnowanym. Dowodem tego są wyniki prezentowane w tabeli 5. Są to czasy odczytania pliku o rozmiarze 10MB za pomocą różnych mechanizmów strażników. Sądzę, że wydłużenia czasu zapisu bądź odczytu pliku z 0.031 sekundy do 0.3 sekundy nikt nie odczuje w codziennej pracy. Prezentowane wyniki są podane w sekundach.

	bez strażnika	moduł	proces
bezpośrednio z dysku	1.000	1.005	1.005
z buforów dyskowych	0.031	0.106	0.325

Tab 5: Porównanie czasów odczytu pliku
czas odczytu pliku 10MB przez strażnika fg_cryp,
bezpośrednio z dysku — dane były odczytywane fizycznie z dysku,
z buforów systemowych — dane były odczytywane z pamięci operacyjnej.

Rozdział 5

Dyskusja prezentowanych projektów

5.1. Zalety i wady strażników plików w projekcie FileGuards

W tym rozdziale chciałbym poruszyć kilka kwestii, które dotyczą sposobów, jakie zostały wybrane do rozwiązania problemów postawionych przed projektem. Chciałbym wyjaśnić dlaczego wybrałem takie rozwiązania, a nie inne — być może lepsze. Tu również przedstawię zauważone braki projektu i jeśli potrafię — wskażę drogę do lepszego rozwiązania.

Zacznijmy od zasadniczej kwestii, czyli sensu wprowadzenia do systemu tak uprzywilejowanych strażników. Ich działanie jest działaniem systemu (konkretnie podsystemu plików). Swoje systemowe uprawnienia mogą wykorzytać na przykład po to, aby przekłamywać zawartości plików. Szczególnie te ładowane jako moduły są potencjalnym zagrożeniem. Mają również zupełną kontrolę nad systemem operacyjnym. Wykonują się w trybie jądra, co oznacza dostęp do struktur systemu, łącznie z prawem ich modyfikacji. Powstaje pytanie, czy można przekazać kontrolę nad systemem do kodu nie napisanego przez twórców systemu? Sądzę, że odpowiedź powinna być dla wszystkich oczywista i brzmieć "tak". Otóż moduł jest kodem uprzywilejowanym i tylko administrator systemu może go załadować do systemu. Jeśli administrator nada dla jądra prawo ładowania modułów do pamięci, to plik zawierający kod modułu musi należeć do administratora. Jeśli zatem ktoś, kto ma prawa administratora zainstaluje w systemie taki moduł, to musi się liczyć z następstwami takiego postępowania. Jeśli nie wiadomo jak działa kod zawarty w module — lepiej go nie instalować.

Istnieje także możliwość nieautoryzowanego dostępu do konta superużytkownika. Jednak, jeśli ktoś chce zepsuć system, nie musi ładować niebezpiecznego kodu z modułu — są dużo prostsze sposoby na spowodowanie strat. Jest jeszcze inna sytuacja, która może spowodować krytyczną dla systemu sytuację. Jest to problem nie tylko mojego rozwiązania, a ogólnie mechanizmu modułów. Problem może zaistnieć, jeśli moduł rejestruje strażnika, a następnie usuwa go z pamięci systemu operacyjnego. Jest to możliwe, jeśli aktualnie nie obsługuje żadnego otwartego pliku. Ten problem istnieje dla wszystkich usług realizowanych za pomocą modułów systemowych. Biorąc pod uwagę, że system operacyjny jest kodem wiarygodnym (moduł to część systemu), a procesy nie, tak na prawdę nie ma tu żadnego niebezpieczeństwa, ponad to wprowadzone przez system Linux.

Problem wygląda inaczej w przypadku strażników realizowanych przez procesy. Jeśli proces strażnika otrzyma zbyt duże uprawnienia, może coś zepsuć w systemie, wbrew oczekiwaniom administratora i innych użytkowników. W związku z tym zostały nałożone pewne ograniczenia. Po pierwsze program wykonuje się z prawami użytkownika, który go uruchomił. Oczywiście do wykorzystania jest mechanizm *SUID*[1], dzięki któremu użytkownik może

[1] Jeśli plik ma ustawiony bit SUID, to system przy uruchamianiu ustawia efektywnego właściciela procesu

uruchomić program z prawami administratora, a superużytkownik z prawami dowolnego użytkownika[2]. Strażnik uruchomiony przez użytkownika działa z jego prawami od początku do końca. Fakt, że jest on strażnikiem nie daje mu żadnych dodatkowych praw, z wyjątkiem możliwości realizowania funkcji systemowych na plikach, które chroni. W szczególności, jeśli strażnik jest przypisany do pliku, do którego nie posiada żadnych praw, nie będzie on w stanie korzystać z tego pliku. W ten sposób mamy pewność, że strażnik nie uzyska zbyt dużych uprawnień do pliku. Procesom korzystającym z jego usług może zaoferować tylko podzbiór swoich uprawnień (dotyczy to tylko realizowanych funkcji). Jądro chroni się przed złymi zamiarami strażnika poprzez restrykcyjny protokół wymiany danych z procesami strażników. Sposób komunikacji wymusza na strażnikach dokładne przygotowywanie danych i trzymanie się określonego schematu komunikacji. Jądro nie odczyta ani mniej, ani więcej danych od strażnika niż oczekuje. To sprawia, że jedyne co strażnik może zrobić źle, to podać fałszywe informacje. Te jednak, albo zostaną zignorowane przez jądro (nie będą pasować do żadnego wysłanego żądania), albo informacja ta zostanie przekazana do procesu wykonującego na pliku operację. Dane odbierane przez jądro od strażnika nigdy nie są wykorzystywane dla działania systemu. Nie mogą one w żaden sposób sterować systemem ani być traktowane jako kod w uprzywilejowanym środowisku.

Kolejnym pytaniem jest, dlaczego strażnicy rejestrują się sami w jądrze, a nie są wołani wtedy, gdy istnieje taka potrzeba. To wyeliminowałoby sytuację, w której użytkownik uruchamia strażnika, podszywającego się pod jakiegoś innego. Ponieważ lista strażników jest globalna dla systemu, użytkownik może wykorzystać fakt, że właściwy strażnik nie jest jeszcze zarejestrowany i podstawić na jego miejsce innego z taką samą nazwą. Oczywiście nie daje mu to większych praw w systemie, ale skutecznie blokuje prawdziwego strażnika. Dodatkowo, jeśli strażnik ma prawo odczytu pilnowanego przez siebie pliku, może modyfikować na własny sposób funkcje czytania oryginalnie zaprojektowanego strażnika. Uważam, że wprowadzanie mechanizmu uruchamiania strażników przez system jest niepotrzebną komplikacją implementacji. Takie rozwiązanie wymaga pisania dodatkowego kodu w jądrze, uruchamiającego procesy użytkowników. Nie jest to niestety proste, ponieważ w jądrze systemu istnieje jedynie wsparcie dla uruchamiania wątków systemowych, działających z prawami systemu operacyjnego. Poza tym problem można rozwiązać jeszcze na dwa inne sposoby. Po pierwsze można wymagać, aby wszystkie pliki strażników znajdowały się w jakimś katalogu systemowym, gdzie prawo zapisu ma tylko administrator (podobnie jak w Watchdogs). Wtedy w module *fg_user* można włączyć sprawdzanie ścieżki do pliku wykonywalnego, aktualnie rejestrującego się strażnika. Jeśli nie pasuje ona do wzorca, to można odmawiać rejestracji. Drugi sposób, to zabronienie rejestracji strażników innemu użytkownikowi niż administrator. Jest to zachowanie domyślne. Aby je zmienić, należy ustawić odpowiednio flagę w pliku */proc/fs/fgu_non_root*. W ten sposób administratorzy systemów mogą zabezpieczyć się przed użytkownikami próbującymi nadużyć swych uprawnień. Jest to podobne rozwiązanie do zastosowanego z modułami, w którym superużytkownik ustala, które moduły mogą działać w systemie. Jeśli okaże się to konieczne, to można w przyszłości zaimplementować uruchamianie strażników z jądra systemu lub za pomocą pomocniczego programu z przestrzeni użytkownika (zarządcy strażników).

