Wykład jedenasty

Wirtualny system plików

Oprócz alokatora plastrowego, twórcy jądra Linuksa zapożyczyli z systemów operacyjnych firmy Sun wirtualny system plików (*ang. Virtual File System – VFS*). Jest to warstwa pośrednicząca między rzeczywistym systemem plików, a resztą jądra. Dzięki niej Linux może obsługiwać różne, czasami diametralnie różniące się systemy plików. To rozwiązanie jest również ciekawe pod innym względem – kod wirtualnego systemu plików ma większość cech charakterystycznych dla techniki programowania obiektowego, mimo że nie został napisany w języku obiektowym. Wirtualny system plików pozwala na unifikację rzeczywistych systemów plików. Oznacza to, że procesy użytkownika obsługują pliki, niezależnie od tego, w jakim systemie plików są one zapisane, zawsze za pomocą takich wywołań systemowych jak *open(), read(), write(), close()* itd. Warstwa VFS „przekłada" te wywołania, na operacje charakterystyczne dla danego systemu plików. Powyższy opis można podsumować stwierdzeniem, że wirtualny system plików tworzy wspólny model (abstrakcję) systemu plików, pozwalający na reprezentację cech i operacji jakie są możliwe w rzeczywistych systemach plików.
Konstrukcja owego wspólnego modelu plików jest ściśle związana z macierzystym systemem plików oryginalnego systemu Unix. Podstawowymi elementami w takim systemie były: pliki, katalogi, i-węzły (*ang. i-node = index node*) oraz punkty montowania. System plików jest strukturą przechowującą dane, w której obowiązuje pewna hierarchia. W systemach uniksowych systemy plików montowane są w określonym punkcie wspólnego drzewa katalogowego[1] - tworzą przestrzeń nazw wspólną dla całego systemu, choć w Linuksie, od wersji 2.4 każdy z procesów może mieć swoją własną, określoną przestrzeń nazw. Pliki są uporządkowanymi ciągami bajtów[2]. Z każdym z plików jest związana identyfikująca go nazwa. Na plikach można wykonywać takie operacje jak: otwarcie, zamknięcie, zapis, odczyt. Katalogi są plikami, które przechowują informacje o innych plikach. Mogą one posiadać inne, zagnieżdżone katalogi nazywane podkatalogami. Kolejne nazwy zagnieżdżonych katalogów tworzą ścieżki dostępu. Każdy z elementów takiej ścieżki tworzy wpis katalogowy (*ang. dentry = directory entry*). Linux traktuje katalogi jak zwykłe pliki, w związku z tym można na nich przeprowadzić część tych samych operacji, co na plikach. Informacje o pliku (np.: data modyfikacji, rozmiar, czyli metadane pliku) w systemach uniksowych gromadzone są także w osobnych blokach na nośniku, zwanych i-węzłami. Informacje sterujące i kontrolne związane z całym systemem plików (metadane systemu plików) przechowywane są w bloku głównym, który jest nazywany superblokiem. Niektóre systemy plików nie są zgodne z przedstawionym wyżej modelem, mimo to możliwe jest ich użycie w systemie Linux. Jest to „zasługą" VFS, który przedstawia wszystkie niezbędne elementy tych systemów, tak aby pasowały one do ogólnego modelu.
Wirtualny system plików jest oparty na modelu obiektowym, mimo że jest zrealizowany od początku do końca w języku C, który takiego modelu bezpośrednio nie wspiera. Obiekty VFS są po prostu zmiennymi, których typ określają struktury, które są z kolei odpowiednikami klas[3]. Każda z takich struktur zawiera wskaźnik do struktury wskaźników na funkcje realizujące określone operacje na elementach użytkowanego systemu plików (odpowiedniki metod). W VFS istnieją cztery typy tych obiektów: obiekty bloku głównego reprezentujące zamontowane systemy plików, obiekty i-węzła reprezentujące pojedynczy plik, obiekty wpisu katalogowego reprezentujące pojedynczy wpis katalogowy oraz obiekty pliku skojarzone z otwartymi przez proces plikami. Każdy z obiektów tych typów zawiera charakterystyczne dla jego typu obiekty operacji: *super_operations* zawiera metody właściwe dla danego systemu plików, *inode_operations* – metody dotyczące konkretnego pliku, *dentry_operations* – metody właściwe dla wpisu katalogowego i w końcu obiekty plikowe grupują metody do obsługi otwartych plików. Część tych metod może być „dziedziczona" z grupy funkcji rodzajowych, które implementują wersje tych metod wspólne dla wszystkich systemów plików.
