Mikrojądra. Omówienie i przegląd

Marek Materzok

22 stycznia 2007

1 Wprowadzenie

Tradycyjnie znaczną część funkcjonalności systemu operacyjnego skupia się w monolitycznym bloku kodu, zwanym jądrem, pracującym z uprawnieniami monitora. Wśród funkcjonalności zawartej w jądrze monolitycznym znajdują się: zarządzanie procesami i pamięcią, sterowniki urządzeń, obsługa systemu plików, stos protokołów sieciowych i wiele innych. Podejście to ma swoje zalety: jest dobrze poznane, co ułatwia pracę nad takimi systemami, a także umożliwia ono dowolnego rodzaju optymalizacje. Jednak ma ono też liczne wady, między innymi:

- Brak izolacji komponentów systemu. Błąd w jednym z komponentów, jakkolwiek mało ważnym, może doprowadzić do załamania systemu bądź przejęcia kontroli nad nim przez atakującego.
- Utrudniona modernizacja i rozbudowa systemu. Nie jest możliwe łatwe dodawanie i wymienianie komponentów systemu w trakcie jego pracy. Wprawdzie problem ten został częściowo rozwiązany przez dynamicznie ładowane moduły, jest to jednak tylko półśrodek.
- Brak możliwości rozszerzenia funkcjonalności systemu przez indywidualnych użytkowników. Wbrew pozorom może to być bardzo pożądane: przykładowo, użytkownik może chcieć pracować na archiwum czy zdalnym systemie plików tak samo, jak na dysku lokalnym.
- Niepotrzebnie skomplikowanie kodu komponentów systemu. Nawet będące koncepcyjnie odległe od sprzętu komponenty, takie jak obsługa protokołów sieciowych, pracują w trybie monitora, przez co muszą radzić sobie z całą złożonością z tym związaną, ponadto mogą korzystać jedynie z uproszczonego środowiska programistycznego dostępnego w jądrze.
- Utrudnione zarządzanie pamięcią wykorzystywaną przez system. Bardzo nieliczne systemy implementują stronicowanie pamięci jądra.
- Problemy z czasem odpowiedzi systemu. Ponieważ jądro z reguły jest niewywłaszczalne, wysokopriorytetowe procesy użytkownika są zmuszone czekać, aż aktualnie pracujący proces opuści jądro, co - biorąc pod uwagę złożoność zadań wykonywanych przez jądro - może potrwać nawet kilka milisekund.

Z tych powodów uzasadnione jest poszukiwanie lepszych metod konstrukcji systemów operacyjnych. Jedną z nich, a którą zajmę się w tej pracy, jest architektura z mikrojądrem.

2 Bliżej o mikrojądrach

Ideą mikrojąder jest zminimalizowanie funkcjonalności wymagającej pracy w trybie monitora. Mikrojądra z reguły zawierają tylko proste operacje zarządzania procesami i pamięcią oraz mechanizmy komunikacji międzyprocesowej, reszta funkcjonalności przeniesiona jest do procesów systemowych pracujących w trybie użytkownika, zwanych serwerami. Serwery komunikują się między sobą i z aplikacjami za pośrednictwem operacji komunikacji międzyprocesowej dostarczanych przez mikrojądro.

Podejście to przynosi wiele korzyści w porównaniu do tradycyjnej architektury z jądrem monolitycznym. Przede wszystkim, architektura z mikrojądrem narzuca ustrukturalizowanie systemu w niezależne komponenty, które komunikują się ze sobą przez dobrze zdefiniowane interfejsy. Komponenty te można testować w oddzieleniu od reszty systemu, co ułatwia pracę nad nimi. Co więcej, poszczególne serwery pracują w osobnych przestrzeniach adresowych, oddzielone od siebie, dzięki czemu błąd w jednym z nich nie może spowodować problemów w innych. A jeśli już z powodu jakiegoś błędu serwer przestanie działać, można go uruchomić ponownie bez szkody dla reszty systemu.