Zupełnie innym problemem jest zagarnięcie do swoich potrzeb kodu błędu *ENOSYS*. Problem został już poruszony w punkcie 4.3.1. Dodam, że nie chciałem w żaden istotny sposób ingerować w struktury jądra. Wydaje mi się, że dodanie nowego kodu błędu, spowodowałoby

na właściciela pliku.

[2]Ponieważ administrator ma dostęp do uprzywilejowanej funkcji *setuid()*, może również ją wykorzystać do tego celu. To samo robi program *su*.

w przyszłości konflikt z jakimś innym, nowo zdefiniowanym w oficjalnym jądrze kodem błędu. Nie jest to sytuacja krytyczna, ale chciałem jej uniknąć. Dlatego zdecydowałem się na wykorzystanie istniejącej już wartości. Jak opisałem wcześniej, ponieważ jest to kod informujący o braku implementacji operacji, nie wprowadza on żadnych zagrożeń dla działania systemu.

Sytuacja wygląda podobnie z wykorzystaniem zarezerwowanego pola w strukturze metryczki systemu plików extended2 (na potrzebę zapisania nazwy strażnika). Istnieje możliwość, że w przyszłości ktoś inny (lub nawet twórcy Linuksa) będzie chciał je wykorzystać. Ponieważ bez używania go nie jest możliwa implementacja rozwiązania, jedyne co mogłem zrobić, to zmienić nazwę pola na inną. Jeśli ktoś będzie chciał użyć w kodzie starej nazwy pola, system nie skompiluje się, ponieważ nie ma takiego symbolu. Jeśli ktoś napisze łatę, używającą tego pola, kompilacja zakończy się błędem z tych samych powodów. Jeśli w końcu, jakaś łata również będzie zmieniała nazwę tego pola, nie zaaplikuje się. Jest to niestety tylko połowiczne rozwiązanie, ale nie widać dobrego pomysłu na inne.

Można rozważać możliwość implementacji przypisań strażników do plików w osobnym, specjalnie dedykowanym do tego pliku, tak jak zostało to zrobione w przypadku implementacji extended3 i pliku księgowania. Plik z dziennikiem transakcyjnym jest normalnym plikiem w systemie. Podobne rozwiązanie jest stosowane dla ograniczeń dyskowych *quota*. Ale takie rozwiązanie wymaga albo zakodowania nowego systemu plików (tak jak extended3 i MonA), albo wprowadzenia poważnych zmian w aktualnym systemie plików.

Kolejnym problemem, jaki chciałbym tu opisać, jest podmiana struktury *file_operations* dla otwartego pliku. Główne pytanie brzmi, dlaczego po prostu nie podmieniam wskaźnika w strukturze *file* na strukturę, którą dostarcza strażnik, zamiast na tę wypełnioną funkcjami-opakowaniami. Przede wszystkim jest ona niezgodna z oryginalną. Ale w pierwszych projektach implementacji strażnik dostarczał po prostu strukturę *file_operations* oraz osobno funkcję *permission*. Zastanawiałem się, czy takie rozwiązanie jest możliwe. Strażnik obsługuje tylko wybrane funkcje, zatem nie obsługiwane w ogóle nie byłyby wołane. W miejscu na wskaźniki do nich są wartości NULL. Ale to można naprawić. Brakujące wartości można skopiować z oryginalnej struktury. Obligowałoby to jednak strażnika do wołania oryginalnych funkcji obsługi we własnym zakresie, co byłoby niezgodne z założeniami projektu. Nie miałby zastosowania mechanizm oparty na kodzie błędu *ENOSYS*. Takie podejście jest niedopuszczalne, ponieważ oznaczałoby to, że strażnik może pilnować pliki w ramach tylko jednego systemu plików. Przecież skopiowane wartości nie będą w stanie obsłużyć innych systemów niż ten, z którego zostały pobrane. Tworzenie osobnej kopii dla każdej instancji otwartego pliku z oczywistych powodów nie jest akceptowalne. Można ewentualnie w miejsce nieobsługiwanych funkcji kopiować odnośniki do naszych funkcji-opakowań. Prowadzi to jednak do innego problemu, który dyskwalifikuje to rozwiązanie. Rozpatrzmy sytuację, w której strażnik ma kilka otwartych plików i nagle, z jakichś powodów wyrejestrowuje strażnika i znika z pamięci. Instancje otwartych plików mają teraz odwołania do struktury, której nie ma w pamięci! To na pewno doprowadzi do błędnego działania systemu. Oczywiście można by w takiej sytuacji przeszukać systemową tablicę otwartych plików i przywrócić oryginalne wskaźniki (lub pamiętać w dodatkowej strukturze otwarte przez strażnika pliki). Jest to jednak rozwiązanie skomplikowane, czasochłonne i co chyba najważniejsze, nie można by kontrolować dostępu do plików za pomocą flagi *fg_allow*, co jest wymogiem stawianym w projekcie. Dlatego uważam, że narzut związany z dodatkowym wywołaniem funkcji jest akceptowalny w sytuacji, gdy w prosty sposób możemy osiągnąć o wiele lepszą funkcjonalność, prostotę i elastyczność. Ostatecznie kod zawarty w opakowaniach wywołań i tak gdzieś musiałby być zawarty, zapewne więc byłby powielany w każdym module.

Problematyczny jest też przypadek, w którym proces strażnika otwiera pilnowany przez siebie samego plik. System powinien jakoś reagować, aby uniknąć zapętlenia odwołań, a w efekcie unieruchomienia procesu-klienta. Aktualnie taka sytuacja jest wykrywana poprzez porównanie identyfikatora procesu żądającego dostępu i procesu rejestrującego strażnika. Nie jest to jednak najlepsza metoda, ponieważ proces obsługujący wywołania systemowe może być dzieckiem tego, który zarejestrował strażnika. Taka sytuacja nie jest wykrywana, co prowadzi wprost do zapętlenia odwołań. Jedną z metod rozwiązania jest założenie, że strażnik będzie dawał dostęp do pliku w standardowy sposób, jeśli żądanie pochodzi od niego samego. Niestety, kod użytkownika nie powinien być obdarzany takim zaufaniem przez system. Innym (i chyba jedynym) rozwiązaniem jest przeszukanie systemowej tablicy otwartych plików, w poszukiwaniu procesów mających otwarte to samo urządzenie komunikacyjne, co strażnik. Jeśli wśród nich jest proces klienta, to oznacza, że jest on strażnikiem. Test jest poprawny, ponieważ tylko procesy spokrewnione mogą mieć otwarte to samo urządzenie. Nie ma możliwości, że otworzy je jeszcze inny proces. Niestety, takie przeszukiwanie przy każdym wywołaniu strażnika jest bardzo kosztowne. Skomplikowałoby kod oraz wydłużyło czas realizacji żądania. Pomysł trzymania tych informacji w pamięci podręcznej też nie jest wystarczający, ponieważ w najgorszym przypadku strażnik może uruchamiać oddzielny proces potomny do obsługi każdego z pojedynczych żądań. Wynika stąd pewne ograniczenie. Jeśli strażnik ma zamiar korzystać z pilnowanych przez siebie plików, odbieranie od jądra komunikatów z żądaniami powinno się odbywać w tym samym procesie, który zarejestrował strażnika (otworzył główne urządzenie komunikacyjne). Odpowiedzi mogą pochodzić od innego procesu lub wątku.

Jest jeszcze jedno rozwiązanie, które uważam za dobre, ale ogranicza ono skalowalność projektu. Chodzi tu o sposób komunikacji ze strażnikami poprzez nowe urządzenie o nazwie */dev/fguX* (tak nazwałem urządzenia komunikacji ze strażnikami; X jest kolejnym numerem urządzenia logicznego). Sama komunikacja jest wygodna i sprawdza się bardzo dobrze. Jednak wprowadza ograniczenie na liczbę jednocześnie zarejestrowanych strażników. Ze względów bezpieczeństwa (również prostoty) każdy strażnik ma przydzielony osobny kanał komunikacji. Ponieważ używamy jednego urządzenia, możemy użyć tylko tylu strażników, ile jest możliwych urządzeń logicznych. Niestety, standard obsługi urządzeń w Linuksie zakłada, że numer drugorzędny może mieć maksymalną wartość 255. Oznacza to tylko 255 jednocześnie zarejestrowanych strażników trybu użytkownika. W najnowszych wersjach Linuksa jest dostępny dodatkowy system plików o nazwie *dev_fs*. Jest to wirtualny system plików, który udostępnia pliki urządzeń znanych systemowi. Znosi on ograniczenie na maksymalną liczbę urządzeń logicznych, ale jest on w wersji eksperymentalnej i nie jest powszechnie używany (moje rozwiązanie współpracuje z tymi wersjami systemu, ale nie wspiera *dev_fs*). Dlatego w przyszłości należy zastanowić się nad inną drogą komunikacji, która będzie bardziej ogólna i z powodzeniem zastąpi ten mechanizm.