Każdy zamontowany system plików posiada swój własny obiekt bloku głównego (superbloku), który przechowuje wszystkie niezbędne informacje dotyczące tego systemu plików. Zazwyczaj jego zawartość całkowicie lub w dużej mierze pokrywa się z zawartością odpowiedniego bloku na dysku twardym lub innym nośniku danych. Istnieją jednak pewne systemy plików, które nie mają swojej fizycznej implementacji np.: sysfs lub procfs. Te systemy plików istnieją wyłącznie jako dane w pamięci operacyjnej. Typ obiektów superbloku określony jest przez strukturę *struct super_block*. Zawiera ona między innymi takie pola, jak identyfikator urządzenia na którym znajduje się opisywany przez obiekt system plików, maksymalny rozmiar plików, typ systemu plików, liczba aktywnych odwołań, itd. Obiekt bloku głównego jest równocześnie tworzony i inicjalizowany przez funkcję *alloc_super()*. Najważniejszym polem tego obiektu jest pole *s_op* wskazujące tablicę operacji na obiekcie bloku głównego, która jest reprezentowana przez typ *struct super_operations*. Każdy element tej tablicy jest wskaźnikiem na funkcję wywoływaną wtedy, gdy jądro chce wykonać jakieś operacje na systemie plików, np. zmniejszenie liczby odwołań do obiektu bloku głównego odbywa się za pomocą instrukcji *sb->s_op->put_super(sb)*. Przekazanie wskaźnika na strukturę superbloku do funkcji *put_super()* jest konieczne, gdyż w języku C nie ma wskaźnika *this*. Do metod obiektu operacji superbloku należą między innymi: *alloc_inode()*, która tworzy i inicjalizuje obiekt i-węzła, *destroy_inode()* - zwalnia obiekt i-węzła, *read_inode()* - służy do odczytania z nośnika bloku zawierającego i-węzeł, *dirty_inode()* - funkcja wywoływana przez VFS po modyfikacji obiektu i-węzła, *write_inode()* - służy do zapisu i-węzła na nośnik, *drop_inode()* - funkcja wywoływana przez VFS w momencie zwolnienia ostatniego odwołania do i-węzła, *evict_inode()* - funkcja usuwa i-węzeł z nośnika, *put_super()* - funkcja wywoływana przy odmontowywaniu systemu plików w celu zmniejszenia liczby odwołań do obiektu superbloku, jeśli wartość tej liczby osiągnie zero, to funkcja usunie obiekt, *sync_fs()* - funkcja służąca do uaktualnienia metadanych systemu plików, *statfs()* - funkcja pozwala uzyskać informacje na temat systemu plików, *remount_fs()* - służy do ponownego montowania systemu plików, *umount_begin()* – funkcja służy do przerwania operacji montowania, jest wykorzystywana przez sieciowe systemy plików, takie jak np. NFS. Nie wszystkie te operacje muszą być zaimplementowane w poszczególnych systemach plików. Jeśli nie są, to wskaźniki na odpowiadające im funkcje mają wartość NULL.
Obiekty i-węzła przechowują wszystkie informacje niezbędne do przeprowadzenia operacji na plikach i katalogach, z którymi są związane. W przypadku niektórych nieuniksowych systemów plików te informacje są pobierane wprost z pliku lub z innego miejsca na nośniku, gdzie są przechowywane i umieszczane w obiekcie. Część z tych informacji nie jest obecna w niektórych systemach plików. W takich wypadkach pola im odpowiadające wypełniane są dowolnymi wartościami. Obiekty i-węzłów mogą być związane nie tylko z fizycznymi plikami, ale również z plikami specjalnymi, np. plikami urządzeń lub kolejek FIFO. Pojedynczy obiekt i-węzła zawiera między innymi takie pola, jak np.: identyfikator właściciela pliku, rzeczywisty numer urządzenia na którym plik jest zapisany, prawa dostępu, numer i-węzła, rozmiar pliku. Zawiera on również pole *i_fop* wskazujące na obiekt operacji, które mogą zostać przeprowadzone na otwartych plikach. Oprócz tego pola istnieje inne pole wskaźnikowe o nazwie *i_op* wskazujące na strukturę zawierającą wskaźniki na funkcje realizujące operacje na obiekcie i-węzła. Do tych operacji należą między innymi: *create()* - stworzenie nowego i-węzła związanego z zadanym obiektem wpisu