Systemy operacyjne

mgr inż. Maciej Karaszewski

Warszawa, 2009

Spis treści

Systemy operacyjne – wprowadzenie

Geneza oraz historia systemów operacyjnych

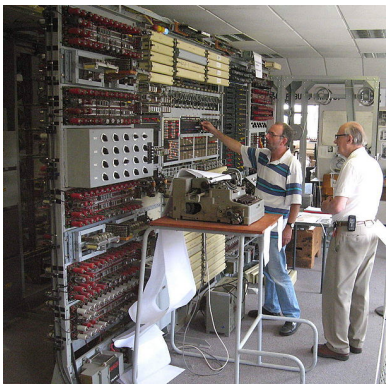
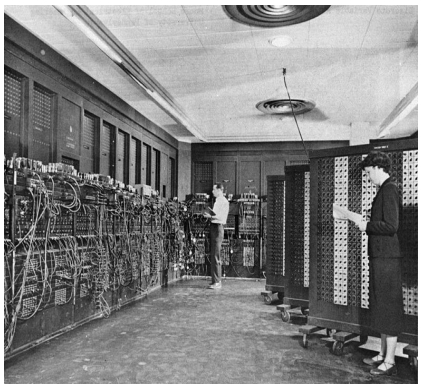
Druga połowa dwudziestego wieku to epoka wciąż przyspieszającego rozwoju elektroniki, a w szczególności komputerów. W ciągu ostatnich kilkunastu lat urządzenia elektroniczne stały się tak popularne i powszechne, że większość ludzi nie wyobraża sobie bez nich życia. Komputery, konsole do gier, telefony komórkowe, są narzędziami pracy i rozrywki dla ogromnej grupy osób. Elektronika trafiła również do samochodów, „inteligentnych domów" itp. zastępując często skomplikowane układy mechaniczne. Projektowane układy elektroniczne są coraz wydajniejsze, mają coraz większe możliwości. Ceną skomplikowania układów jest jednak skomplikowanie ich oprogramowania, bez którego nie mogą one pełnić swoich funkcji. Z tego powodu do kolejnych grup urządzeń elektronicznych zaczęto wprowadzać podział na różne warstwy oprogramowania, które najczęściej tworzą różne, specjalizujące się w tym zespoły. I tak projektanci i programiści aplikacji końcowych nie muszą zajmować się niskopoziomową obsługą elementów danego urządzenia (np. portów wejścia / wyjścia, pamięci operacyjnej itd.).

Za **pośredniczenie między fizycznymi elementami systemu a programami końcowymi odpowiada system operacyjny (SO)**. Wprowadzenie tej warstwy, oprócz zdjęcia z programistów końcowych niebanalnego problemu oprogramowywania (w ramach każdej aplikacji!) sprzętu, pozwoliło na wprowadzeniu pewnej standaryzacji dzięki czemu np. kolejne linie urządzeń są wzajemnie kompatybilne i aplikacje końcowe działające na jednym modelu można najczęściej uruchamiać na jego następcach, bez względu na różnice sprzętowo architekturalne. Dobrym przykładem tego ułatwienia jest np. dowolny program do odtwarzania muzyki z plików mp3 działający pod kontrolą systemu Microsoft Windows – programista nie musi przewidywać jaki model karty dźwiękowej znajdzie się w komputerze na którym program ma działać, nie musi też zastanawiać się jakich przerwań i adresów IO dana karta używa, chcąc odtwarzać dźwięk odwołuje się do funkcji systemu operacyjnego, który sam przekierowuje dane do odpowiedniej karty w odpowiedni sposób.

Oprócz pośredniczenia między urządzeniami a programami SO zajmuje się także odpowiednim p**rzydziałem dostępu do elementów składowych systemu poszczególnym programom**. Pierwsze komputery mogły mieć uruchomiony tylko jeden program (do momentu jego zakończenia uruchomienie innego było niemożliwe). W ten sposób działają wciąż niektóre bardzo proste urządzenia, jednak przeważająca większość systemów może uruchamiać wiele programów równolegle. Z racji tego, że niemożliwym jest aby programista aplikacji mógł przewidzieć w jakich warunkach będzie działał jego program (tzn. jakie programy będą z nim uruchomione), nie jest on w stanie zabezpieczyć programu przed np. nadpisywaniem danych przez niego przechowywanych w pamięci RAM przez inny program. Tą funkcję musi zapewniać system operacyjny.

Rozwój systemów operacyjnych nie rozpoczął się jednocześnie z pojawieniem się komputerów. Pierwsze tego typu urządzenia, które trafniej byłoby określić kalkulatorami nie posiadały żadnego systemu operacyjnego. Budowali je, uruchamiali i oprogramowywali najczęściej ci sami ludzie. Komputery takie jak Z3 Konrada Zuse, Collosus używany w Wielkiej Brytanii do łamania szyfrów niemieckich w czasie II Wojny Światowej lub ENIAC wykorzystywany do

obliczeń przy konstrukcji bomby wodorowej w USA wykonywały proste programy pobierające przy uruchomieniu dane wejściowe, wykonywały obliczenia i zwracały wynik, zwykle w postaci wydruku. Programowanie tych komputerów odbywało się w języku maszynowym poprzez wykonanie odpowiednich połączeń na patch-panelach lub przygotowanie odpowiednio perforowanych kart metalowych.

Pierwsze komputery mechaniczno – elektryczne. Role elementów logicznych pełniły mechaniczne lub elektryczne przekaźniki i przełączniki, wykorzystywano również lampy próżniowe. Od lewej – Zuse Z3, Collosus MKII (zrekonstruowany), ENIAC (źródło – Wikipedia).

Wynalezienie tranzystora i wyprodukowanie pierwszego takiego elementu z krzemu przez Texas Instruments w 1954 roku zapoczątkowało nową erę elektroniki. Układy logiczne dotąd budowane z elementów mechanicznych lub lampowych były bardzo zawodne, przykładowo komputer ENIAC zawierał 17 468 lamp próżniowych, z których w początkowym okresie działania dziennie przepalało się kilkanaście powodując, że czas awaryjnego przestoju systemu był równy czasowi jego działania. Zastąpienie lamp tranzystorami pozwoliło uzyskać układy elektroniczne na tyle niezawodne, że urządzenia z nich zbudowane mogły zacząć być sprzedawane (oczywiście cena komputerów tranzystorowych była tak duża że w zasadzie stać na nie było tylko agencje rządowe i wielkie korporacje). Komputery te obsługiwane były przez grupę operatorów, którzy odbierali od programistów perforowane karty z programami, przenosili je do pomieszczenia gdzie znajdowały się czytniki kart i w ten sposób wprowadzali dane i program do komputera. Po wykonaniu obliczeń wyniki były drukowane lub zapisywane na kartach perforowanych, które były odnoszone przez operatorów programistom. Oczywiście ówczesne komputery mogły wykonywać tylko jeden program jednocześnie. Aby zoptymalizować czasowo proces ładowania kolejnych programów wymyślono więc **system wsadowy**. Kolekcja programów od różnych programistów była nagrywana na taśmę magnetyczną; kolejne programy rozdzielone były instrukcjami oznaczającymi początek i koniec programu, załadowanie kompilatora, załadowanie danych, uruchomienie programu itp. Program wykonywany na komputerze odczytywał te informacje z taśmy i wykonywał wskazane operacje. Był to prekursor dzisiejszych systemów operacyjnych. Często programy szeregowane były w ten sposób, że dane wyjściowe jednego programu były danymi wejściowymi kolejnego. Przykładowym komputerem, wykorzystującym ów system był mainframe[1] IBM 7094.

1 Mainframe – komputer używany dla krytycznych zastosowań o wysokiej wydajności przetwarzania danych i możliwości obsługi wysokiego obciążenia systemów wejścia/wyjścia niezależnych od głównego procesora.

Podwójny mainframe IBM 7094 w centrum NASA wykorzystywany w Projekcie Mercury. (źródło – Wikipedia).

W latach sześćdziesiątych wśród producentów komputerów pojawił się trend ujednolicania produkowanych przez nich linii sprzętu. Dotąd bowiem jednostki z różnych linii (mainframe, komputery do prostych zastosowań) były wzajemnie niekompatybilne, czyli niemożliwym było uruchomienie na nich tego samego programu bez zmiany jego struktury (w zasadzie napisania go od nowa). Z racji niewygody i dużych kosztów utrzymania tego typu rozwiązania, najpierw firma IBM, a za nią kolejni producenci zdecydowali się na ujednolicenie architekturalne swoich produktów. W ten sposób powstał IBM System/360, seria komputerów różniących się wydajnością i ceną ale tej samej architekturze. Komputery tej serii obsługiwane były przez system operacyjny OS/360. Najważniejszą cechą tego systemu było wprowadzenie możliwości wykonania (części lub całego) innego programu w czasie, kiedy bieżący program oczekiwał na zakończenie operacji wejścia wyjścia (np. pobrania danych z taśmy) zwane **multiprogrammingiem**. Oczywiście system operacyjny musiał zarządzać dostępem do komponentów komputera (szczególnie pamięci) aby wiele uruchomionych jednocześnie programów nie nadpisywało wzajemnie swoich danych. Zarządzanie to było wspomagane przez odpowiednie rozwiązania sprzętowe wprowadzone w serii System/360.

Komputer IBM System/360-20 z IBM 2560 MFCM (Multi-Function Card Machine) (źródło – Wikipedia).

Obecnie koncern IBM sprzedaje komputery mainframe pracujące pod kontrolą spadkobiercy OS/360 zwanego z/OS.

Sukces OS/360 i ogromne zainteresowanie multiprogrammingiem zapoczątkowały prace nad systemami operacyjnymi umożliwiającymi pełną wieloprocesowość. Zaowocowało to projektem systemu CTSS na uniwersytecie M.I.T, a następnie szeroko zakrojonym projektem MULTICS (1969r) realizowanym przez MIT, GE i Bell Labs. System ten był projektowany w celu umożliwienia korzystania z jednego komputera typu mainframe (GE 645) tysiącom użytkowników jednocześnie (w oryginalnym projekcie docelową grupą byli mieszkańcy miasta Boston). Projekt ten nie odniósł sukcesu komercyjnego, jednakże system był wykorzystywany przez takie jednostki jak NSA (Agencja Bezpieczeństwa Narodowego USA), firmy Ford i GM, Air Force Data Services

Center u USA, Industrial Nucleonics oraz Kanadyjski Departament Obrony Narodowej działający do 30 października 2000. System ten posiadał wiele rozwiązań, które zostały wykorzystane w SO następnej generacji i wykorzystywane są do dzisiaj.

Jeden z twórców systemu MULTICS postanowił wykorzystać swoje doświadczenia z pracy nad tym projektem i opracować system operacyjny umożliwiający jednoczesną obsługę dwóch klientów (początkowo tylko jednego) pracujący na prostym i tanim (rzędu 1/20 ceny mainframe) komputerze DEC PDP-7. System ten został nazwany UNIXem. Kod źródłowy systemu został upubliczniony co doprowadziło do powstania wielu odmian systemu dla przeróżnych platform sprzętowych. Najbardziej znane dystrybucje Unixa to BSD (dystrybuowany przez Uniwersytet Berkeley) i System V (sprzedawany przez AT&T). Popularność systemu i mnogość jego odmian wymusiła potrzebę standaryzacji grupy odwołań systemowych (system calls – rozdział...), który każdy z systemów deklarowany jako „kompatybilny z Unixem" musiał obsługiwać. Standard ten, wprowadzony przez IEEE nosi oficjalną nazwę IEEE 1003 / oraz ISO/IEC 9945. W świecie komputerów jest on bardziej znany pod nazwą POSIX[2]. Ze standardem tym zgodna jest większość współcześnie występujących systemów operacyjnych (A/UX, AIX, BSD/OS, Mac OS X, HP-UX, INTEGRITY, IRIX, LynxOS, MINIX, MPE/iX, QNX, RTEMS, Solaris, UnixWare, VxWorks). Częściowo zgodny z POSIXem jest m.in. Linux i FreeBSD.

Napisany w 1987 roku w celach edukacyjnych przez A. Tannenbauma system MINIX jest uproszczoną wersją UNIXa, wyróżniającą się architekturą opartą na jądrze monolitycznym (Patrz rozdział 2). Projekt ten, przeznaczony głównie dla studentów stał się źródłem inspiracji dla Linusa Torvaldsa, który w 1991 roku rozpoczął pracę nad swoim własnym systemem operacyjnym opartym na UNIXie i MINIXie nazwanym później Linuxem. Główną różnicą w początkowym stadium rozwoju Linuxa między tym systemem a MINIXem była architektura jądra, w Linuxie monolityczna. Nowy system był też dostępny za darmo dla wszystkich użytkowników, bez ograniczeń edukacyjnych.

W roku 1979 Steve Jobs zwiedzając laboratoria Xerox PARC dostrzegł potencjał opracowanego tam systemu komputerowego z graficznym interfejsem użytkownika (GUI). Xerox nie uznał tego projektu za ważny natomiast Jobs wkrótce rozpoczął pracę nad nowym systemem z GUI w swojej macierzystej firmie Apple Computers Inc. System ten został nazwany Apple Lisa, nie odniósł jednak sukcesu komercyjnego z racji wysokiej ceny i braku wystarczająco zróżnicowanego oprogramowania. Jobs, straciwszy na skutek wewnętrznych tarć dostęp do Apple Lisa rozpoczął pracę nad niskobudżetowym komputerem z GUI. Tak powstał Apple Macintosh. Te dwa komputery i ich systemy operacyjne z graficznym interfejsem stały się prekursorami współczesnych systemów operacyjnych. W obecnej chwili komputery Apple pracują pod kontrolą dedykowanego dla nich systemu Mac OS X opartego na UNIXie i projektach firmy NeXT, w której stworzono Macintosha.

Apple Macintosh

IBM 5150 PC

2 POSIX - Portable Operating System Interface [for Unix] – rozdział 3

W 1981 roku IBM wyprodukował komputer oznaczony symbolem 5150, znany szerzej jako IBM PC. Poszukując do niego oprogramowania, koncern zgłosił się do producenta interpretera języka BASIC, firmy Billa Gatesa Microsoft. Wkrótce po sprzedaży wspomnianego interpretera, Gates zaoferował IBM odkupiony od firmy Seattle Computer Products prosty system operacyjny DOS (Disk Operating System). Wraz z pojawieniem się GUI w Apple Lisa, Gates również dostrzegł potencjał drzemiący w graficznych interfejsach. W związku z tym Microsoft rozpoczął pracę nad graficzną nakładką na MS-DOS sprzedawaną pod nazwą Windows (w wersjach 1.0 – 3.11). Kolejna wersja systemu, Windows 95, 98 i ME wykorzystywał MS-DOS tylko w momencie uruchomienia i dla zachowania kompatybilności z programami DOS. Z kolei wydany w 1993 system Windows NT 3.1 został zaprojektowany od nowa, z pominięciem podstawy MS-DOS. System ten był w pełni 32 bitowy. Tą wersję zastąpiły kolejne odmiany NT (3.5, 4.0), następnie Windows 2000, XP, 2003, Vista, 7 i 2008 Server. Warte podkreślenia jest to, że o ile wczesne wersje NT były wydawane również na platformy PowerPC, DEC Alpha i MIPS R4000, to pozostałe wymienione systemy Microsoft są związane z architekturą x86 Intela. Wydawane są również wersje tych systemów na architekturę IA-64 czyli Intel Itanium.

W powyższym przeglądzie systemów operacyjnych skupiono się głównie na systemach kontrolujących komputery mainframe i PC. Od lat dziewięćdziesiątych równolegle rozwijane były również systemy operacyjne dla urządzeń przenośnych, takich jak palmtopy, telefony itp. (przykładowe systemy to Microsoft Windows CE, BlackBerry, Microsoft Windows Mobile, iPhone OS, Palm OS, Symbian, Android).

Rodzaje systemów operacyjnych

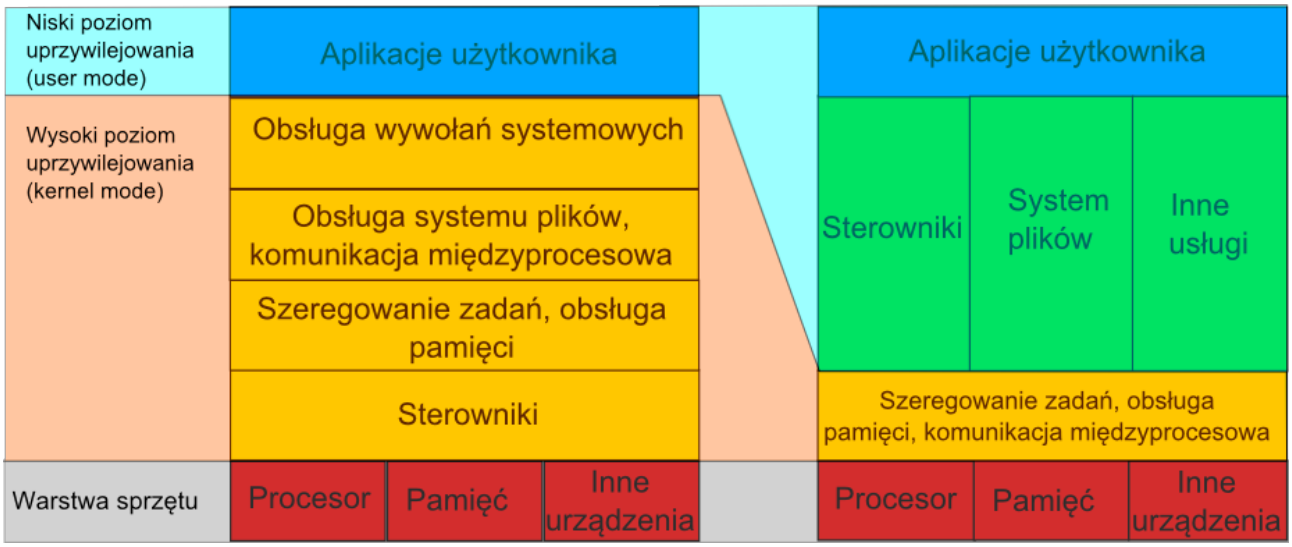
Każdy system operacyjny można podzielić na pewne warstwy, które realizują odpowiednie funkcjonalności takie jak warstwa powłoki (*shell*), której rolą jest przedstawienie interfejsu użytkownika, jądro realizujące funkcje systemowe (*kernel*) oraz warstwa komunikacji z urządzeniami fizycznymi systemu (tzw. *hardware abstraction layer*).

Systemy operacyjne można szeregować według wielu kryteriów. Pierwszym z nich jest przyjęta przez twórców **architektura jądra systemowego** (kernela). Podział ten bazuje na tym jak duża część operacji charakterystycznych dla systemu operacyjnego jest wykonywana w tzw. trybie jądra, a jak duża w trybie użytkownika. W trybie jądra (kernel mode) każda operacja dozwolona przez architekturę procesora może być wykonana (dowolna instrukcja, dowolna operacja wejścia/wyjścia, zapis lub odczyt dowolnego miejsca w pamięci). W trybie użytkownika niektóre z tych operacji są zabronione, przy czym wykonania niedozwolonej operacji w tym trybie zapobiega sprzętowe ograniczenie wbudowane w procesor. Podział systemów operacyjnych wg kryterium kernela przedstawia się następująco:

- monolityczne (wszystkie operacje SO wykonywane są w trybie jądra) – Linux, z/OS, Solaris
- mikrokernelowe (niektóre operacje wykonywane są w trybie jądra – partycjonowanie pamięci, przydzielanie zasobów, podstawowa komunikacja międzyprocesowa, pozostałe – komunikacja między aplikacjami, sterowniki urządzeń i systemy plików – w trybie użytkownika) – Amiga OS Classic / OS4, Minix
- nanokernelowe (tylko bardzo mała część operacji wykonywana jest w trybie kernela. Czasem tylko warstwa dostępu do urządzeń fizycznych wykonywana jest w tym trybie w celu dostarczenia funkcjonalności systemów czasu rzeczywistego do normalnych systemów

operacyjnych) - Adeos

- exokernelowe (minimalistyczny kernel umożliwiający uruchomienie tzw. maszyn wirtualnych, z których każda jest kopią fizycznego systemu z ograniczonymi zasobami. Rolą jądra jest odpowiedni przydział tych zasobów do poszczególnej wirtualnej maszyny i pośredniczenie wymianie informacji między nimi a fizycznymi urządzeniami) – Nemesis, ExOS
- hybrydowe (łączące cechy wielu architektur) – Mac OS X, Microsoft Windows
- warstwowe



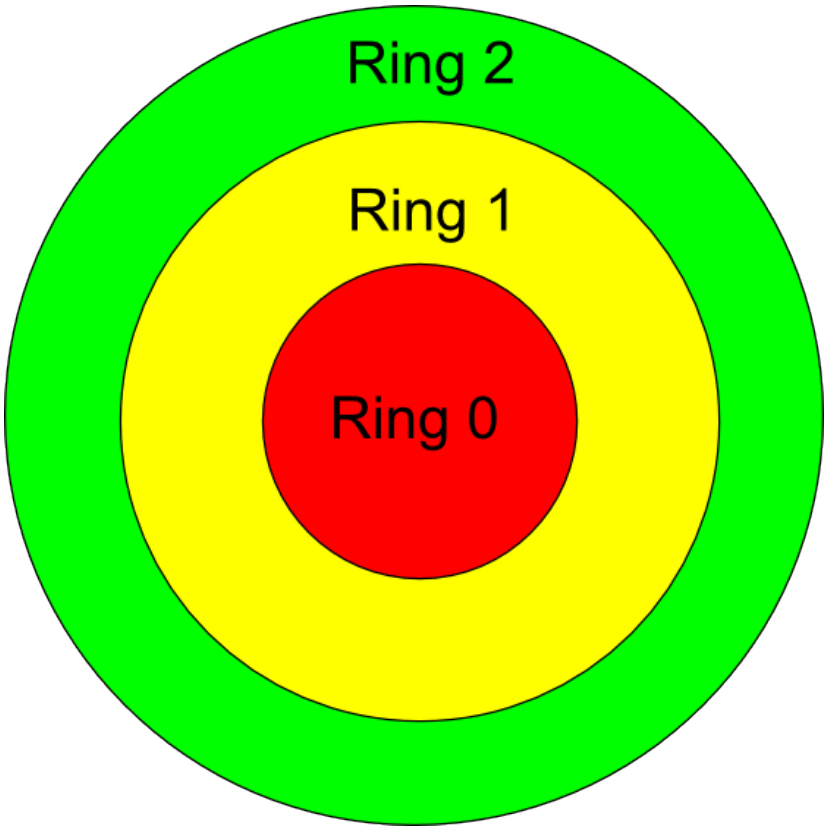
Schemat organizacji poziomów uprzywilejowania poszczególnych części systemu operacyjnego w systemie monolitycznym (z lewej) i mikrokernelowym (z prawej)

Podstawowymi zaletami minimalizacji liczby funkcjonalności pracujących w trybie jądra jest zmniejszenie wymagań dotyczących pamięci operacyjnej zajmowanej przez kernel oraz większe bezpieczeństwo systemu – błędy programowe zawarte w sterownikach itp. pracując w trybie użytkownika nie są w stanie naruszyć stabilności systemu operacyjnego ani zafałszować informacji przechowywanych przez inne aplikacje w pamięci. Wadą tego rozwiązania jest jednak zmniejszenie szybkości działania systemu, ponieważ sterowniki są traktowane jak zwykłe aplikacje (serwery usług).

Rozwinięciem idei podziału programów ze względu na „zaufanie" jakim można je „obdarzyć" jest mechanizm **systemu warstwowego** (***layer system***), czy też pierścieniowego (***protection rings*** – pojęcie to wprowadzono w systemie Multics, dla którego zdefiniowano osiem warstw uprzywilejowania). W tym ujęciu system operacyjny przedstawiony jest jako hierarchia warstw, lub zbiór koncentrycznych okręgów, obejmujących różne grupy oprogramowania i określających poziom uprzywilejowania (dostępu). W zależności od implementacji liczba poziomów hierarchii może być różna, w większości systemów jednak pierścień 0 jest najbardziej uprzywilejowaną warstwą programową, dla której dozwolona jest bezpośrednia współpraca z fizycznymi urządzeniami (procesor, pamięć operacyjna itp.). Na kolejnych poziomach znajdują się sterowniki urządzeń wejścia / wyjścia. Na najwyższym poziomie znajdują się zwykłe aplikacje.

W celu umożliwienia odwołań do programów znajdujących się na wyższym poziomie uprzywilejowania (np. ring 1) przez aplikacje z niższych poziomów (np. ring 3) wywoływana jest instrukcja **TRAP / GATE** (strukturalnie podobna do wywołania systemowego). Parametry

przekazywane do tej instrukcji przez aplikację są sprawdzane (często sprzętowo) i w przypadku kiedy weryfikacja przeprowadzona zostanie pomyślnie, aplikacja jest dopuszczana do oczekiwanej funkcji.



Schemat pierścieniowej organizacji systemu operacyjnego. Znajdujące się w centrum (Ring0) usługi mają największy stopień uprzywilejowania, kolejne warstwy – coraz mniejszy

Organizacja systemu operacyjnego w warstwy pozwala na uzyskanie bardzo dobrego stopnia zabezpieczenia przed konsekwencjami wykonywania niedozwolonych operacji przez aplikacje. Sprzętowe zabezpieczenie przed nieautoryzowanym dostępem do niższej warstwy pozwala na limitowanie nieautoryzowanego dostępu do wrażliwych warstw systemu. Ponadto, awaria oprogramowania na wyższym poziomie wpływa tylko na aplikacje pracujące na nim lub powyżej, podstawowe funkcje systemowe znajdujące się na niższych poziomach pozostają w pełni sprawne.

Niestety, jak przedstawiono powyżej, zastosowanie modelu warstwowego czy pierścieniowego wymaga współpracy pomiędzy systemem operacyjnym a fizycznym sprzętem (moduły sprawdzające autoryzację przejść itd.) co ogranicza przenośność systemu (możliwość uruchomienia go na różnych architekturach). Z tego powodu najczęściej systemy operacyjne używają dwóch warstw uprzywilejowania – jądra (*kernel*) i użytkownika.

Innym kryterium podziału systemów operacyjnych przeznaczonych dla komputerów jest konfiguracja urządzeń jakie system ów kontroluje. W tym wypadku podział wygląda następująco:

1. systemy jednostanowiskowe
2. systemy rozproszone
3. systemy klastrowe

4. systemy równoległe
5. systemy czasu rzeczywistego
6. systemy *embedded*

Ad1. Systemy jednostanowiskowe

Systemy jednostanowiskowe są systemami przeznaczonymi do obsługi komputerów osobistych. Obecnie w zasadzie każdy współczesny system operacyjny potrafi obsługiwać więcej niż jeden procesor. Rolą systemu jednostanowiskowego jest zapewnienie interfejsu dla pojedynczego użytkownika. Systemy te umożliwiają uruchamianie wielu programów jednocześnie. Przykładem oprogramowania należącego do tej grupy jest Microsoft Windows XP, Mac OS X i Linux, przy czym ten ostatni równie dobrze sprawdza się jako system serwerowy, zapewniając obsługę więcej niż jednego użytkownika.

Ad2. Systemy rozproszone

Jak wspomniano powyżej, system operacyjny nie musi być przypisany do pojedynczego urządzenia (komputera) a może kontrolować pewną ich grupę. Systemy komputerowe mogą składać się z zespołów urządzeń posiadających oddzielne procesory, zegary i pamięć (zarówno operacyjną jak i stałą). Komunikacja między tymi urządzeniami odbywa się najczęściej poprzez szybką sieć lokalną, w szczególnych przypadkach przez zwykłe sieci komunikacyjne. Często systemy operacyjne kontrolujące takie zespoły urządzeń są tworzone w ten sposób, że ich użytkownicy nie są świadomi tego, że korzystają z systemu składającego się z więcej niż jednego komputera. Przykładem systemu rozproszonego jest sieciowy system operacyjny Novell Open Enterprise Server, następca Novell Netware.

Ad3. Systemy klastrowe

System klastrowy jest pojęciem zawężającym pojęcie systemu rozproszonego. Kontroluje on pracę komputerów, które stanowią klaster, czyli zbiór powiązanych ze sobą komputerów, którego celem jest poprawienie wydajności, dostępności i odporności na awarię w stosunku do pojedynczego komputera. Najbardziej znaną klasą komputerów klastrowych jest klaster Beowulf będący zbiorem identycznych komputerów PC połączonych siecią Ethernet. Systemem operacyjnym kontrolującym pracę Beowulfa jest najczęściej Linux (np. ClusterKnoppix), BSD (Dragonfly BSD) lub Solaris wyposażone w specjalne oprogramowanie wspomagające przekazywanie komunikatów między poszczególnymi komputerami klastra, takie jak Open Source Cluster Application Resources (OSCAR).