5.2. Porównanie z innymi rozwiązaniami

5.2.1. Projekt Watchdogs

Zaprezentowane w tej pracy rozwiązanie jest bardzo podobne do opracowanego przez Bershada i Pinkertona. Wynika to stąd, że jest wzorowane na ich pracy. Ale tylko schemat i założenia są podobne, szczegóły oraz sama implementacja różnią się. Przede wszystkim w moim rozwiązaniu nie ma ogólnosystemowego procesu zarządzającego strażnikami. W projekcie Watchdogs zakłada się, że wszystkie pliki strażników znajdują się w odpowiednim katalogu i proces zarządcy uruchamia je w razie potrzeby. W moim rozwiązaniu nie jest istotne gdzie znajduje się program strażnika. Może zostać uruchomiony w każdej chwili, przez jakiegokolwiek użytkownika, z dowolnego miejsca systemu plików. Nie ma także wymogu, aby nazwa pliku zgadzała się z nazwą programu strażnika (pliku z kodem programu).

W obu rozwiązaniach istnieje ograniczenie na długość nazwy strażnika, ale jest to uzależnione tylko od systemu plików, z jakim działa implementacja. W rzeczywistości w moim rozwiązaniu nie ma z góry narzuconego ograniczenia do czterech znaków, ponieważ nazwa trzymana na poziomie VFS może mieć dowolną długość[3]. Dopiero fakt, że użyłem systemu extended2 narzucił ograniczenie do czterech znaków. Dla innych systemów plików mogą to być zupełnie inne wartości i mogą razem istnieć we wspólnym systemie strażników. Dla systemu 4.3BSD to ograniczenie wynosiło dwadzieścia znaków.

Trudno określić, który sposób uruchamiania strażników jest lepszy. Mój z pewnością jest o wiele prostszy w realizacji, wymaga za to uruchomienia strażnika na nowo przy każdym starcie systemu. Z drugiej strony narzuca mniej restrykcji związanych ze sposobem ich instalowania. Administrator nie musi kopiować ich do nowego katalogu o nowym systemowym znaczeniu. W zasadzie obydwa rozwiązania są akceptowalne, choć moje wnosi pewne dodatkowe niebezpieczeństwo związane z faktem, że dwaj użytkownicy mogą próbować tak samo nazwać swych strażników (opisane w punkcie 5.1).

Również w inny sposób rozwiązana została sprawa komunikacji między jądrem i procesami. Bershad i Pinkerton zdecydowali o utworzeniu nowego typu pliku dedykowanego specjalnie dla komunikacji ze strażnikami. Uważam, że jest to niepotrzebne, a wykorzystanie urządzeń jest lepszym pomysłem, nie wymagającym pisania dodatkowego kodu. Innymi możliwościami są *netlink*, komunikacja za pomocą *ioctl* lub nawet poprzez system plików *procfs*. Dla procesu strażnika nie ma różnicy czy otwiera plik, urządzenie systemowe, czy może gniazdo sieciowe typu *UNIX* (*netlink*) — zawsze działa w ten sam sposób. Sądzę, że jedyną akceptowalną alternatywą jest wymiana danych poprzez pamięć procesu, za pomocą funkcji *ioctl*. Sam system wymiany danych za pomocą komunikatów wysyłanych przez jakiś kanał, jest podobny w obu rozwiązaniach.

Rozwiązanie z projektu Watchdogs opiera się w głównej mierze na wykorzystaniu systemowej tablicy otwartych plików, ale kod obsługi znajduje się również w implementacji systemu plików. Istnieje zagrożenie, że przy zmianie systemu plików dość duże porcje kodu trzeba będzie napisać na nowo. Moje rozwiązanie całkowicie korzysta z VFS, który na pewno zostanie w Linuksie podstawą dla wszystkich innych systemów plików. Nawet jeśli w przyszłości wszyscy zrezygnują z extended2, to niewiele pracy trzeba będzie włożyć na przeniesienie strażników na inny system plików. Modyfikacje w kodzie extended2 zamykają się w dwóch wierszach kodu. W związku z tym uważam, że przedstawione podejście jest właściwe. Wprowadzenie systemu strażników do innych systemów plików (vfat, extended3, reiserfs) jest prostym zadaniem. Oczywiście aby realizacja tego zadania była możliwa, trzeba znaleźć miejsce na nazwę bądź identyfikator strażnika w metryczce pliku przechowywanej na dysku.

Wyniki wydajności Watchdogs prezentowane w tabelach 6 i 7 (tabele pochodzą z pracy [1]) wykazują około 10% opóźnienia dla funkcji *read()* przekazującej sterowanie do oryginalnych funkcji systemowych, ponad 300% opóźnienia dla funkcji otwarcia pliku, gdy strażnik jest uruchomiony, oraz 3600% w przypadku konieczności uruchomienia strażnika. Te wyniki zaskakują, szczególnie 10% przy czytaniu danych (u mnie jest to 2500%), jako że jest to opóźnienie dodane wyłącznie przez komunikację strażników z jądrem systemu. Jeśli ów wynik jest prawdziwy, to jest on lepszy, nawet od przypadku strażnika załadowanego jako moduł systemowy (gdzie opóźnienie jest rzędu 300%)! Można to tłumaczyć faktem, że testy były przeprowadzane na diametralnie różnych maszynach co ma wpływ na wyniki pomiarów. Na przykład czytanie danych z dużo wolniejszego dysku pochłaniało większość czasu operacji, więc czas komunikacji z procesami przestaje być istotny (pokazuje to porównanie z tabeli 5).

[3] Co prawda aktualna implementacja narzuca takie ograniczenie, ale bez żadnych zmian w projekcie jak i sposobie używania strażników można to zmienić.

Niestety nie dotarłem do specyfikacji maszyny, na której były wykonane testy implementacji Watchdogs, jak i sposobu ich wykonania.

	brak strażnika	z pustą funkcją	funkcja czytająca dane
1KB danych (read-ahead)	5.064	5.572	5.348
1KB danych (no read-ahead)	6.872	10.272	6.872

Tab 6: Opóźnienia funkcji *read()* dla strażników w systemie Watchdogs
wyniki w milisekundach
brak strażnika — brak przypisanego strażnika do pliku,
z pustą funkcja — funkcja *open()* nie robi nic i od razu przekazuje sterowanie do oryginalnych procedur,
funkcja czytająca dane — strażnik sam odczytuje dane z pliku, następnie przekazuje je do procesu klienta.

	brak strażnika	z pustą funkcją	z pustą funkcją + start strażnika
absolutna nazwa	3.070	9.591	108.0
relatywna nazwa	1.398	21.356	117.0

Tab 7: Opóźnienia funkcji *open()* dla strażników w systemie Watchdogs
wyniki w milisekundach
brak strażnika — brak przypisanego strażnika do pliku,
z pustą funkcja — funkcja *open()* nie robi nic i od razu przekazuje sterowanie do oryginalnych procedur,
start strażnika — strażnik nie działa w systemie, trzeba uruchomić jego proces.

Chciałbym zaznaczyć, że zastosowanie strażników działających w trybie jądra całkowicie redukuje wspomniane narzuty czasowe. Oczywiście nie można tego argumentu użyć w porównaniu implementacji ponieważ projekt Watchdogs nie zawierał założenia o możliwości działania strażników jako modułów.

