Wykład piąty

Wywołania systemowe

Każdy system operacyjny zarządza zasobami systemu komputerowego, na którym jest uruchomiony oraz dostarcza pewnych usług procesom użytkownika. Jednocześnie większość nowoczesnych systemów operacyjnych zabrania zadaniom z przestrzeni użytkownika bezpośredniej interakcji z innymi zadaniami tego typu, lub wykonywania operacji na sprzęcie. Ma to na celu zapobieżenie nadużyciom ze strony tych zadań i stanowi część mechanizmu ochrony. Oznacza to także, że jedynie system operacyjny może wykonywać niektóre czynności, takie jak np.: odczyt i zapis danych z urządzeń wejścia-wyjścia. Jeśli proces użytkownika chce pobrać lub zapamiętać dane na takim urządzeniu, to musi to zrobić za pośrednictwem systemu operacyjnego, uruchamiając odpowiednie *wywołanie systemowe*. Wywołanie systemowe jest funkcją jądra, która może być uruchomiona przez proces użytkownika, celem zlecenia systemowi operacyjnemu wykonania jakiejś czynności w imieniu tego procesu[1]. Zbiór wszystkich wywołań systemowych stanowi interfejs między aplikacją a jądrem systemu operacyjnego. Dzięki jego istnieniu zapewniona jest stabilność systemu, możliwa jest praca wielozadaniowa oraz łatwiej jest pisać programy, które będą wykonywane w przestrzeni użytkownika.

Powyższy opis należy uzupełnić o element, który jest określany mianem interfejsu aplikacji – w skrócie API (ang. *Application Programming Interface*). Zadania użytkownika najczęściej nie korzystają bezpośrednio z wywołań systemowych lecz robią to za pomocą podprogramów języka wysokiego poziomu. W przypadku systemów uniksowych API jest określone standardem POSIX, zdefiniowanym przez organizację IEEE. Standard ten opisuje również wywołania systemowe. Każdy system, który ma być uważany za zgodny z Uniksem musi ten standard implementować[2], przy czym nie jest określony sposób tej implementacji. Podstawowym językiem programowania w każdym systemie uniksowym jest język C. Funkcje należące do API zawarte są w standardowej bibliotece tego języka, o nazwie *libc*[3]. Inne języki programowania wysokiego poziomu mają własne biblioteki standardowe, które najczęściej bazują na bibliotece języka C[4]. Część funkcji określonych standardem POSIX stanowią tzw. funkcje opakowujące (*ang. wrapping routines*), których jedynym zadaniem jest uruchomienie wywołania systemowego. Inną część stanowią funkcje korzystające z więcej niż jednego wywołania systemowego, a jeszcze inną funkcje, które wcale nie korzystają z wywołań systemowych. Zgodnie z tym co zostało napisane wcześniej różne kompilatory języka C, działające na różnych systemach uniksowych mogą w różny sposób implementować każdą z tych funkcji.