Ad4. Systemy równoległe (symetryczne i asymetryczne)

Komputery z przetwarzaniem równoległym symetrycznym (***symmetric multiprocessing - SMP***) to urządzenia w których kilka (najczęściej 2 – 8) identycznych procesorów podłączonych jest do wspólnego systemu (pamięci). Termin ten obejmuje popularne ostatnio architektury takie jak Intel Core2 Duo oraz AMD Phenom. Taka organizacja pozwala na łatwe przenoszenie wykonywanych procesów pomiędzy poszczególnymi procesorami (rdzeniami) przez system operacyjny co pozwala na uzyskanie dobrego rozkładu obciążenia na wszystkie dostępne procesory. Asymetryczne przetwarzanie równoległe (***asymmetric multiprocessing – ASMP)*** pozwala na użycie różnych procesorów (często specjalizowanych) znajdujących się w komputerze do specyficznych zadań (przykładowym urządzeniem pracującym w trybie ASMP jest konsola do gier Sony PlayStation 3 z procesorem Cell Broadband Engine, składający się z głównego dwurdzeniowego

procesora Power ISA 2.03 oraz ośmiu specjalizowanych jednostkach SPE).

Systemy operacyjne pracujące na komputerach wieloprocesorowych / wielordzeniowych muszą uwzględniać możliwość rozmieszczania uruchomionych procesów na różnych jednostkach obliczeniowych. W zasadzie wszystkie dostępne obecnie OS mogą pracować na architekturach SMP.

Ad5. Systemy operacyjne czasu rzeczywistego

Specyficznym typem systemu operacyjnego jest tzw. system czasu rzeczywistego (**real-time OS - RTOS**). Jak wskazuje nazwa, kluczowym wymaganiem jest tutaj wykonywanie zadań w określonym reżimie czasowym, przy czym sam system operacyjny nie jest w stanie w każdym wypadku zagwarantować pracy w czasie rzeczywistym do czego wymagane jest odpowiednie zaprojektowanie uruchamianych na nim aplikacji. Istnieją dwa rodzaje systemów RT – soft i hard real time OS. Różnica między tymi systemami polega na tym, że w systemach hard zachowanie systemu (łącznie z czasami i kolejnością wywołań) jest w pełni zdeterminowane, to znaczy zawsze możliwe do określenia. W systemach soft RT wymagania co do determinizmu zdarzeń są nieco obniżone, choć zwykle można określić czas i kolejność wywołań.

Głównymi środkami do uzyskania systemów RT są specjalizowane algorytmy planowania wykonania wątków (*scheduling algorithms*) a także minimalizacja opóźnienia wprowadzanego przez przełączanie wątków oraz przerwania.

Systemy RTOS wykorzystywane są m.in. w urządzeniach przemysłowych (SCADA), robotach i kontrolerach, w urządzeniach do edycji dźwięku (zwykle soft RTOS). Przykładowymi systemami RT są QNX i RTLinux.

Zagadnienia charakterystyczne dla systemów czasu rzeczywistego zostały omówione w rodziale 3.

Ad6. Systemy operacyjne typu wbudowanego (*embedded*)

Systemy typu *embedded* są przeznaczone do kontrolowania prostych urządzeń takich jak palmtopy, telefony, konsole nawigacji GPS itp. Częstokroć systemy te zaprojektowane są podobnie jak systemy czasu rzeczywistego z uwzględnieniem zwykle małej wydajności sprzętowej urządzeń dla których są przeznaczone. Funkcjonalność tych systemów zazwyczaj jest minimalna co pozwala na minimalizację zajmowanej przez nie pamięci operacyjnej i innych zasobów. Przykładowymi systemami operacyjnymi typu wbudowanego są Embedded Linux, Minix3, VxWorks, Windows CE, PalmOS, Windows Mobile, Android, iPhone OS, DD-WRT (system operacyjny sterujący pracą access pointów Linksys), Cisco IOS, BrickOS (LEGO Mindstorms), Robotic Operating System.

Budowa systemów operacyjnych

Jak wspomniano wcześniej klasyczny system operacyjny składa się z kilku części / warstw. Podstawową jego elementem jest jądro czyli część oprogramowania zarządzająca kluczowymi składnikami systemu.

Jądro

Jądro w większości systemów zarządza przydzielaniem poszczególnym procesom prawa do wykorzystania procesora (tzw. scheduling), pamięci ulotnej (RAM), pamięci stałej (dysków twardych, stacji dyskietek itd.) wejść i wyjść systemowych a także przechwytywaniem przerwań i odpowiednią ich obsługą oraz komunikacją międzyprocesową. Jądro umożliwia dostęp do wymienionych elementów procesom (programom) poprzez tzw. odwołania systemowe (**system calls**).

Procesy

Podstawowym pojęciem wykorzystywanym w ramach organizacji systemów komputerowych jest proces (**process**).Przyczyną powstania tego pojęcia była idea multiprogrammingu wprowadzona w System/360 IBM i CTSS a rozwinięta w systemie MULTICS. W systemach nie umożliwiających uruchomienia więcej niż jednego programu naraz nie istniał problem rozdzielania dostępu do poszczególnych urządzeń składowych komputera różnym programom, nie było też potrzeby zatrzymywania i uruchamiania kolejnych procesów. Wprowadzenie możliwości uruchomienia i wykonywania więcej niż jednego programu jednocześnie spowodowało konieczność rozwiązania przedstawionych wyżej zagadnień. Przez proces rozumiana jest każda instancja dowolnego programu uruchomionego w systemie. Instancja ta składa się z kodu maszynowego wykonywanego programu, danych stowarzyszonych z programem – stosu[3], sterty[4] oraz innych, odwołań do zasobów przydzielonych przez system operacyjny np. uchwytów (**handle**) w Microsoft Windows i kontekstu wywołania **(execution context)**. Kontekst ten, inaczej zwany stanem procesu (**process state**) to zestaw danych, dzięki którym SO jest w stanie nadzorować wykonywaniem programu. Do kontekstu należy zawartość rejestrów procesora (licznika programu i rejestrów danych), priorytet danego procesu oraz informacja o tym czy proces oczekuje na wystąpienie określonego zdarzenia (**event**). Każdy proces charakteryzuje się jednocześnie stanem w jakim się znajduje (utworzony, oczekujący na uruchomienie, uruchomiony, zablokowany / oczekujący na zdarzenie, oczekujący) [SILBERSCHATZ].

Wątki

Wątki często nazywane są lekkimi procesami co dobrze oddaje ich naturę. Wątek (**thread**) jest fragmentem programu, który jest wykonywany oddzielnie (w sensie czasowym), równolegle z

3 Stos – (ang. *stack*) konstrukcja bufora danych oparta na zasadzie kolejki LIFO (*last in first out*) służąca do przechowywania m.in. rejestrów procesora.

4 Sterta – (ang. *heap*) obszar pamięci zarezerwowany dla danego procesu przez SO służący do przechowywania zmiennych alokowanych dynamicznie

pozostałą częścią programu używającym tej samej przestrzeni adresowej (danych). W związku z tym wątek w systemie operacyjnym nie musi być charakteryzowany przez taki sam zestaw danych jak proces – wystarczającymi elementami są licznik programu, rejestry procesora, stos i stan.

Główną przyczyną powstania pojęcia wątku, podobnie jak w przypadku procesów potrzeba wykonywania różnych czynności przez program jednocześnie. O ile część tego typu zagadnień może być rozwiązana przy użyciu procesów, to niekoniecznie rozwiązanie to musi być korzystne ze względu na wydajność aplikacji. W wielu zastosowaniach niezbędna jest wymiana danych pomiędzy różnymi częściami wykonywanego programu. W wypadku procesów, wymiana ta wiąże się z transportem często dużych porcji danych pomiędzy oddzielnymi przestrzeniami adresowymi (co jest wykonalne na różne sposoby, ale bardzo często nieefektywne). Wątki natomiast pracują w tej samej przestrzeni adresowej i używają współdzielonych danych. Oczywiście to rozwiązanie wymaga poprawnego zarządzania synchronizacją danych przez programistę (aby np. jeden wątek nie nadpisał bloku pamięci, która jest czytana przez inny), niemniej w większości zastosowań jest to bardzo korzystne.

Ze względu na ograniczoną ilość danych stowarzyszonych z wątkiem (niezbędną do sterowania nim przez system operacyjny) tworzenie i niszczenie wątków jest wielokrotnie szybsze niż tworzenie procesów, niejednokrotnie różnica ta jest kilkudziesięciokrotna. W niektórych przypadkach (kiedy program wielokrotnie tworzy i niszczy dużą liczbę wątków / procesów – np. serwery baz danych, WWW) zastosowanie lżejszych wątków prowadzi do znacznej poprawy wydajności. Nie należy również zapominać o możliwości wykonywania poszczególnych wątków na różnych procesorach (jeżeli system jest wyposażony w więcej niż jeden CPU) co zdecydowanie przyspiesza wykonanie programu.

Zarządzanie procesami

System operacyjny przechowuje informacje o uruchomionych procesach w specjalnym miejscu pamięci operacyjnej zwanym tablicą procesów (**process table**). Dla każdego procesu przechowywana jest tam informacja o jego stanie, zwartość jego licznika rozkazów, dane na temat alokacji pamięci operacyjnej i plikach otwartych przez proces. Przechowywany jest również wskaźnik stosu oraz informacja *schedulera*[5]. Dzięki tym informacjom SO może w dowolnym momencie zatrzymywać i uruchamiać procesy, chronić dane obszary pamięci co pozwala na uruchamianie wielu procesów równolegle. Oczywiście procesy te nie są wykonywane jednocześnie (uogólniając – w systemie może jednocześnie działać tyle procesów ile procesorów jest w nim fizycznie obecnych). W systemie, w którym uruchomione jest więcej procesów niż procesorów jądro systemu operacyjnego wykonuje je na przemian, uruchamiając każdy z procesów na krótki okres czasu (rysunek). Czasy przełączania oraz wykonywania poszczególnych fragmentów procesów są na tyle krótkie, że dla użytkownika systemu wykonywanie procesów wydaje się być jednoczesne. Opisany mechanizm przemiennego wykonywania programów wykonywany jest przez systemy operacyjne z tzw. wywłaszczającą wielozadaniowością (**preemptive multitasking**). W tym trybie działają wszystkie współczesne systemy obsługujące komputery (Microsoft Windows, Linux, Mac OS X). Pierwsze wersje Windows i Mac OS X, a także specjalizowane systemy operacyjne przeznaczone głównie dla prostych komputerów pracują w trybie wielozadaniowości współpracującej (**cooperative multitasking**). Różnica w działaniu polega na tym, że w przypadku uruchomienia wielu procesów, pierwszy z nich jest wykonywany ciągle do momentu przejścia przez niego w tryb oczekiwania na zdarzenie lub komunikat od innego procesu. W tym momencie uruchamiany jest kolejny proces, który jest znowu wykonywany do momentu przejścia w stan oczekiwania.

Planowanie kolejki uruchomiania poszczególnych fragmentów procesów lub wątków jest

5 Nie mam pojęcia jak w polskiej terminologii nazywa się scheduler

bardzo skomplikowanym zadaniem. Część systemu operacyjnego odpowiedzialna za tą funkcjonalność nazywana jest *schedulerem*. W zależności od przeznaczenia systemu operacyjnego (np. dla systemu operacyjnego, serwera) algorytmy sterujące wykonaniem procesu mogą być różne, bardziej lub mniej skomplikowane. Generalnie *scheduler* ma za zadanie:

- wybranie odpowiedniego procesu do uruchomienia (w większości systemów operacyjnych procesy mogą mieć różne priorytety, oznaczające ich istotność w systemie i warunkujące w pewnym stopniu kolejność ich uruchamiania)
- zapewnienie jak najlepszego wykorzystania czasu procesora
- minimalizację czasu jaki procesy oczekują na uruchomienie
- minimalizację liczby procesów oczekujących (czyli możliwie szybkie zakończenie procesów uruchomionych)
- przydział takiego samego czasu procesora każdemu procesowi (z uwzględnieniem jego priorytetu)
- w szczególnych przypadkach (systemy czasu rzeczywistego) zapewnienie wykonania procesu przed upłynięciem wyznaczonego czasu
- maksymalizację wykorzystania urządzeń systemowych

Dodatkowo, ze względu na mnogość operacji, które są wykonywane przy przełączaniu procesów (zmiana trybu pracy do trybu jądra, zapisanie bieżącego stanu procesu, zachowaniem mapy pamięci przypisanej procesowi, wczytanie tych danych dla nowo uruchamianego procesu, , jest to operacja kosztowna czasowo i z tego względu powinna być wykonywana najrzadziej jak to możliwe co z kolei stoi często w konflikcie z wymienionymi powyżej zadaniami *schedulera*.

W systemie operacyjnym istnieje kilka elementów zaangażowanych w zarządzanie procesami. Jest to szereg kolejek (list) procesów oczekujących na pewne zasoby. Kolejki te to tzw. kolejka krótkoterminowa (**short term queue**), kolejka długoterminowa (**long term queue**) oraz kolejki obsługi urządzeń wejścia / wyjścia (**I/O queues**). Poza wspomnianymi kolejkami system operacyjny posiada odbiornik[6] odwołań systemowych (**service call handler**) oraz odbiornik przerwań. Innym elementem jest tzw. *scheduler* wybierającym proces do uruchomienia oraz *dispatcher* którego zadaniem jest przekazywanie kontroli nad systemem aktualnie wybranemu procesowi.

Kolejki obsługi urządzeń wejścia / wyjścia służą do szeregowania odwołań poszczególnych procesów do konkretnych urządzeń. Procesy, które chcą odwołać się do tego samego urządzenia są umieszczane w przypisanej do niego kolejce, a system operacyjny decyduje o kolejności ich dopuszczeń.

Długoterminowa kolejka zadań to lista procesów rozpoczynających uruchamianie. System operacyjny na tym etapie przydziela startującemu procesowi oczekiwany zakres pamięci RAM sprawdzając jednocześnie czy ten zakres nie przekracza dostępnej w systemie pojemności. Po udanym uruchomieniu procesu przekazywany jest on do kolejki krótkoterminowej.

Kolejka krótkoterminowa to lista procesów uruchomionych (znajdujących się w pamięci operacyjnej), które w każdej chwili mogą wykonać się na procesorze. To, który proces zostanie wykonany określa algorytm *schedulingu*.

Odbiorniki przerwań i odwołań systemowych służą do przekazania kontroli procesora systemowi operacyjnego w momencie wystąpienia któregoś z przerwań lub wykonaniu przez dowolny proces odwołania systemowego (**system call**).

6 Nie wiem jak w polskiej nomenklaturze tłumaczy się pojęcie handler

Istnieją różne typy algorytmów *schedulingu*, stosowane w zależności od przeznaczenia systemu operacyjnego. Poniżej zostaną przedstawione trzy z nich:

1. Systemy wsadowe

Jak wspomniano w pierwszym rozdziale tego podręcznika pierwsze systemy komputerowe pracowały w trybie wsadowym, wykonując jednocześnie jeden program i przekazując (najczęściej) jego wyniki kolejnemu programowi jako dane wejściowe. Z racji wykonywania tylko jednego procesu jednocześnie planowanie wykonania zarządzanie procesami było bardzo proste i ograniczone w zasadzie do wyboru kolejnego procesu do uruchomienia. Podstawowym algorytmem jest algorytm obsługujący procesy według kolejności ich zgłoszenia, według zasady FIFO (**first in, first out**). Jego największą zaletą jest prostota działania. W niektórych jednak przypadkach tego typu planowanie uruchomień jest nieefektywne, szczególnie jeżeli zadania oczekujące w kolejce wymagają różnego czasu obliczeń. W celu zapobiegania temu problemowi w systemach wsadowych występują również algorytmy *schedulingu* planujące kolejność zadań według czasu wymaganego do wykonania procesu (**shortest job first**). W ten sposób wynik działania prostych, krótkotrwałych procesów uzyskiwany jest możliwie szybko. Niestety, niezbędnym wymaganiem tego typu planowania jest znajomość czasu wykonania danej procedury, co często, ale nie zawsze jest możliwe do określenia. Rozwinięciem algorytmu *shortest job first* jest algorytm ***shortest remaining time next***. Polega on na tym, że w momencie przybycia nowego procesu do listy oczekujących na uruchomienie czas potrzebny do jego wykonania jest porównywany z czasem, który pozostał do wykonania bieżącego procesu. Jeżeli ten pierwszy jest krótszy, to proces bieżący zostaje zawieszony a nowo przybyły – uruchomiony.

2. Systemy czasu rzeczywistego

W systemach tego typu zachowanie „czasu rzeczywistego” jest osiągane poprzez podzielenie programu na fragmenty (procesy), dla których zachowanie systemu jest przewidywalne co pozwala na określenie czasu niezbędnego do ich wykonania. Wykonywanie procesów jest tak planowane aby w momencie wystąpienia zewnętrznego zdarzenia na które system ma reagować, były one wykonywane tak, aby założone ramy czasowe działania zostały zachowane.

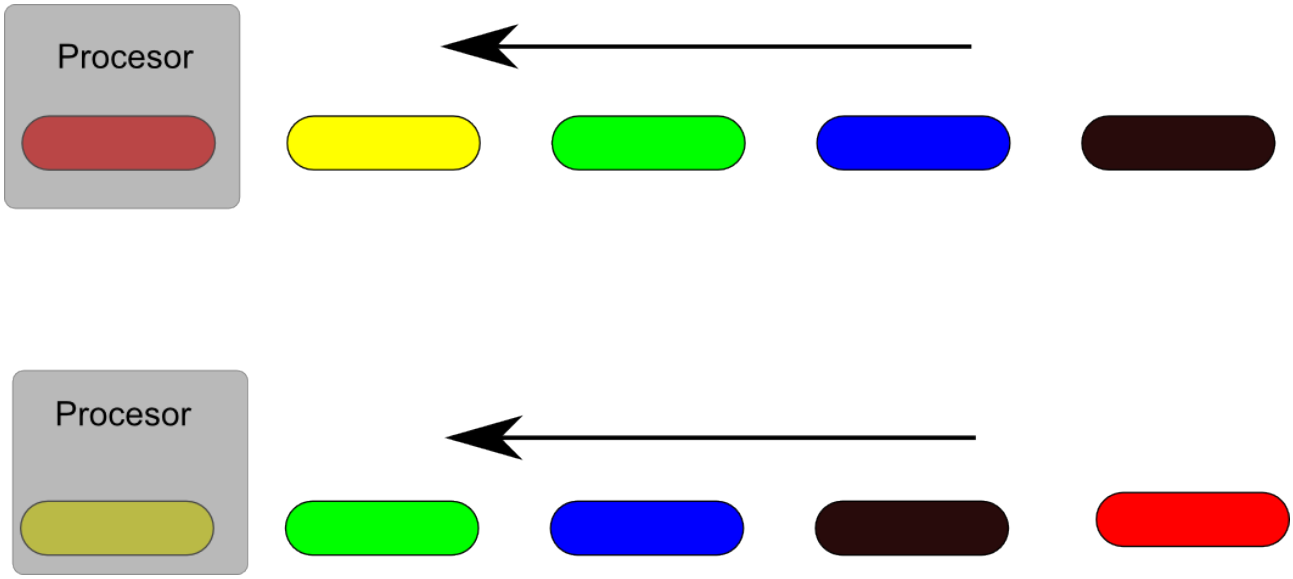
Zwykle algorytmy *schedulingu* w systemach RT są implementowane na zasadzie wywłaszczania (czyli przełączania między różnymi, jednocześnie uruchomionymi procesami), przy czym różne procesy mogą mieć określone różne priorytety określające istotność ich wykonania. W prostych systemach używane są również algorytmy szeregujące zadania według czasu jaki pozostał do momentu, w którym muszą one zostać zakończone. Tego typu algorytmy działają poprawnie tylko w wypadku zapewnienia obciążenia systemu poniżej 100% to znaczy wtedy, kiedy częstotliwość występowania zdarzeń wywołujących dany proces w systemie jest mniejsza niż czas konieczny do reakcji systemu na pojedyncze zdarzenie.

Najbardziej zaawansowanym algorytmem planującym wykonywanie zadań w RTOS jest algorytm ***Adaptive partition scheduler*** [APS-QNX] zaimplementowany np. w systemie QNX. System ten, w przypadku nie występowania przeciążeń procesora (częstszych wywołań zdarzeń) pracuje tak jak system z planowaniem *pre-emptive*. Na wypadek wystąpienia przeciążeń, projektant systemu określa jednak pulę procesów, dla których zarezerwowana jest jakaś część (np. 10%) zasobów systemowych. W momencie wystąpienia przeciążenia *scheduler* gwarantuje procesom z tej puli przynajmniej 10% czasu procesora i innych zasobów systemowych zapewniając im wykonanie w założonym czasie, mniej istotne procesy przesuwając dalej w czasie.

3. Systemy generalnego użycia (interaktywne)

Systemy tej grupy obejmują największą grupę systemów operacyjnych dla komputerów ogólnego użycia, serwerów itd. Z racji szerokiego spektrum zastosowań opracowano wiele algorytmów *schedulingu*. Omówione zostaną najpopularniejsze z nich [TANNENBAUM] - *round robin, priority scheduling, multilevel feedback queues, lottery O(1)* i *completely fair scheduler*. Ich nazwy zostały podane w języku angielskim, bo nie istnieją dobre odpowiedniki nazw wszystkich tych algorytmów w języku polskim.

Podstawowym algorytmem kolejkowania procesów jest algorytm ***round robin***. Polega on na uruchamianiu poszczególnych zadań według kolejności umieszczenia ich w kolejce krótkoterminowej przy czym wykonanie procesu przerywane jest po upłynięciu czasu zwanego kwantem. Po przerwaniu działania proces jest umieszczany na samym końcu listy procesów oczekujących na uruchomienie. W przypadku zakończenia procesu przed upłynięciem całego kwantu system uruchamia kolejny proces z listy.



Schemat szeregowania zadań wg algorytmu round-robin. Po wywłaszczeniu procesu symbolizowanego przez czerwony segment (powyżej) jest on przesuwany na ostatnie miejsce kolejki, a jego miejsce zajmuje kolejny w kolejce proces (poniżej)

Istotnym zagadnieniem w przypadku *schedulera round robin* jest dobór czasu uruchomienia fragmentów procesu – kwantu. Zbyt krótki czas powoduje słabe wykorzystanie procesora, ponieważ spędza on dużo czasu na przełączaniu się między procesami. Z punktu widzenia wydajności najkorzystniejszy jest możliwie długi kwant (wtedy system dąży do działania systemu wsadowego), jednak w systemach interaktywnych może on spowodować duże opóźnienia działania poszczególnych procesów co jest niedopuszczalne z punktu widzenia użytkownika. Przeciętnie w systemach operacyjnych wartość kwantu ustawiana jest na poziomie 20 – 50 milisekund.

Dużą zaletą *round robin* jest nie występowanie zjawiska **głodu**[7] [DIJKSTRA-DPP] czyli nieskończonego (czy bardzo długiego) odmawiania jakiemuś procesowi dostępu do procesora. W podstawowej implementacji wszystkie procesy są jednakowo traktowane i uruchamiane kolejno.

7 W angielskiej nomenklaturze zjawisko to nazywane jest starvation

Większość użytkowników komputerów osobistych zauważyła z pewnością że wskazane jest aby niektóre zadania wykonywane były szybciej niż inne, które mogą być przesunięte w czasie. Prostym przykładem jest reakcja komputera na ruch myszy czy naciśnięcie klawisza – jeżeli system reaguje na te zdarzenia z widocznym opóźnieniem to komfort pracy jest znacznie obniżony, a użytkownik narzeka na „wolny komputer". Jednocześnie to czy np. defragmentacja dysku lub indeksowanie jego zawartości działające w tle wykona się wolniej czy szybciej jest z punktu widzenia użytkownika nieistotne. Wynika z tego wniosek, że każde zadanie wykonywane przez komputer powinno mieć swój priorytet aby uszeregować je w zbiorze innych, równolegle wykonywanych. Aby wykorzystać informację o priorytecie procesu podstawowy algorytm *round robin* musi zostać nieco zmodyfikowany. W tym celu stosuje się organizację uruchomionych procesów w szereg kolejek, jednej dla każdego priorytetu (***priority scheduling***). W ramach każdej kolejki szeregowanie zadań odbywa się według zasad zwykłego algorytmu *round robin,* natomiast numer kolejki do uruchomienia dobierany jest tak aby faworyzowane były te z wyższym priorytetem. Wiąże się z tym zagrożenie wstrzymywania zadań niskopriorytetowych w nieskończoność (*starving*). Z tego powodu często stosuje się dynamiczną regulację priorytetów procesów, często bazującą na ilości odwołań do systemów wejścia / wyjścia w danym procesie lub czasie oczekiwania na uruchomienie (priorytet długo oczekujących procesów jest zwiększany)

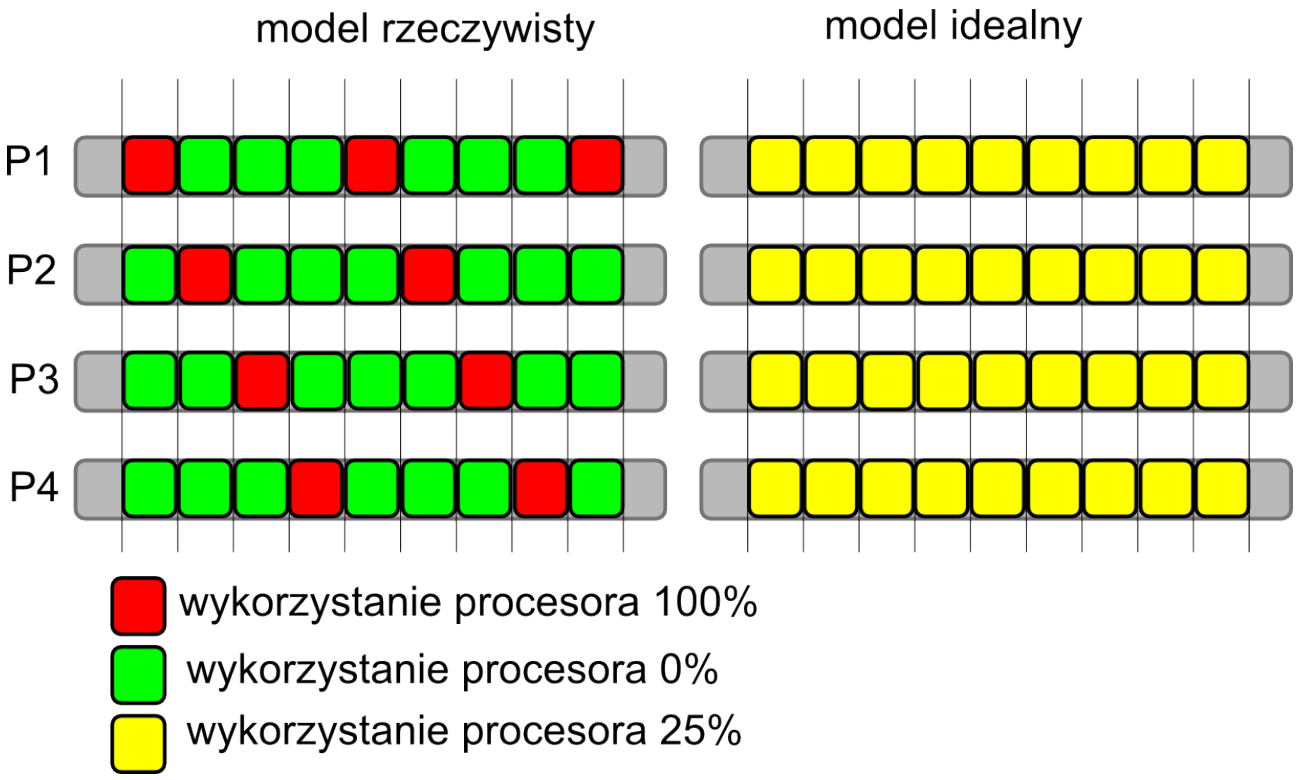
Rozwinięciem idei *priority schedulingu* jest algorytm ***multi-level feedback queues***. Podobnie jak w przypadku szeregowania priorytetowego tutaj także procesy umieszczane są w kilku listach tworzonych dla każdego poziomu priorytetów. Każda z kolejek może mieć jednak inny model zarządzania (niekoniecznie *round robin*, a na przykład *shortest job first*) oraz wielkość kwantu czasu procesora. Generalną zasadą jest otrzymywanie krótkich kwantów przez procesy o wysokim priorytecie. Nowy proces otrzymuje wysoki priorytet i jest uruchamiany na bardzo krótki czas. Jeżeli w tym fragmencie wykonania proces nie wykonuje odwołań blokujących (czyli CPU jest zajęty przez cały kwant), to priorytet procesu jest obniżany, zwiększany za to jest jego kwant. Jeżeli proces odwołuje się do urządzeń wejścia / wyjścia to jego priorytet jest podnoszony. W efekcie najszybciej wykonywane są krótkotrwałe zadania, a także takie, które często odnoszą się do urządzeń wejścia / wyjścia (np. wprowadzanie danych przez użytkownika).