5.2.2. System MonAFS

Porównanie z projektem MonA może sprawić więcej problemów. W MonA zastosowano inne podejście do zagadnienia niż w projekcie FileGuards. Jest to po prostu nowy system plików bazujący na extended2. Implementuje wszystkie zawarte w extended2 funkcje tak, aby osiągnąć zamierzony cel, czyli transformacje danych w locie.

Przede wszystkim MonA implementuje bardzo mały zestaw operacji na plikach, co uniemożliwia całkowitą zamianę semantyki dostępu. Można kontrolować działanie funkcji *read()* i *write()*, ale inne funkcje pozostają niezmienione. Pozwala to tylko na zmianę zawartości pliku i na nic więcej. Również ogólność rozwiązania nie jest osiągnięta. Reiserfs z czasem wyprze stary już system extended2, a wtedy cały kod MonA będzie bezużyteczny, ponieważ jest mocno uzależniony od kodu extended2. Aby dla reiserfs uzyskać podobne właściwości, trzeba będzie od nowa napisać cały kod MonA, uwzględniając wszystkie różnice między systemami plików. Na przykład reiserfs wprowadza zupełnie nowe rozmieszczenie danych na dysku, co może stanowić poważny problem w implementacji MonA na jego podstawie.

Tak samo jak w systemie Watchdogs, w MonA strażnicy są ładowani do pamięci, gdy istnieje taka potrzeba. Już wcześniej pisałem zaletach i wadach takiego podejścia. Do zalet MonA trzeba zaliczyć możliwość realizacji transformacji w zwykłych skryptach powłoki systemowej (ang. shell script). Jest to możliwe również w przypadku strażników, ale nie sądzę aby wiele osób potrafiło napisać skrypt, który potrafiłby wykonywać zadania strażnika. W przypadku MonA jest to łatwe, ponieważ podprogramy transformacji zawsze operują na strumieniach danych, które nie mają struktury wewnętrznej. W strażnikach dane przesyłane są za pomocą komunikatów.

Zaletą rozwiązania MonA jest możliwość zastosowania na pliku wielu transformacji na raz. Aby to osiągnąć dla strażników plików, trzeba napisać nowego strażnika wołającego innych, bądź realizującego wymagane funkcje.

System MonA nie zawiera mechanizmów ochrony oryginalnych danych z pliku (tych nieprzetworzonych przez transformację). Nowe możliwości możemy osiągnąć tylko tworząc dowiązanie symboliczne do surowego pliku. Jeśli zatem zależy nam na ukryciu surowych danych z pliku, nie uda się. W takim przypadku nie ma sensu tworzenie transformacji polegającej na zmianie zawartości pliku lub ukrywaniu części informacji. To wyklucza właściwie wszelkie zastosowania dla celów administracyjnych, takich jak ukrywanie przed użytkownikami hasła z pliku */etc/passwd*. Wyjątkiem są transformacje wymagające dodatkowych danych, takie jak szyfrowanie z hasłem.

Dodatkowo MonA nie jest do końca przezroczysty dla procesów. Można tylko udawać przezroczystość w ten sposób, że oryginalny plik schowamy pod inna nazwą i utworzymy w jego miejsce dowiązanie transformacyjne. Jest to jednak niewygodne rozwiązanie i jak wcześniej zauważyłem, dostęp do ukrytego pliku jest nadal możliwy.

Jeśli chodzi o dość istotną kwestię komunikacji między procesami i jądrem, to twórcy MonA zdecydowali się na wymianę danych poprzez pamięć procesu. Polega ona na tym, że proces wstawia do swojej pamięci komunikat, a potem za pomocą wywołania *ioctl()* przekazuje do systemu jego adres. Gdy wykonuje funkcję systemową w celu pobrania komunikatu, przekazuje wskaźnik do zarezerwowanej na wiadomość pamięci. Uważam, że jest to wystarczająco proste rozwiązanie, a przy tym tak samo efektywne, jak korzystanie z podsystemu urządzeń. Jedyne wady, to trochę bardziej skomplikowana implementacja i brak zdefiniowanego standardu wymiany danych, zatem w jądrze musi się znaleźć dodatkowy kod, specyficzny dla tego mechanizmu. W przypadku urządzeń jest to standard istniejący od powstania systemu.

Na koniec przedstawiam porównanie wydajności rozwiązań. Testy MonA wykonano na maszynie z procesorem Pentium 200MHz z 64MB pamięci RAM. System nie był obciążony dodatkowymi czynnościami i działał na Linuksie 2.0.27. Wyniki są prezentowane w tabelach 8 i 9. Zostały wykonane przez twórców systemu MonA. Więcej informacji można znaleźć pod adresem http://www.cse.nd.edu/~ssr/papers/linc99/node8.html. Już samo zastosowanie systemu plików MonA zamiast extended2 powoduje zwiększenie czasu, jaki jest potrzebny na obsługę żądania, o około 13%. Jest to istotne ponieważ rozwiązanie ze strażnikami nie wnosi żadnych opóźnień przy obsłudze niepilnowanych plików. Następnie przy otwieraniu pliku z transformacją identyczności, opóźnienie w stosunku do extended2 sięga już 300%. Opóźnienie jest porównywalne z tym, jakie produkuje system strażników dla zdefiniowanej pustej funkcji *open()*. Wprowadzanie dodatkowych transformacji powoduje opóźnienia odpowiednio 480% dla pięciu identyczności i 1210% dla dwudziestu pięciu. Podobne wyniki uzyskano dla wywołań funkcji czytania i pisania. Samo użycie MonA powoduje wydłużenie czasów realizacji funkcji o około 10% w porównaniu z extended2. Dodatkowy narzut związany z obsługą transformacji to kolejne 5%. Zmniejsza się on wraz ze wzrostem liczby dowiązanych transformacji. W przypadku strażników brak obsługi funkcji czytania jest porównywalny z wynikami funkcji *open()*, czyli około 250%. Jest to duża różnica w stosunku do MonA, spowodowana potrzebą wyszukiwania strażnika w systemowych strukturach. Sądzę że rozwiązaniem tego problemu będzie znalezienie sposobu na zmiejszenie użycia funkcji opakowujących właściwe wywołania strażników. Ta różnica zmniejsza się, jeśli porównanie wykonuje się z inną transformacją niż identyczność. Wtedy większość narzutu czasowego jest powodowana przez funkcję obsługi żądania, co powoduje, że narzuty związane z obsługą systemów (MonA, strażnicy) stają się mało istotne. Dla transformacji eksportującej funkcje do przestrzeni użytkownika, czasy są ponad sześćdziesiąt razy dłuższe niż dla obsługi funkcji rezydującej w jądrze systemu. Takie same wyniki osiągnąłem w swojej implementacji.

rozmiar pliku	1 KB	4 KB	64 KB
extended2	8.5	11	14
bez transformacji	11	12.5	14
1 transformacja	8.6	11.1	14
5 transformacji	8.7	11.2	14.5
25 transformacji	10.1	12.5	15

Tab 8: Opóźnienia funkcji *read()* dla strażników w systemie MonA
wyniki w mikrosekundach,
używana transformacja to transformacja pusta (identyczność).

extended2	bez transformacji	1 transformacja	5 transformacji	25 transformacji
15.9	17.6	38.8	71.9	182.2

Tab 9: Opóźnienia funkcji *open()* dla strażników w systemie MonA
wyniki w mikrosekundach,
używana transformacja to transformacja pusta (identyczność).

5.3. Wnioski

Na pierwszy plan wysuwa się spostrzeżenie, że rozwiązanie nie jest zbyt wydajne. Jedne części wprowadzają taki sam (lub nawet mniejszy) narzut czasowy co MonA, ale testy pustych funkcji czytania i pisania dają gorsze wyniki od systemu transformacji. W związku z tym należy zastanowić się nad możliwością zrezygnowania z pewnych założeń projektu, przerzucając część pracy na strażników. Chodzi o to, aby strażnik dostarczał pełną strukturę *file_operations*, którą wpisywałoby się przy funkcji *open* do instancji otwarcia pliku (do struktury *file*). To wyrównałoby wyniki poprzez eliminację potrzeby wyszukiwania strażnika przy odwołaniu do niego. W kodzie VFS od razu wołano by funkcję strażnika, omijając zaimplementowany system funkcji-opakowań. Jest to możliwe do osiągnięcia, ale kosztem zmniejszenia ogólności i elastyczności rozwiązania (patrz rozważania w punkcie 5.1).