Oczywiście, oparcie funkcjonowania systemu na komunikacji międzyprocesowej oznacza, że ta funkcjonalność w systemach z mikrojądrem musi być szczególnie dobrze zaprojektowana. Wczesne mikrojądra stosowały komunikację asynchroniczną, jednak okazało się, że oznacza to problemy z wydajnością związane z nadmiernym kopiowaniem i opóźnieniami w szeregowaniu procesów. Z tego powodu współczesne systemy z mikrojądrem używają przede wszystkim komunikacji synchronicznej. W wielkim uproszczeniu, synchroniczna komunikacja polega na tym, że nadawca komunikatu może nadać komunikat tylko wtedy, gdy odbiorca jest gotowy do odbierania. Ponieważ odbiorca otrzymuje czas procesora od razu po odebraniu komunikatu, możliwe jest zredukowanie do minimum lub nawet całkowite wyeliminowanie kopiowania.

3 Wybrane systemy

3.1 Minix

System Minix jest systemem edukacyjnym opracowanym przez Andrew Tanenbauma dla potrzeb jego książki pt. „Operating Systems: Design and Implementation", wydanej w roku 1987. W założeniach miał on służyć za przeznaczoną dla studentów ilustrację koncepcji omówionych w książce, w związku z tym jest on napisany w sposób prosty i klarowny.

Rozwój systemu Minix nie zatrzymał się w miejscu. Obecnie Tanenbaum pracuje nad trzecią wersją tego systemu, który - oprócz bycia systemem edukacyjnym, jak poprzednie wersje - będzie systemem o bardzo wysokiej bezawaryjności. Został on zaprojektowany w ten sposób, aby automatycznie odbudowywał uszkodzenia wywołane przez awarie komponentów systemu, nie przerywając przy tym pracy.

3.2 QNX

QNX jest rozwijanym od 1982 roku komercyjnym systemem czasu rzeczywistego projektowanym pod kątem sterowania pracą urządzeń. Jest to bardzo lekki, wydajny system, który wiele swoich pozytywnych cech - w tym spełnienie ostrych ograniczeń czasowych, wymaganych w wielu specjalizowanych zastosowaniach, oraz bardzo dobre wsparcie dla wielu procesorów, zawdzięcza architekturze z mikrojądrem.

Z zewnątrz QNX przypomina system uniksowy - wspiera interfejs POSIX, obsługuje programy wykorzystujące interfejs graficzny X Window (choć sam używa własny, lekki interfejs Photon microGUI). Jednak wewnątrz jest nowoczesnym systemem z mikrojądrem, drastycznie innym od Uniksów. Funkcjonalność jądra jest ograniczona do minimum, nawet sterowniki sprzętu i elementy zarządzania pamięcią i procesami zostały przeniesione do przestrzeni użytkownika. Mechanizmem komunikacji jest wysoko zoptymalizowane synchroniczne przesyłanie komunikatów.

3.3 GNU HURD

Prace nad GNU Hurd rozpoczęły się w roku 1990 celem uzupełnienia systemu GNU - otwartego klonu Uniksa - o jedyny brakujący element, mianowicie jądro. Postanowiono oprzeć się o mikrojądro Mach, co ostatecznie okazało się nienajlepszą decyzją. Ponadto projekt okazał się zbyt ambitny, co odbiło się na tempie prac. Ostatecznym ciosem było opracowanie przez Linusa Torvaldsa systemu Linux w 1991. Ostatecznie Linux został zaadoptowany jako jądro GNU, a zainteresowanie systemem Hurd zmalało tak bardzo, że stał się on zdatny do użytku dopiero po roku 2000.

Hurd różni się od innych systemów tym, że duży nacisk kładzie się w nim na swobodę użytkownika. Każdy użytkownik w systemie może uruchamiać własne serwery, które pracują na jego poziomie uprawnień, oraz udostępniać je innym użytkownikom. Możliwe jest nawet zrezygnowanie z użytkowania systemowych serwerów i korzystanie z własnych, co doprowadzone jest do ekstremum: użytkownik może nawet uruchomić całego Hurda pod swoją kontrolą.