Zupełnie innym podejściem do procesu uruchamiania procesów jest algorytm loterii (**lottery***) [WALDSPURGER]*. Polega on na tym, że każdy z procesów uruchomionych w systemie otrzymuje pewną liczbę tzw. losów, przy czym liczba ta jest ekwiwalentem priorytetów zadań w systemach kolejkowych. W każdym momencie decyzji SO o zmianie wykonywanego procesu (czyli upłynięciu kwantu czasu) dokonywane jest losowanie numeru losu. Proces, który posiada los o danym numerze jest wykonywany w kolejnym cyklu pracy. W ten sposób w łatwy sposób zapewniony jest przydział określonej „części procesora" procesowi – w odpowiednio długim czasie proces posiadający 33% losów wykorzysta procesor przez 33% całkowitego czasu pracy. W odpowiednich warunkach procesy współpracujące mogą wymieniać między sobą losy zwiększając szybkość wykonania programu.

System operacyjny Linux w wersjach jądra od 2.6 do 2.6.23 wykorzystywał *scheduler* nazywany O(1). Cechą charakterystyczną (od której zresztą algorytm ten wziął nazwę) było to, że czas planowania uruchomienia dowolnego procesu był niezmienny, bez względu na liczbę procesów uruchomionych w systemie, co było osiągnięciem bardzo istotnym, zwłaszcza w przypadku zastosowań w systemie czasu rzeczywistego (niezmienność czasu wyznaczania *schedulingu* w dużym stopniu poprawia determinizm wykonywanych przez system czynności). Zasada działania tego algorytmu opiera się na dwóch tablicach procesów – aktywnych i wygasłych (**active, expired**). Uruchomiony proces (znajdujący się w tablicy aktywnej) wykonywany jest przez określony czas (zdefiniowany w systemie) po czym jest on usypiany, a informacja o nim

przeniesiona z tablicy procesów aktywnych do uśpionych. W ten sposób tablica aktywna jest opróżniana i w momencie kiedy staje się pusta zostaje podmieniona tablicą procesów uśpionych, która z kolei zostaje zamieniona tablicą aktywną. Algorytm ten automatycznie reguluje również priorytet danego procesu, analizując jego czasy aktywności (oczekiwania na dane), przyjmując że ten proces, który czeka długo jest procesem związanym z interfejsem użytkownika, który dla zachowania komfortu obsługi powinien mieć wysoki priorytet. Zadania obciążające procesor w dużym stopniu i nieoczekujące na dane mają ustawiany niski priorytet. Niestety użyta metoda identyfikacji zadań oczekujących na dane (opierająca się na heurystyce) powoduje potrzebę wykonywania dużej liczby obliczeń (niepotrzebnie obciążających procesor) dodatkowo często dając niewłaściwe wyniki.

Z powodu wyżej wymienionych wad opracowano nowy algorytm planowania procesów, zwany CFS czyli ***completely fair scheduler*** (całkowicie sprawiedliwy *scheduler*). „Sprawiedliwość" oznacza tutaj przydzielanie każdemu procesowi jednakowej, wynikającej z liczby procesów, mocy obliczeniowej procesora symulując działanie „idealnie wielozadaniowego procesora" (cyt. Ingo Molnar, autor CFS).



Przydział procesora procesom (P1, P2, P3, P4) w sytuacji rzeczywistej i idealnej (procesor wykorzystywany w takim samym stopniu przez wszystkie procesy w każdej jednostce czasu)

Założenia CFS realizowane są poprzez śledzenie całkowitego czasu oczekiwania na procesor (**wait_runtime**), który minął od rozpoczęcia procesu. Czas ten, dzielony przez współczynnik wynikający z liczby równolegle uruchomionych procesów oraz priorytetu analizowanego procesu. Proces, którego czas oczekiwania jest największy, otrzymuje dostęp do procesora, a jego wait_runtime jest odpowiednio zmniejszany. W momencie kiedy jakikolwiek proces ma współczynnik oczekiwania większy niż bieżący, następuje zmiana przydziału procesora. Tego typu szeregowanie zadań pozwala na rezygnację z zaawansowanych i czasochłonnych algorytmów heurystycznych jak w *schedulerze* O(1).

Algorytm CFS zamiast standardowych kolejek zadań do organizacji listy zadań używa tzw. struktury binarnych, czerwono-czarnych drzew. Struktura ta, pomimo pewnych trudności

implementacyjnych, pozwala efektywnie wykonywać najczęściej wykonywane operacje (wstawienie, usunięcie, przeszukanie) w czasie O(log n), gdzie n – liczba zadań w strukturze.

Interesującą właściwością algorytmu CFS jest to, że czas, na który procesor przypisywany jest poszczególnemu procesowi wynika z liczby aktualnie uruchomionych procesów i jest określany dynamicznie.

Nowoczesne systemy operacyjne używają różnych technik zarządzania procesami. Krótkie podsumowanie znajduje się w tabeli poniżej:

System operacyjny	Wywłaszczenie (preemption)	Algorytm
Microsoft Windows 3.1x	Brak	Wielozadaniowość kooperatywna
Microsoft Windows 95,98,ME	Częściowe	Podstawowy, tylko dla operacji 32 bitowych (prawdopodobnie *priority scheduling*)
Microsoft Windows NT,XP,Vista	Tak	*Multilevel Feedback Queue*
Mac OS (przed wersją 9)	Brak	Wielozadaniowość kooperatywna
Mac OS X	Tak	Mach (kernel)[8]
Linux przed wersją kernela 2.5	Tak	*Multilevel Feedback Queue*
Linux 2.5-2.6.23	Tak	*O(1)*
Linux po wersji kernela 2.6.23	Tak	*Completely Fair Scheduler*
Solaris	Tak	*Multilevel feedback queue*
NetBSD	Tak	*Multilevel feedback queue*
FreeBSD	Tak	*Multilevel feedback queue*

Źródło: Wikipedia

Porównanie mechanizmów zarządzania procesami w Microsoft Windows i Linuxie [STALLING]:

System	Microsoft Windows	Linux
Algorytm zarządzania procesami	O(1), z listami priorytetów dla każdego z procesorów (*Multilevel feedback queue)*	Do wersji 2.6.23 O(1), z listami priorytetów dla każdego z procesorów, po 2.6.23 - CFS
Priorytety zadań	16 poziomów zadań (bez czasu rzeczywistego), przy czym poziom 0 odpowiada najmniej ważnemu zadaniu.	40 poziomów priorytetów (bez czasu rzeczywistego), większy numer oznacza mniejszą ważność procesu.

8 Konstrukcja jądra Mach jest inna niż standardowych kerneli systemów operacyjnych. Z tego powodu *scheduling* jest tam rozwiązany w odmienny sposób. Więcej informacji na np. http://en.wikipedia.org/wiki/Mach_(kernel)

	16 poziomów zadań czasu rzeczywistego z pierwszeństwem w stosunku do zadań poprzedniej grupy. Zadania te są szeregowane wg algorytmu *round-robin*	99 poziomów priorytetów dla zadań czasu rzeczywistego, z pierwszeństwem w stosunku do zadań z poprzedniej grupy. Zadania RT są szeregowane według algorytmu *round-robin* lub FIFO (w tym wypadku zadanie jest wykonywane przez procesor bez przerywania do momentu naturalnego zakończenia lub nadejścia procesu o większym priorytecie)
Kontrola użytkownika	Procesy mogą być przypisywane do procesorów, przy czym *scheduler* wybiera procesor tak, aby zoptymalizować wydajność pamięci cache. Wątki są przenoszone na inne procesory jeżeli są one wolne lub wykonują zadania o niskim priorytecie	-
Postępowanie ze zjawiskiem głodu	Automatyczne zwiększanie priorytetu zadaniom, które długo oczekują na przydział procesora	Procesy o niskim priorytecie są uruchamiane przed ważniejszymi zadaniami pod warunkiem, że te wyczerpały swój kwant przydziału do procesora. Pozwala to na uniknięcie zbyt długiego czasu oczekiwania na procesor przez procesy o bardzo niskim priorytecie

Zarządzanie pamięcią operacyjną

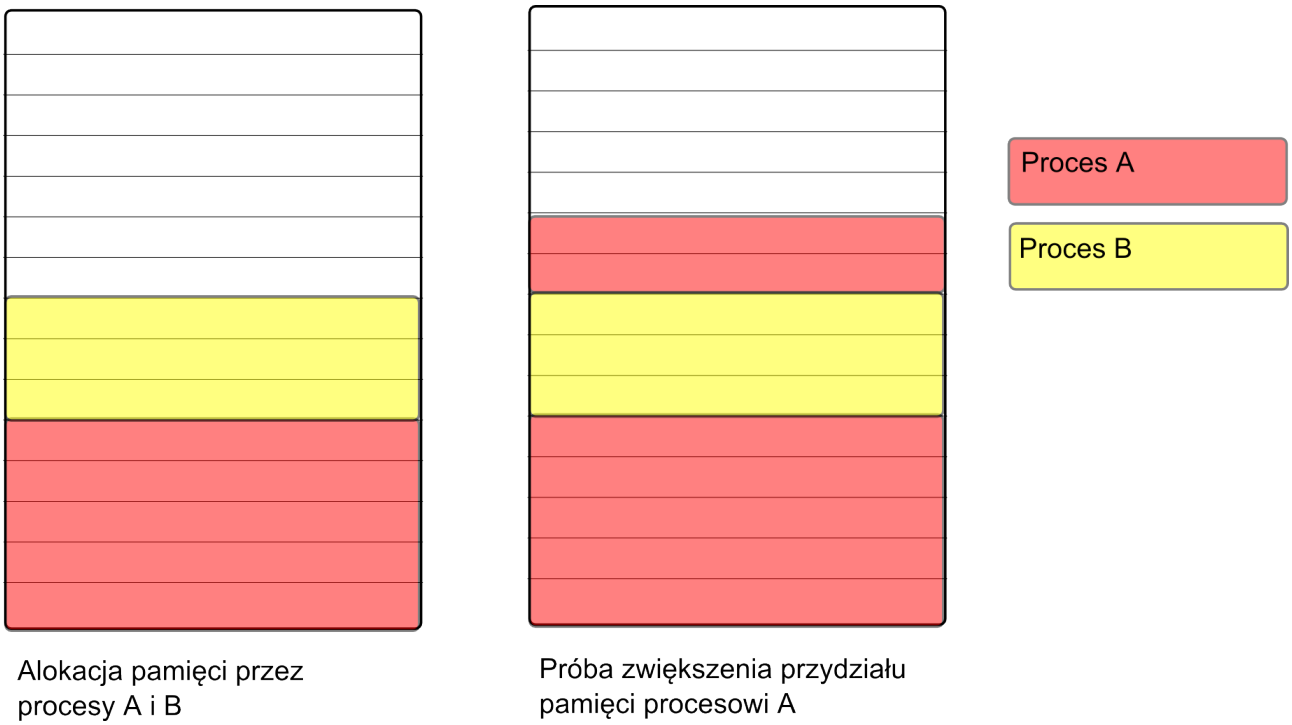
W pierwszych systemach operacyjnych, zdolnych do uruchomienia tylko jednego procesu cała pamięć operacyjna zamontowana w systemie była dostępna dla tego procesu. Z momentem pojawienia się systemów operacyjnych pojawiło się zagadnienie dzielenia pamięci na obszar systemu i programu. Istnieją różne rozwiązania (OS w pamięci ROM[9], pamięć RAM przeznaczona dla programu, OS w pamięci RAM, na początku przestrzeni adresowej, część OS w RAM, sterowniki urządzeń w ROM). W obecnych systemach typu *embedded* lub niektórych palmtopach i telefonach używany jest model pierwszy, czyli przechowujący system operacyjny w pamięci ROM.

Jak wspomniano, na początku, w systemach jednoprocesowych nie istniał problem przydziału i chronienia fragmentów pamięci przed różnymi procesami. W momencie powstania pierwszych systemów wieloprogramowych, np. OS/360 zagadnienie to zaczęło być analizowane a

9 ROM – pamięć nieulotna, tylko do odczytu (*read – only memory*)

pierwsze, dość proste rozwiązania optymalizowane.

System operacyjny musi przydzielać pamięć aplikacji w postaci ciągłego segmentu. Niedopuszczalna jest sytuacja jak na poniższym rysunku, kiedy pamięć procesu A jest rozdzielona przez blok pamięci procesu B.



Niedopuszczalny sposób alokacji pamięci dla procesu A

Niedopuszczalność takiej sytuacji wynika stąd, że dowolny program komputerowy w postaci skompilowanej i zlinkowanej (czyli języka maszynowego) jest umieszczany w jednej przestrzeni adresowej, która nie może być podzielona. Dodatkowo, linker adresuje pamięć w ten sposób jakby program miał dostęp do niej od adresu 0. W rzeczywistości, system operacyjny umieszcza uruchamiane programy w pewnym miejscu fizycznej pamięci, a adres startowy różni się od zakładanego przez linker. Z tego powodu SO musi modyfikować adresy pamięci z których korzysta uruchomiony proces. Może się to odbywać na dwa sposoby – poprzez modyfikację kodu maszynowego procesu (rozwiązanie obecnie niewykorzystywane) lub poprzez modyfikację adresów w momencie odwoływania się do nich przez proces (najczęściej poprzez dodawanie do oryginalnego adresu wartości specjalnego rejestru sprzętowego przechowującego informację o początku przestrzeni adresowej aktualnego procesu). Dodanie kolejnego rejestru, określającego koniec przestrzeni adresowej procesu pozwala na sprzętowe zabezpieczenie nadpisywania pamięci należącej do innych procesów przez niewłaściwie napisany program.

W systemie IBM OS/360 pamięć operacyjna dzielona była na bloki o stałym, niekoniecznie tym samym rozmiarze (***Fixed-size-blocks***). Wielkość tych bloków była ustalana przez operatora komputera w momencie jego uruchomienia. System operacyjny przydzielał każdemu procesowi jeden blok pamięci, o rozmiarze nie mniejszym niż deklarowane zapotrzebowanie procesu. Metoda ta miała jednak trzy bardzo istotne wady. Ustalenie stałej wielkości bloków pamięci nie pozwalało na uruchomienie procesu, który potrzebował więcej pamięci niż maksymalnej wielkości blok. Zmiana wielkości bloków wymagała ponownego uruchomienia komputera, a więc przerwy w pracy. Drugą, równie istotną wadą tej metody było słabe wykorzystanie pamięci operacyjnej, ponieważ procesy często korzystają z mniejszej ilości pamięci niż dostępna w bloku, a niewykorzystana pamięć nie sumuje się tworząc nowy blok. Ostatnią, ale równie ważną wadą była wynikająca ze stałej ilości bloków maksymalna liczba procesów, które mogły być uruchomione na komputerze.

Statyczne ustalanie wielkości bloków pamięci przypisanych do procesu jest stosunkowo proste jednak mało efektywne i obarczone dużą liczbą wad. W związku z tym kolejne propozycje rozwiązań zagadnienia zarządzania pamięcią operacyjną przez system implementowały już dynamiczne przypisywanie pamięci procesom. Pierwszym z tych rozwiązań było zastosowanie map bitowych (***bitmap***), przechowujących informacje o tym które bloki pamięci (tworzone przez system operacyjny, najczęściej wielkości kilku bajtów do kilkudziesięciu kilobajtów) są używane (1), a które nie (0). Uruchomienie procesu powoduje przeszukanie mapy bitowej do momentu znalezienia bloku wypełnionego zerami (czyli nieużywanego) o długości równej (lub większej) niż wymagana przez proces. Konieczność przeszukania mapy przy każdej alokacji pamięci implikuje wymaganie jak najmniejszej jej wielkości (w celu optymalizacji szybkości) z czego z kolei wynika maksymalnie duży rozmiar pojedynczego bloku pamięci. Z drugiej strony patrząc duże bloki oznaczają nieoptymalne wykorzystanie pamięci (proces nie wypełnia całego bloku).

Kolejnym algorytmem zarządzania pamięcią operacyjną jest algorytm sąsiednich bloków (***buddy blocks***). Pamięć jest tutaj reprezentowana jako dwukierunkowa lista (lub drzewo) łącząca bloki pamięci używanej przez procesy. Każdy taki blok opisany jest przez miejsce w pamięci gdzie znajduje się jego początek, długość (liczbę bajtów) oraz wskaźnik do następnego zajętego bloku. W analogiczny sposób przechowywane są informacje o niewykorzystywanych blokach pamięci. W prostych implementacjach dla obu typu bloków wykorzystywana jest ta sama lista, a pojedyncze pole zawiera informacje o tym czy jest używane, czy nie. W momencie uruchamiania procesu lista wolnych obszarów pamięci jest skanowana w poszukiwaniu obszaru o rozmiarze większym lub równym wielkości pamięci deklarowanej przez uruchamiany proces. W przypadku gdy obszar ten jest większy niż wymagana przez proces pamięć, dzielony jest on na dwie części a informacja o nich (tzn. o części używanej przez proces i wolnej) uaktualnia listy. Istnieje kilka sposobów poszukiwania obszaru do alokacji – znajdujące pierwszy, najlepszy, lub najgorszy spełniający warunek wielkości obszar. Najlepszy w tym przypadku oznacza obszar możliwie najmniej większy niż wymagany przez proces (w tym wypadku w pamięci często powstaje duży zbiór małych bloków pamięci, które są zbyt małe dla większości procesów). Najgorszy obszar oznacza możliwie największy obszar, co gwarantuje że pozostały po alokacji obszar może zostać wykorzystany dla innych projektów.

Dotychczas rozważane były sytuacje kiedy uruchamiany proces żądał pamięci nie przekraczającej wielkości dostępnej fizycznie w systemie. Istnieją jednak (stosunkowo często) sytuacje kiedy sytuacja taka występuje. W celu umożliwienia pracy systemu w takich warunkach stosuje się dwa typy rozwiązań – pamięć wirtualną (**virtual memory**) i ***swapping***.

Najprostszym rozwiązaniem umożliwiającym przezwyciężenie ograniczenia wielkości fizycznie dostępnej pamięci operacyjnej jest zastosowanie *swappingu*. Mechanizm ten polega na przenoszeniu informacji przechowywanej w pamięci RAM przez proces nieaktywny na twardy dysk. W ten sposób udostępniane jest miejsce dla bieżącego procesu. Niestety, ze względu na swoją prostotę metoda ta nie jest efektywna, szczególnie jeżeli zmiana aktywnych procesów jest częsta (operacja przenoszenia danych między pamięcią dyskową i RAM jest wolna). Dodatkowo, w wyniku *swappingu* w pamięci operacyjnej powstają niewypełnione bloki między blokami zaalokowanymi przez inne procesy co prowadzi do tzw. fragmentacji pamięci i powstawania obszarów, które nie mogą być wykorzystane przez inne procesy (są zbyt małe). Radą na to jest stosowanie defragmentacji pamięci (przesuwania procesów w dół przestrzeni adresowej tak aby wypełnić powstałe dziury). Mała wydajność systemów stosujących s*wapping* spowodowana jest również tym, że przy każdym przełączeniu procesu, cała pamięć należąca do procesów przenoszonych między dyskiem i pamięcią RAM musi być przeniesiona, niemożliwe jest przechowywanie najczęściej używanych fragmentów procesu czy danych w szybkim RAMie a rzadko używanych – na dysku. Niemożliwe jest również z oczywistych powodów uruchomienie programu wymagającego więcej pamięci niż dostępna pamięć RAM.

Rozwiązaniem powyżej przedstawionego problemu jest metoda rozszerzania pamięci operacyjnej zwana pamięcią wirtualną (***virtual memory***). W metodzie tej pamięć dyskowa jest

przedłużeniem pamięci RAM objętym tą samą przestrzenią adresową. Z punktu widzenia aplikacji system ma pamięć operacyjną o pojemności będącej sumą RAM i zadeklarowanego pliku buforowego na dysku twardym a więc możliwym jest uruchomienie programu wymagającego więcej pamięci niż fizycznie obecne w komputerze. W zasadzie od wprowadzenia procesorów rodziny Intel 80386 wirtualna pamięć w systemach operacyjnych implementowana jest zgodnie z techniką zwaną stronnicowaniem (**paging**). Metoda ta polega na podziale przestrzeni adresowej procesu na tzw. strony czyli bloki pamięci, najczęściej o rozmiarze 4kB. Informacja o stronach jest przechowywana w tablicy stronnic (**page table**), która zawiera fizyczny adres danej strony. Adres ten może być adresem w rzeczywistej pamięci RAM ale równie dobrze wskaźnikiem do fragmentu pliku na dysku (**page file**). Rozkaz odwołania się do jakiegoś adresu w wirtualnej przestrzeni adresowej przetwarzany jest przez specjalizowany komponent komputera (***memory management unit - MMU***), który przeszukuje tablicę stronnicowania i przekazuje właściwy (fizyczny) adres wybranego fragmentu pamięci RAM. W przypadku gdy wskaźnik w tablicy stron wskazuje na dysk twardy, MMU zgłasza wyjątek błędu stronnicowania (***page fault***), który powoduje wywołanie części systemu operacyjnego obsługującej pamięć wirtualną. System szuka w pamięci RAM wolnego miejsca, w które może skopiować dane z dysku, kopiuje je i uaktualnia tablicę stronnicowania tak aby adres, który dotychczas odnosił się do dysku twardego, wskazywał teraz na odpowiednie miejsce w pamięci RAM. Analogicznie, dane znajdujące się w RAM, a nie wykorzystywane przez długi czas zostają przez system przeniesione na dysk twardy. Wybór właściwych do usunięcia danych może odbywać się według różnych algorytmów, dobry ich opis znajduje się w [TANNENBAUM].

Technika wirtualnej pamięci jest używana przez wszystkie współczesne systemy operacyjne z wyjątkiem systemów RTOS oraz *embedded*.

Obsługa przerwań

W normalnym toku działania komputera, kiedy wykonywany jest jeden lub więcej procesów czasami wskazane jest zatrzymanie ciągu przetwarzania i zażądanie od procesora wykonania określonych operacji, na przykład obsłużenia urządzenia zewnętrznego. Operacje te wykonywane są już w normalnym ciągu przetwarzania, a szybkość ich wykonania zależna jest od używanego w systemie *schedulera* oraz priorytetu przypisanego do tych operacji. Zasygnalizowanie takiej sytuacji może wydarzyć się w dowolnym momencie, jest zatem sygnałem asynchronicznym zwanym przerwaniem (***interrupt***), z kolei operacje obsługujące wykonywane są synchronicznie. Alternatywą stosowania przerwań jest cykliczne (w pętli) sprawdzanie stanu określonych flag (sygnałów), które mogą być modyfikowane przez urządzenia zewnętrzne sygnalizujące wystąpienie określonych zdarzeń. Rozwiązanie to jest jednak bardzo nieefektywne ze względu na niepotrzebne obciążenie procesora operacjami odpytywania.

Rozróżniane są dwa typy przerwań, sprzętowe oraz programowe. Efekty ich działania są takie same, przy czym sprzętowe przerwania wywoływane są przez fizyczne urządzenia podłączone do systemu komputerowego, programowe zaś – poprzez wywołanie odpowiedniego zestawu instrukcji (przerwania systemowe często wykorzystywane są w implementacji wywołań systemowych, które wymagają zmiany pracy procesora z trybu użytkownika na tryb jądra) . Przerwania sprzętowe mogą z kolei być zgłaszane dwojako – albo przez modyfikację danego obszaru pamięci operacyjnej albo przez zmianę stanów sygnałów na specjalnych obwodach logicznych doprowadzonych bezpośrednio do procesora.

Z racji tego, że przerwania wywoływane mogą być przez różne urządzenia, a informacje przez nie zgłaszane mogą mieć różną istotność (od krytycznych do pomijalnych) istnieje możliwość maskowania niektórych przerwań, to znaczy oznaczania ich jako ignorowane. Przerwania krytyczne dla działania systemu (np. generowane przez zegar systemowy) nie mogą być ustawiane jako ignorowane.

Ogólny schemat obsługi przerwania można przedstawić w następujących punktach [TANNENBAUM]:

1. Zapisanie wartości rejestrów łącznie z licznikiem programów (PSW)
2. Zestawienie kontekstu procesu obsługi przerwania
3. Zestawienie stosu procesu obsługi przerwania
4. Powiadomienie kontrolera przerwań o przechwyceniu bieżącego przerwania
5. Skopiowanie zachowanych rejestrów do tablicy procesów
6. Uruchomienie procedury obsługującej przerwanie, która pobierze dodatkowe dane (jeżeli takowe są) z rejestrów urządzenia, które wywołało przerwanie.
7. Ponowne uruchomienie ciągu wykonania procesów (wg *schedulera*). Proces obsługujący przerwanie zostanie z dużym prawdopodobieństwem wykonany jako jeden z pierwszych z racji jego wysokiego priorytetu.

Synchronizacja

Systemy operacyjne umożliwiające uruchamianie wielu procesów jednocześnie muszą być wyposażone w szereg środków służących do ochrony współdzielonych zasobów (pamięci operacyjnej, procesora, plików, urządzeń wejścia / wyjścia). Konieczność tej ochrony wynika z tego, że niemożliwe jest przewidzenie sekwencji operacji wykonywanych przez komputer na skutek najczęściej dynamicznego zarządzania procesami wykonywanymi na przemian przez procesor (lub wiele procesorów). Ochrona zasobów nie byłaby potrzebna gdyby programista aplikacji mógłby założyć (i założenie to zostało spełnione) jakie procesy będą uruchomione w czasie pracy jego aplikacji, do których zasobów i kiedy będą się odnosić. Jak łatwo się domyślić poza bardzo szczególnymi przypadkami spełnienie tych warunków jest niemożliwe. W przeciętnym systemie, realny czas wykonania jakiegokolwiek procesu nie może być jednoznacznie zdefiniowany (tzn. przy każdym uruchomieniu może być różny) ponieważ zależy od innych procesów jednocześnie uruchomionych, stosowanego algorytmu szeregowania zdarzeń, występowania przerwań itd. W efekcie kiedy np. dwa procesy używają jednej, globalnie dostępnej zmiennej (jeden zapisując, a drugi odczytując) to wartość odczytana może być różna, w zależności od tego, czy proces zapisujący został wykonany wcześniej niż odczyt[10]. Z drugiej strony blokowanie dostępu do zasobów na cały okres wykonania jednego zadania jest niekorzystne ponieważ może prowadzić do znacznego zmniejszenia wydajności systemu, jak również zjawiska zakleszczenia procesów[11].

Mechanizm powstrzymywania procesów przed dostępem do danego zasobu jeżeli jest on używany przez jakikolwiek inny proces, nazywa się wzajemnym wykluczaniem (*mutual exclusion*) a zasoby chronione – zasobami krytycznymi. Blokowanie to nie może odbywać się w pełni automatycznie (tzn. przez system operacyjny) ponieważ realizacja automatycznego mechanizmu wykluczeń jest zbyt trudna, system operacyjny udostępnia jednakże mechanizmy blokowania i kontroluje je. Odwołanie do zasobu krytycznego w procesie odbywa się w tzw. sekcji krytycznej (***critical section***). Mechanizm sekcji krytycznych musi być zaimplementowany w kodzie procesu, a realizowany jest najczęściej poprzez użycie udostępnianych przez system operacyjny metod synchronizacyjnych.