Inną kwestią są nieefektywne struktury danych, zarówno w kodzie strażników, jak i module dostępu dla programów użytkownika. Listy, które są tam użyte do utrzymywania strażników są strukturami, w których wyszukiwanie kosztuje czas liniowy. Wymiana ich na tablice z funkcją mieszającą lub po prostu drzewa, na pewno poprawiłaby wydajność wyszukiwania, a co za tym idzie narzuty na wykonywanie funkcji byłyby mniejsze. Im więcej strażników zarejestrowanych w systemie, tym więcej czasu można zaoszczędzić dzięki takiemu podejściu.

Dopracowania wymaga również sposób uruchamiania strażników. Problemem jest możliwość podszywania się pod nazwę cudzego strażnika. Ta niejednoznaczność nie ma znaczenia, jeśli strażników może uruchamiać tylko jeden użytkownik (administrator). Jednak w ogólnym przypadku problem istnieje. Należy zatem w przyszłości opracować lepszy system rejestracji strażników trybu użytkownika. Może okazać się, że rozwiązanie z zarządcą strażników jest optymalne, przy uwzględnieniu potrzeby bezpieczeństwa i jednoznaczności. Ponieważ to zarządca uruchamia strażników, jest jednoznacznie określone, który program zostanie uruchomiony.

Można się również zastanowić nad innymi sposobami realizacji przyporządkowania strażników do plików. Może okazać się konieczne zastosowanie dodatkowego pliku, o specjalnym znaczeniu systemowym, gdzie będą zapisane powiązania strażników z plikami. Rozwiązanie z wpisywaniem nazwy do metryczki jest najlepsze i najszybsze, ale niestety, systemy plików nie były projektowane z myślą o wspieraniu takich rozwiązań jak strażnicy. Efekt jest taki, że dla extended2 mamy tylko cztery znaki, a dla reiserfs dwa. Wpisywanie do metryczki tylko identyfikatora strażnika rozwiązałoby ten problem. Zaradziłoby to również na wspomniany w poprzednim akapicie problem niejednoznaczności przy rejestracji strażnika. Wtedy, zanim można by użyć strażnika, należałoby dodatkowo zarejestrować go w specjalnym pliku systemowym, zawierającym listę pełnych nazw strażników wraz z ich identyfikatorami. W tym kroku można by wychwycić rejestrację drugiego strażnika o takiej samej nazwie. Tacy strażnicy byliby odrzucani.

Kolejny nasuwający się wniosek, to dobre umiejscowienie implementacji strażników. VFS jest idealnym miejscem dla takiego rozwiązania. Dzięki temu strażnicy mogą współpracować z każdym obsługiwanym przez Linux systemem plików. Jest to duża zaleta w porównaniu z ograniczeniami jakie posiadają alternatywne rozwiązania.

Również komunikacja poprzez strukturę urządzeń jest ogólna, wydajna i prosta w obsłudze. System dodatkowych dedykowanych rozwiązań jest słabszy oraz bardziej skomplikowany. System komunikacji wykorzystujący pamięć procesów jest tak samo wydajny jak urządzenia, ale nie jest udokumentowany i brak w nim dobrze zdefiniowanego protokołu. Jeśli jednak wziąć pod uwagę użyteczność, to jest ona tak samo dobra jak w prezentowanym tutaj rozwiązaniu.

Rozdział 6

Podsumowanie

Celem tej pracy było umożliwienie użytkownikom możliwości zmiany zachowań plików oraz semantyki dostępu do nich. Rozwiązanie miało być proste i efektywne, dzięki umożliwieniu personalizacji funkcji obsługi plików. Uważam, że zadanie postawione przed projektem zostało wykonane (czego dowodem jest załączona do pracy implementacja). Zadanie zrealizowano na dwa sposoby: personalizowana obsługa plików może się odbywać w trybie systemowym bądź w trybie użytkownika. Strażnicy mają swoje konkretnie określone miejsce w systemie operacyjnym, a implementacja działa poprawnie oraz z akceptowalną wydajnością. Wprowadzenie do projektu niewielkich modyfikacji zmniejszających trochę elastyczność systemu, powinno zaowocować poprawą wydajności. Sądzę, że wprowadzenie tych zmian, jeśli pożądane, jest zadaniem na przyszłość dla mnie lub moich następców.

Projekt FileGuards pokazuje, w jak łatwy sposób można uzyskać funkcjonalność, której nie ma w żadnym istniejącym systemie plików. Podobny cel realizują opisane projekty MonA i Watchdogs. Projektując strażników starałem się zapożyczyć z nich to co najlepsze, unikając jednocześnie popełnionych tam błędów. Rezultat jest zadowalający, choć nie jest doskonały. Niedociągnięcia pojawiają się głównie w implementacji, ale pewne z nich zostały narzucone przez zbyt wymagający projekt. Praca pokazuje jednak, jaką drogą należy podążać w przyszłości, aby uzyskać nieograniczone możliwości systemów plików bez modyfikowania ich implementacji.

Choć rozwiązanie nie było testowane na obciążonych systemach produkcyjnych, nie powinno być z nim większych problemów. Wprowadzone przez projekt obciążenie zasobów systemowych jest na minimalnym poziomie. Główne obciążenie dla systemu generują funkcje strażników, które obsługują wywołania systemowe. Ponieważ na serwerach produkcyjnych szybkość systemu jest bardzo ważna, a strażnicy realizują postawione im zadania szybciej niż przetwarzanie potokowe z pomocą łączy nienazwanych, strażnicy mają szansę zaspokojenia potrzeby wydajności. Oczywiście system strażników musi być stabilny i bezpieczny. O wydajności przekonują wykonane testy, o stabilności można się przekonać używając systemu (sam używam go już kilka miesięcy), a o bezpieczeństwie dobrze zdefiniowane prawa strażników w systemie operacyjnym.

W porównaniu z rozwiązaniami technicznymi sprzed dziesięciu lat, komputery są obecnie dziesiątki razy szybsze. W dodatku ten rozwój postępuje coraz dynamiczniej. Przy tak wydajnych maszynach może okazać się opłacalne zrezygnowanie ze standardowych systemów plików na rzecz plików realizowanych przez strażników. Jest to rozwiązanie dające duże możliwości, uwalniające z ograniczeń, jakie narzuca statyczny widok plików w systemie. Dlatego sądzę, że ta praca powinna zainteresować ludzi projektujących nowe systemy plików. Być może taki mechanizm wbudowany bezpośrednio do systemu plików miałby szansę zwiększyć użyteczność systemu plików, przy minimalnej stracie efektywności. To zupełnie zmieniłoby myślenie programistów i administratorów na temat systemów plików, które wreszcie przestałyby być pasywnymi kontenerami na surowe dane. Takie podejście do zagadnienia mogłoby rozwiązać od zawsze istniejący problem, że w każdym systemie plików brakuje jakiejś właściwości lub mechanizmu. Ze strażnikami plików każdy administrator systemu lub nawet użytkownik może sam dołożyć funkcjonalność, której mu brakuje.

Bibliografia

[1] Brian N. Bershad i C. Brian Pinkerton, "Watchdogs — Extending the UNIX File System". Raport techniczny. Zimowa konferencja USENIX 1988, strony 267-275.

[2] George I. Davida i Brian J. Matt, "UNIX GUARDIANS: Active User Intervention in Data Protection", Uniwersytet Wisconsin-Milwaukee, WI53201, luty 1989.

[3] Galen C. Hunt, "Creating User-Mode Device Drivers with a Proxy". Konferencja USENIX Windows NT Workshop, Seattle, Washington 1997.
http://www.usenix.org/publications/library/proceedings/
/usenix-nt97/fill_papers/hunt/hunt_html/hunt/html

[4] Richard Kendall i Vincent W. Freech, "The Modify-on-Access File System: An Extensible Linux File System". http://www.cs.nd.edu/~ssr/projects/mona.