3.4 L4

Ze względu na porażkę wydajnościową wczesnych mikrojąder, w tym mikrojądra Mach, konieczne było opracowanie sprawnego systemu komunikacji międzyprocesowej. Cel ten zrealizował w mikrojądrze L3, które później wyewoluowało w L4. Mikrojądro L4 odniosło duży sukces w środowisku akademickim i jest dalej rozwijane na Uniwersytecie w Karlsruhe i Uniwersytecie Nowej Południowej Walii.

Swoją wydajność mikrojądro L4 zawdzięcza ekstremalnemu uproszczeniu jądra. Mechanizmem komunikacji jest wysoko zoptymalizowane synchroniczne przekazywanie komunikatów rozbudowane o dodatkową funkcjonalność zarządzania przestrzeniami adresowymi, która polega na tym, że w ramach przesłania komunikatu można udostępnić lub oddać fragment swojej pamięci innemu procesowi. Wywołania systemowe L4 zostały starannie zaprojektowane tak, aby udostępniać procesom wyłącznie mechanizmy, politykę zostawiając wymaganiom konkretnych procesów.

Mikrojądro L4 jest używane w badaniach dotyczących systemów trwałych (ang. persistent), wirtualizacji, reużycia sterowników sprzętu, systemów z pojedynczą przestrzenią adresową i innych.

3.5 KeyKOS

Istotną dla bezpieczeństwa komputera zasadą jest reguła najmniejszych uprawnień, która mówi, że każdy program powinien mieć dostęp tylko i wyłącznie do tego, co jest mu potrzebne do prawidłowego działania. We współcześnie używanych systemach operacyjnych reguła ta jest notorycznie łamana. Kontrola uprawnień jest bardzo gruboziarnista, a często brak jej w ogóle, co powoduje, że najmniejszy błąd w systemie może prowadzić do katastrofy - i prowadzi,

co doskonale pokazuje wciąż rosnąca plaga wszelkiej maści programów typu spyware, adware i trojan.

Metodą na zaimplementowanie reguły najmniejszych uprawnień może być zastosowanie koncepcji zdolności (ang. capabilities). Zdolność jest obiektem systemowym, dającym procesowi go posiadającemu możliwość operowania na innym obiekcie, którego zdolność dotyczy. O zdolności można myśleć jak o kluczu - można ją przekazywać, kopiować, można ją też unieważnić („zmiana zamka"). Zdolności sprawdzają się dobrze, jeśli system pracuje w sposób ciągły, jednak restart systemu stwarza ryzyko zaginięcia pewnych zdolności.

W jaki sposób system KeyKOS rozwiązuje ten problem? Bardzo prosto - tworzy procesom iluzję systemu pracującego w sposób ciągły. Okresowo system zapisuje cały stan procesów na dysku, tak, że po awarii bądź restarcie system zostanie odtworzony w stanie, w jakim był wcześniej, łącznie ze zdolnościami procesów.

U podstaw systemu KeyKOS znajduje się proste, wydajne mikrojądro. To właśnie ono gwarantuje sprawne odtwarzanie stanu systemu. Zaimplementowana w jądrze została koncepcja pamięci trwałej, która z punktu widzenia procesów jest jedynym rodzajem pamięci. Jest w niej zapisywane wszystko - kod i dane procesów, pliki, dane jądra itp.

Następcami systemu KeyKOS są EROS i CapROS.

4 Pokrewne rozwiązania

Istnieje kilka rozwiązań pokrewnych do architektury z mikrojądrem, które, mimo bycia uznawanym za odrębne, są warte wspomnienia.

4.1 Xen

Ostatnio coraz większą popularnością cieszy się tzw. wirtualizacja. Polega ona na uruchamianiu kilku programów na tym samym komputerze, stwarzając iluzję, że każdy z nich ma do dyspozycji cały komputer. Pierwsze rozwiązania tego rodzaju (przykładowo popularny VMWare) polegały na tym, że maszyna wirtualna wydawała się być identyczna z maszyną fizyczną, co umożliwiło uruchamianie niezmodyfikowanych programów na maszynie wirtualnej. Niestety, ponieważ architektura IA32 nie była projektowana pod kątem wirtualizacji, wiązało się to ze znacznym obniżeniem wydajności.