Typowym problemem obrazującym problem synchronizacji dostępu do danych jest tzw. problem producenta – konsumenta. Przykładowe implementacje procesów producenta (zapisującego dane) i konsumenta (odczytującego dane) zostały zamieszczone poniżej:

10 Zjawisko przedstawione w przykładzie nosi nazwę wyścigu (***race condition***)

11 Zakleszczenie – sytuacja kiedy dwa lub więcej procesów nie może kontynuować wykonania ponieważ oczekuje na zwolnienie zasobu zablokowanego przez inny proces. Przykładem może być zdarzenie kiedy proces A ma wyłączny dostęp do zmiennej V1 ale oczekuje na dostęp do zmiennej V2, która została wcześniej zablokowana przez proces B, który nie może kontynuować pracy bez dostępu do zmiennej V1. Dokładny opis zjawiska i sposoby zabezpieczenia przed nim omówione zostały w dalszej części bieżącego rozdziału

Proces producenta (uproszczony)	Proces konsumenta (uproszczony)

`#define BUFFER_SIZE 100`
`int g_iItemCount = 0; // współdzielona zmienna – liczba zapełnionych bloków bufora`
`char g_pucBuffer[BUFFER_SIZE]; // bufor`
 	
`void Producer ()`
`{`
`  int v_iCurrentItem;`
 	
`  while (TRUE)`
`  {`
`  char * g_pucCurrentItem = new char;`
`  if ( g_iItemCount == BUFFER_SIZE)`
`  SuspendProcess(this);`
`  g_pucBuffer[v_iCurrentItem] = data;`
`  delete data;`
`  g_iItemCount++;`
`  if ( iItemCount == 1)`
`  WakeProcess(g_psConsumer);`
`  }`
`}`
	`void Consumer()`
	`{`
	`  int iCurrentItem;`
	
	`  while (TRUE)`
	`  {`
	`  if (g_iItemCount == 0)`
	`  SuspendProcess(this);`
	`  char v_ucCurrentItem=g_pucBuffer[g_iItemCount-1];`
	`  printf( v_ucCurrentItem );`
	`  g_iItemCount = g_iItemCount – 1;`
	
	`  if ( g_iItemCount == BUFFER_SIZE – 1)`
	`  WakeProcess(g_psProducer);`
	
	`  }`
	`}`

Dwa procesy dzielą wspólny bufor danych o z góry ustalonym rozmiarze, przy czym proces producenta zapisuje do tego bufora dane w pętli, jeden blok w każdym przebiegu. Jeżeli bufor zostaje zapełniony (jest w nim tyle elementów ile wynosi jego rozmiar), to proces producenta zostaje wstrzymany i może być wznowiony przez konsumenta jeżeli ten zwolni jeden blok danych. Konsument w nieskończonej pętli próbuje odczytać dane z bufora, również po jednym bloku w przebiegu, przy czym jeżeli bufor jest pusty, to proces czytania zostaje wstrzymany i może zostać wznowiony jeżeli do bufora producent wpisze jeden znak (i wyśle do konsumenta rozkaz wznowienia przetwarzania). Ponieważ dostęp do zmiennej mówiącej o zapełnieniu bufora (g_iItemCount) jest niesynchronizowany, to może wystąpić sytuacja wyścigu (*race condition*): konsument odczytuje wartość g_iItemCount, która równa jest 0 i w tym momencie następuje przełączenie procesu wykonywanego przez *scheduler* systemowy. Uruchamiany jest proces producenta, który zapełnia bufor jednym znakiem i wysyła zgodnie z kodem programu rozkaz wznowienia działania konsumenta. Rozkaz ten jest jednak zignorowany, bo konsument nie został uśpiony (jedynie wywłaszczony przez *scheduler*). Po ponownym przełączeniu procesów wykonanie konsumenta rozpoczyna się od momentu, w którym został on wywłaszczony, czyli zostaje on uśpiony (ponieważ pamięta wartość g_iItemCount = 0 sprzed wywłaszczenia). Po pewnym czasie przydział procesora dostaje proces producenta i napełnia bufor kolejnym znakiem, nie wysyłając już jednak kolejnych rozkazów wznowienia konsumenta (wysyłanie w każdej pętli wiązałoby się z dużym narzutem czasu procesora, z tego powodu komunikat jest wysyłany tylko w momencie napełnienia pustego bufora jednym znakiem). Konsument nie jest pracuje, więc producent napełnia bufora aż do jego wypełnienia i wtedy zostaje wstrzymany (g_iCount == BUFFER_SIZE). Oba procesy są więc wstrzymane i nie ma możliwości aby któryś z nich został uruchomiony (poza zewnętrzną ingerencją).

Ze względu na model współdziałania procesów przy dostępie do współdzielonych danych można wyróżnić dostęp konkurencyjny (czyli taki, który wymaga zablokowania danych w każdym wypadku) oraz współpracujący (czyli taki, który wymaga zablokowania danych tylko przy zapisie). Przykładem pracy w trybie drugim jest na przykład baza danych, w której wiele procesów może odczytywać jednocześnie, niemniej jednak w momencie wystąpienia zapisu proces zapisujący powinien być jedynym, który ma w danej chwili dostęp do danych. Ponadto, aby nie wystąpił błąd niespójności danych, wszystkie procesy odczytujące muszą zostać zakończone (a nie wstrzymane) przed rozpoczęciem zapisu, w innym wypadku mogłoby nastąpić przekłamanie i odczytanie fragmentu starych danych (przed zapisem) i fragmentu danych uaktualnionych (po zapisie).

Istnieją różne sposoby implementacji metod służących do wzajemnego blokowania procesów. Można je podzielić na:

Rozwiązania sprzętowe:

- wyłączanie przerwań na czas trwania danego procesu – zabezpiecza przed przełączeniem aktualnie wykonywanego procesu (nie działa *scheduler*), wiąże się jednak ze znacznym spadkiem wydajności (nie wykonywane są inne procesy) a jego skuteczność ogranicza się do systemów jednoprocesorowych (w wieloprocesorowych jednocześnie wykonywany jest więcej niż jeden proces)
- blokowanie dostępu do obszaru czytanej / zapisywanej pamięci (w ramach jednej, nieprzerywalnej operacji[12], zwykle odczytu i zapisu lub odczytu i sprawdzenia). W czasie trwania tej operacji żaden inny proces nie może uzyskać dostępu do obszaru pamięci przechowującego daną zmienną. Najczęściej używane instrukcje to ***Compare&Swap*** oraz ***Exchange (XCHG)***, obie dostępne na popularnych platformach sprzętowych (x86, IA64).

  Zaletą rozwiązań sprzętowych jest ich prostota, możliwość zabezpieczania wielu współdzielonych zasobów oraz niezależność (oczywiście poza wyłączaniem przerwań) od liczby uruchomionych procesów i procesorów obecnych w systemie. Wadami są natomiast spadek wydajności przetwarzania (oczekiwanie na dostęp wiąże się z dodatkowymi cyklami procesora) oraz możliwość wystąpienia zarówno zjawiska zakleszczenia jak i głodu (ponieważ użytkownik nie ma kontroli nad przydzielaniem krytycznych sekcji procesom, co może czasem prowadzić do zaniedbania niektórych procesów).

Rozwiązania programowe – implementowane przez programistę, które mogą być najlepiej dostosowane do konkretnego problemu, jednak często wymagają dużego narzutu obliczeniowego. W rozwiązaniach tych istnieje ponadto duże ryzyko istnienia błędów implementacyjnych.

Rozwiązania programowe ze wsparciem systemu operacyjnego, który udostępnia narzędzia i metody synchronizacji, często zaimplementowane w oparciu o rozwiązania sprzętowe. To rozwiązanie jest najczęściej spotykane, ponieważ łączy zalety obu wcześniej przedstawionych. Popularnymi mechanizmami synchronizacji udostępnianymi przez większość systemów są muteksy, semafory i monitory.

Muteksy

Muteks można rozumieć jako chronioną, globalnie dostępną zmienną, która służy jako swojego rodzaju flaga informująca o tym, czy zasób do niej przypisany jest wykorzystywany przez jakiś proces, czy nie. W swojej istocie jest on zbliżony do semafora binarnego (patrz dalej), jednakże w przeciwności do niego, muteks może być odwołany (dostęp do zasobu odblokowany) tylko przez proces, który go założył (zablokował dostęp). W większości implementacji, muteksy są bardzo szybkimi konstrukcjami synchronizacyjnymi. W systemach Microsoft Windows, specjalna odmiana muteksów wykorzystywana jest jako domyślna metoda synchronizacji wątków (czyli dostępu do zmiennych w ramach jednej przestrzeni adresowej – procesu). Odmiana ta nosi nieco mylącą nazwę ***CriticalSection***.

Przykładowy program obrazujący użycie muteksów przedstawiony jest poniżej:

12 Operacja nieprzerywalna, inaczej atomowa (***atomic operation***) – operacja, która nie może być w żaden sposób przerwana ani zmodyfikowana po rozpoczęciu (niekoniecznie wykonywana w jednym kroku)

Główna funkcja programu

```
#include <windows.h>
#include <stdio.h>
#define THREADCOUNT 2
HANDLE ghMutex;
DWORD WINAPI WriteToDatabase( LPVOID );
void main()
{
   HANDLE aThread[THREADCOUNT];
   DWORD ThreadID;
   int i;
   // Create a mutex with no initial owner
   ghMutex = CreateMutex(
      NULL,              // default security attributes
      FALSE,             // initially not owned
      NULL);             // unnamed mutex
   if (ghMutex == NULL)
   {
      printf("CreateMutex error: %d\n", GetLastError());
      return;
   }
   // Create worker threads
   for( i=0; i < THREADCOUNT; i++ )
   {
      aThread[i] = CreateThread(
                NULL,       // default security attributes
                0,          // default stack size
                (LPTHREAD_START_ROUTINE) WriteToDatabase,
                NULL,       // no thread function arguments
                0,          // default creation flags
                &ThreadID); // receive thread identifier
      if( aThread[i] == NULL )
      {
         printf("CreateThread error: %d\n", GetLastError());
         return;
      }
   }
   // Wait for all threads to terminate
   WaitForMultipleObjects(THREADCOUNT, aThread, TRUE,
INFINITE);
   // Close thread and mutex handles
   for( i=0; i < THREADCOUNT; i++ )
      CloseHandle(aThread[i]);
   CloseHandle(ghMutex);
}
```

Funkcja zapisująca dane

```
DWORD WINAPI WriteToDatabase( LPVOID lpParam )
{
   DWORD dwCount=0, dwWaitResult;
   // Request ownership of mutex.
   while( dwCount < 20 )
   {
      dwWaitResult = WaitForSingleObject(
         ghMutex,    // handle to mutex
         INFINITE);  // no time-out interval
      switch (dwWaitResult)
      {
         // The thread got ownership of the mutex
         case WAIT_OBJECT_0:
            __try {
               // TODO: Write to the database
               printf("Thread %d writing to database...\n",
                     GetCurrentThreadId());
               dwCount++;
            }
            __finally {
               // Release ownership of the mutex object
               if (! ReleaseMutex(ghMutex))
               {
                  // Handle error.
               }
            }
            break;
         // The thread got ownership of an abandoned mutex
         // The database is in an indeterminate state
         case WAIT_ABANDONED:
            return FALSE;
      }
   }
   return TRUE;
}
```

Źródło: Microsoft Developer Network

Semafory

Semafor, podobnie jak muteks jest globalną zmienną o chronionym dostępie, która może służyć do informowania o tym, że dany zasób jest wykorzystywany przez jakiś proces. Istnieją dwa typy semaforów – binarne (czyli takie, które mogą jedynie wskazywać zablokowanie lub odblokowanie zasobu) oraz liczące (które potrafią zarządzać zbiorami zasobów). Kluczowym środkiem, z którego korzysta system operacyjny do implementacji semaforów jest obecność specjalnych instrukcji, które powodują że operacje na semaforach są operacjami atomowymi (niepodzielnymi). Spośród trzech funkcji obsługi semaforów (inicjalizacji, uzyskania oraz zwolnienia dostępu do danych) dwie ostatnie muszą być niepodzielne aby uniknąć zjawiska wyścigu między procesami:

```
Initialize (Semaphore p_sS, int p_iCount) //Inicjalizacja semafora (nie musi być operacją atomową)
{
        p_sS = p_iCount;
}

Wait(Semaphore p_sS) // Zażądaj dostępu do zasobów
{
        wait until p_sS > 0 then p_sS--;  // sprawdzenie wartości semafora i jej zmniejszenie musi być operacją niepodzielną
}
```

```
Signal(Semaphore p_sS)  // Zwolnij dostęp do zasobów
{
        p_sS++;   // zwiększenie wartości semafora musi być operacją atomową
}
```

Funkcje, które muszą być wykonywane jako operacje atomowe muszą mieć oparcie w rozwiązaniach sprzętowych, takich jak np. opisane wyżej instrukcje *Compare&Swap* i *Exchange*.

Mechanizm semaforów został wprowadzony przez E.Dijkstrę jako rozwiązanie modelu problemu synchronizacyjnego zwanego problemem ucztujących filozofów[13].

Przykład użycia semaforów w środowisku Microsoft Windows znajduje się poniżej:

Główna funkcja programu

```
#include <windows.h>
#include <stdio.h>

#define MAX_SEM_COUNT 10
#define THREADCOUNT 12

HANDLE ghSemaphore;

DWORD WINAPI ThreadProc( LPVOID );

void main()
{
   HANDLE aThread[THREADCOUNT];
   DWORD ThreadID;
   int i;

   // Create a semaphore with initial and max counts of
MAX_SEM_COUNT

   ghSemaphore = CreateSemaphore(
      NULL,           // default security attributes
      MAX_SEM_COUNT,  // initial count
      MAX_SEM_COUNT,  // maximum count
      NULL);          // unnamed semaphore

   if (ghSemaphore == NULL)
   {
      printf("CreateSemaphore error: %d\n", GetLastError());
      return;
   }

   // Create worker threads

   for( i=0; i < THREADCOUNT; i++ )
   {
      aThread[i] = CreateThread(
                NULL,       // default security attributes
                0,          // default stack size
                (LPTHREAD_START_ROUTINE) ThreadProc,
                NULL,       // no thread function arguments
                0,          // default creation flags
                &ThreadID); // receive thread identifier

      if( aThread[i] == NULL )
      {
         printf("CreateThread error: %d\n", GetLastError());
         return;
      }
   }

   // Wait for all threads to terminate

   WaitForMultipleObjects(THREADCOUNT, aThread, TRUE,
```

Funkcja wykonywana przez każdy z uruchomionych wątków

```
DWORD WINAPI ThreadProc( LPVOID lpParam )
{
   DWORD dwWaitResult;
   BOOL bContinue=TRUE;

   while(bContinue)
   {
      // Try to enter the semaphore gate.

      dwWaitResult = WaitForSingleObject(
         ghSemaphore,   // handle to semaphore
         0L);           // zero-second time-out interval

      switch (dwWaitResult)
      {
         // The semaphore object was signaled.
         case WAIT_OBJECT_0:
            // TODO: Perform task
            printf("Thread %d: wait succeeded\n", GetCurrentThreadId());
            bContinue=FALSE;

            // Simulate thread spending time on task
            Sleep(5);

            // Release the semaphore when task is finished

            if (!ReleaseSemaphore(
                  ghSemaphore,  // handle to semaphore
                  1,            // increase count by one
                  NULL) )       // not interested in previous count
            {
               printf("ReleaseSemaphore error: %d\n", GetLastError());
            }
            break;

         // The semaphore was nonsignaled, so a time-out occurred.
         case WAIT_TIMEOUT:
            printf("Thread %d: wait timed out\n", GetCurrentThreadId());
            break;
      }
   }
   return TRUE;
}
```

13 Problem ucztujących filozofów – modelowy przykład obrazujący mechanizm powstawania wyścigu (*race condition*) i zakleszczenia (deadlock) w programowaniu współbieżnym. Opis problemu i jego rozwiązania przy użyciu semaforów znajduje się np. w [SILBERSCHATZ]

```
INFINITE);

   // Close thread and semaphore handles

   for( i=0; i < THREADCOUNT; i++ )
      CloseHandle(aThread[i]);

   CloseHandle(ghSemaphore);
}
```

Źródło: Microsoft Developer Network

Monitory

Bardziej rozbudowanym środkiem służącym do synchronizacji jest obiekt wysokiego poziomu abstrakcji zwany monitorem. Jest obiekt (instancja) przeznaczony do jednoczesnego użycia przez procesy, które chcą uzyskać dostęp do danego zasobu, przy czym użycie nie oznacza bezpośredniego wykorzystania zmiennych synchronizujących (tak jak na przykład w wypadku muteksów), a wywołania odpowiednich funkcji, które odnoszą się do wewnętrznych struktur monitora. Struktury te najczęściej oparte są o proste elementy takie jak semafory, muteksy lub zmienne warunkowe, jednak dzięki temu że są one opakowane klasą monitora, ich wykorzystaniem zarządza kompilator programu, nie sam programista, co zdecydowanie wpływa na poprawienie niezawodności programu i zmniejsza liczbę błędów, które można popełnić. Monitor może być obsługiwany przez dwie funkcje – wait (usypiająca wołający proces do momentu uzyskania dostępu do zasobu) i signal (zwalniający blokadę zasobu i uruchamiający kolejny proces, który na ten zasób oczekuje. W przypadku jeżeli nie ma takiego procesu, komunikat uruchamiający jest tracony). Tylko jeden proces może w danej chwili mieć dostęp do monitora, z kolei monitor może mieć kolejkę procesów, które czekają na ów dostęp. Przykład wykorzystania monitora znajduje się poniżej:

```
#define BUFFER_SIZE 100
monitor g_sBuffer; // monitor
cond g_sNotFull // zmienna warunkowa kontrolowana przez monitor
cond g_sNotEmpty; // zmienna warunkowa kontrolowana przez monitor

char p_pucBuffer[BUFFER_SIZE];
char * v_pcNextIn = NULL;
char * v_pcNextOut = NULL;

int g_iItemCount = 0;

void Producer()
{
	char v_pcChar = 0;
	while (true)
	{
		v_pcChar++;
		if (g_iItemCount == BUFFER_SIZE)
			g_sBuffer.cwait(g_sNotFull);

		p_pucBuffer[v_pcNextIn] = v_pcChar;
		v_pcNextIn = (v_pcNextIn++);
		if (v_pcNextIn >= BUFFER_SIZE)
			v_pcNextIn = 0;
		g_iItemCount++;

		g_sBuffer.csignal (g_sNotEmpty); // sygnał do odbiorcy, że dane są w buforze
	}
}

void Consumer()
{
	char v_cReceivedChar;
	while (true)
	{
```

```
                        if (g_iItemCount == 0)
                                g_sBuffer.cwait(g_sNotEmpty); // bufor jest pusty, oczekuj na sygnał g_sNotEmpty

                        v_cReceivedChar = p_pucBuffer[v_pcNextOut];
                        v_pcNextOut++;
                        if (v_pcNextOut >= BUFFER_SIZE)
                                v_pcNextOut = 0;

                        g_iItemCount--; // jeden obiekt zdjęty z bufora
                        g_sBuffer.csignal (g_sNotFull); // jeżeli proces producenta jest uśpiony to obudź go
                        printf(v_cReceivedChar);
                }
}
```

Implementacja procesów producenta i konsumenta z synchronizacją przy użyciu monitora

Zdarzenia

Zdarzenia (***events***) są kolejnymi obiektami mogącymi służyć do synchronizacji. Ich ogólnym przeznaczeniem jest komunikacja międzyprocesowa, a dokładnie przekazywanie sygnałów do procesów lub wątków poprzez ustawienie danego obiektu zdarzenia (*event object*) w stan sygnalizujący (*signalled state*). Tworzenie obiektu zdarzenia inicjowane jest przez pierwszy z synchronizowanych procesów *(CreateEvent),* w ramach inicjalizacji ustawiane są również jego parametry takie jak nazwa, tryb kasowania stanu (automatyczny lub ręczny). Wszystkie inne procesy i wątki które chcą korzystać z tego obiektu mogą uzyskać do niego wskaźnik poprzez wywołanie funkcji *OpenEvent*, której jedynym wymaganym parametrem jest nazwa oczekiwanego zdarzenia. Z racji zdolności sygnalizacji stanów procesom obiekty te mogą być również wykorzystywane do synchronizacji dostępu do danych. Przykładowy program wykorzystujący zdarzenia w środowisku Microsoft Windows znajduje się poniżej:

Funkcje pomocnicze i funkcja główna	Funkcje wątków

#include <windows.h>
#include <stdio.h>
#define THREADCOUNT 4
HANDLE ghWriteEvent;
HANDLE ghThreads[THREADCOUNT];
DWORD WINAPI ThreadProc(LPVOID);
void CreateEventsAndThreads(void)
{
  int i;
  DWORD dwThreadID;
  // Create a manual-reset event object. The write thread sets this
  // object to the nonsignaled state when it finishes writing to a
  // shared buffer.
  ghWriteEvent = CreateEvent(
  NULL, // default security attributes
  TRUE, // manual-reset event
  FALSE, // initial state is nonsignaled
  TEXT("WriteEvent") // object name
  );
  if (ghWriteEvent == NULL)
  {
  printf("CreateEvent failed (%d)\n", GetLastError());
  return;
  }
  // Create multiple threads to read from the buffer.
  for(i = 0; i < THREADCOUNT; i++)
  {
  // TODO: More complex scenarios may require use of a parameter
  // to the thread procedure, such as an event per thread to
  // be used for synchronization.
  ghThreads[i] = CreateThread(
  NULL, // default security
  0, // default stack size
  ThreadProc, // name of the thread function
  NULL, // no thread parameters
  0, // default startup flags
  &dwThreadID);
	void WriteToBuffer(VOID)
	{
	  // TODO: Write to the shared buffer.
	  printf("Main thread writing to the shared buffer...\n");
	  // Set ghWriteEvent to signaled
	  if (! SetEvent(ghWriteEvent))
	  {
	  printf("SetEvent failed (%d)\n", GetLastError());
	  return;
	  }
	}
	void CloseEvents()
	{
	  // Close all event handles (currently, only one global handle).
	  CloseHandle(ghWriteEvent);
	}
	DWORD WINAPI ThreadProc(LPVOID lpParam)
	{
	  DWORD dwWaitResult;
	  printf("Thread %d waiting for write event...\n",
	GetCurrentThreadId());
	  dwWaitResult = WaitForSingleObject(
	  ghWriteEvent, // event handle
	  INFINITE); // indefinite wait
	
	  switch (dwWaitResult)
	  {
	  // Event object was signaled
	  case WAIT_OBJECT_0:
	  //
	  // TODO: Read from the shared buffer
	  //
	  printf("Thread %d reading from buffer\n",
	  GetCurrentThreadId());
	  break;
	  // An error occurred

```
      if (ghThreads[i] == NULL)
      {
         printf("CreateThread failed (%d)\n", GetLastError());
         return;
      }
   }
}

void main()
{
   DWORD dwWaitResult;
   // TODO: Create the shared buffer
   // Create events and THREADCOUNT threads to read from the buffer
   CreateEventsAndThreads();

   // At this point, the reader threads have started and are most
   // likely waiting for the global event to be signaled. However,
   // it is safe to write to the buffer because the event is a
   // manual-reset event.

   WriteToBuffer();
   printf("Main thread waiting for threads to exit...\n");
   // The handle for each thread is signaled when the thread is
   // terminated.
   dwWaitResult = WaitForMultipleObjects(
      THREADCOUNT,   // number of handles in array
      ghThreads,     // array of thread handles
      TRUE,          // wait until all are signaled
      INFINITE);
   switch (dwWaitResult)
   {
      // All thread objects were signaled
      case WAIT_OBJECT_0:
         printf("All threads ended, cleaning up for application exit...\n");
         break;
      // An error occurred
      default:
         printf("WaitForMultipleObjects failed (%d)\n", GetLastError());
         return;
   }
    // Close the events to clean up
   CloseEvents();
}
```

```
      default:
         printf("Wait error (%d)\n", GetLastError());
         return 0;
   }
   // Now that we are done reading the buffer, we could use another
   // event to signal that this thread is no longer reading. This
   // example simply uses the thread handle for synchronization (the
   // handle is signaled when the thread terminates.)
   printf("Thread %d exiting\n", GetCurrentThreadId());
   return 1;
}
```

Źródło – Microsoft Developer Network

Bariery

Bariera jest specyficzną konstrukcją programową służącą do synchronizacji grupy procesów. Definiowana jest jako obiekt, który umożliwia zatrzymanie poszczególnych procesów w momencie kiedy do niej docierają na czas tak długi, aby wszystkie procesy z synchronizowanej grupy dotarły do tego etapu wykonania. Bariera najczęściej służy do synchronizowania długotrwałych obliczeń wykonywanych na bardzo wielu procesorach przez wiele procesów (np. symulacji naukowych).

Obsługa urządzeń zewnętrznych

Wszystkie urządzenie zewnętrzne, w które wyposażony może być system komputerowy posiadają pewnego rodzaju kontroler zawierający zestaw rejestrów, dzięki którym urządzenie może być sterowane oraz może ono przesyłać (zwracać) charakterystyczne dla siebie informacje. Z racji różnorodności urządzeń, kontrolery, zestawy rejestrów i dane, które w nich się znajdują są zupełnie różne w wyniku czego praktycznie każde z urządzeń wymaga odrębnego programu do obsługi, zwanego zwykle sterownikiem (***driver***), który umożliwia komunikację kontrolera z jądrem systemu

operacyjnego. Najczęściej sterowniki urządzeń nie są kompatybilne z więcej niż jednym systemem operacyjnym. Jak wspomniano w rozdziale poświęconym architekturze jądra systemu, sterowniki mogą należeć do jądra (procedury z nimi związane są wtedy wykonywane w najwyższym możliwym poziomie uprzywilejowania) lub znajdować się poza jądrem (pracujące w trybie użytkownika). Oba rozwiązania mają swoje wady i zalety, główną zaletą uruchamiania sterowników z ograniczonym dostępem do zasobów komputera jest automatyczne zwiększenie stabilności pracy systemu operacyjnego (błędy w sterownikach, które pojawiają się stosunkowo często). Główną zaletą uruchamiania sterowników w trybie jądra jest łatwiejsza ich implementacja (możliwe jest bezpośrednie odwołanie do rejestrów kontrolera zamiast odwołanie poprzez wywołania systemowe systemu operacyjnego).

Jak wspomniano wcześniej, zadaniem sterownika urządzenia jest wymiana danych w odpowiednim formacie pomiędzy komputerem, a kontrolerem urządzenia. Odpowiednie serie danych powodują inicjalizację urządzenia, sprawdzanie jego stanu, rozkaz wykonania pewnych czynności lub pobrania oczekujących danych. Ogólnie sterowniki można podzielić na te, które wymieniają dane pod postacią znaków (np. drukarki igłowe, klawiatury, wyświetlacze tekstowe) oraz wymieniające dane w postaci bloków.

Dawniej sterowniki urządzeń były wkompilowywane w system operacyjny, dodanie nowego urządzenia do komputera wiązało się z przebudowaniem jądra systemu (sterowniki najczęściej dostarczane były w postaci kodów źródłowych). Jak łatwo sobie wyobrazić instalacja nowych urządzeń była bardzo problematyczna i wymagała dużej znajomości systemu, wiązała się również z czasochłonną procedurą rekompilacji jądra systemowego. W obecnie istniejących systemach sterowniki ładowane są dynamicznie, najczęściej w czasie uruchamiania systemu operacyjnego.

Zarządzanie procesem obsługi urządzeń może się odbywać na trzy sposoby:

- programowej obsługi wejścia / wyjścia
- poprzez przerwania
- poprzez bezpośredni dostęp do pamięci (DMA)

W przypadku programowej obsługi wejścia / wyjścia całością procesu obsługi zarządza procesor komputera – wykonuje on procedury, które uruchamiają odpowiednie akcje w module obsługi urządzenia. Od momentu przekazania rozkazu przez procesor do tego modułu, rozpoczyna on cykliczne odpytywanie rejestrów przechowujących flagi informujące o stanie urządzenia po to, aby sprawdzić, czy wywołana akcja już się zakończyła, a jeżeli tak, to z jakim efektem. Głównym problemem w tym trybie obsługi urządzeń jest znaczny spadek wydajności komputera w momencie oczekiwania na wynik operacji – kolejne cykle jego pracy są marnotrawione na odpytywanie rejestrów urządzenia zamiast wykorzystania ich do przetwarzania innych procesów.

Mechanizm przerwań (omówiony wcześniej w tym rozdziale) likwiduje problem czasu traconego przez procesor w wyniku oczekiwania na rezultat operacji. W tym trybie, po wysłaniu danego rozkazu do modułu obsługi urządzenia, procesor przestaje się zajmować tą obsługą i powraca do innych zadań. W momencie zakończenia operacji (zarówno sukcesem jak i błędem) kontroler urządzenia generuje przerwanie, które przerywa normalny ciąg wykonania procesów przez procesor i uruchamia procedurę obsługi wyników operacji wykonanej przez urządzenie zewnętrzne. Niestety, obsługa przerwania wiąże się z wieloma dodatkowymi cyklami pracy procesora (zachowanie kontekstu bieżąco wykonywanego procesu, sprawdzenie procedury do której odnosi się przerwanie, wczytanie jej kontekstu, wykonanie, wczytanie kontekstu uprzednio wykonywanego procesu).

Dla urządzeń, które wykorzystując przerwania generowałyby ich bardzo dużą liczbę (co powodowałoby znaczny spadek wydajności całego systemu) opracowano trzecią metodę współpracy, mianowicie bezpośredni dostęp do pamięci operacyjnej dzięki dodatkowemu modułowi sprzętowemu podłączonemu do szyny systemowej. Moduł ten przyjmuje od procesora rozkaz

obsługi urządzenia z parametrami takimi jak adres urządzenia, kierunek wymiany danych (zapis czy odczyt z urządzenia), a także początkowy adres w pamięci operacyjnej, przeznaczonej na dane z tego urządzenia i ich rozmiar. Od tego momentu procesor nie zajmuje się wymianą danych, a kontroler DMA obsługuje cały transfer po szynie danych. Po zakończeniu transferu (albo wystąpieniu błędu) generowane jest przerwanie.

Obsługa błędów i wyjątków

W idealnych warunkach, w systemie komputerowym nie powinny występować żadne błędy. W rzeczywistości jednak tak nie jest, błędy pojawiają się przy obsłudze urządzeń zewnętrznych (np. uszkodzenie dysku, przerwy w zasilaniu, zakłócenie elektromagnetyczne itd.) ale również w wyniku błędów programistów w nieprzewidzianych przez nich sytuacjach itp.