[5] Alessandro Rubini i Jonathan Corbet, "Linux Device Drivers, 2nd edition", rozdział 3 "Char drivers", O'Reilly, czerwiec 2001.
http://www.oreilly.com/catalog/linuxdrive2/chapter/book

[6] Abraham Silberschatz i Peter B. Galvin, "Podstawy systemów operacyjnych", wydanie trzecie zmienione, rozszerzone, rozdziały 10, 11, 21.7, 22.7, WNT 1999.

[7] Uresh Vahalia, "Jądro systemu UNIX. Nowe horyzonty", rozdział 11 "Zaawansowane systemy plików", Warszawa, WNT 2001.

[8] Linux Documentation Project — praca zbiorowa, http://www.linuxdoc.org.

Dodatek A

Dokumentacja techniczna

A.1. Struktury danych i funkcje

Zmodyfikowane struktury danych:

- struktura ***inode*** w pliku *include/linux/fs.h* otrzymała dodatkowe pole o nazwie ***i_fg***. Jest to nazwa strażnika pilnującego pliku, do którego należy ta metryczka. Są to maksymalnie cztery litery odwzorowane na typ *integer*. Jeśli pole ma wartość zero, to plik nie jest pilnowany,
- struktura ***file*** w tym samym pliku otrzymała dodatkowe pole o nazwie ***f_op_fg***. Jest to wskaźnik do oryginalnej struktury *file* dla danej instancji otwarcia pliku. Pole jest używane wyłącznie dla plików pilnowanych i zawiera wartość przypisaną przez system do pola *f_op* tej samej struktury, jeszcze przed otwarciem pliku,
- w pliku *include/linux/ext2_fs.h* zmiana nazwy jednego z zarezerwowanych pól struktury *ext2_inode* z ***i_reserved1*** na ***i_ext2fg***. Jest to pole, w którym faktycznie jest trzymana nazwa strażnika na dysku. Dzięki temu zabiegowi inne modyfikacje jądra, wykorzystujące to pole, nie będą się kompilować. Tak samo zmodyfikowano plik *include/linux/ext3_fs.h*.

Zmodyfikowane funkcje:

- w ***start_kernel()*** w pliku *init/main.c*, dodano wywołanie funkcji *fg_init*() inicjującej dane systemu strażników,
- do funkcji ***ext2_read_inode()*** w pliku *fs/ext2/inode.c* dodano przepisywanie nazwy strażnika ze struktury wczytanej z dysku do struktury *inode* w VFS, a w funkcji ***ext2_update_inode()*** odwrotnie. Tak samo zmodyfikowano plik *fs/ext3/inode.c*,
- funkcja ***clean_inode()*** w pliku *fs/inode.c* dodatkowo czyści pole *i_fg* ze struktury *inode*,
- w funkcji ***permission()*** w pliku *fs/namei.c* na początku sprawdza się, czy metryczka, której dotyczy operacja, ma ustawione pole *i_fg*. Jeśli tak, oznacza to, że plik jest pilnowany i jest wywoływana funkcja *fg_permission()*,
- oraz ***dentry_open()*** w pliku *fs/open.c*. Tak jak w poprzedniej funkcji tylko w przypadku pilnowanego pliku następuje wywołanie *fg_open()*.

Nowe struktury z pliku *include/linux/fg.h:*

- ***fgf_ operations*** jest strukturą zawierającą wskaźniki do wszystkich funkcji możliwych do przedefiniowania przez strażnika. Strażnik dostarcza tę strukturę rejestrując się w systemie. Wartość *NULL* oznacza, że strażnik nie obsługuje funkcji i system ma wykonać operację w normalny sposób (tak jakby nie było strażnika). Lista pól to: *owner, llseek, read, write, readdir, poll, ioctl, mmap, open, flush, release, fsync, fasync, lock, permission.* Pole *owner* jest wskaźnikiem na moduł rejestrujący strażnika, reszta jest oczywista,

- ***fg_ fo_ list*** jest listą strażników zarejestrowanych w systemie. Zawiera nazwę strażnika, liczbę otwartych plików oraz wyżej opisaną strukturę z obsługiwanymi funkcjami,

- ***fgu_ operations*** jest strukturą podobną do *fgf_ operations*. Jednak zamiast wskaźników zawiera taką samą liczbę wartości całkowitych (bez pola *owner*). Jest to struktura dostarczana przez proces, który w ten sposób informuje o sposobie obsługi funkcji. Dla każdego odpowiednika z *fgf_ operations* pole może mieć wartość (w nawiasie wartość liczbowa):
  - FGU_MYSELF (0) oznacza, że strażnik obsługuje funkcję. Wywołania tej funkcji realizuje strażnik,
  - FGU_NORMAL (-ENOSYS) oznacza, że strażnik nie implementuje funkcji. Funkcja jest wykonywana w standardowy sposób,
  - FGU_REJECT (-EPERM) oznacza, że wykonywanie funkcji jest niedozwolone,
  - dowolna wartość mniejsza od zera oznacza kod błędu jaki zostanie przekazany do wywołującego procesu,
  - FGU_WDATA (1) oznacza, że strażnik chce, aby jądro wykonało operacje i dostarczyło wyników do weryfikacji oraz ewentualnej zmiany. W tej chwili jest to możliwe tylko dla funkcji *open()*, dla innych zaś jest ignorowane,

- ***fgu_ mesg*** jest strukturą definiującą nagłówek wiadomości przesyłanej między procesem i jądrem. Kolejne pola mają następujące znaczenia i możliwe wartości:
  - *magic* musi zawsze być ustawione na wartość *FGU_MAGIC*,
  - *cmd* jest kodem operacji jakiej dotyczy komunikat. Dozwolone kody to:
  FGU_REG, FGU_SENDMIN, FGU_LLSEEK, FGU_READ, FGU_WRITE, FGU_READDIR, FGU_POLL, FGU_IOCTL, FGU_MMAP, FGU_OPEN, FGU_FLUSH, FGU_RELEASE, FGU_FSYNC, FGU_FASYNC, FGU_LOCK, FGU_PERMISSION,
  - *no* jest numerem komunikatu. Jest to losowa wartość, którą proces strażnika musi znać odpowiadając na żądanie i zawrzeć w odpowiedzi. Numer wskazuje również, na które żądanie ta wiadomość jest odpowiedzią,
  - *sid* jest identyfikatorem sesji, do której należy komunikat. Dzięki temu strażnik wie, której instancji otwarcia pliku dotyczy komunikat. Jest to wartość wskaźnika do struktury *file* (lub *inode* w przypadku *permission*),
  - *pid* jest identyfikatorem procesu zgłaszającego żądanie. Za pomocą tego numeru i systemu *procfs* strażnik może uzyskać wiele informacji o procesie klienckim,
  - *len* jest rozmiarem dodatkowych danych w komunikacie.

  Wszystkie pola z wyjątkiem *len* muszą mieć taką samą wartość w komunikacie odpowiedzi,

- ***fgu_ mesx*** jest strukturą zawierającą komunikaty strażnika wraz z dodatkowymi danymi potrzebnymi dla modułu obsługi procesów. Zawiera przede wszystkim wskaźnik na strukturę *fgu_mesg*, ale również wskaźnik na dodatkowe dane (te o rozmiarze *len*), pole *next* implementujące listę wiadomości, a pole *wq* kolejkę oczekiwania. W tej kolejce będzie oczekiwał proces kliencki, aż do otrzymania od strażnika odpowiedzi na komunikat,

- ***fgu_data*** jest strukturą zawierającą komplet informacji potrzebnych do obsługi strażnika-procesu. Są to:
  - identyfikator procesu, który zarejestrował strażnika,
  - flaga informująca czy urządzenie pomocnicze do komunikacji ze strażnikiem zostało już otwarte,
  - nazwa strażnika,
  - jego tablica operacji *fgu_operations*,
  - spinlock do wyłącznego dostępu do kolejek komunikatów,
  - cztery wskaźniki definiujące dwie kolejki komunikatów oczekujących na wysłanie do strażnika oraz czekających na odpowiedź od niego — po dwa wskaźniki na listę, odpowiednio głowa i ogon (są wskaźnikami na strukturę *fgu_mesx*),
  - kolejka oczekiwania, na której strażnik będzie oczekiwał na pojawienie się danych w kolejce komunikatów do odebrania.