katalogowego, *lookup()* - przeszukuje katalog w celu znalezienia i-węzła związanego z określonym wpisem katalogowym, *link()* - utworzenie dowiązania twardego, *unlink()* - usunięcie dowiązania, *symlink()* - tworzy dowiązanie symboliczne, *mkdir()* - tworzy plik będący katalogiem, *rmdir()* - usuwa katalog, *mknod()* - funkcja tworzy plik specjalny (reprezentujący np. urządzenie), *rename()* - zmienia nazwę pliku, *readlink()* - odczytuje określoną część pełnej ścieżki związanej z dowiązaniem symbolicznym, *follow_link()* - konwertuje dowiązanie do i-węzła docelowego, *permission()* - obsługuje prawa dostępu w niektórych systemach plików, *setattr()* - inicjuje powiadomienie o zmianie zawartości i-węzła, *getattr()* - powiadamia, że i-węzeł powinien zostać ponownie odczytany z nośnika, *setxattr()* - ustawia atrybuty rozszerzone, *getxattr()* - odczytuje wartość atrybutu rozszerzonego, *listxattr()* - kopiuje listę wszystkich rozszerzonych atrybutów do bufora, *removexattr()* - funkcja usuwa wskazany atrybut rozszerzony z listy.
Obiekty wpisu katalogowego związane są z każdą nazwą katalogu pojawiającą się w ścieżce dostępu. Tak więc np. dla ścieżki */usr/java* stworzone zostaną trzy takie obiekty: dla katalogu głównego „/", dla katalogu „usr" i dla katalogu „java". Służą one do realizowania operacji, które są charakterystyczne dla katalogów, a nie dla plików, jak, np. przeszukiwanie katalogów. Jeśli ścieżkę kończy zwykły plik, to jest on również reprezentowany przez obiekt wpisu katalogowego. Do ścieżki dostępu mogą również należeć punkty montowania. Obiekty wpisu katalogowego są tworzone na bieżąco, w trakcie analizowania ścieżki dostępu. Typ takich obiektów jest określony strukturą *struct dentry*. Te obiekty nie są związane z żadnymi danymi na nośniku, więc struktura je opisująca nie zawiera żadnego znacznika informującego o zmianie ich zawartości. Obiekt wpisu katalogowego może znajdować się w jednym z trzech stanów: używanym, nieużywanym lub ujemnym. Obiekt w stanie „używany" odpowiada poprawnemu i-węzłowi i jest bieżąco używany przez system. Obiekt w stanie „nieużywany" również odpowiada poprawnemu i-węzłowi, ale obecnie nie jest używany. Ten obiekt nie jest niszczony, na wypadek, gdyby trzeba go było użyć w niedalekiej przyszłości, chyba że zaczyna brakować wolnej pamięci operacyjnej. Obiekt w stanie „ujemny" związany jest z i-węzłem, który został usunięty, lub nie istnieje. Ten obiekt również nie jest usuwany, jeśli nie ma takiej potrzeby, gdyż jego obecność może zapobiec niepotrzebnym operacjom przeszukiwania, które nie zakończą się sukcesem. Zwolnione obiekty wpisów trafiają do dedykowanej pamięci podręcznej obsługiwanej przez alokator plastrowy.
Jądro utrzymuje również bufor wpisów katalogowych , przechowujący dotychczas utworzone wpisy katalogowe. Składa się on z trzech elementów: listy używanych wpisów katalogowych, listy ostatnio wykorzystywanych obiektów wpisów, która zawiera głównie obiekty wpisów nieużywanych i ujemnych, oraz tablicy skrótów (*ang. hash table*). Ta ostatnia wykorzystuje technikę haszowania, celem przyspieszenia odnajdywania obiektów wpisów katalogowych w buforze. Należy zaznaczyć, że jako pierwszy wyszukiwany jest obiekt związany z plikiem docelowym. Oprócz bufora obiektów wpisów katalogowych istnieje również bufor obiektów i-węzłów związanych ze zbuforowanymi wpisami katalogowymi. Operacje na wpisach katalogowych zgromadzone są w strukturze *struct dentry_operations*. Należą do nich między innymi: *d_revalidate()* - określa poprawność wskazywanego obiektu wpisu katalogowego, *d_hash()* - funkcja haszująca, *d_compare()* - porównuje dwie nazwy plików, *d_delete()* - wywoływana gdy licznik odwołań do obiektu osiągnie wartość zerową, *d_release()* - zwalnia obiekt wpisu katalogowego, *d_iput()* - wywoływana, gdy obiekt wpisu katalogowego traci związany z nim i-węzeł.
Z punktu widzenia procesów przestrzeni użytkownika obiekty plików są najważniejsze ze wszystkich obiektów VFS. Tworzone są one przez wywołanie systemowe *open()*,