Wywołania systemowe podobnie jak zwykłe funkcje mogą przyjmować pewną liczbę argumentów, lub nie przyjmować ich w ogóle. Mogą one również, oprócz podstawowej operacji wykonywać czynności dodatkowe, które wpływają na stan sytemu. Oznacza to wówczas, że wywołanie ma skutki uboczne. Każde wywołanie systemowe zwraca wartość typu *long*, która stanowi kod wykonania. Zazwyczaj poprawne zakończenie wywołania sygnalizowane jest liczbą dodatnią lub (najczęściej) zerem, a błędne, ujemną. W przestrzeni użytkownika kod wykonania zapisywany jest do specjalnej zmiennej globalnej o nazwie *errno*. Jej zawartość może zostać przetworzona na komunikat czytelny dla użytkownika dzięki funkcji *perror*(). Wywołanie systemowe implementowane jest za pomocą funkcji napisanej w języku C. Umieszczana jest ona najczęściej w pliku z kodem źródłowym tej części jądra, z którą jest powiązane określone wywołanie. Kod wszystkich funkcji implementujących wywołania systemowe ma pewne cechy wspólne. Przyjęto konwencję, że nazwy takich funkcji konstruowane są według schematu *sys_**, gdzie znak „*" oznacza nazwę implementowanego wywołania systemowego. Dodatkowo nazwy te są poprzedzone modyfikatorem *asmlinkage*, celem poinformowania kompilatora, że argumenty dla tych funkcji są przekazywane wyłącznie za pomocą stosu. Nie każda architektura procesora tego wymaga, więc dla niektórych z nich ten modyfikator jest pusty. Każde wywołanie systemowe posiada swój numer, który jest równocześnie indeksem w tablicy *sys_call_table*, zawierającej adresy wszystkich zarejestrowanych wywołań systemowych. Jej implementacja zależna jest od sprzętu na którym uruchomiony jest system. W przypadku sprzętu opartego na 32-bitowych procesorach Intela znajduje się ona w pliku *syscall_table_32.S*, który włączany jest do pliku *entry_32.S,* również zależnego od architektury. Dla sprzętu z procesorem opartym na architekturze x86_64 tablica ta jest implementowana w pliku *unistd_64.h*, który następnie jest włączany do pliku *syscall_64.c*, a ten załączany jest do *entry_64.S*. Linux udostępnia także funkcję *sys_ni_call*(), zwracającą wartość -ENOSYS, oznaczającą że wywołanie o podanym numerze nie zostało zaimplementowane, lub zostało z jakiś przyczyn usunięte, co zdarza się niezmiernie rzadko[5]. W nowszych wersjach jądra wprowadzono makrodefinicje, które rozwijane są automatycznie przez preprocesor do określonego nagłówka funkcji realizującej wywołanie systemowe. Nazwy tych makr są utworzone według schematu SYSCALL_DEFINE*n*, gdzie *n* jest liczbą parametrów wspomnianej funkcji. Jeśli nie ma ona żadnych parametrów, to *n* jest równe zero. Pierwszym argumentem opisywanych makr jest nazwa wywołania systemowego (bez przedrostka *sys_*). W przypadku, gdy makro pozwala na określenie parametrów funkcji realizującej wywołanie systemowe, to należy do niego przekazać nie tylko nazwę parametru, ale trzeba ją poprzedzić jego typem.

Proces użytkownika nie może bezpośrednio wywołać funkcji implementującej wywołanie systemowe, gdyż znajduje się ona w chronionej przestrzeni jądra. Może to uczynić wyłącznie poprzez mechanizm przerwań. W wersji Linuksa dla 32-bitowych procesorów Intela lub zgodnych istnieje specjalne przerwanie programowe o numerze 128 (80h), które służy do uruchamiania wywołań systemowych. Skojarzona jest z nim funkcja *system_call*(), która stanowi jego procedurę obsługi (ang. *handler*) i jednocześnie punkt wejścia dla wywołań systemowych (ang. *call gate*). Kod tej funkcji jest umieszczony w pliku *entry_32.S*. Nowsze, 32-bitowe procesory tej firmy (od Pentium II wzwyż) dostarczają rozkazu *sysenter*, który pełni tę samą funkcję, co rozkaz przerwania programowego *int*, ale jest szybszy w działaniu. W momencie wykonania przerwania 128 następuje przejście procesora do trybu jądra. Funkcja *system_call*() sprawdza poprawność numeru wywołania, który jest jej przekazywany przez rejestr *eax*. Jeśli ten numer nie jest prawidłowy, to *system_call*() zwraca błąd -ENOSYS. W przeciwnym przypadku jego wartość jest mnożona przez wielkość adresu wyrażoną w bajtach (4 w tym przypadku) i odczytywany jest adres funkcji wywołania systemowego z tablicy *sys_call_table*, a następnie ta funkcja jest wywoływana za pomocą zwykłego rozkazu *call*. Argumenty wywołania systemowego przekazywane są do *system_call*() za pomocą rejestrów *ebx, ecx, edx, esi, edi*. Zanim zostanie uruchomiona funkcja implementująca określone wywołanie, to *system_call*() odkłada wartości z rejestrów na stos. Jeśli trzeba wywołaniu przekazać więcej niż pięć argumentów, to w jednym z rejestrów umieszczany jest adres obszaru pamięci w przestrzeni użytkownika, gdzie umieszczona jest reszta argumentów. Ta sama metoda jest stosowana dla argumentów, które nie mieszczą się w rejestrach. Wartości wszystkich argumentów muszą zostać zweryfikowane celem sprawdzenia, czy nie są one błędne i czy nie spowodują naruszenia ochrony. Szczególnie ważne jest sprawdzenie argumentów wskaźnikowych. W ich przypadku jądro wykonuje trzy testy:

1. czy wskaźnik wskazuje na obszar pamięci przestrzeni użytkownika,
2. czy wskaźnik wskazuje obszar pamięci w przestrzeni procesu na zlecenie którego zostało uruchomione wywołanie,
3. jeśli ma zostać wykonana operacja odczytu, to sprawdzane jest, czy obszar na który wskazuje wskaźnik jest obszarem do odczytu, a jeśli ma być wykonany zapis, to sprawdzane jest, czy do tego obszaru można zapisywać, jeśli ma nastąpić wykonanie kodu, to sprawdzane jest, czy można tę operację zrealizować.