Format części komunikatu dla danych spoza nagłówka w zależności od typu:

- *FGU_LLSEEK* wysyłane do procesu *(int nowa_pozycja, int względem_pozycji)*, oczekiwane w odpowiedzi *(int nowa_pozycja)*,
- *FGU_READ* wysyłane do procesu *(int ilość_danych_do_odczytu, int aktualna_pozycja_w_pliku)*, oczekiwane w odpowiedzi *(int ilość_danych, int nowa_pozycja, char dane_odczytane[ilość_danych])*,
- *FGU_WRITE* wysyłane do procesu *(int ilość_danych, int pozycja, char dane_do_zapisu[ilość_danych])*, oczekiwane w odpowiedzi *(int ilość_zapisanych, int nowa_pozycja)*,
- *FGU_READDIR* — nie obsługiwane,
- *FGU_POLL* — nie obsługiwane,
- *FGU_IOCTL* wysyłane do procesu *(int polecenie, int argument)*, oczekiwane w odpowiedzi *(int wynik)*,
- *FGU_MMAP* — nie obsługiwane,
- *FGU_FLUSH* brak wysyłanych danych, oczekiwane w odpowiedzi *(int wynik)*,
- *FGU_FSYNC* wysyłane do procesu *(int dane_synchronizacji)*, oczekiwane w odpowiedzi *(int wynik)*,
- *FGU_FASYNC* wysyłane do procesu *(int na_czym)*, oczekiwane w odpowiedzi *(int wynik)*,
- *FGU_LOCK* — nie obsługiwane,
- *FGU_OPEN* wysyłane do procesu *(int wynik_open, char[] nazwa_otwieranego_pliku)*, oczekiwane w odpowiedzi *(int wynik)*; nazwa jest absolutną nazwą pliku, którego dotyczy operacja; pierwszy parametr jest ustawiany na wynik oryginalnego wywołania *open()* (jeśli proces prosi o to) albo na wartość '////',

- *FGU_RELEASE* brak wysyłanych danych, oczekiwane w odpowiedzi *(int wynik)*; *wynik* jest ignorowany,
- *FGU_PERMISSION* wysyłane do procesu *(int maska_żądanego_dostępu)*, oczekiwane w odpowiedzi *(int wynik)*,
- *FGU_SENDMIN* wysyłane do procesu *(int numer_urządzenia)*, oczekiwane w odpowiedzi *(struct fgu_operations fgops, char *nazwa_strażnika)*,
- *FGU_REG* — jest odpowiedzią na komunikat typu *FGU_SENDMIN*.

Dane w pliku *fs/fg/fg_main.c*:

- *struct fgf_operations* ***fgf_null*** jest strukturą używaną tylko do zainicjowania listy strażników w systemie,
- *struct fg_fo_list* ***fg_list*** jest listą zarejestrowanych w systemie strażników. Zawiera co najmniej strażnika zdefiniowanego w *fgf_null*. Do blokady współbieżnego dostępu służy *spinlock* o nazwie ***fg_list_lock***,
- *int* ***failed_probe[]*** zawiera nazwy strażników, których próba załadowania nie powiodła się. Jest to tablica z funkcją mieszającą. Używana tylko wtedy, gdy w jądrze jest włączona możliwość automatycznego ładowania modułów,
- *int* ***fg_allow*** zawiera informację determinującą zachowanie systemu w przypadku dostępu do pilnowanego pliku, dla którego nie ma zarejestrowanego strażnika. Jeśli wartość flagi jest różna od zera, to dostęp jest realizowany w normalny sposób, w przeciwnym przypadku tylko superużytkownik ma dostęp do pliku,
- *struct file_operations* ***fg_default_fops*** jest strukturą, która jest przypisywana pilnowanym instancjom plików jako ich *file_operations* (pole *f_op* struktury *file*). Zawiera wskaźniki do funkcji opakowujących wywołania strażników.

Funkcje w pliku *fs/fg/fg_main.c*:

- *struct fg_fo_list* ****lookup_fgfunc****(int name, int lock)* jest używana do odszukania strażnika o podanej nazwie. Jeśli taki istnieje, to jest przekazywany wskaźnik do odpowiedniego elementu list *fg_list*, jeśli nie — wartość *NULL*. Dodatkowy parametr *lock* oznacza potrzebę zwiększenia licznika otwartych plików dla znalezionego strażnika (jeśli licznik był równy zero) i zablokowaniu w pamięci modułu. W funkcji, jeśli strażnika nie ma w systemie, próbuje się załadować moduł ze strażnikiem. Jeśli po tej operacji modułu nadal nie ma w systemie, to zapamiętuje się nazwę w tablicy *failed_probe*, co pozwala na oszczędzenie czasu przy następnej próbie,
- *int* ***register_fgfunc****(char *name, struct fgf_operations *fgops)* oraz ***unregister_fgfunc****(...)* są eksportowanymi symbolami jądra, za pomocą których strażnicy rejestrują się (i wyrejestrowywują) w systemie. W pierwszej z nich, po sprawdzeniu kilku warunków, tworzy się nowy wpis na liście *fg_list*, w drugiej oprócz likwidacji tego wpisu, sprawdza się czy moduł jest zablokowany w pamięci *(licznik otwarć większy od zera)* i ewentualnie zwalnia blokadę,
- funkcje *int* ***proc_fg_allow_read/write****(...)* służą do realizacji odczytu i zapisu do pliku */proc/fs/fg_allow*. Plik zawiera wartość flagi *fg_allow*,

- funkcja *int* ***proc_fg_stat_read****(...)* służy do generowania statystyk o strażnikach i umieszczania ich w pliku */proc/fs/fg_stat*,

- *void* ***fg_init****(void)* jest funkcją inicjującą, wołaną podczas startu systemu operacyjnego. Tu odbywa się rejestracja plików w *procfs*,

- funkcje *int* ***fg_*** **(...)* są opakowaniami wywołań funkcji ze strażników. Wszystkie z wyjątkiem *fg_open()* i *fg_release()* mają ten sam schemat. W dwóch wyróżnionych dodatkowo dba się o modyfikację licznika otwartych plików i blokowanie strażnika w pamięci. Funkcje te są wołane przez system VFS w przypadku dostępu do pilnowanego pliku. W funkcjach sprawdza się jak powinno potoczyć się wykonanie funkcji systemowej w danym przypadku i wykonywane są odpowiednie akcje,

- *int* ***sys_fglink****(const char *filename, char *fg)* jest funkcją systemową, umożliwiającą ustawianie, zmianę i kasowanie strażników dla plików. Jeśli *fg* ma wartość NULL, to przypisanie strażnika do pliku *filename* jest kasowane, jeśli jest niepustym ciągiem znaków, to jest ustawiany nowy strażnik, jeśli jest pustym ciągiem znaków, to aktualna nazwa jest kopiowana do użytkownika.

Dane w pliku *fs/fg/fg_user.c*:

- *struct fgu_data* ***fgu_list****[FGU_MAX+1]* jest tablicą strażników. Liczba obsługiwanych strażników jest ograniczona. Istnieje ścisłe odwzorowanie między pozycją w tablicy a numerem urządzenie, przez które odbywa się komunikacja z tym strażnikiem. Pozycja zero jest nieużywana. Do uzyskania wyłącznego dostępu do tablicy służy *spinlock* o nazwie ***fgu_data_lock***,

- *int* ***fgu_non_root*** jest flagą wskazującą, czy procesy zwykłych użytkowników mogą rejestrować strażników,

- *int* ***fgu_timeout*** jest wartością w sekundach oznaczającą czas, jaki może upłynąć między żądaniem procesu do strażnika, a przybyciem odpowiedzi. Jeśli ten czas zostanie przekroczony, to system zakłada, że strażnik nie wykonał operacji.

- *struct file_operations* ***_fgops*** jest strukturą zawierającą wskaźniki do funkcji obsługujących urządzenia komunikacji z procesami */dev/fguX*,

- *struct fgf_operations* ***fgu_fgops*** jest strukturą używaną w procesie rejestracji strażników w jądrze. Jest ona wypełniona wskaźnikami do funkcji-opakowań zdefiniowanych w tym samym pliku,

- *struct fgf_operations **___fgu_fgops_p***, jest pomocniczym wskaźnikiem na strukturę *fgu_fgops*.