Źródłem błędów sprzętowych mogą być w zasadzie wszystkie urządzenia komputera, niemniej wyróżnić można następujące grupy:

- błędy MC (*processor machine check*) generowane przez procesor w wyniku problemów z CPU, pamięcią cache i RAM oraz szyną systemową
- błędy SERR# i SMI (*chipset error signal*) – generowane przez główny układ płyty głównej. Sygnały SMI (*system management interrupt*) nie są nigdy korygowane sprzętowo
- błędy szyn danych (np. PCI, PCI Express)
- błędy urządzeń wejścia / wyjścia

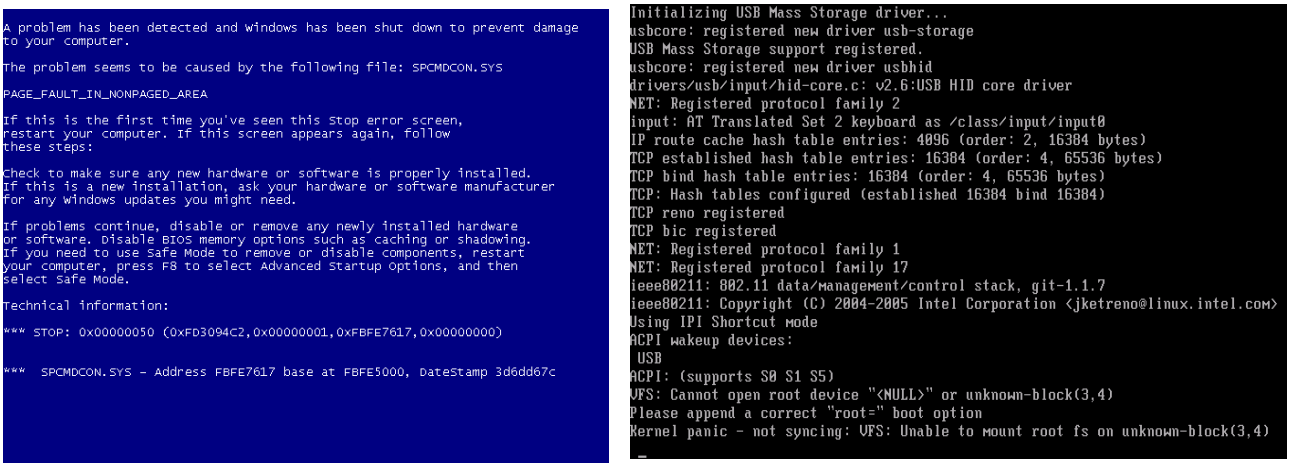
Najczęściej błędy zgłaszane są przez odpowiednie moduły systemu w formie niemaskowalnych przerwań (NMI). Jednocześnie ustawiane wartości specjalnych rejestrów urządzeń reprezentujące szczegółowe informacje o problemie.

Błędy sprzętowe są wykrywane najczęściej przez kontrolery urządzeń, które zwykle posiadają określone systemy ich detekcji. W momencie wykrycia błędu kontroler określa jego rodzaj i wagę oraz (jeżeli jest to możliwe) próbuje naprawić ten błąd. W zależności od rezultatu tej próby oraz wagi, błędy sprzętowe można podzielić na:

- poprawione przez kontroler i firmware (*correctable*) – kontroler wykrywa błąd a jego oprogramowanie (firmware) dokonuje korekcji
- niemożliwe do naprawy przez kontroler i firmware ale nie krytyczne (*non – fatal*) – kontroler wykrywa błąd, a jego oprogramowanie nie jest w stanie dokonać korekcji. Informacja o błędzie przekazywana jest do systemu operacyjnego, który podejmuje próbę naprawy sytuacji. Jeżeli próba ta powiedzie się, to system kontynuuje pracę, w innym wypadku zostaje awaryjnie zatrzymany
- krytyczne (*fatal*) – system operacyjny nie podejmuje próby naprawy błędu lecz od razu wykonuje awaryjne zatrzymanie

W przypadku błędów, które zostały poprawione na poziomie sprzętowym do systemu operacyjnego przekazywana jest jedynie informacja o ich wystąpieniu. Najczęściej jest ona zapisywana w dziennikach systemowych, czasami wyświetlany jest komunikat informacyjny dla użytkownika, niemniej po wystąpieniu tego rodzaju błędu praca oprogramowania jest kontynuowana bez

zakłóceń. Błędy, które nie mogą być naprawione przez sterownik powodują najczęściej awaryjne zamknięcie systemu, które wykonywane jest w w celu zapobieżenia uszkodzeniu danych po wystąpieniu poważnego błędu (efektem błędu może być nadpisanie pamięci operacyjnej co może prowadzić do nieprzewidzianych operacji wykonywanych przez programy i system operacyjny). Awaryjne zatrzymanie wiąże się w większości systemów operacyjnych z tzw. zrzutem pamięci[14] i wyświetleniem komunikatu o wystąpieniu błędu. Jedyną możliwością kontynuacji pracy jest restart całego systemu. W systemach Microsoft komunikat o awaryjnym zatrzymaniu jest wyświetlany w postaci charakterystycznego niebieskiego ekranu z tekstem zawierającym szczegóły awarii (tzw. *Blue Screen of Death* – niebieski ekran śmierci). W systemach Linux i Mac OS X najczęściej wyświetlany jest komunikat *Kernel Panic*:



Komunikaty informujące o awaryjnym zamknięciu systemu po wystąpieniu błędu krytycznego. Z lewej strony Microsoft Windows XP, z prawej – Linux

W zależności od konfiguracji systemu po wystąpieniu błędu krytycznego system może zainicjować restart komputera lub zatrzymanie pracy na ekranie awaryjnym.

W przypadku wystąpienia programowego (nie sprzętowego) błędu w obszarze jądra systemu operacyjnego (na przykład z powodu błędu w samym jądrze lub, co częstsze, błędu w sterownikach urządzeń) system postępuje dokładnie w ten sam sposób co przy wystąpieniu krytycznego błędu sprzętowego czyli wykonuje awaryjne zatrzymanie.

W przypadku błędów w programach pracujących w przestrzeni użytkownika, system operacyjny monitorujący ich pracę najczęściej dokonuje ich zamknięcia, bez inicjowania awaryjnego zatrzymania. Ze względu na pracę w izolowanej przestrzeni adresowej procesy użytkownika nie są w stanie naruszyć pamięci należącej do systemu operacyjnego.

Zamknięcie, zarówno systemu operacyjnego jak i programu użytkownika jest jednak niewygodnym rozwiązaniem z punktu widzenia użytkowników ponieważ często wiąże się z utratą niezapisanych na dysku twardym danych (np. tekstu napisanego w edytorze). Istnieją jednak pewne mechanizmy pozwalające na rozwiązanie tego problemu. Mechanizmami tymi są m.in. mikrorestarty, automatyczna naprawa danych, restarty sterowników oraz zabezpieczenie pamięci procesu przed niewłaściwymi operacjami.

Mikrorestarty (*micro-reboots*) to technika polegająca na zamknięciu i ponownym

14 Zrzut pamięci – zapis na dysku twardym obejmujący zawartość pamięci operacyjnej (danych i rejestrów procesora, licznika programu itd.) występujący przy awaryjnym zatrzymaniu systemu. Zawartość zrzutu pamięci (ang. *memory dump, core dump*) może służyć do analizy źródła problemu skutkującego zatrzymaniem systemu.

uruchomieniu fragmentu procesu, w którym wystąpił błąd [MREBOOT]. Ze względu na to, że restartowi nie podlega cały proces, a tylko aktualnie wykonywany przez komputer fragment, dane użytkownika przechowywane w aplikacji nie są tracone. Metoda ta wymaga oczywiście wsparcia w systemie operacyjnym oraz specjalnej konstrukcji oprogramowania jednakże w dużej liczbie przypadków jest skuteczna. W obecnej chwili rozwiązanie bazujące na mikrorestartach zastosowane jest w serwerze aplikacji J2EE (Java Platform, Enterprise Edition).

Interesującym rozwiązaniem pozwalającym w wielu wypadkach uniknąć utraty danych użytkownika (na skutek błędów aplikacji) jest zastosowany w Microsoft Windows 7 mechanizm FTH (*Fault Tolerant Heap* – Odporna na błędy sterta) [FTH]. Polega on na tym, że system zapamiętuje wystąpienie błędów polegających na próbie dostępu do zwolnionej uprzednio pamięci oraz przekroczenia zakresu zaalokowanej pamięci w aplikacji. Po kilkakrotnym wystąpieniu błędu tej samej natury w aplikacji, system automatycznie podejmuje środki zabezpieczające przed tymi zdarzeniami alokując większą ilość pamięci niż deklarowana przez aplikację oraz przechowując kopię zwolnionych obszarów pamięci na wypadek odwołań do niej. Oczywiście, rozwiązanie to wiąże się z niższą wydajnością i większym użyciem zasobów systemowych, niemniej wielokrotnie umożliwia pracę z aplikacją zawierającą błędy programistyczne.

Wyjątki w terminologii systemów operacyjnych to zdarzenia przerywające normalny ciąg wykonania programów, najczęściej stosowany do wywoływania funkcji obsługujących niestandardowe sytuacje (np. błędy). Można wyróżnić wyjątki wykrywane przez procesor komputera jak również wyjątki programowane:

- wyjątki wykrywane przez procesor:
  - niepowodzenie (*fault*) – błąd tego typu może być naprawiony w systemie, a proces go wywołujący wznowiony bez utraty ciągłości wykonania (błąd ten nie powoduje naruszenia zawartości rejestrów, po zakończeniu obsługi błędu proces może wrócić do normalnego ciągu wykonania)
  - pułapka (*trap*) – wyjątek podnoszony po wywołaniu operacji, powodującej wystąpienie błędu (np. dzielenia przez zero), który to błąd skutkuje przechwyceniem kontroli przez system operacyjny (i przejściem w tryb jądra), wykonujący pewne operacje mające na celu korekcję błędu. W momencie powrotu na tryb użytkownika i zwrócenia kontroli programowi wywoływany jest wyjątek typu *trap*. Głównym mechanizmem wykorzystującym te wyjątki są programy pomagające wykrywać błędy oprogramowania (*debuggery*)
  - zaniechanie (*abort*) – wyjątek podnoszony w momencie wystąpienia poważnego błędu (np. związany z awarią urządzenia). Wyjątki te są zgłaszane wtedy kiedy istnieje ryzyko że wewnętrzna struktura pamięci programu została naruszona. Po wystąpieniu tego wyjątku proces który go wywołał musi zostać przerwany
- wyjątki programowe – zdefiniowane w aplikacji użytkownika. Definicja wyjątku polega na określeniu sposobu jego obsłużenia (poprzez wywołanie funkcji służącej do wykonania ciągu operacji w razie wystąpienia zdarzenia wyjątkowego, najczęściej zachowania danych, wyświetlanie informacji dla użytkownika itd.) oraz na zaprogramowaniu miejsc, w których ten wyjątek może zostać rzucony (po sprawdzeniu jakiegoś warunku wywołania polecenia użytego języka programowania podnoszącego wyjątek, na przykład *throw* w języku C++). Przechwytywanie wyjątków odbywa się poprzez umieszczenia wywołania funkcji, która ma możliwość rzucania wyjątków w bloku oznaczonym specjalnymi instrukcjami (np. *try, catch* w C++). Przykład użycia wyjątków programowych znajduje się poniżej:

```
int v_iVariable = 1;
try
{

      if (v_iVariable > 0)
         throw 1;
      else if (v_iVariable < 0)
         throw 2;

 }
 catch (int v_iException)
 {
       printf("Wystąpił wyjątek nr %d ", v_iException);
 }
 return 0;
```

Wyjątki, z punktu widzenia procesu obsługi są podobne do przerwań systemowych, przerwania jednak najczęściej wywoływane są przez urządzenia fizyczne i są, przeciwnie do wyjątków, asynchroniczne w stosunku do wykonywanych procesów.

Komunikacja międzyprocesowa

W systemie komputerowym, w którym uruchomiane są różnorodne aplikacje często zachodzi potrzeba przekazywania komunikatów i danych pomiędzy tymi aplikacjami, wynikająca zarówno z wymogu współdzielenia informacji między procesami, ale także z wygody i niejednokrotnie możliwego przyspieszenia działania aplikacji. Istnieją różne metody komunikacji między procesami (**inter process communication – IPC**), różniące się wieloma parametrami, takimi jak ilość danych możliwych do przekazania, zasięg (tzn. działanie tylko w obrębie jednego lub wielu komputerów), opóźnienia transmisji itp.

Najprostszą metodą wymiany informacji, jest wymiana danych poprzez wspólny plik. Jest to bardzo mało wydajna i wymagająca synchronizacji metoda polegająca na zapisywaniu informacji do przekazania przez jeden z procesów do określonego pliku w pamięci (np. dysku) i odczytaniu tej informacji przez inny proces.

Popularną i wygodną metodą wymiany informacji jest tzw. pamięć dzielona (***shared memory***). Polega on na rezerwacji pewnego obszaru pamięci przez proces uruchamiający komunikację i uczynieniu go dostępnym dla innych procesów. Komunikujące się procesy odwołują się do tej pamięci w ten sam sposób (również fizycznie) jak do przynależnej sobie pamięci, dzięki czemu wydajność tej metody jest bardzo dobra, ograniczona przez szybkość transferu danych pomiędzy procesorem i pamięcią operacyjną. Większość systemów operacyjnych umożliwia wymianę danych poprzez pamięć dzieloną, standard API opisujący funkcje wykorzystywane w tej transmisji został opisany w standardzie POSIX. Co ważne, zarezerwowany jako dzielony segment pamięci jest obecny w systemie do momentu jego usunięcia przez proces, który ów segment stworzył lub do momentu wyłączenia systemu operacyjnego. Przykładowy kod źródłowy wykorzystujący pamięć dzieloną w systemie HP-UX znajduje się poniżej:

```
#include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/shm.h>

  #define SHM_SIZE 1024  /* make it a 1K shared memory segment */

  int main(int argc, char *argv[])
  {
    key_t key;
    int shmid;
    char *data;
    int mode;
```

```
    if (argc > 2) {
        fprintf(stderr, "usage: shmdemo [data_to_write]\n");
        exit(1);
    }

    /* make the key: */
    if ((key = ftok("shmdemo.c", 'R')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* connect to (and possibly create) the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
        perror("shmget");
        exit(1);
    }

    /* attach to the segment to get a pointer to it: */
    data = shmat(shmid, (void *)0, 0);
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }

    /* read or modify the segment, based on the command line: */
    if (argc == 2) {
        printf("writing to segment: \"%s\"\n", argv[1]);
        strncpy(data, argv[1], SHM_SIZE);
    } else
        printf("segment contains: \"%s\"\n", data);

    /* detach from the segment: */
    if (shmdt(data) == -1) {
        perror("shmdt");
        exit(1);
    }

    return 0;
}
```


(Źródło - http://beej.us/guide/)

Rozwinięciem tej metody jest obecna w systemach zgodnych z POSIX oraz Microsoft Windows technika mapowania pamięci (***memory-mapped file***), która pozwala na traktowanie zawartości pliku (niekoniecznie fizycznie istniejącego na nośniku, plikiem może być również fragment pamięci operacyjnej) przez współdzielące go procesy jako bloku pamięci należącego do przestrzeni adresowej każdego z nich. Odwołania do pliku muszą być jednak chronione przez obiekty synchronizacyjne aby uniknąć wzajemnego nadpisywania danych. Metoda ta pozwala osiągnąć dobrą wydajność wymiany danych w ramach tego samego komputera, jednak wymiana danych pomiędzy różnymi maszynami jest niemożliwa. Przeciwnie niż w przypadku pamięci dzielonej, plik wymiany (o ile nie jest zapisany na dysku) jest nietrwały, znika po zakończeniu procesu, dzięki któremu został utworzony.

Przykładowy kod źródłowy w języku C++ wykorzystujący mapowanie pamięci:

Tworzenie mapowanego pliku i wysyłanie wiadomości (proces 1)	Otwieranie mapowanego pliku, odbieranie wiadomości (proces 2)

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>
 	
#define BUF_SIZE 256
TCHAR szName[]=TEXT("Global\\MyFileMappingObject");
TCHAR szMsg[]=TEXT("Message from first process.");
 	
int _tmain()
{
  HANDLE hMapFile;
  LPCTSTR pBuf;
 	
  hMapFile = CreateFileMapping(
  INVALID_HANDLE_VALUE, // use paging file
	#include <windows.h>
	#include <stdio.h>
	#include <conio.h>
	#include <tchar.h>
	#pragma comment(lib, "user32.lib")
	
	#define BUF_SIZE 256
	TCHAR szName[]=TEXT("Global\\MyFileMappingObject");
	
	int _tmain()
	{
	  HANDLE hMapFile;
	  LPCTSTR pBuf;
	
	  hMapFile = OpenFileMapping(
	  FILE_MAP_ALL_ACCESS, // read/write access
	  FALSE, // do not inherit the name
	  szName); // name of mapping object

```
            NULL,                    // default security
            PAGE_READWRITE,          // read/write access
            0,                       // max. object size
            BUF_SIZE,                // buffer size
            szName);                 // name of mapping object
  if (hMapFile == NULL)
  {
        _tprintf(TEXT("Could not create file mapping object (%d).\n"),
        GetLastError());
        return 1;
  }
  pBuf = (LPTSTR) MapViewOfFile(hMapFile,   //handle to map object
                FILE_MAP_ALL_ACCESS,        //read/write permission
                0,   0,   BUF_SIZE);
  if (pBuf == NULL)
  {
             _tprintf(TEXT("Could not map view of file (%d).\n"),
             GetLastError());
             CloseHandle(hMapFile);
            return 1;
  }
  CopyMemory((PVOID)pBuf, szMsg, (_tcslen(szMsg) *
sizeof(TCHAR)));
   _getch();
  UnmapViewOfFile(pBuf);
  CloseHandle(hMapFile);
  return 0;
}
```

```
  if (hMapFile == NULL)
  {
        _tprintf(TEXT("Could not open file mapping object (%d).\n"),
        GetLastError());
        return 1;
  }

  pBuf = (LPTSTR) MapViewOfFile(hMapFile,    // handle to map object
           FILE_MAP_ALL_ACCESS,              // read/write permission
           0, 0,  BUF_SIZE);

  if (pBuf == NULL)
  {
        _tprintf(TEXT("Could not map view of file (%d).\n"),
        GetLastError());
        CloseHandle(hMapFile);
        return 1;
  }

  MessageBox(NULL, pBuf, TEXT("Process2"), MB_OK);
  UnmapViewOfFile(pBuf);
  CloseHandle(hMapFile);
  return 0;
}
```

(źródło – Microsoft Developer Network, Windows Developer Center)

Prostym sposobem komunikacji z procesami (przeznaczonym do wysyłania informacji, ale nie danych) są tzw. sygnały (***signals***). Sygnał charakteryzuje się tym, że przesłany do procesu (jako informacja o pewnym zdarzeniu) powoduje przerwanie ciągu wykonywania się procesu przez system operacyjny. W wypadku, kiedy w procesie zaimplementowano procedurę przechwytującą dany sygnał, jest ona wykonywana, w przeciwnym razie wykonywana jest procedura domyślna dla sygnału. Nie można przechwycić jedynie sygnału SIGSTOP (zatrzymaj wykonanie procesu) i SIGKILL (zakończ proces natychmiast). Sygnały występują głównie w środowiskach zgodnych z POSIX, szeroko znanymi oprócz wyżej wymienionych są np. SIGTERM (zakończ proces) i SIGTSTP (zatrzymaj proces wywoływany z poziomu terminala przez CTRL + Z). Pliki źródłowe obrazujące wykorzystanie sygnałów w systemie Linux znajdują się poniżej:

Wysyłanie sygnału

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

int main(int argc, char *argv[])
{
    pid_t v_iPID = getpid(); // pobierz ID procesu
    kill(v_iPID, SIGSTOP); // wyślij do procesu sygnał STOP

    return 0;
}
```

Przechwytywanie sygnału

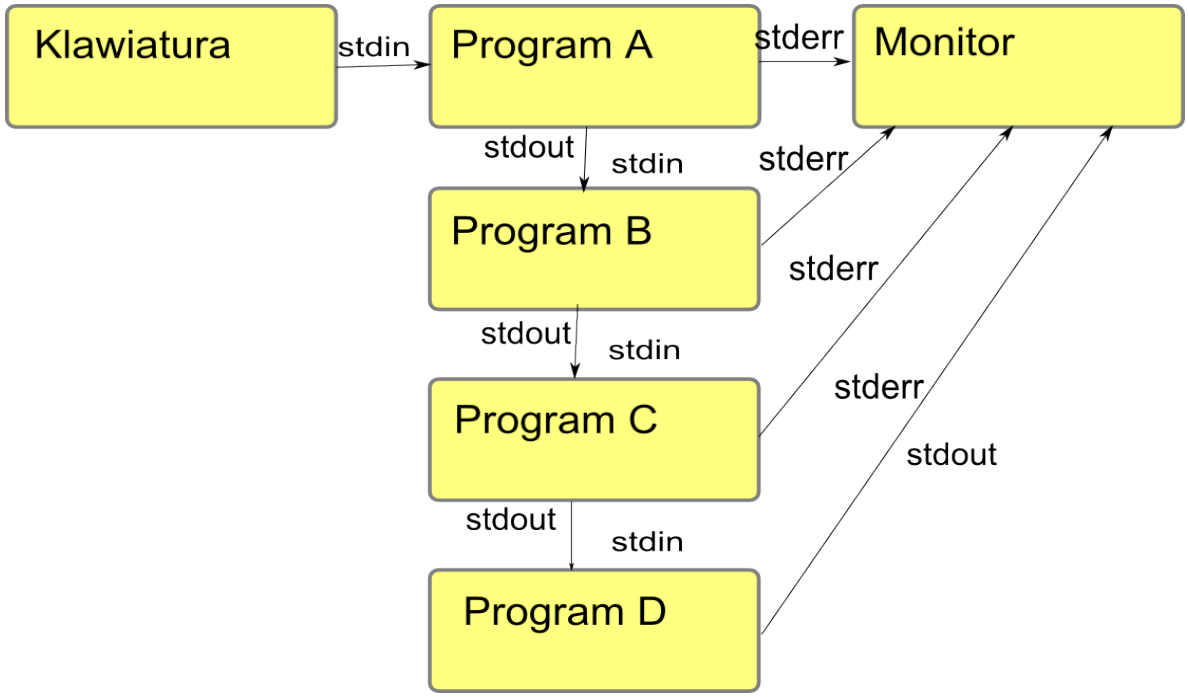
```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

// funkcja przechwytująca sygnał
void SigINTHandler(int p_iSignal)
{
   printf("Sygnał przechwycony");
   signal(SIGTSTP, SigINTHandler); /* uruchom przechwytywanie na nowo */
}

int main(int argc, char *argv[])
{
    signal(SIGTSTP, SigINTHandler); /* uruchom przechwytywanie sygnału
SIGTSTP */

    while (true)
   {
        Sleep(10);
   }

    return 0;
}
```

W celu przeciwdziałania nakładaniu się sygnałów[15] system operacyjny zapewnia mechanizmy maskowania sygnałów. Więcej informacji na ten temat można znaleźć w manualu

15 Sytuacja, w której kolejny sygnał zostanie wysłany do aplikacji w trakcie przetwarzania poprzedniego.

Linuxa.

W systemach POSIXowych do przekazywania danych pomiędzy procesami często wykorzystywane są tzw. rury (***pipes***), czego szczególnie znanym przykładem jest łączenie poleceń w konsoli np.: ls |grep test.dat (wywołanie listingu katalogu i ukrycie wszystkich linii z wyjątkiem tej zawierającej tekst „test.dat").



Przykład wykorzystania pipe do przekazywania danych między programami, wejściem i wyjściem terminala

Pipe'y służą do przekazywania wyjściowych danych z jednego procesu do innego (jak na powyższym rysunku), ale także do obsługi wyjścia diagnostycznego (stderr). Transfer danych jest buforowany aby uniknąć strat informacji w przypadku wolnego odbierania ich przez kolejny proces. Przykładowy kod źródłowy pokazujący użycie rur w programie znajduje się poniżej:

```
void ProcessFiles()
{
   char   psBuffer[128];
   FILE   *pPipe;

            char path[200];
            sprintf(path,"%s \"%s\"", "dir /A-D /B", "c:\\temp\\");
            if( (pPipe = _popen( path , "rt" )) == NULL )
                        exit( 1 );

            while( !feof( pPipe ) )
            {
                        if( fgets( psBuffer, 128, pPipe ) != NULL )
                        {

                                    printf( psBuffer );  // wypisz nazwy plików w konsoli

                        }
            }
}
```

Kolejną metodą wymiany danych między procesami jest tzw. kolejka komunikatów (***message queue***). Poszczególne komunikaty są wysyłane przez aplikację do kolejki i przebywają tam do momentu aż proces, do którego są przeznaczone je odbierze. Najczęściej technika te jest używana w systemach operacyjnych czasu rzeczywistego z racji jej ścisłego powiązania z procesem zarządzania czasem procesora oraz tego, że kolejki komunikatów pracują w trybie FIFO (*first in, first out* czyli pierwszy na wejściu, pierwszy na wyjściu) a zatem komunikaty odbierane są przez

proces w tej samej kolejności, w której były do niej wysłane przez proces nadający[16]. Kolejka utworzona przez dowolny proces nie zostaje zniszczona razem z zakończeniem procesu, musi zostać usunięta, podobnie jak segmenty pamięci dzielonej. Co ważne, kolejkami można wysyłać dowolne dane, zarówno proste typy jak i struktury, jednakże zawsze pierwsze pole struktury musi być zdefiniowane jako zmienna typu long. Przykładowy kod źródłowy programów wysyłających i odbierających dane z kolejki zamieszczony został poniżej:

Wysyłanie danych

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
   long mtype;
   char mtext[200];
};

int main(void)
{
   struct my_msgbuf buf;
   int msqid;
   key_t key;

   if ((key = ftok("kirk.c", 'B')) == -1) {
      perror("ftok");
      exit(1);
   }

   if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
      perror("msgget");
      exit(1);
   }

   printf("Enter lines of text, ^D to quit:\n");

   buf.mtype = 1; /* we don't really care in this case */

   while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
      int len = strlen(buf.mtext);

      /* ditch newline at end, if it exists */
      if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';

      if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
         perror("msgsnd");
   }

   if (msgctl(msqid, IPC_RMID, NULL) == -1) {
      perror("msgctl");
      exit(1);
   }

   return 0;
}
```