Kopiowanie informacji z obszaru pamięci jądra do obszaru pamięci procesu użytkownika wykonywane jest za pomocą funkcji *copy_to_user*(). Przyjmuje ona trzy argumenty. Pierwszym z nich jest adres obszaru docelowego, drugi to adres obszaru źródłowego, a trzeci to liczba bajtów, które trzeba przekopiować. Jeśli należy skopiować dane z obszaru procesu użytkownika do obszaru pamięci jądra, to wówczas używana jest funkcja *copy_from_user*(). Podobnie jak poprzedniczka przyjmuje ona trzy argumenty, o takim samym znaczeniu. Obie funkcje w przypadku niepowodzenia zwracają liczbę bajtów, których nie udało się skopiować, a w przypadku powodzenia, wartość zero. Weryfikację uprawnień procesu wywołującego wywołanie systemowe w jądrach Linuksa serii 2.6 i nowszych przeprowadza się za pomocą wywołania funkcji *capable*(). Jeśli proces uruchamiający dane wywołanie systemowe nie ma wymaganych uprawnień do zasobu obsługiwanego przez to wywołanie, to *capable*() zwraca wartość zero, w przeciwnym wypadku wartość większą od zera. Lista uprawnień przechowywana jest w pliku *linux/capability.h*. W starszych wersjach jądra sprawdzane było jedynie, czy proces jest procesem użytkownika uprzywilejowanego. Dokonywane to było z pomocą funkcji *suser*(). Wywołanie jest wykonywane w kontekście procesu użytkownika, oznacza to, że ma ono dostęp do deskryptora procesu, który je uruchomił za pomocą makrodefinicji *current*. Wywołanie systemowe można zawiesić i tym samym wprowadzić

1 W literaturze dotyczącej systemów operacyjnych wywołania systemowe określa się niekiedy mianem funkcji systemowych. W niniejszym tekście terminem tym określa się podprogramy jądra. Wywołania systemowe są tylko podzbiorem zbioru tych podprogramów.
2 Niektóre z funkcji opisanych tym standardem implementują nie tylko systemy kompatybilne z Uniksami. Do tych wyjątków zalicza się między innymi Windows NT.
3 W przypadku systemu Linux ta biblioteka nazywa się *glibc* – skrót od *GNU libc*.
4 Translatory tych języków są najczęściej pisane bezpośrednio, lub przy wykorzystaniu odpowiednich narzędzi, w języku C.
5 Litery *ni* w nazwie funkcji są skrótem od angielskich słów *not implemented*.

proces, który je uruchomił w stan oczekiwania (uśpienia), jeśli musi ono poczekać na jakieś zdarzenie (np. realizację operacji wejścia-wyjścia). Po zakończeniu jego wykonania sterowanie wraca do funkcji *system_call*(), a wartość przez nie zwrócona zapisywana jest w rejestrze *eax*.

Sposób uruchamiania i działania wywołań systemowych dla procesorów opartych na innych architekturach jest bardzo podobny. Takie procesory posiadają odpowiednie mechanizmy, które pełnią taką samą rolę jak przerwanie 128. Przykładowo, procesory oparte na architekturze x86_64 posiadają rozkaz o nazwie *syscall*. W rezultacie jego wykonania uruchamiana jest wspomniana funkcja *system_call()*, której implementacja jest zapisana w pliku *entry_64.S*. Działanie tej funkcji jest podobne jak jej odpowiedniczki dla 32-bitowych procesorów x86, ale występują drobne różnice. Numer wywołania jest jej przekazywany w rejestrze o nazwie *rax*, a parametry dla funkcji implementującej wywołanie systemowe są przekazywane za pomocą rejestrów *rdi, rsi, rdx, r10, r8* i *r9*. Jeśli numer wywołania jest poprawny, to mnożony jest nie przez 4, a przez 8. Wartość zwracana przez funkcję realizującą wywołanie systemowe zapisywana jest do rejestru *rax*. Ostatnia różnica polega na tym, że wartości rejestrów przed uruchomieniem funkcji realizującej wywołanie nie są odkładane na stos, gdyż te architektury procesorów nie wymagają tego. Dla nich modyfikator *asmlinkage* będący makrem napisanym w języku C jest zdefiniowany jako pusty.