Na problem warto popatrzeć w sposób bardziej abstrakcyjny. Czym monitor maszyn wirtualnych, taki jak VMware, różni się od typowego systemu operacyjnego? Oba udostępniają dla uruchomionych pod nimi programów abstrakcyjny model maszyny, różnica jest tylko taka, że pierwszy model jest blisko sprzętu, a drugi - blisko użytkownika. Dlaczego nie spróbować wariantu pośredniego? Xen jest właśnie realizacją takiego wariantu.

Model maszyny udostępniany przez system Xen jest wciąż bardzo niskopoziomowy - nie ma w nim mowy o plikach, uprawnieniach, sieci, czy nawet przekazywaniu komunikatów, jak w mikrojądrach. Jest on jednak opracowany pod kątem jak najsprawniejszej implementacji, tak, żeby uruchamiane pod Xenem programy - które muszą być zmodyfikowane pod kątem współpracy z nim - działały jak najefektywniej.

Obecnie Xen jest najczęściej wykorzystywany w celu uruchomienia wielu kopii tradycyjnych systemów operacyjnych (takich jak Linux, BSD czy Solaris) obok siebie. Jednak koncepcja maszyn wirtualnych jest jeszcze młoda, więc przyszłość może przynieść jeszcze wiele niespodzianek.

4.2 Singularity

Duża część złożoności współczesnych systemów operacyjnych jest związana z obchodzeniem takich czy innych ograniczeń niskopoziomowej natury. Nawet w systemach z mikrojądrem zarządzanie pamięcią i mechanizmami ochrony pamięci jest zwykle skomplikowane. Jednym z możliwych rozwiązań jest całkowite porzucenie możliwości wykonywania kodu maszynowego i zastąpienie go znacznie prostszym kodem maszyny wirtualnej. Na tej idei oparty został eksperymentalny system operacyjny opracowany przez laboratoria firmy Microsoft, Singularity.

W sercu Singularity znajduje się maszyna wirtualna .NET. Jest to jedyny komponent, w którym znajduje się kod maszynowy. Cała reszta systemu oraz aplikacje muszą być zapisane w bajtkodzie .NET i pracują pod kontrolą maszyny wirtualnej. Można więc myśleć o niej jak o specyficznym mikrojądrze, które dostarcza mechanizmów wykonania, zarządzania pamięcią i komunikacji, jednak o znacznie bardziej wysokopoziomowym charakterze. Czynności możliwe do wykonania przez procesy są przy tym ograniczone na tyle, że nie mogą one szkodzić sobie wzajemnie.

Wydawać by się mogło, że tak napisany system musi być ekstremalnie powolny. Otóż nie. Po pierwsze, kontrola poprawności programów następuje przez statyczną weryfikację, a nie w trakcie pracy, przez co programy nie ponoszą związanych z tym kosztów wydajnościowych. Po drugie, bajtkod .NET nie jest interpretowany, tylko dynamicznie kompilowany do kodu maszynowego, co zapewnia równie dobrą - a często lepszą, ze względu na dynamiczne optymalizacje - wydajność pracy programów. Zaś po trzecie, i chyba najważniejsze, kontrola poprawności na poziomie kodu programów pozwala na całkowite zrezygnowanie ze sprzętowej kontroli dostępu do pamięci, co pozwala na zaimplementowanie bardzo wydajnego programowego przełączania kontekstu. Jest to o tyle istotne, że to właśnie efektywność przełączania kontekstu stanowi większość kosztów związanych z architekturą z mikrojądrem.

5 Podsumowanie

Jak widać, koncepcja mikrojądra, kiedyś uznana za chybioną, wraca dziś ze zdwojoną siłą. Zredukowanie do minimum ilości niskopoziomowego kodu pracującego w jądrze otworzyło rozmaite drogi rozwoju, które są teraz intensywnie badane. Niektóre z tych dróg zapewne prowadzą donikąd, jednak jestem przekonany, że prace w ogólności podążają w dobrym kierunku i następnej rewolucji w dziedzinie systemów operacyjnych doczekamy się właśnie ze strony mikrojąder.