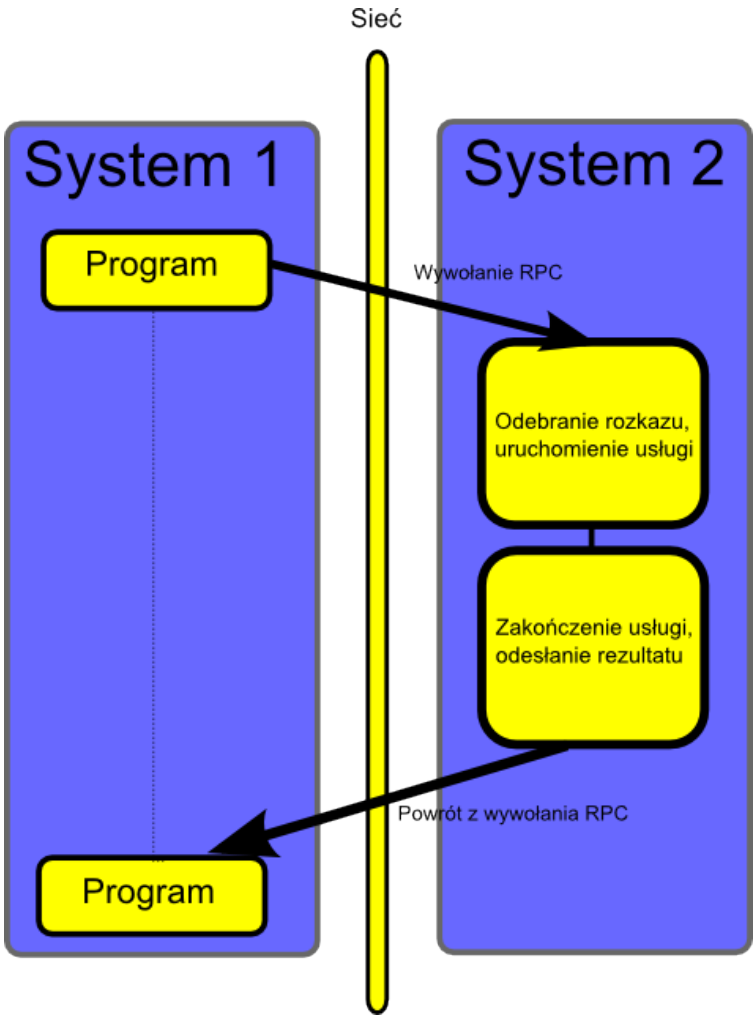
Odbieranie danych

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
   long mtype;
   char mtext[200];
};

int main(void)
{
   struct my_msgbuf buf;
   int msqid;
   key_t key;

   if ((key = ftok("kirk.c", 'B')) == -1) {  /* same key as kirk.c */
      perror("ftok");
      exit(1);
   }

   if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue */
      perror("msgget");
      exit(1);
   }

   printf("spock: ready to receive messages, captain.\n");

   for(;;) { /* Spock never quits! */
      if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1) {
         perror("msgrcv");
         exit(1);
      }
      printf("spock: \"%s\"\n", buf.mtext);
   }

   return 0;
}
```

Źródło: http://beej.us

W komunikacji międzyprocesowej wykorzystywane są również takie mechanizmy jak zdalne wywołanie procedury (***Remote Procedure Call – RPC***) oraz gniazda (***sockets***). Zdalne wywołanie procedury polega na wykonaniu pewnej procedury przez proces w innej przestrzeni adresowej, przy czym programista nie musi przewidzieć i zaprogramować szczegółów tego zdarzenia, aplikacja uruchamiana lokalnie i zdalnie wygląda tak samo. Poprzez inną przestrzeń adresową rozumie się tutaj również inny komputer dostępny poprzez sieć. RPC implementuje model komunikacyjny klient-serwer bez wymagania od programisty uwzględnienia tego, że komunikacja ta może odbywać się poprzez sieć.

16 Zachowanie to można jednak zmieniać poprzez podanie specjalnych argumentów do funkcji odbierającej (więcej informacji w manualu funkcji linuxa msgrcv()).



Model zdalnego wywołania procedury (RPC)

Przekazywanie danych poprzez RPC odbywa się w ciekawy sposób, mianowicie poprzez użycie mechanizmu zwanego *External Data Representation* (XDR). Polega on na serializacji danych, przesłaniu pakietu przez sieć i reasemblacji danych przed przekazaniem ich do procesu odbierającego. W ten sposób można przekazywać między modułami nie tylko proste typy danych, ale nawet wskaźniki do pamięci zawierającej zaawansowane struktury danych. Kody źródłowe przykładowych aplikacji wymieniających się danymi dzięki RPC znajdują się poniżej:

Serwer	klient

#include <cassert>
#include <stdexcept>
#include <iostream>
#ifdef WIN32
#include <windows.h>
#else
#include <unistd.h>
#endif
#include <xmlrpc-c/base.hpp>
#include <xmlrpc-c/registry.hpp>
#include <xmlrpc-c/server_abyss.hpp>
using namespace std;
 	
#ifdef WIN32
#define SLEEP(seconds) SleepEx(seconds * 1000);
#else
#define SLEEP(seconds) sleep(seconds);
#endif
class sampleAddMethod : public xmlrpc_c::method {
public:
sampleAddMethod() {
// signature and help strings are documentation -- the client
// can query this information with a system.methodSignature and
// system.methodHelp RPC.
this->_signature = "i:ii";
	#include <cstdlib>
	#include <string>
	#include <iostream>
	#include <xmlrpc-c/girerr.hpp>
	#include <xmlrpc-c/base.hpp>
	#include <xmlrpc-c/client_simple.hpp>
	using namespace std;
	int main(int argc, char **) {
	if (argc-1 > 0) {
	cerr << "This program has no arguments" << endl;
	exit(1);
	}
	try {
	string const serverUrl("http://localhost:8080/RPC2");
	string const methodName("sample.add");
	xmlrpc_c::clientSimple myClient;
	xmlrpc_c::value result;
	myClient.call(serverUrl, methodName, "ii", &result, 5, 7);
	int const sum = xmlrpc_c::value_int(result);
	// Assume the method returned an integer; throws error if not
	cout << "Result of RPC (sum of 5 and 7): " << sum << endl;
	} catch (exception const& e) {
	cerr << "Client threw error: " << e.what() << endl;
	} catch (...) {
	cerr << "Client threw unexpected error." << endl;