Linux jako system dostępny na licencji GPL, umożliwia modyfikowanie, dodawanie i usuwanie wywołań systemowych każdemu programiście, który ma dostęp do kodu źródłowego jądra i odpowiednie uprawnienia by zmodyfikowane jądro uruchomić. W przypadku procesorów opartych na architekturze x86_32, w celu dodania nowego wywołania należy oprogramować funkcję, która je obsłuży, dodać odpowiedni wpis do tablicy wywołań systemowych, która znajduje się w pliku *syscall_table_32.S*, określić numer wywołania w pliku *include/asm/unistd_32.h* i skompilować kod źródłowy zmodyfikowanego jądra. Dla procesorów o architekturze x86_64 nie trzeba modyfikować pliku *entry_64.S*, a jedynie plik *include/asm/unistd_64.h*. W obu przypadkach kod funkcji realizującej wywołanie nie może być umieszczony w module. Dodane przez nas wywołanie nie jest oczywiście uwzględnione w standardowej bibliotece języka C, mimo to możemy z niego skorzystać w naszych aplikacjach. We wcześniejszych wersjach jądra można było w tym celu użyć jednej z makrodefinicji *_syscall**n***(), gdzie „n" oznacza liczbę argumentów przyjmowanych przez nasze wywołanie. Każda taka makrodefinicja przyjmowała 2*n+2 argumentów. Były to: typ wartości zwracanej przez wywołanie, nazwa wywołania i argumenty poprzedzone ich typami. Te makrodefinicje nie były dostępne na wszystkich platformach sprzętowych. Począwszy od wersji 2.6.18 jądra zostały one zastąpione przez funkcję *syscall*() i przestały być w ogóle dostępne. Funkcja *syscall*() przyjmuje jeden parametr obowiązkowy jakim jest numer wywołania systemowego oraz dowolną liczbę argumentów przekazywanych do niego. Zwraca status wykonania wywołania.

Dodanie do jądra nowych wywołań systemowych jest kuszącym pomysłem, jednakże wśród twórców jądra Linuksa panuje silna tendencja, aby tego nie robić. Po dokładnych rozważaniach można określić następującą listę wad i zalet tworzenia nowych wywołań:

Wady:

1. Konieczność przypisania wywołaniu numeru, który musi zostać zaakceptowany przez głównych programistów jądra Linuksa.

2. Interfejs wywołania powinien zostać zdefiniowany tak, aby nie trzeba go było zmieniać w przyszłości, bo mogłoby to spowodować problemy z wykonywaniem programów, które z tych wywołań korzystają.

3. W niektórych przypadkach trzeba będzie dostarczyć różnych implementacji tego wywołania dla różnych platform sprzętowych, dodatkowo może wystąpić potrzeba określenia dla każdej z nich osobnego numer wywołania.

4. Nie należy implementować wywołania jako środka prostej komunikacji między procesami.

Zalety:

1. Wywołania można w prosty sposób implementować i również prosto się z nich korzysta.

2. Ze względu na krótki czas przełączania kontekstu w Linuksie wywołania systemowe są bardzo wydajne.

W jądrze Linuksa serii 2.6 i nowszych dostępnych jest średnio kilkaset[6] wywołań. Świadczy to o „dojrzałości rozwojowej" tego systemu. Przypadki usuwania istniejących wywołań są bardzo rzadkie. Większość wywołań jest prosta i dostarcza prostych usług. Jako wyjątek od tej reguły może zostać przedstawione wywołanie *ioctl()*. Aby uniknąć dodawania nowych wywołań można posłużyć się, rozwiązując jakiś problem, tym co dostarcza podsystem obsługi urządzeń i plików.

6 Liczba ta zmienia się w zależności od platformy. Część sprzętu na którym pracuje Linux wymaga specyficznych wywołań systemowych.