1 Jest to bardzo jednolity opis. Sięgając do określonego pliku nie musimy wiedzieć na jakim fizyczne urządzeniu się znajduje. Systemy plików na wszystkich takich urządzeniach współtworzą wspólne drzewo katalogów. Przykładem innego podejścia są systemy MS Windows.

2 Należy zaznaczyć, że pojęcie pliku jest podstawowym, obok procesu, pojęciem w systemie Unix. Nie wszystkie pliki są „pojemnikami" na dane, niektóre z nich reprezentują np.: urządzenia.

3 W języku C++ można tworzyć klasy przy użyciu słowa kluczowego *struct* zamiast *class*. Osobom zainteresowanym bardziej szczegółowymi informacjami na ten temat polecam książkę Bruce'a Eckela „Thinking in C++".

a niszczone przez *close()*. Obiekty takie wskazują na obiekty wpisu katalogowego, a te z kolei wskazują na obiekty i-węzłów związane z plikami otwartymi przez proces. Z każdym z plików może być związanych kilka obiektów plików, w zależności od tego ile procesów go otworzyło. Typ takiego obiektu jest opisany strukturą *struct file*. Ta struktura zawiera również pole wskazujące na obiekt operacji, które można zrealizować na pliku. Ten obiekt jest określony strukturą *struct file_operations* i może zawierać wskaźniki na funkcje realizujące między innymi następujące operacje: *llseek()* - aktualizuje wskaźnik pozycji pliku, *read()* - odczytuje plik, *write()* - zapis do pliku, *poll()* - zawiesza proces w oczekiwaniu na sygnał aktywności pliku, *unlocked_ioctl()* i *compat_ioctl()* - służą do realizacji operacji charakterystycznych dla urządzenia reprezentowanego przez plik[4], *mmap()* - odwzorowuje plik w pamięci, *open()* - otwiera plik, *flush()* - jej działanie jest zależne od systemu plików, ale zawsze jest wywoływana przy zmniejszeniu liczby odwołań do pliku, *release()* - wywoływana przy wyzerowaniu licznika odwołań do pliku, *fsync()* - zapisuje wszystkie buforowane zmiany na nośnik, *aio_fsync()* - jak poprzedniczka, ale zapisuje w sposób nieblokujący, *fasync()* - funkcja uaktywnia lub dezaktywuje sygnały powiadamiające o zakończeniu asynchronicznych operacji wejścia – wyjścia, *read_iter()* - funkcja odczytuje dane z pliku i umieszcza je we wskazanych buforach, *write_iter()* - funkcja zapisuje dane do pliku ze wskazanych buforów, *sendpage()* - realizuje przekaz danych między plikami, *get_unmapped_area()* - funkcja odwzorowująca wskazany plik na niewykorzystaną pamięć, *lock()* - umożliwia manipulowanie blokadą pliku, *flock()* - używana do implementacji wywołania systemowego o tej samej nazwie, które zapewnia blokowanie doradcze pliku, *check_flags()* - sprawdza flagi ustawione za pomocą funkcji *fcntl()*.

Poza opisanymi wcześniej obiektami VFS wykorzystuje jeszcze kilka innych struktur. Struktura *file_system_type* zawiera informacje dotyczące poszczególnych typów systemów plików i jest wykorzystywana przez funkcję *get_sb()*, służącą do odczytu zawartości bloku głównego określonego systemu plików. Z każdym typem systemu plików obsługiwanego przez Linuksa powiązana jest jedna taka struktura. Po zamontowaniu systemu plików w systemie tworzona jest zmienna, której typ jest określony strukturą *struct vfsmount*. Zmienna ta zawiera informacje na temat punktu montownia, do których należą między innymi znaczniki montowania określające jakie operacje można w tym systemie przeprowadzić. Z każdym procesem związane są trzy struktury danych: *struct files_struct* – zawierająca wszystkie informacje związane z otwartymi przez proces plikami i ich deskryptorami, między innymi wskaźniki do obiektów plików, *struct fs_struct* – zawierająca informacje o związanym z procesem systemie plików (najważniejsze z nich to katalog bieżący i katalog główny) i *struct mnt_namespace* – określa dla procesu unikalną perspektywę systemu plików. Dwie pierwsze struktury mogą być współużytkowane przez procesy spokrewnione, ostatnia jest domyślnie współdzielona przez wszystkie procesy, ale istnieje możliwość określenia dla procesu odrębnej takiej struktury podczas jego tworzenia.

4 W starszych wersjach jądra istniała metoda funkcja *ioctl()*, która była synchronizowana za pomocą BKL. Została ona zastąpiona dwoma wymienionymi funkcjami. Funkcja *unlocked_ioctl()* nie korzysta z BKL. Funkcja *compat_ioctl()* również nie korzysta z BKL, została wprowadzona celem zachowania kompatybilności między 64 i 32-bitowymi komputerami, tzn. pozwala uruchamiać 32-bitowe aplikacje na 64-bitowych platformach sprzętowych.