```
            // method's result and two arguments are integers
        this->_help = "This method adds two integers together";
    }
    void
    execute(xmlrpc_c::paramList const& paramList,
            xmlrpc_c::value *   const  retvalP) {
        int const addend(paramList.getInt(0));
        int const adder(paramList.getInt(1));
        paramList.verifyEnd(2);
        *retvalP = xmlrpc_c::value_int(addend + adder);
        // Sometimes, make it look hard (so client can see what it's like
        // to do an RPC that takes a while).
        if (adder == 1)
            SLEEP(2);
    }
};
int  main(int const,  const char ** const)
{
    try {
        xmlrpc_c::registry myRegistry;
        xmlrpc_c::methodPtr const sampleAddMethodP(new sampleAddMethod);
        myRegistry.addMethod("sample.add", sampleAddMethodP);
        xmlrpc_c::serverAbyss myAbyssServer(
            myRegistry,
            8080,              // TCP port on which to listen
            "/tmp/xmlrpc_log"  // Log file
            );
        myAbyssServer.run();
        // xmlrpc_c::serverAbyss.run() never returns
        assert(false);
    } catch (exception const& e) {
        cerr << "Something failed.  " << e.what() << endl;
    }
    return 0;
}
```

Źródło: SCM repositories, Sourceforge.net, 3.X.2007, http://xmlrpc-c.sourceforge.net/example-code.php

Podobną do RPC metodą komunikacji między aplikacjami, szczególnie znajdującymi się na różnych komputerach jest komunikacja z użyciem gniazd (***sockets***). Uogólniając, procesy komunikują się wówczas z wykorzystaniem protokołów sieciowych takich jak UDP oraz TCP, a także ICMP i IGMP oraz protokołów przeznaczonych tylko do komunikacji międzyprocesowej takich jak UDS (*Unix Domain Sockets*).

Komunikacja międzyprocesowa jest bardzo ważnym zagadnieniem, bez którego niemożliwe byłoby działanie przeważającej części dzisiejszych systemów komputerowych i aplikacji. W bieżącym rozdziale przedstawiono kilka najczęściej używanych metod IPC, z których większość jest obsługiwana przez wszystkie główne systemy operacyjne. Temat implementacji metod komunikacji IPC jest jednak dużo szerszy i zainteresowanych poszerzeniem wiedzy autor odsyła do zapoznania się z technologiami takimi jak: Microsoft Component Object Model, Distributed Component Object Model, Dynamic Data Exchange, Windows Communication Foundation, Apple Events, Mach Ports (Mac OS X), System V message queues, DCOP, CORBA, D-Bus.

Powłoka

Powłoka systemu operacyjnego (***operating system shell***) to oprogramowanie, które zapewnia interfejs pomiędzy użytkownikiem a jądrem systemu operacyjnego. Jest najbardziej zewnętrzną warstwą OS, ukrywającą wszystkie jego mechanizmy wewnątrz (stąd nazwa). Zapewniany przez powłokę interfejs może być interfejsem tekstowym (tzw. interfejs linii poleceń – ***Command Line Interface, CLI***) lub graficznym (***Graphical User Interface – GUI***). Generalnym zastosowaniem powłoki jest uruchamianie programów użytkownika poprzez utworzenie procesu potomnego, uruchomienie wskazanego programu w ramach tego procesu, wyświetlenie ewentualnej informacji zwracanej przez program i oczekiwanie na jego zakończenie. Powłoka prezentuje system operacyjny jako zbiór rozmaitych usług, dostęp do których odbywa się poprzez wywołania systemowe (*system calls*).

Najbardziej znane powłoki systemowe przedstawione są w poniższej tabeli:

Powłoki tekstowe	Powłoki graficzne
sh (Unix, Linux) bash (Unix, Linux) ksh (Unix, Linux) 4DOS (Microsoft DOS, Microsoft Windows NT, OS/2) command.com (Microsoft DOS) cmd (Microsoft Windows) PowerShell (Microsoft Windows) Amiga CLI (AmigaOS) GoogleShell (interfejs wyszukiwarki google search)	Blackbox (X Window) Gnome (X Window) KDE (X Window) Microsoft Windows Shell (Microsoft Windows) Stardock WindowBlinds (Microsoft Windows) LiteStep (Microsoft Windows) Amiga Workbench (AmigaOS) Finder (Mac OS X)

System plików

Pierwsze komputery, pracujące w trybie wsadowym pobierały dane i zwracały wyniki obliczeń w postaci kart perforowanych, taśm magnetycznych itp. Nie posiadały więc żadnego mechanizmu przechowywania jakichkolwiek zbiorów danych. Kolejny poziom ewolucji komputerów i systemów operacyjnych umożliwiał już jednak zapamiętywanie i odczytywanie danych użytkowników tak, aby były one dostępne po np. zakończeniu procesów, wyłączeniu urządzenia itd. Obecnie w zasadzie każde urządzenie komputerowe potrafi odczytywać i zapisywać dane z jakiejś formy pamięci stałej (taśmach, dyskietkach, dyskach twardych, dyskach optycznych, pamięci flash itd.). Narzędziem umożliwiającym organizację danych na nośnikach, jest system plików (czyli zbiorów danych), często zintegrowany z systemem operacyjnym z racji głębokich powiązań z jądrem systemu i wywołaniami pracującymi w trybie specjalnych uprawnień (kernela).

Głównym celem systemów plików (SP) jest jak wspomniano przechowywanie danych w sposób zorganizowany, umożliwiający łatwe ich zapisanie oraz odnalezienie i odczytanie. Ważnymi cechami SP są:

- długoterminowość (przechowywanie danych po wylogowaniu się użytkownika je tworzącego, wyłączeniu zasilania itp.)
- możliwość wykorzystywania danych przez różne procesy
- możliwość organizacji plików w struktury odzwierciedlające wzajemne relacje między nimi (np. w strukturę katalogów i podkatalogów)[17]

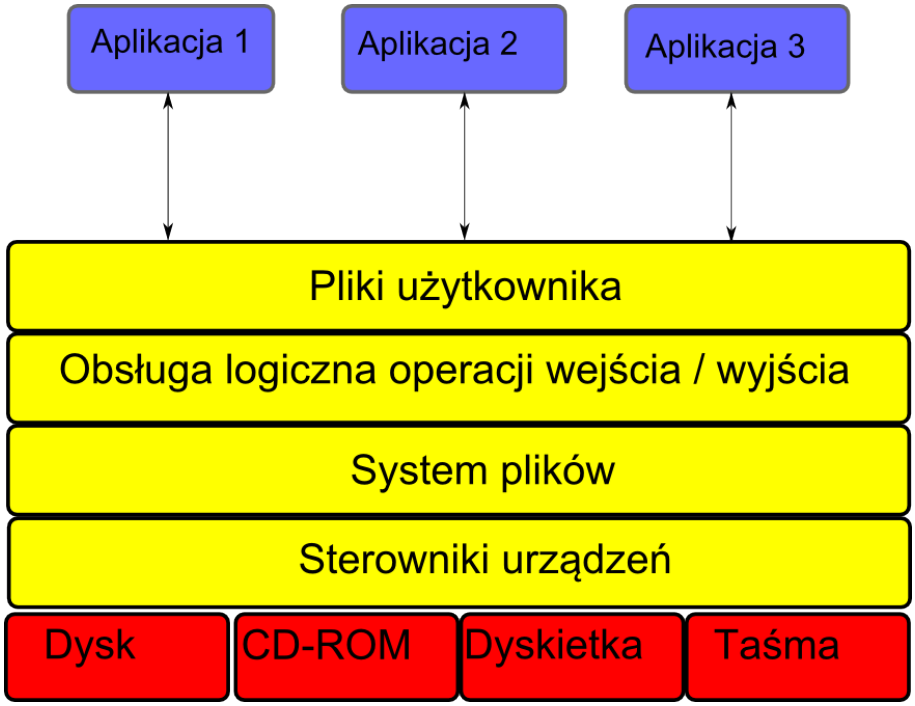
Większość współczesnych systemów plików posiada wspólny zestaw operacji, które są możliwe do wykonania. Są to operacje:

- tworzenia (***create***) – w określonym miejscu powstaje plik z określonym zestawem atrybutów
- otwarcia (***open***) – załadowana do pamięci operacyjnej zostaje lista atrybutów oraz adresy fizyczne lokalizacji w której znajduje się plik (np. numery bloków na dysku twardym)
- czytania (***read***) – zawartość (lub jej część) zostaje wczytana do pamięci operacyjnej
- zapisu (***write***) – zawartość określonego obszaru pamięci operacyjnej zostaje zapisana w pliku

17 Klasyfikacja za [STALLING]

- odwołania do konkretnego miejsca w pliku (***seek***) – wskaźnik pliku zostaje ustawiony w żądanym miejscu co pozwala czytać interesujący fragment danych
- pobrania listy atrybutów przypisanych do pliku (***get attributes***)
- ustawienia jednego lub grupy atrybutów przypisanych do pliku (***set attributes***)
- zamknięcia (***close***) – informacje zostają zapisane fizycznie w urządzeniu przechowującym dane, a pamięć operacyjna przechowująca informacje związane z otwartym plikiem zostaje zwolniona
- zmiany nazwy (***rename***)
- usuwania (***delete***) – usunięcie pliku z dysku najczęściej polega na usunięciu informacji o jego fizycznym położeniu (np. z tablicy alokacji plików). Miejsce zajmowane przez dane nie zostaje wyczyszczone (np. nadpisane zerami). Miejsce to zostaje nadpisane dopiero w momencie kiedy system operacyjny zapisze tam nowe dane (inny plik).[18]

System plików wraz z biblioteką funkcji służących do jego obsługi można przedstawić za pomocą schematu warstwowego jak poniższym rysunku:



Schemat warstwowy obsługi systemu plików

Aplikacje i programy użytkowników komunikują się z systemem poprzez wywołania systemowe. W zależności od programisty, pliki mogą być traktowane jako binarne, tekstowe, przechowujące rekordy itd. Wywołania systemowe przetwarzane są przez warstwę odpowiedzialną za komunikację z urządzeniem, przetwarzającą informacje określające plik użytkownikom (nazwę, położenie, atrybuty) na informacje o fizycznym lokalizacji danych w nim zawartych na nośniku. Warstwa ta zarządza również ewentualnym buforowaniem pliku (lub jego części) w pamięci operacyjnej komputera. Odwołania do fizycznych parametrów nośnika (np. numer sektora) są przekazywane do

18 Lista operacji za Tannenbaum

sterownika urządzenia fizycznego, który dba o przekazanie odpowiednich sygnałów do układów elektronicznych nośnika (np. pozycjonowania głowicy w dysku twardym).

Logiczna organizacja plików

W pierwszych rozwiązaniach wszystkie pliki użytkowników przechowywane były w tym samym miejscu. W niektórych współczesnych systemach spotyka się takie rozwiązania (np. w prostych mikrokontrolerach), przeważająca jednak większość umożliwia utworzenie pewnej struktury, która ułatwia rozmieszczanie plików, udostępnianie ich różnym użytkownikom itd. Przechowywanie wszystkich danych w jednym katalogu (tzw. ***root directory***) poza brakiem organizacji i istotnymi ograniczeniami (np. niemożnością wystąpienia więcej niż jednego pliku o danej nazwie) łączy się z bardzo istotnym niebezpieczeństwem w systemach przeznaczonych dla wielu użytkowników, kiedy mogą oni sobie nawzajem nadpisywać istniejące pliki. Wydaje się niemożliwe aby tego typu rozwiązania mogły być stosowane w systemach obsługujących więcej niż jednego użytkownika, ale np. pierwszy superkomputer z tranzystorami krzemowymi (Control Data Corporation 6600 – 1963 rok) posiadał właśnie tego typu system plików. Jeden poziom katalogów posiadał również pierwszy komputer Apple Macintosh.

Prostym z punktu widzenia implementacji rozwinięciem jednokatalogowego systemu plików był system dwupoziomowy, umożliwiający utworzenie w głównym katalogu podkatalogów o nazwach przypisanych do każdego z użytkowników, w których mogli oni zapisywać swoje dane. Czasem implementacja ta umożliwiała również tworzenie kilku dodatkowych poziomów podkatalogów (np. VAX firmy Digital Equipment Corporation umożliwiał utworzenie 9 poziomowej struktury katalogów).

Współczesne systemy plików umożliwiają tworzenie praktycznie nieograniczonej liczby poziomów podkatalogów. Struktura tych katalogów jest określana mianem drzewiastej, którą obrazuje RYSUNEK. Przykładem systemu plików o strukturze drzewiastej jest rozwiązanie FAT obecne np. w systemie operacyjnym MSDOS. Pewną wadą takiej struktury jest niemożność współdzielenia plików w różnych katalogach, co umożliwia system plików zbudowany na zasadzie grafów acyklicznych umożliwiający tworzenie odnośników (***hard link***) do plików znajdujących się w innych katalogach. Tego typu rozwiązanie jest wygodniejsze i oferuje więcej możliwości niż zwykła struktura drzewiasta, stwarza jednak jednocześnie więcej problemów w implementacji. Przykładem systemu plików opartego o grafy acykliczne jest system plików Unixa.

W większości systemów operacyjnych struktura katalogów rozpoczyna się od głównego katalogu (***root directory***), w którym znajdują się np. podkatalogi użytkowników, urządzeń itd. Systemy Microsoftu używają jednak innej notacji, katalog *root* jest w nich ukryty, a użytkownik ma dostęp do poszczególnych dysków (lub partycji) przypisanych do kolejnych liter alfabetu.

Pliki pozwalają na organizację danych użytkowników, z czego wynika konieczność ich identyfikacji. W praktycznie wszystkich systemach mogą być one nazywane wedle życzenia użytkownika. Proste systemy plików, takie jak np. FAT16 zezwalały na tworzenie nazw według ściśle określonego szablonu – tzw. standardu 8+3, gdzie nazwa pliku mogła składać się z 8 znaków, a jego rozszerzenie – z 3. Większość obecnie używanych systemów plików umożliwia nadawanie nazw o długości 255 znaków, które, w zależności od implementacji, mogą być tylko literami i cyframi, ewentualnie również znakami specjalnymi (np. &$^#). W niektórych rozwiązaniach system odróżnia małe i duże litery (np. Unix), w niektórych nie (MSDOS). W rodzinie systemów Microsoftu rozszerzenie pliku wskazuje systemowi co należy z nim zrobić (np. exe – plik wykonywalny, dll – biblioteka ładowana dynamicznie, txt – plik tekstowy), w Linuxie z kolei rola pliku jest określana na podstawie jego zawartości, pomimo że plikom można nadawać rozszerzenia.

Z plikiem, oprócz jego nazwy może być skojarzona pewna liczba atrybutów. Najczęściej stosowane atrybuty to:

- identyfikator właściciela
- uprawnienia (właściciela, grupy, innych użytkowników) takie jak odczyt, zapis, wykonanie
- flaga „tylko do odczytu”
- flaga „ukryty”
- flaga „systemowy” - określająca plik niezbędny do działania systemu operacyjnego
- flaga „do archiwizacji” - określająca że plik został zmodyfikowany od momentu ostatniej archiwizacji
- data i czas utworzenia
- data i czas ostatniej modyfikacji
- data i czas ostatniego dostępu

Część flag służy celom informacyjnym (np. data i czas utworzenia i modyfikacji), inne z kolei chronią dane zawarte w pliku przed nieautoryzowanym użyciem lub zmianami (uprawnienia, flaga „tylko do odczytu”).

Fizyczna organizacja danych

Wszystkie urządzenia służące do przechowywania danych (dyski twarde, dyski optyczne, pamięć flash) są zorganizowane w formie tablicy sektorów (***physical sector***) o pewnym rozmiarze (liczbie bajtów). Rolą systemu plików obsługującego dane urządzenie jest powiązanie tych bloków ze zbiorami danych czyli plikami, które są logiczną reprezentacją zestawów danych użytkownika. Najczęściej sektory są organizowane w grupy zwane klastrami (***cluster***), które reprezentują najmniejszą porcję danych, która może być wykorzystana do przechowywania danych w danym systemie plików.

Zagadnienie fizycznego rozmieszczania danych na dysku jest podobne do zagadnienia alokacji pamięci operacyjnej dla procesów. Najprostszą metodą jest tworzenie ciągłych plików (***contigous allocation***), z których nowo tworzony rozpoczyna się od razu za ostatnio utworzonym (RYSUNEK). Niestety, to rozwiązanie ma dwie wady, mianowicie brak możliwości zwiększania objętości pliku (ponieważ, z wyjątkiem ostatniego pliku, miejsce na rozrost danych jest zajęte przez kolejne pliki) a także szybką fragmentację przestrzeni dyskowej (wystąpienie dziur po usuniętych plikach, które mogą być wykorzystane tylko dla danych o rozmiarze nie większym niż dziura – RYSUNEK). Zaletami tego typu alokacji przestrzeni dyskowej są prostota implementacji (plik jest określany dwoma wartościami – adresem jego początku oraz rozmiarem) oraz szybkim odczytem i zapisem danych – wymagającym tylko jednej operacji przeszukiwania dysku – do pozycjonowania głowicy na początku pliku, kolejne bajty są czytane sekwencyjnie, bez konieczności wyszukiwania. Z racji przedstawionych powyżej krytycznych wad tego rozwiązania jego zastosowanie ogranicza się do nośników, na których dane zapisywane są tylko raz i nie ma możliwości ich modyfikacji – dyskach optycznych (CD, DVD).

Metodą niejako odwrotną w stosunku do przedstawionej powyżej jest metoda połączonej listy bloków (***linked list of blocks***), w której dane z pliku przechowywane są w blokach w ogólności

rozrzuconych losowo na dysku twardym, przy czym każdy z bloków oprócz porcji danych zawiera wskaźnik (adres) kolejnego w kolejności bloku. Rozwiązanie to nie jest obarczone wadami *contigous allocation* (pliki mogą rozrastać się dowolnie w ramach dysku, a nowy plik zapisywany po usunięciu poprzedniego może mieć dowolny rozmiar). Poważną wadą listy bloków jest jednak bardzo słaba wydajność w przypadku chęci odczytywania plików nie od początku, a od wskazanego miejsca – wiąże się to z przeczytaniem wszystkich poprzedzających interesujący fragment bloków, bez tej operacji niemożliwe jest odnalezienie fizycznego miejsca na dysku, w którym ten fragment jest zapisany. Dodatkowo pewna część przestrzeni dyskowej jest tracona na przechowywanie adresów kolejnych bloków. Wady te można jednak zniwelować poprzez przeniesienie informacji o adresach bloków do pamięci operacyjnej (tablica przechowująca takie dane zwana jest tablicą alokacji plików – ***file allocation table*** - ***FAT***), do której dostęp jest zdecydowanie szybszy niż do dysku. Niestety, optymalizacja ta powoduje powstanie kolejnych problemów, mianowicie zwiększa w znacznym stopniu zajętość pamięci operacyjnej (np. dla 100 gigabajtowego dysku o blokach 4 kilobajtowych potrzeba 25 milionów rekordów, co z kolei przy założeniu rozmiaru rekordu (4 B) powoduje konieczność zarezerwowania 100 MB pamięci RAM. W obecnych czasach taka ilość pamięci nie jest wprawdzie niespełnialnym wymaganiem, jednak mając np. dysk 500GB, strata 0.5 GB RAM na samą informację o plikach jest nieakceptowalna. Kolejny problem związany z przechowywaniem takiej tablicy w pamięci RAM jest konieczność jej zapisywania na dysku (aby informacja o położeniu plików przetrwała np. wyłączenie komputera) co przy takiej zajętości pamięci w znacznym stopniu opóźnia np. proces wyłączania systemu.

Microsoft Windows (New Technology File System - NTFS)

Następcą systemu plików FAT (w różnych odmianach: FAT12, FAT16, FAT32) jest, używany aktualnie w systemach Microsoft Windows (od NT 3.1) system NTFS. Przestrzeń dyskowa jest zorganizowana jako lista bloków, których rozmiar jest ustalony, najczęściej na 4kB (ale w zależności od zastosowań może być dowolną wartością będącą potęgą liczby 2 w zakresie od 512 do 65536 bajtów). Maksymalny rozmiar woluminu to 256 TB przy blokach 64kB.

Informacje na temat plików i katalogów, ich rozmieszczenia na dysku i przyporządkowania do fizycznych bloków przechowywane są w tzw. Głównej Tablicy Plików (***Master File Table – MFT***). Struktura ta to lista rekordów o rozmiarze 1kB przechowywana jako zwykły plik na dysku, którego zawartość opisuje. System informacyjny odnajduje tą strukturę przy uruchomieniu w wyniku sprawdzenia adresu jej początku zamieszczonego w *boot block*[19]. Informacja ta jest umieszczana podczas formatowania dysku (tworzenia systemu plików). MFT jest strukturą dynamiczną, to znaczy mogącą zmieniać rozmiar jeżeli zachodzi ku temu konieczność (np. przybywa plików). Maksymalna liczba przechowywanych rekordów to 2^{48} czyli 281474976710656. Każdy z rekordów przechowuje informację o jednym pliku (i jego atrybutach). Ponieważ w skład tej informacji wchodzą również adresy bloków, w których fizycznie przechowywane są dane zawarte w pliku, to przy dużej fragmentacji zapisu (i odpowiednio dużym rozmiarze pliku) adresów tych może być więcej niż możliwe do zmieszczenia w 1kB. W tym przypadku informacja o adresach bloków przechowywana jest w dwóch (lub więcej) blokach, przy czym każdy z nich przechowuje również informację o tym czy, i w którym miejscu znajduje się kolejny rekord MFT.

Plik MFT poza wspomnianymi powyżej rekordami odnoszącymi się do zwykłych plików i katalogów na opisywanym dysku (partycji) przechowuje wskaźniki do plików specjalnych, zawierających dodatkowe informacje o partycji (ich nazwy rozpoczynają się od znaku $). Są to (w kolejności licząc od początku pliku MFT):

19 Boot block (boot sector)

0: $MFT – plik MFT (położenia wszystkich jego segmentów)

1: $MFTMirr – kopia pliku MFT (niepełna, przechowywane są tylko najistotniejsze dane)[20]

2: $LogFile – plik zawierający opis operacji wykonywanych na plikach[21]

3: $Volume – plik zawierający dane woluminu – unikalny identyfikator, nazwę, wersję użytego systemu plików oraz inne parametry

4: $AttrDef – tablica wiążąca nazwy atrybutów, które mogą być nadawane plikom i katalogom z numerycznymi odpowiednikami

5: . - plik reprezentujący główny katalog woluminu

6: $Bitmap – mapa reprezentująca używane i wolne bloki dysku

7: $Boot – plik służący do uruchamiania systemu (zawiera kod ładujący i uruchamiający system operacyjny)

8: $BadClus – plik przechowujący informacje o uszkodzonych blokach dysku

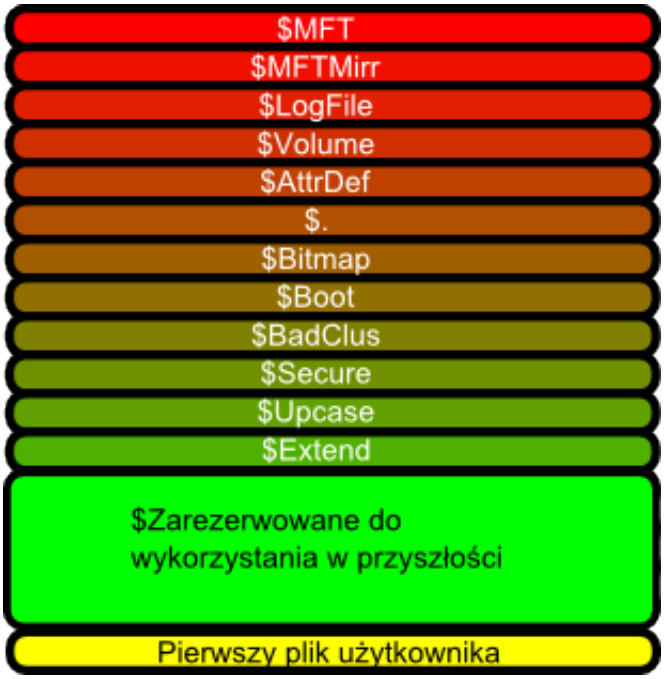
9: $Secure – baza danych przechowująca listę uprawnień (ACL – access control list), które mogą być nadawane plikom i katalogom

10: $UpCase – Tablica znaków Unicode (tzw. dużych liter) służąca do zapewnienia kompatybilności w programach opartych o MSDOS i Win32 (nierozróżnialność nazw plików pisanych dużymi i małymi literami)

11: $Extend – katalog zawierający dodatkowe dane jak informacje o przydziałach przestrzeni dyskowej dla poszczególnych użytkowników, punktach parsowania woluminu (niezbędnych do implementacji np. dowiązań symbolicznych)

12 – 15: zarezerwowane dla przyszłych rozwiązań

16: pierwszy rekord opisujący zwykłe pliki na dysku



Schemat rozmieszczenia metadanych w systemie NTFS

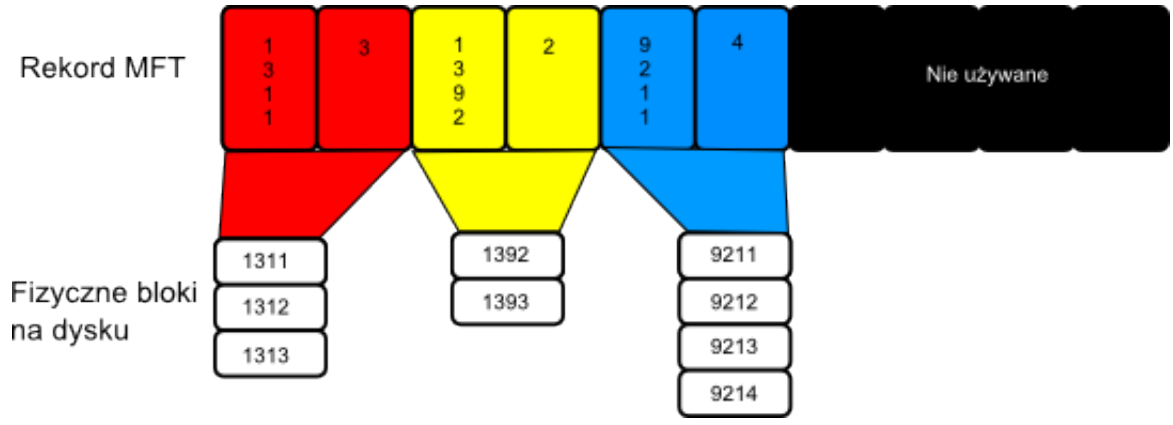
20 Brak pełnej kopii MFT powoduje że w przypadku uszkodzenia głównej struktury tablicy plików (np. fizycznego uszkodzenia dysku) nie jest możliwe jej naprawienie co może prowadzić do utraty części (lub wszystkich) danych przechowywanych na partycji, w zależności od miejsca wystąpienia błędu

21 Każda operacja, która ma zostać wykonana na strukturze plików jest najpierw odnotowywana w dzienniku, w przypadku awarii przy wykonywaniu tej operacji możliwe jest późniejsze usunięcie błędów w systemie plików

Jak wspomniano, każdy z plików umieszczonych na dysku określany jest przez jeden (w szczególnych przypadkach więcej) rekord w tablicy MFT. Struktura pojedynczego rekordu przedstawiona jest poniżej:

Struktura rekordu NFTS

Rekord rozpoczyna się nagłówkiem, w którym przechowywane są podstawowe informacje dotyczące pliku, np. czasy utworzenia, modyfikacji i dostępu, liczba powiązań itd. Parametry te przechowywane są na zasadzie sekwencji nazwa_atrybutu – wartość. Następnym polem jest nazwa pliku lub folderu, pełna (maksymalnie 255 znaków Unicode) oraz jej wersja kompatybilna z MSDOS (8 znaków nazwy + 3 znaki rozszerzenia, bez uwzględniania wielkości liter). Za polem określającym nazwę znajduje się opis atrybutów i uprawnień pliku lub katalogu (jako odnośniki do tablicy ACL znajdującej się w nagłówku pliku MFT). Poza tymi danymi rozpoczyna się przestrzeń przechowująca właściwą informację o danych. W tym miejscu należy wspomnieć, że informacja ta może być zapisana dwojako – w przypadku małych plików (kilkaset bajtów) zawartość pliku jest zapisywana bezpośrednio w rekordzie MFT co w ogromnym stopniu zwiększa szybkość dostępu do tego rodzaju plików. Pliki o większych rozmiarach są zapisywane na dysku twardym w blokach, a informacja o numerach tych bloków przechowywana jest w rekordzie (jako indeks na powyższym schemacie). Algorytmy rozmieszczania danych na dysku dążą do uzyskania ich jak największej spójności, tzn. rozmieszczenia ciągłego (kolejne partie pliku zapisywane w sąsiadujących blokach) co pozwala na jednoczesne przyspieszenie dostępu do danych (czytanie sekwencyjne jest dużo szybsze niż czytanie danych rozrzuconych po całym dysku) oraz minimalizacji liczby wpisów w rekordzie MFT, który musi przechowywać informację o lokalizacji wszystkich segmentów pliku. Informacja ta jest reprezentowana przez dwa wpisy – adres bloku, w którym rozpoczyna się dany segment i liczbę bloków, które zawierają kolejne części pliku:



Rekord MTF dla dużego pliku

Jeżeli rekord MFT dla danego pliku nie jest wypełniony do końca, uzupełniany jest do rozmiaru 1kB.

Jeżeli rekord przechowuje informację o katalogu, to w polach odpowiadających indeksom bloków dla plików przechowywana jest informacja (indeks MFT, długość nazwy pliku, nazwa) dotycząca plików zawartych w tym katalogu. W przypadku folderów zawierających niewielką liczbę plików

jest to prosta lista, w przypadku folderów o bardzo dużej liczbie wpisów struktura listy zamieniana jest na umożliwiającą szybkie wyszukiwanie informacji strukturę drzewa B+.

Linux (Second Extended File System - ext2)

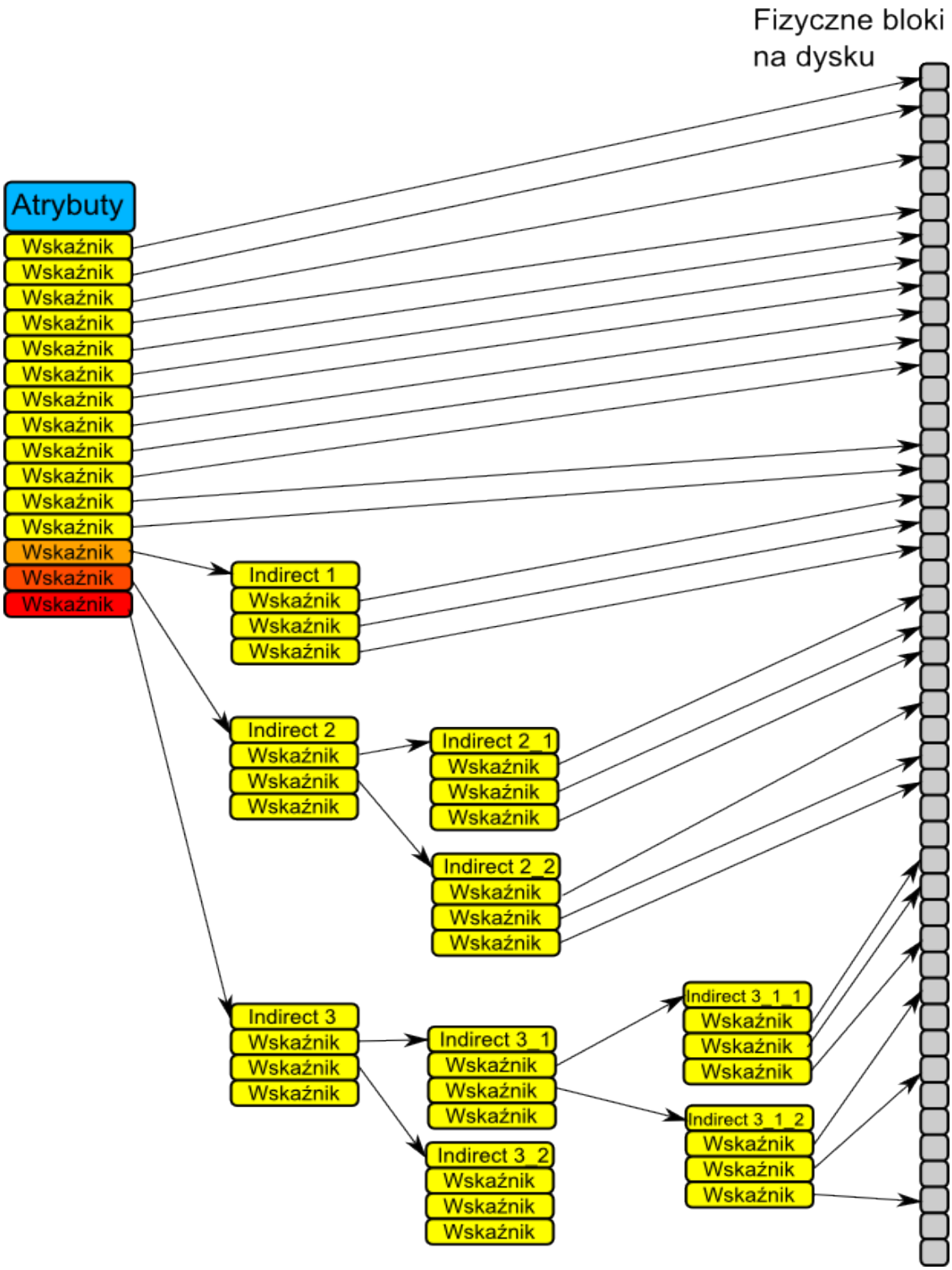
System plików używany przez różne dystrybucje Linuxa nazwany jest Rozszerzonym Systemem Plików (***Extended File System – ext***). W zależności od wersji, może to być ext, ext2, ext3 lub ext4. System ten jest rozszerzeniem systemu plików zaimplementowanego w Minixie, który z kolei swoje źródła ma w UFS czyli systemie plików Unixa, którego ogólna struktura przedstawiona została na rysunku***:***

Struktura systemu plików Unix

Na początku woluminu znajduje się *boot block*, który zawiera program ładujący system operacyjny. Znajdujący się za nim blok przechowuje informację o systemie plików, takie jak liczba bloków, adres początku listy wolnych bloków oraz liczba tzw ***i-node'ów***. Blok ten zwany jest superblokiem (*superblock*) a informacje w nim zawarte, jako że krytyczne dla działania systemu operacyjnego przechowywane są w kilku miejscach na dysku i mogą być odtworzone w przypadku awarii.

Za superblokiem znajduje się lista *i-node'ów*, które są strukturami przechowującymi informacje o plikach i katalogach. Struktura wewnętrzna i-node'a przedstawiona jest na rysunku (RYSUNEK). I-node'y są zbliżone ideowo do rekordów w tablicy MFT (zawierają określenie atrybutów, zezwoleń, ID użytkownika i grupy, rozmiaru, czasy tworzenia, modyfikacji itd. – pole *informacja* i listę bloków w których umieszczone są dane zapisane w pliku - *wskaźniki*). Każdy z i-node'ów jest identyfikowany w ramach systemu plików przez swój unikalny numer (nie zawiera jednak informacji o nazwie pliku, który opisuje). W unixowym systemie plików, nazwy plików są wiązane z poszczególnymi i-node'ami dzięki katalogom (katalog jest i-nodem, który przechowuje listę plików w nim zawartych, wiążąc nazwy z numerami i-node'ów).

W obecnie istniejących implementacjach systemów opartych na i-node'ach (np. ext2), wskaźników, które opisują położenie bloków danych jest piętnaście. Pierwsze 12 wskazuje bezpośrednio na adresy bloków (*bezpośrednich*). Kolejny wskaźnik odnosi się do bloku wskaźników adresujących kolejne bloki dysku (*niebezpośrednich*). Czternasty wskaźnik odnosi się do grupy wskaźników adresujących kolejne grupy wskaźników adresujących bloki dysku (tzw. *podwójnie niebezpośrednie*). Ostatni wskaźnik zwiększa poziom indeksowania o jeden do *potrójnie niebezpośrednich* bloków. Ta struktura umożliwia przechowywanie bardzo dużych plików przy jednoczesnym łatwym poruszaniu się po ich zawartości (alokowane bloki są stałej wartości, typowo 4kB, a więc łatwym do policzenia jest wskaźnik prowadzący do danych w dowolnej części pliku).



Struktura i-node'a w systemie ext2

Organizacja systemu plików bazując na i-node'ach w znacznym stopniu zmniejsza wymagania co do objętości pamięci operacyjnej komputera obsługującego dany wolumin. Z racji swojej specyfiki niepotrzebne jest przechowywanie całej tablicy i-node'ów (która standardowo zajmuje około 1% całkowitej przestrzeni woluminu) w czasie działania systemu, wystarczające jest przeczytanie zawartości i-node'a opisującego plik, na którym ma zostać wykonana jakaś operacja. W związku z tym zajętość pamięci operacyjnej określa iloczyn rozmiaru poszczególnego i-node'a (64 bajty) i liczby otwartych plików oraz katalogów.

Drugą korzyścią z identyfikacji plików poprzez i-node'y jest możliwość stosowania tzw. twardych odnośników (***hard links***), czyli konstrukcji odnoszących się do tego samego i-node'a. Efektem *hard link'ów* jest np. obecność jednego pliku (tego samego) w dwóch lokalizacjach (folderach) na raz. Ponieważ konkretne dane zawarte w pliku są silniej skojarzone z i-nodem niż z nazwą pliku możliwe jest usunięcie używanego pliku (np. biblioteki) i podmienienie jest nową wersją. Dzięki temu większość modyfikacji i uaktualnień bibliotek systemowych nie wymaga restartu systemu operacyjnego – odnośniki do starych bibliotek są usuwane (nawet jeżeli jakieś programy z nich obecnie korzystają) i na ich miejsce tworzone są nowe. I-node, który jest skojarzony ze starą wersją biblioteki zostanie automatycznie usunięty kiedy wszystkie używające go programy zostaną zamknięte. Nowo uruchamiane programy (od momentu wgrania nowej wersji biblioteki) będą korzystały z uaktualnionych danych.

W systemie ext2 pomiędzy superblokiem a tablicą i-node'ów umieszczane są dodatkowe dane. Pierwszą strukturą jest tzw. ***group descriptor*****,** który przechowuje dane o położeniu map bloków i i-node'ów oraz liczbę wolnych bloków, i-node'ów oraz katalogów należących do danej grupy. Dalej znajdują się bitmapy, które przechowują informację o wolnych blokach oraz i-node'ach. W dalszej cześci przechowywane są już same i-node'y a dalej – dane. W tej implementacji rozmiar i-node'a to 128 bajtów. Bloki danych adresowane są 4 bajtami (32 bity).

System plików ext2 został zastąpiony przez ext3, w którym głównym dodatkiem było wprowadzenie dziennika wykonywanych operacji (patrz dalej). W systemach z jądrem w wersji 2.6.28 dystrybuowany jest jego następca, ext4, który oferuje lepszą wydajność niż ext3 i możliwość tworzenia woluminów o rozmiarze 1 EiB.

Porównanie systemów plików Microsoft Windows NTFS i Linux ext4

W poniższej tabeli znajduje się krótkie porównanie charakterystycznych cech i ograniczeń systemów plików używanych przez najpopularniejsze obecnie systemy operacyjne.

Parametry	Microsoft Windows NTFS	Linux ext4
Maksymalny rozmiar pliku	2^{64} bajtów (16 EiB) minus 1 KiB	16 TiB (dla 4kB bloków)
Maksymalna liczba plików	4,294,967,295	4,294,967,295
Maksymalna długość nazwy pliku	255 UTF-16 code units	256
Maksymalny rozmiar woluminu	$2^{64} - 1$ clusters	1 EiB
Dozwolone znaki w nazwach plików	W trybie zgodnym z POSIX wszystkie kody UTF-16 z wyjątkiem NULL i '/' W trybie zgodnym z Win32 API wszystkie kody UTF-16 z wyjątkiem NULL, '/', '\', ':', '*', '?', '"', '<', '>' i '|'	Wszystkie bajty z wyjątkiem NULL ('\0') oraz '/'

Informacja o czasie operacji	Utworzenia, modyfikacji, ostatniego dostępu, zmiany POSIX	Utworzenia, modyfikacji, modyfikacji atrybutów, ostatniego dostępu, usunięcia
Możliwe atrybuty czasu (zakres)	1.01.1601 – 28.5.60056	14.12.1901 – 25.04.2514
Rozdzielczość czasu w atrybutach	100ns	1ns
Dodatkowe metadane związane z plikiem	Tak, w postaci alternatywnych strumieni danych	Nie
Atrybuty	Tylko do odczytu, ukryty, systemowy, do archwizacji, tymczasowy, skompresowany, nie do indeksowania zawartości	extents, noextents, mballoc, nomballoc, delalloc, nodelalloc, data=journal, data=ordered, data=writeback, commit=nrsec, orlov, oldalloc, user_xattr, nouser_xattr, acl, noacl, bsddf, minixdf, bh, nobh, journal_dev
Uprawnienia	ACL	POSIX
Kompresja	Tak, LZ77	Nie
Kodowanie	Tak, DESX (od Windows 2000), 3DES (od Windows XP), AES (od Windows XP Service Pack 1 i Windows Server 2003)	Nie
Systemy operacyjne	Windows NT 3.1 - Windows NT 4.0, Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, Windows Server 2008 R2	Linux

Źródło - Wikipedia

Zabezpieczenie przed utratą informacji

W ramach opisu systemu plików Microsoft Windows, poruszona została kwestia zapisu (logowania) informacji odnoszącej się do operacji na systemie plików w tzw. dzienniku (***journal***), przed jej faktycznym wykonaniem. Pozwala to na zabezpieczenie systemu przed stratą danych w wyniku błędu (np. przerwy w zasilaniu) występującego w trakcie operacji, zwłaszcza jeżeli operacja ta jest wieloetapowa (np. usunięcie pliku wiąże się z usunięciem rekordu go opisującego oraz informacji o nim z rekordu opisującego katalog, w którym się znajduje). Jeżeli zanik napięcia wystąpi pomiędzy operacjami, to powstanie niespójność w systemie plików mogąca prowadzić do utraty danych. W systemach plików nie przechowujących informacji o wykonywanych informacjach naprawa systemu plików łączy się z analizą całej struktury go opisującej, w przypadku systemów z dziennikiem naprawa wiąże się ze sprawdzeniem czy operacje tam

umieszczone zostały w rzeczywistości wykonane co jest znacznie szybsze.

Wydajność systemów plików wykorzystujących logowanie operacji jest gorsza od tych, które tej funkcji nie posiadają ponieważ operacje wymagają zwykle dwukrotnie większej liczby zapisów na dysk (do dziennika i do tablicy plików).

Istnieją systemy plików, które ściśle kontrolują kolejność wykonywania operacji (w normalnych okolicznościach kolejność ich jest w zasadzie dowolna, wynikająca z optymalizacji i obecnych warunków w systemie) tak, aby nigdy nie wystąpiło zjawisko nadpisania informacji po awarii (obecnie zaimplementowane w systemie FreeBSD).

Uwaga – przedstawiony tutaj sposób zabezpieczania informacji na dyskach dotyczy tylko struktury systemu plików, a nie fizycznych uszkodzeń nośników!

Usługi sieciowe

Nowoczesne systemy operacyjne (także te kontrolujące pracę prostych urządzeń) w większości posiadają zdolność komunikacji z innymi urządzeniami dzięki sieciom komputerowym. W wielu systemach obsługa podstawowych funkcji sieciowych jest wbudowana w jądro systemu operacyjnego (np. obsługa popularnych protokołów). Dzięki standaryzacji tych protokołów, komputery mogą wymieniać dane bez względu na rodzaj systemu operacyjnego, który je obsługuje. Istnieją przeróżne usług sieciowych, które mogą być udostępniane przez sieć, począwszy od udostępniania plików, poprzez zdalną administrację (tekstową lub graficzną) do urządzeń (drukarki, skanery, karty dźwiękowe) czy samej mocy obliczeniowej. Obsługa sieci komputerowych jest najczęściej bazowana na modelu OSI, którego schemat został zamieszczony poniżej:

Model OSI

Model OSI obejmuje siedem warstw komunikacji sieciowej. Pierwsza warstwa (Fizyczna - ***Physical Link***), obejmuje fizyczną transmisję danych w postaci sygnałów elektrycznych między urządzeniami. Warstwa druga (Łącza danych - ***Data Link***) zapewnia mechanizmy wymianych między urządzaniami (rozpoznawanie ramek danych, kontrolę przepływu danych, detekcję i korekcję błędów na poziomie pojedynczych ramek). Kolejna warstwa (sieciowa – ***Network***), odpowiada za nawiązywanie, podtrzymanie i zakończenie połączeń, fragmentację i routing danych. Najpopularniejszym protokołem należącym do trzeciej warstwy OSI jest IP (***Internet Protocol***), przedstawiony dokładniej w dalszej części bieżącego rozdziału. Warstwa czwarta, transportowa (***Transport***) odpowiada za pośredniczenie pomiędzy dalszymi warstwami a poziomem sieciowym zapewniając mechanizmy nawiązania określonych połączeń wraz z kontrolą ich stanu, błędów i