Funkcje w pliku *fs/fg/fg_user.c*:

- funkcje *int* ***proc_*** **(...)* są odpowiedzialne za prezentację zmiennych *fgu_non_root* oraz *fgu_timeout* w plikach, odpowiednio */proc/fs/fgu_non_root* oraz */proc/fs/fgu_timeout*,

- *void* ***fgu_init****(void)* jest funkcją inicjującą struktury danych. Jest wołana po załadowaniu modułu do pamięci,

- *int* ***_fgu_mesx_put****(struct fgu_data *md, struct fgu_mesx *mx, int s)* służy do wstawiania komunikatu *mx* do kolejki komunikatów strażnika *md*. Parametr *s* może przyjmować wartości *FGU_TOSEND* i *FGU_TORECV*, odpowiednio wskazując kolejkę,

- *struct fgu_mesx ** ***_fgu_mesx_get****(struct fgu_data *md, int s)* służy do pobierania komunikatu z kolejki oznaczonej przez parametry o takim samym znaczeniu jak poprzednio. Komunikat jest usuwany z kolejki,

- *struct fgu_mesx ** ***_fgu_mesx_cmdnosid****(int minor, int cmd, int no, int sid)* służy do pobierania z kolejki *FGU_TORECV* od strażnika z numerem *minor* komunikatu o typie *cmd*, numerze *no* i należącego do sesji *sid*. Komunikat jest usuwany z kolejki.

- *struct fgu_mesx ** ***fgu_mesx_new****(int size)* służy do budowania nowego nagłówka komunikatu oraz rezerwowania pamięci na dane dodatkowe o rozmiarze *size*. Wynikiem jest odpowiednio wypełniona struktura *fgu_mesx*,

- *void* ***fgu_mesx_del****(struct fgu_mesx *mx)* służy do usuwania komunikatu *mx* i zwolnienia zarezerwowanej pamięci. Jeśli kolejka oczekiwania zawarta w strukturze wiadomości jest aktualnie używana, to komunikat jest oznaczany jako zły, a proces czekający w niej jest budzony. W takim przypadku nie cała pamięć jest zwalniana. Obudzony proces ponownie wywoła tę funkcję,

- *int* ***_reserve_minor****(void)* służy do wyszukiwania pierwszego wolnego wpis w *fgu_list*. Jest on rezerwowany dla aktualnego procesu,

- *int* ***_open_minor****(int minor)* służy do oznaczenia, że strażnik *fgu_list[minor]* otworzył pomocnicze urządzenie komunikacji. Strażnik musi mieć wcześniej zarezerwowany ten numer dla siebie,

- funkcja *void* ***_clear_minor****(int minor)* jest wołana podczas wyrejestrowywania strażnika. Dane dotyczące strażnika są czyszczone, a komunikaty z kolejek są kasowane,

- w funkcji *void* ***_try_clear_pid****(void)* jest wołana *_clear_minor()* dla aktualnego procesu, chyba że proces otworzył już zarezerwowane dla niego pomocnicze urządzenie komunikacji,

- *struct fgu_data ** ***_get_pid_data****(pid_t pid)* służy do wyszukiwania strażnika, którego identyfikator procesu jest równy *pid*,

- *struct fgu_data ** ***_get_name_data****(int name)* służy do wyszukiwania strażnika o nazwie *name*,

- *int* ***_get_pid_minor****(pid_t pid)* służy do wyszukiwania numeru *minor* (numer urządzenia używanego do komunikacji) strażnika, którego identyfikator procesu jest równy *pid*.

- *int* ***_open****(...)* jest funkcją realizującą otwieranie urządzeń komunikacji */dev/fguX*. Jeśli jest otwierane urządzenie główne, to dla procesu jest rezerwowany kanał obsługi (*_reserve_minor*), jeśli otwieranie dotyczy urządzenia pobocznego, to próba otwarcia udaje się tylko wtedy, gdy proces miał wcześniej zarezerwowane to urządzenie,

- *int* ***_close****(...)* jest funkcją realizującą zamykanie urządzeń komunikacji. Jeśli proces zamyka urządzenie główne, to jest on wyrejestrowywany ze struktury strażników, chyba że ma już otwarte pomocnicze urządzenie komunikacji (*_try_clear_pid*). Jeśli zamykane jest urządzenie poboczne, to strażnik jest wyrejestrowywany (*_clear_minor*),

- *int **_read**(...)* jest funkcją realizującą odczyt danych z urządzeń komunikacji. Jeśli jest to urządzenie główne, to zarezerwowany *minor* jest przesyłany do procesu. W przeciwnym przypadku, proces jest usypiany aż do momentu, gdy komunikat w kolejce *FGU_TOSEND* jest gotowy. Wtedy jest on przesyłany do procesu i przesuwany do kolejki *FGU_TORECV*,

- *int **_write**(...)* jest funkcją realizującą zapis danych do urządzeń komunikacji. Nie jest dozwolony zapis do urządzenia głównego. Odbierany od procesu komunikat musi być dobrze skonstruowany i komunikat z tym samym numerem musi istnieć w kolejce *FGU_TORECV*. Jeśli dane zgadzają się, to proces czekający na ukończenie tego zadania jest budzony. Ta funkcja realizuje również przypadek rejestracji strażnika,

- *char *__get_full_path__(struct file *f)* służy do budowania bezwzględnej nazwy pliku, na który wskazuje parametr *f*,

- *int **send_and_wait**(struct fgu_data *md, struct fgu_mesx *mx)* jest funkcją, w której wysyła się żądanie zawarte w komunikacie *mx* do strażnika *md*. Komunikat jest wkładany do kolejki *FGU_TOSEND* strażnika, strażnik jest budzony i zaczyna oczekiwać na odpowiedź bądź wyczerpanie się czasu oczekiwania. W przypadku otrzymania odpowiedzi, następuje wyjście z funkcji. W przypadku wyczerpania czasu oczekiwania lub otrzymania komunikatu oznaczonego jako zły, komunikat jest kasowany z kolejki oraz zostaje przekazany błąd do procesu wywołującego funkcję,

- funkcje *int **_fgu_***()* są opakowaniami na wywołania strażników. To one są wołane przez jądro jako funkcje strażników. Wszystkie mają taki sam schemat. W funkcji sprawdza się w jaki sposób dany strażnik obsługuje tę funkcję. Jeśli istnieje taka potrzeba, to jest budowany komunikat żądania, który zostaje wysyłany do strażnika za pomocą funkcji *send_and_wait()*. Po otrzymaniu odpowiedzi jest ona przekazywana do jądra, a komunikat jest kasowany. Jeśli odpowiedź jest znana przed wysłaniem komunikatu, to udziela się jej z pominięciem strażnika. Jest tu wykrywana sytuacja, w której proces strażnika woła funkcję na pilnowanym przez siebie pliku. Wtedy przyznawany jest bezpośredni dostęp do pliku,

- *int **init_module**(void)* oraz *void **cleanup_module**(void)* są wołane przez system odpowiednio przy rejestracji modułu w jądrze i opuszczaniu systemu.

A.2. Przykłady strażników

Przykłady strażników zarówno dla trybu jądra, jak i użytkownika, są załączone wraz kodem źródłowym systemu strażników. Można je również otrzymać od autora pracy dostępnego pod adresem *rafal@wijata.com*. Zawarte w nich komentarze (w języku angielskim) powinny pomóc w zrozumieniu działania strażników.

Dodatek B

Zawartość dołączonej dyskietki

Na dołączonym do pracy nośniku znajdują się kody źródłowe systemu strażników, opisanego w tej pracy. Mają one postać łaty na jądro systemu Linux 2.4. Są również załączone kompletne, załatane i skomentowane źródła Linuksa. Najnowsze wersje są dostępne poprzez stronę internetową pod adresem *http://www.wijata.com/fileguard*, bądź u autora pracy pod adresem poczty elektronicznej *rafal@wijata.com*.

Wszelkie pytania i sugestie proszę kierować pod wyżej wymieniony adres.