stanu transmisji. Najbardziej znanymi protokołami tej warstwy są TCP oraz UDP. Kolejnym poziomem modelu jest warstwa sesji (***Session Layer***), kontrolująca logiczne połączenia między komunikującymi się komputerami. Zapewnia między innymi mechanizmy nawiązywania i kończenia połączenia. Warstwa szósta, prezentacji (***Presentation Layer***), służy do zapewnienia niewrażliwości różnic w reprezentacji danych używanych na wyższym poziomie (w warstwie siódmej) a reprezentacją określoną na niższych poziomach poprzez odpowiednie formatowanie i kodowanie danych. Umożliwia to komunikację sieciową między różnymi systemami, chociażby używającymi różnych notacji danych (np. architektury x86 używają tzw. formatu ***little-endian***[22], podczas gdy urządzenia sieciowe oraz inne procesory jak np. PowerPC, Motorola 68000 – ***big-endian***). Ostatnia z warstw, aplikacyjna (***Application Layer***) obejmuje mechanizmy, z ktorymi użytownik ma bezpośredni kontakt. Przykładowymi protokołami, które należą do tej warstwy są http (***HyperText Transfer Protocol***) i ftp (***File Transfer Protocol***).

Protokół internetowy (***Internet Protocol – IP***) należy do trzeciej warstwy modelu OSI. Jego główną rolą jest zapewnienie transportu pakietów od urządzenia nadawczego do odbiorczego, które są identyfikowalne tylko poprzez adres sieciowy. Przesyłane dane podlegają enkapsulacji czyli obudowaniu dodatkowymi elementami charakterystycznymi dla warstwy sieciowej. Protokół ten nie zapewnia samoistnej kontroli przesyłania danych, to znaczy właściwej kolejności docierania pakietów do odbiorcy (np. drugi pakiet może przybyć przed pierwszym) ani ich liczby (możliwe jest gubienie pakietów, ale także ich dublowanie) oraz zawartości (dane wewnątrz pakietu mogą zostać uszkodzone). Protokół ten dodaje do danych nagłówek, w którym znajduje się szereg danych, takich jak numer wersji protokołu IP, rozmiar pakietu, wartość TTL (*time to live* – wartość określająca po przejściu ilu urządzeń aktywnych, takich jak karty sieciowe, routery, dany pakiet zostanie zniszczony), określenie typu protokołu z warstwy transportowej (np. TCP), suma kontrolna pakietu oraz oczywiście adres IP nadawcy i odbiorcy. Protokół IP w wersji 4 jest szeroko stosowany w sieciach lokalnych i Internecie, przy czym w związku z jego ograniczeniami podejmowane są próby zastąpienia go kolejną wersją, mianowicie IPv6. Konfiguracja parametrów protokołu IP w systemach Microsoft Windows i Linux została przedstawiona w rozdziale____.

W powiązaniu z protokołem IP często występuje protokół warstwy czwartej, transportowej, UDP (***User Datagram Protocol*** – datagramowy[23] protokół użytkownika). Jest to stosunkowo prosty protokół zapewniający podstawową komunikację między urządzeniami, o bardzo dobrej wydajności (małym narzucie obliczeniowym), która wiąże się z brakiem kontroli przesyłania danych. UDP nie zapewnia zarówno poprawności przesłanych danych, jak i kolejności ich nadejścia oraz tego, że w ogóle dotrą do odbiorcy. Z tego względu wykorzystywany jest tylko w tych zastosowaniach, w których utrata lub przekłamanie porcji danych nie jest błędem krytycznym – na przykład w przesyłaniu dźwięku i obrazu, grach sieciowych itd. Co ważne, przesłanie danych z wykorzystaniem tego protokołu nie wymaga wcześniejszego zestawienia połączenia między nadawcą i odbiorcą. Dużą zaletą UDP jest także możliwość jednoczesnego wysłania datagramów do wielu urządzeń odbiorczych (broadcast, multicast). Istotną cechą tego protokołu, podobnie jak TCP jest możliwość identyfikacji aplikacji odbiorczych i nadawczych poprzez tzw. porty (datagram jest wysyłany do odbiorcy na kokretny port, a więc każde urządzenie może mieć tyle różnych aplikacji odbiorczych oczekujących na różne komunikaty, ile jest dostępnych portów). W UDP i TCP porty numerowane są od 0 do 65535.

Drugim z popularnych protokołów warstwy transportowej jest tzw. Protokół Kontroli Transmisji (***Transmission Control Protocol – TCP***). W przeciwności do UDP, przesłanie danych przy użyciu TCP wymaga zestawienia połączenia między nadawcą i odbiorcą, co wykonywane jest w charakterystycznym trybie podwójnego potwierdzenia. Nadawca wysyła pakiet z żądaniem

22 Endian – kolejność bajtów w określonej strukturze. *Little-endian* oznacza przechowywanie bajtów w kolejności odpowiadającej rosnącej ważności, tzn. najmniej istotny bajt (LSB) przechowywany jest na początku. *Big-endian* to struktura, w której dane są przechowywane w kolejności od najważniejszego do mniej ważnego bajtu.

23 Datagram – inne określenie pakietu danych

połączenia do odbiorcy (charakteryzowany przez flagę TCP SYN). Jeżeli ten decyduje się go obsłużyć, to odsyła nadawcy odpowiedź z flagami SYN+ACK (w przeciwnym razie odsyła pakiet z flagą RST). Nadawca potwierdza nawiązanie połączenia pakietem z flagą ACK.

Protokół TCP, zgodnie ze swoją nazwą zapewnia pełną kontrolę transmisji danych, to znaczy zarówno dostarczenie wszystkich pakietów bez zmian ich zawartości, kolejności oraz liczby. W wypadku wystąpienia błędu transmisji (np. zagubienia pakietu), nadawca wysyła go ponownie, a odbiorca kompletuje i szereguje odebrane dane we właściwej kolejności. Oczywiście w celu zapewnienia wolnej od błędów transmisji konieczne jest zwiększenie liczby danych przesyłanych w nagłówku każdego pakietu. Ponadto transmisja danych wymaga wykonania pewnych obliczeń przez procesor (lub układy kart sieciowych). Weryfikacja danych przez odbiorcę jest wykonywana m.in. na podstawie sumy kontrolnej pakietu zawartej w jego nagłówku i w przypadku poprawnego odbioru potwierdzona poprzez wysłanie pakietu z numerem identyfikacyjnym otrzymanego pakietu i flagą ACK. W przypadku braku potwierdzenia otrzymania pakietu, dane są automatycznie wysyłane ponownie. W normalnych warunkach połączenie jest zamykane w sposób podobny do jego nawiązania, przy czym flagę SYN zastępuje flaga FIN. Protokół ten jest najczęściej spotykanym protokołem transmisji danych i wykorzystują go w zasadzie wszystkie główne protokoły warstwy aplikacyjnej jak http, ftp, ssh itd.

Obecnie istniejące systemy operacyjne w przeważającej większości posiadają zdolność komunikacji z innymi urządzeniami poprzez sieci różnego rodzaju. Jednakże praca sieciowa nie była dotąd głównym założeniem twórców systemów, jak dotąd są one mimo wszystko projektowane jako zarządzające lokalnymi zasobami, elementy sieciowe są raczej dodatkiem, a nie podstawą działania. Istnieje duże prawdopodobieństwo, że ten model systemów operacyjnych zostanie jednak zastąpiony przez nowy, bazujący na powszechnej łączności (na przykład przechowywanie danych będzie zorganizowane zupełnie inaczej, bez podziału na dane lokalne i zdalne). Pierwszym tego typu systemem, który został zademonstrowany, ale nie jest jeszcze ogólnie dostępny (planowany termin wydania – lipiec 2010), jest system operacyjny Chrome OS firmy Google, przeznaczony dla netbooków[24], oparty na jądrze Linuxa. System ten, którego centrum będzie przeglądarka internetowa Chrome będzie przeznaczony do pracy sieciowej, aplikacje natywne (instalowane lokalnie) zostaną zastąpione przez ich wersje sieciowe (np. Google Documents, GMail) pracujące w oknach przeglądarki Chrome.

24 Netbook - mały i lekki komputer przenośny, będący uzupełnieniem luki pomiędzy palmtopami i laptopami. Najczęściej wyposażony w mały ekran, stosunkowo wolny lecz energooszczędny procesor i dysk SSD. Przeznaczony głównie do programów biurowych, poczty elektronicznej, komunikacji, przeglądania internetu przez osoby często podróżujące.

Kontrola dostępu, ochrona danych oraz inne usługi

Systemy operacyjne obsługują sprzęt komputerowy pracujący w najróżniejszych warunkach, wykorzystywany w rozmaitych celach. Częstokroć natura wykonywanych operacji, a w szczególności przechowywane dane są prywatne, tajne i wymagają możliwie zaawansowanej ochrony. Oczywiście, podobnie jak różnorodne są zastosowania komputerów, tak różne są poziomy ochrony danych jak i autoryzacji dostępu, wiadomym jest że komputer osobisty nie wymaga takich zabezpieczeń jak serwer bankowy zarządzający kontami czy też komputer sterujący pracą elektrowni. Przez ochronę danych można rozumieć dwa procesy – ochronę przed dostępem niepowołanych osób / programów, ale także ochronę fizyczną, przed np. uszkodzeniem nośnika na którym są zapisane. Niektóre, najczęściej stosowane metody autoryzacji dostępu, a także ochrony i udostępniania danych (szczególnie te obecne w popularnych systemach operacyjnych) zostaną zaprezentowane w tym rozdziale.

Sposoby ochrony danych i ich udostępnianie

Jak wspomniano we wstępie do bieżącego rozdziału, ochronę danych można rozumieć dwojako, jako ochronę przed losowymi wydarzeniami skutkującymi ich utratą (np. pożarowi, zalaniu, awariom) jak i ochronę przed kradzieżą, nieautoryzowanym skopiowaniem czy przeczytaniem.

Ochrona przed czynnikami losowymi

Czynnikami losowymi mogą być zdarzenia fizyczne (pożar, zalanie, uszkodzenie nośnika, wyładowanie elektryczne, przerwa w zasilaniu), jak również błędy oprogramowania (skutkujące np. nadpisaniem istotnych informacji) oraz błędy użytkowników (np. odłączenie nośnika przed zakończeniem zapisu danych, skasowanie ważnych informacji) oraz awarie urządzeń (np. zasilacza komputera skutkujące uszkodzeniem dysków twardych).

Skutecznym sposobem zabezpieczenia przed awarią dysków twardych jest przechowywanie danych na macierzach RAID, gdzie w zależności od konfiguracji informacja jest przechowywana na dwóch lub większej liczbie nośników i możliwe jest jej odzyskanie w wypadku awarii jednego (lub w szczególnych wypadkach więcej) z nich. Najskuteczniejszym jednak i najczęściej stosowanym sposobem zabezpieczenia danych przed czynnikami losowymi jest częste wykonywanie kopii zapasowych danych na nośnikach gwarantujących długie bezstratne przechowywanie danych (np. taśmach) i przechowywanie tych nośników w bezpiecznym miejscu.

Ochrona przed nieautoryzowanym dostępem

Kontrola dostępu do systemu komputerowego

Pojęcie kontroli dostępu do systemu (niekoniecznie komputerowego) obejmuje trzy aspekty – uwierzytelnienie (***authentication***), autoryzację (***authorization***) i rewizję (***audit***). Ogólnie kontrolę dostępu można zdefiniować jako funkcjonalność określającą kto (lub co) ma dostęp (i jakiej natury) do pewnych danych lub funkcji systemu.

Uwierzytelnienie to sprawdzenie tożsamości osoby lub systemu oczekującego na dostęp. Zgodnie z dokumentem [RFC2828] proces uwierzytelnienia składa się z dwóch etapów –

identyfikacji (podania systemowi określającego obiekt identyfikatora, np. nazwy użytkownika) oraz weryfikacji (podania informacji, która potwierdza tożsamość obiektu, np. hasła, PINu, smart-card, kodu jednorazowego, odcisku palca, skanu siatkówki, próbki głosu itd.). Po poprawnym przejściu procesu uwierzytelnienia następuje etap autoryzacji.

Autoryzacja to określenie praw dostępu do poszczególnych elementów systemu. Po określeniu tych praw, system porównuje je każdorazowo przy próbie wykonania dowolnej czynności lub dostępie do danych przez użytkownika. Jeżeli dostęp do danych / czynności jest dozwolony przez zdefiniowany zestaw uprawnień użytkownik uzyskuje ów dostęp.

Audyt oznacza zapis poszczególnych zdarzeń wykonywanych przez system i użytkowników, przeprowadzany w sposób umożliwiający rekonstrukcję i analizę tych zdarzeń co pozwala na wykrycie prób (zakończonych sukcesem lub nie) nieautoryzowanego dostępu do danych lub części systemu.

W przypadku najczęściej używanej metody uwierzytelnienia, czyli wymagania podania pary identyfikatorów (nazwy użytkownika i hasła) często występuje problem dobierania przez użytkowników haseł o małym stopniu skomplikowania, łatwych do odgadnięcia lub złamania. W przypadku komputerów prywatnych, nazwą użytkownika jest najczęściej imię właściciela (często proponowane przez sam system operacyjny przy instalacji), w przypadku np. serwerów poczty nazwa ta jest generowana na podstawie klucza, np. pierwsza litera imienia i nazwisko, czasem same inicjały itd., w związku z czym odgadnięcie przez niepowołaną osobę pierwszej części danych uwierzytelniających jest zwykle bardzo łatwe. Hasła natomiast, co wynika z chęci ułatwienia sobie ich zapamiętania, nadawane są przez użytkowników również wg klucza, najczęściej są to kombinacje ich imion i nazwisk, inicjałów, imion żony czy męża, dzieci, samochodu, dość popularne są również wyrazy obraźliwe. Niektóre systemy wymagają aby hasło miało np. nie mniej niż 10 znaków, aby znajdowały się w nim cyfry, duże i małe litery. Niestety, najczęściej tego typu hasła są przez użytkowników zapisywane na kartkach znajdujących się w pobliżu komputera, co przeczy wszelkim regułom zabezpieczenia dostępu. Często więc nawet najbardziej zaawansowane systemy zabezpieczeń są pokonywane przez niedbałość i bezmyślność ich użytkowników, a nie przez błędy oprogramowania.

Kontrola dostępu do systemu może być określona na trzy sposoby – obowiązkowy (***mandatory access control - MAC***), swobodny (***discretionary access control - DAC***) i oparty na rolach (***role based access control - RBAC***). Nazwy pierwszych dwóch są trochę mylące, ponieważ nie oznacza „swobodnego" poddawania się kontroli, a jedynie możliwość nadania innemu użytkownikowi prawa do dostępu do zasobu, do którego nadający ma prawo. W przypadku kontroli obowiązkowej takiej możliwości nie ma.

Klasyfikacja typów kontroli dostępu została scharakteryzowana np. w [STALLING]:

MAC – obiekty chronione mają przypisany pewien poziom ważności, a użytkownicy posiadają „przepustki", z którymi skojarzone jest zezwolenie na dostęp do pewnego poziomu.

DAC – dla użytkownika określone są reguły, co dana osoba może, a czego nie może zrobić.

RBAC – użytkownicy systemu należą do określonych grup lub przyjmują określone role (np. administratora, obserwatora itd.), do których to ról przypisane są prawa dostępu do poszczególnych części systemu i danych.

Częścią systemu kontroli dostępu są tak zwane listy kontroli dostępu (***access control lists – ACL***). Lista taka jest obiektem skojarzonym z zasobem, który zawiera listę użytkowników, grup i procesów, które posiadają dostęp do tego zasobu. Dostęp ten jest specyfikowany poprzez określenie operacji, które dany obiekt może wykonać. Model ten jest stosowany w praktycznie wszystkich systemach operacyjnych umożliwiających kontrolę dostępu, przy czym najczęściej spotykany jest model ACL, który określony został w standardzie POSIX.1e. Najpopularniejsze systemy plików, takie jak ext i NTFS wykorzystują model list kontrolnych do określenia praw dostępu

użytkowników i grup do danych plików.

Systemy detekcji włamań (***Intrusion detection system – IDS***) są systemami, które pozwalają na wykrycie naruszenia praw dostępu przez nieautoryzowaną jednostkę, czy to na skutek błędów oprogramowania czy innych zdarzeń. Jeżeli akcja taka zostanie wykryta odpowiednio szybko (najlepiej w trakcie jej trwania), to istnieją duże szanse na wykrycie tożsamości intruza oraz na wystarczająco szybkie pozbycie się go z systemu (lub np. odłączenie systemu od sieci przez którą nastąpiło włamanie) aby włamujący nie zdążył przechwycić najbardziej istotnych informacji. Charakterystyka i sposoby działania systemów IDS wykraczają poza zasięg tego podręcznika, niemniej zainteresowanemu nimi czytelnikowi proponowane jest zapoznanie się np. z dokumentacją otwartego oprogramowania Snort [link].

Dodatkowe środki ochrony systemu komputerowego.

Zabezpieczenie systemu komputerowego przed nieautoryzowanym dostępem wiąże się również, oprócz zapewnienia odpowiedniego systemu uwierzytelniania i autoryzacji z fizycznym i programowym ograniczeniem dostępności chronionych usług czy danych do niezbędnego minimum gwarantującego zapewnienie wymaganej dostępności systemu dla autoryzowanych użytkowników. Oczywistym jest, że przykładowo dane wewnętrzne przedsiębiorstwa powinny być dostępne dla niektórych pracowników (np. księgowości), a niedostępne dla np. kierowców, dozorców itd. Ponadto dane te nie powinny być dostępne poza przedsiębiorstwem, tzn. nie powinno być do nich publicznego dostępu. Komputer przechowujący owe dane powinien może więc być zabezpieczony w opisany w poprzednim podrozdziale sposób (np. poprzez wymóg posiadania karty uprawniającej do dostępu do danych). Jeżeli jednak komputer ten jest przyłączony do sieci, w szegółności do sieci publicznej (Internetu), to ograniczenie to może okazać się niewystarczające, chociażby z tego względu, że oprogramowanie na nim uruchomione (system operacyjny, serwer udostępniający dane itp.) może zawierać pewne luki w systemach autoryzacyjnych, które np. pozwolą na uzyskanie dostępu przez nieautoryzowane osoby. W sieci publicznej istnieje bardzo duże ryzyko ataku na taki komputer, dlatego wskazane jest aby wzajemnie się uzupełniających systemów bezpieczeństwa było kilka. Przykładem prostego ideowo (choć w rzeczywistości podstawowego) środka ograniczenia dostępu do systemu jest tzw. ***firewall*** czyli zapora blokująca dostęp do określonych usług sieciowych. W najprostszym rozwiązaniu zapora ta może być skonfigurowana na całkowitą blokadę lub całkowite dopuszczenie ruchu sieciowego, jednak wszystkie istniejące implementacje umożliwiają dużo bardziej wyrafinowane metody filtracji, począwszy od podziału na adresy sieciowe (IP) komputerów, które mogą być dopuszczone, filtrację adresów fizycznych kart sieciowych, zliczanie liczby prób nawiązania połączenia, analizę lokalizacji w której znajduje się komputer próbujący uzyskać dostęp itd. Najpopularniejsze obecnie systemy operacyjne (ogólnego użycia oraz serwerowe) posiadają wbudowane zapory sieciowe (w systemach Linux jest to pakiet ***iptables***, w Microsoft Windows – ***Windows Firewall***).

Włamania do systemów komputerowych mają różne cele i są wykonywane przez różne klasy osób. Zdarzają się oczywiście profesjonalne włamania do komputerów korporacyjnych czy agencji rządowych wykonywane w celu zdobycia określonych informacji (szpiegostwo przemysłowe) czy też wprowadzenie zamętu w struktury określonej instytucji, są to jednak zdarzenia stosunkowo rzadkie. Zdecydowanie częściej występują włamania do komputerów domowych, pozornie nie zawierających żadnych interesujących informacji. Włamania te najczęściej jednak nie są związane z chęcią pozyskania danych składowanych na komputerze, zdecydowanie częściej mają na celu zaszkodzenie użytkownikowi (np. usunięcie plików, wywołanie awarii systemu operacyjnego) lub instalację oprogramowania kontrolującego pracę docelowego komputera. Oprogramowanie to najczęściej stosowane jest do przechwytywania danych określonego rodzaju (zwykle haseł oraz numerów kart kredytowych) oraz, coraz częściej do zapewnienia możliwości wykorzystania komputera ofiary do określonych celów, takich jak np.

próba wywołania blokady określonych serwerów sieciowych (tzw. atak ***DDOS – Distributed Denial of Service***). Blokada taka powodowana jest „zalaniem" serwera żądaniami od wielu komputerów jednocześnie (przy odpowiednio dużej liczbie żądań usługa oferowana przez serwer staje się praktycznie niedostępna). Przykładem DDOS był np. atak na rządowe i bankowe serwery Estonii w 2007 roku paraliżujący działanie wielu instytucji. Należy tutaj podkreślić, że włamania mające na celu instalację tego typu oprogramowania nie są w zasadzie wykonywane indywidualnie, raczej używane są specjalne programy skanujące widoczne w sieci komputery i próbujące wykorzystać luki w oprogramowaniu na nich pracującym. Do zabezpieczenia systemu komputerowego przed tego typu programami (zwanymi wirusami[25] lub koniami trojańskimi[26]), oprócz zapór sieciowych (blokujących dostęp do niektórych usług, które nie muszą być otwarte[27]) stosowane są również tzw. programy antywirusowe, które pozwalają na wykrycie próby uruchomienia programu zawierającego wirusa lub konia trojańskiego i jego usunięcie. Antywirusy pozwalają również na sprawdzenie znajdujących się na dyskach danych pod kątem obecności w nich podejrzanych programów. Dla systemów Linux dostępnych jest kilka pakietów oprogramowania antywirusowego (np. darmowy ClamAV), w systemach Microsoft Windows, oprócz szeregu komercyjnych i darmowych rozwiązań firm trzecich od niedawna dostępny jest również darmowy (dla użytkowników prywatnych) pakiet Microsoft Security Essentials. Warto w tym miejscu podkreślić, że często obecne w świadomości użytkowników przekonanie o tym, że wirusy i konie trojańskie to problem użytkowników produktów Microsoft, jest fałszywe. Mniejsza liczba wrogiego oprogramowania przeznaczonego na systemy takie jak Linux czy Mac OS X wynika z jednej strony z małej w porównaniu z MS Windows popularności tych systemów, z drugiej – z tego, że przeciętny użytkownik Windows pracuje na koncie z uprawnieniami administratora (co w zasadzie nie jest spotykane w Linuxie), co automatycznie pozwala na większą penetrację systemu przez uruchamiane przez tego użytkownika wirusy.

Osobną kwestią ochrony danych jest sposób ich zabezpieczenia tak, aby nawet w wypadku gdyby dostały się w niepowołane ręce, nie dały się wykorzystać. W przypadku przechowywania danych osiągane jest to najczęściej dzięki szyfrowaniu tych danych. Istnieje wiele metod szyfrowania, zarówno sprzętowych jak i programowych. Główną zaletą rozwiązań sprzętowych, bazujących najczęściej na zasadzie szyfrowania symetrycznego (patrz dalej), jest ich niezależność od pozostałych komponentów komputera (klucze szyfrujące nie są przechowywane w pamięci operacyjnej) co podnosi w znacznym stopniu ich odporność na złamanie. Nie bez znaczenia jest również to, że szyfrowanie nie obciąża w żadnym stopniu procesora ponieważ wykonywane jest wewnętrznie przez układy scalone dysku twardego. W ten sposób szyfrowana może być zawartość dysków twardych jak i pamięci flash, na przykład typu ***pendrive*** czy ***SSD***.

Rozwiązania programowe pozwalają na kodowanie danych w momencie ich zapisu i dekodowanie przy odczycie, przy czym operacja ta odbywa się dzięki specjalnemu oprogramowaniu za pośrednictwem procesora. Powoduje to zmniejszenie wydajności transferu danych między pamięcią operacyjną i dyskiem przy jednoczesnym zwiększeniu użycia procesora (deszyfracja jest zwykle złożoną operacją). Przykładowym oprogramowaniem służącym do szyfrowania dysków jest darmowy pakiet Truecrypt, cryptoloop, czy komercyjny Symantec Endpoint Encryption.

---

25 Wirus – program komputerowy zdolny do rozmnażania (kopiowania się) i zarażania programów znajdujących się na atakowanym komputerze. Wirusy potrafią rozprzestrzeniać się między komputerami poprzez wymianę danych (na nośnikach przenośnych oraz z wykorzystaniem łączącej ich sieci). Często są one niegroźne (tzn. ich jedyną funkcją jest rozprzestrzenianie się), nierzadko jednak mogą uszkadzać zgromadzone dane lub destablizować pracę komputera.

26 Koń trojański – oprogramowanie, które w przeciwności do wirusa nie potrafi się rozmnażać i rozprzestrzeniać. Głównym celem programu jest umożliwienie dostępu do systemu komputerowego przez nieautoryzowanego użytkownika. Najczęściej konie trojańskie służą do kontrolowania komputerów w celu przeprowadzenia ataku DDOS, kradzieży numerów kart kredytowych, instalacji innych koni trojańskich i wirusów.

27 W 2003 roku systemy Microsoft Windows XP masowo zarażane były programem MSBLAST, który wykorzystywał lukę w systemie DCOM RPC dostępną na niezabezpieczonym przed publicznym dostępem portem TCP 135. Po zarażeniu powodował między innymi restart systemu operacyjnego [MSBLAST].

Przedstawione powyżej rozwiązania pozwalają na szyfrowanie wszystkich danych zapisywanych na nośnikach fizycznych. Istnieje jednak potrzeba szyfrowania np. pojedynczych plików (lub zbiorów plików), które użytkownicy mogliby sobie przekazywać (w postaci zaszyfrowanej), na przykład pocztą elektroniczną bez obawy o ich wykorzystanie przez niepowołane osoby nawet w przypadku przechwycenia danych. Przykładem takich aplikacji są między innymi PGP (szyfrowanie RSA i DSA) [PGP], Gnu PrivacyGuard (szyfrowanie RSA i ElGamal) [GPG]oraz archwizatory takie jak arj, zip, 7zip i rar.

Ogólną zasadą szyfrowania danych jest takie przekształcenie kodowanego tekstu (przy użyciu tzw. klucza szyfrującego[28]), na inny tekst, z którego treści nie da się odczytać treści orginalnej bez użycia klucza deszyfrującego. Zdecydowana większość rozwiązań szyfrujących bazuje na jednej z dwóch metod kryptograficznych różniących się mechanizmem tworzenia kluczy, które zostaną przedstawione poniżej.

Szyfrowanie kluczem symetrycznym (***symmetric-key cryptography***) to metoda wykorzystująca ten sam kod do zakodowania i odkodowania wiadomości. Jest ona stosunkowo łatwa do złamania (np. algorytmem siłowym[29] – ***brute force***), ale tylko jeżeli używany klucz jest krótki. Identyfikacja składników klucza o długości 1024 lub więcej znaków jest bardzo długotrwała. Niemniej złamanie klucza dowolnej długości jest 100% pewne, siłą szyfrowania jest czas jego identyfikacji. Dodatkowo, metoda ta wymaga przekazania sobie przez obie komunikujące się strony (szyfrującego i odbierającego) klucza co w wielu wypadkach jest niemalże równoznaczne ze zdradzeniem go osobom niepowołanym.

Szyfrowanie kluczem publicznym (***public-key cryptography***) polega na użyciu do zakodowania wiadomości przez nadawcę klucza publicznego odbiorcy. Klucz ten może być udostępniony publicznie, a odczytanie jakiejkolwiek wiadomości tylko za jego pomocą jest niemożliwe. Do deszyfracji wymagany jest klucz prywatny, generowany jednocześnie z publicznym, który należy tylko do odbiorcy. Istnieje wiele implementacji tego typu algorytmów szyfrujących, najbardziej znane z nich to RSA i ElGamal [CRYPTOGRAPHY]. Pierwszy z nich, którego nazwa pochodzi od pierwszych liter nazwisk twórców opiera swoje działanie na zagadnieniu faktoryzacji[30] liczb, a konkretnie na braku metody, która pozwala na szybkie obliczenie czynników faktoryzujących. Algorytm ElGamal (nazwa pochodzi od nazwiska twórcy) z kolei korzysta z podobnego założenia (długości obliczeń) wyznaczenia logartymu dyskretnego w ciele liczb całkowitych[31].

Udostępnianie danych

Obecne czasy są erą gwałtownego rozwoju technologii szybkich łącz internetowych. To, co jeszcze dziesięć lat temu było trudne do wyobrażenia, jak na przykład kilkumegabitowe łącze bezprzewodowe wykorzystujące infrastrukturę telefonii komórkowej, dziś jest w powszechnym użyciu. Wraz ze wzrostem przepustowości łącz maleje również ich koszt. To wszystko powoduje że w przeszłość odchodzą czasy oddzielnych komputerów, sporadycznie wymieniających dane poprzez fizyczne nośniki takie jak np. dyskietki. Obecne komputery, z wyjątkiem tych używanych

28 Klucz szyfrujący – informacja wykonywanie operacji kryptograficznej – szyfrowania lub deszyfrowania

29 Algorytm siłowy – algorytm, w którym metodą odnalezienia rozwiązania problemu jest sprawdzenie wszystkich możliwości jego rozwiązania i identyfikacji właściwej zamiast zastosowania metody wynikającej z analizy problemu

30 Faktoryzacja – rozkład liczby całkowitej C na czynniki takie, że ich iloczyn jest równy tej liczbie przy czym czynniki nie mogą mieć wartości 1 oraz C

31 Logarytm dyskretny – taka liczba całkowita l, dla której zachodzi równość $p^l=n$, gdzie p – podstawa algorytmu, n – liczba logarytmowana. Interesującą właściwością tej funkcji jest to, że nie zawsze istnieje, a jeżeli istnieje, to jest często niejednoznaczna (np. dwa wyniki są poprawne) [DISCLOG]

w szczególnych zastosowaniach są najczęściej podłączone do sieci (lokalnej, globalnej) i za jej pomocą wymieniają ze sobą dane. Wymiana ta odbywa się na różnych zasadach, w sieciach lokalnych najczęściej za pomocą protokołów takich jak NFS (Network File System), SMB (tzw. Otoczenie sieciowe Microsoft Windows), ftp. W sieci globalnej dane są z kolei najczęściej udostępniane poprzez protokoły http i ftp.

Najbardziej znanym protokołem sieciowym (pracującej w siódmej warstwie – aplikacyjnej modelu OSI) jest protokół HTTP (***HyperText Transfer Protocol***) będący podstawą sieci WWW (***World Wide Web***). Jego rolą jest transmisja do komputerów obsługujących serwisy WWW danych pochodzących z komputerów klienckich, pośród których znajdują się rozkazy udostępnienia przez dany serwer określonej zawartości serwisu oraz ewentualne argumenty funkcji wykonywanej przez serwer (np. dane wypełnione w formularzu na stronie WWW, hasła itd). Oprócz transmisji danych użytkownika, protokół ten służy również do przesyłania odpowiedzi serwerów, głównie dokumentów hipertekstowych (czyli takich, które zawierają odnośniki do innych dokumentów, które w łatwy sposób, np. poprzez wskazanie myszą, mogą być otwarte) oraz innej zawartości serwisów (np. obrazów stanowiących część treści otwieranej strony). Domyślnym portem TCP na którym działają usługi http jest port 80.

Lista poleceń klienta, które wchodzą w skład specyfikacji HTTP jest krótka, zawierająca jedynie siedem pozycji:

HEAD – pobranie informacji o wskazanym zasobie, najczęściej stosowane do sprawdzenia jego dostępności. Niezbędne w serwerach WWW.
GET – pobranie zasobu identyfikowanego przez argument (URL[32]). Niezbędne w serwerach WWW.
POST – przesłanie danych od klienta do serwera (dane dołączone są jako ciąg znaków podany za poleceniem)
PUT – przesłanie danych od klienta do serwera (dane dołączone są jako plik)
DELETE – usunięcie wskazanego zasobu (oczywiście tylko przez uprawnionego użytkownika)
OPTIONS – żądanie informacji o metodach, które serwer udostępnia dla danego adresu URL
TRACE – włączenie odsyłania komunikatów otrzymanych przez serwer klientowi (do celów diagnostycznych)

Dodatkowo, w serwerach http często implementowana jest metoda CONNECT, służąca do zestawiania tzw. tunelu TCP z serwerami z kodowaniem transmisji SSL (HTTPS) poprzez zwykłe serwery HTTP.

Więcej informacji na temat protokołu HTTP można znaleźć między innymi na stronach organizacji standaryzacji WWW [W3C].

Kolejnym bardzo popularnym protokołem stosowanym do wymiany danych jest protokół FTP (***File Transfer Protocol*** – protokół transmisji plików). Protokół ten, jak wskazuje jego nazwa służy do wymiany danych pomiędzy komputerami. Podobnie jak HTTP implementuje on model komunikacji klient-serwer. Standardowym portem TCP, na którym serwery ftp oczekują komunikacji od klienta jest port 21. Po nawiązaniu połączenia na tym porcie tworzone jest drugie połączenie, służace do wymiany danych. Kanał komunikacyjny z portem 21 serwera wykorzystywany jest do przesyłania komunikatów sterujących jego pracą. W zależności od trybu pracy serwera, kanał wymiany danych może być tworzony dwojako – albo przez klienta (metoda ta nosi nazwę pasywnej), który łączy się z podanym mu przez serwer portem (zwykle o numerze ponad 1023), albo przez serwer (wtedy kanał danych zestawiany jest na porcie 20 serwera i

32 URL – *Uniform Resource Locator* – ujednolicony format identyfikacji i adresowania zasobów w sieciach komputerowych. Składa się z części określającej rodzaj zasobu oraz właściwego identyfikatora oddzielonego znakami://. Przykładem URL może być adres serwera Politechniki Warszawskiej http://pw.edu.pl

losowym klienta a metodę określa się mianem aktywnej). Konfiguracja serwera na tryb aktywny lub pasywny uzależniona jest od konfiguracji *firewalla* serwera oraz klienta a także konfiguracji sieci w jakich komputery te się znajdują.

Ważną opcją transmisji danych poprzez ftp jest możliwość rozpoczęcia transmisji od dowolnego miejsca pliku (używane jest to do wznawiania przerwanych transferów – nie wymaga transmisji całego pliku od początku).

Protokół ftp rozróżnia kilka trybów transferu danych, niemniej w rzeczywistości używane są tylko dwa ASCII i BYTE. Pierwszy z nich służy do transmisji plików tekstowych a jego zaletą jest to, że każda litera tekstu przesyłana jest jako numer kodu w tablicy ASCII. Pozwala to na takie zapisanie danych przez serwer, aby były zrozumiałe przez system operacyjny na nim zainstalowany. Jednocześnie po odebraniu tych danych z serwera przez inny system zostaną one automatycznie przetłumaczone na system znakowy właściwy dla kontrolującego pracę klienta systemu operacyjnego. Dobrym przykładem celowości transferu tekstu w trybie ASCII jest automatyczna zamiana znaków końca linii pomiędzy maszynami wyposażonymi w Microsoft Windows (CRLF[33]) oraz Linux / Unix (LF). Z kolei dane nie będące tekstowymi muszą być przesyłane w trybie BYTE (czyli bez żadnej interpretacji), w innym przypadku zostaną uszkodzone w czasie transferu.

Lista najpopularniejszych poleceń protokołu FTP zamieszczona została poniżej:

ABOR – przerwij transfer pliku

CWD – zmień katalog

DELE – usuń plik

LIST – wyświetl listę plików w katalogu

MKD – utwórz katalog

PASS – wyślij hasło

PASV – przejdź w tryb pasywny

PORT – otwórz kanał komunikacyjny do przesyłu danych

QUIT – zakończ połączenie

RETR – pobierz plik z serwera

RMD – usuń katalog

SIZE – podaj rozmiar pliku

STOR – wyślij plik do serwera

TYPE – ustaw typ transferu (ASCII, BYTE)

USER – wyślij nazwę użytkownika

Protokoły HTTP oraz FTP nie należą do protokołów bezpiecznych, wszystkie przesyłane za ich pomocą dane nie są szyfrowane i możliwe jest ich przechwycenie przy pomocy prostych narzędzi zapisujących ruch w sieci komputerowej (*snifferów*). Z tego powodu często do ochrony istotnych informacji używa się bezpiecznej wersji HTTP (HTTPS) i FTP (FTPS), w których transmisja kodowana jest przez protokół SSL[34] (pracujący w warstwie transportowej OSI). Połączenia używające zwykłych protokołów HTTP i FTP mogą być również tunelowane poprzez SSH lub VPN w celu zabezpieczenia ich przed podsłuchem.

W sieciach lokalnych (np. domowych, biurowych) popularnym rozwiązaniem służącym do

33 CR, LF – znaki specjalne określające koniec linii tekstu (LF) oraz tzw. powrót karetki (CR) czyli przejście do kolejnej linii

34 SSL - Secure Sockets Layer (Warstwa Bezpiecznych Gniazd Sieciowych) – rodzina protokołów umożliwiających szyfrowanie transmisji na poziomie warstwy transportowej OSI. Więcej informacji można znaleźć np. w [SSL]

wymiany plików (szczególnie pomiędzy komputerami pracującymi pod kontrolą OS Microsoftu) jest tzw. sieć Microsoft Windows (***Microsoft Windows Network***). Transmisja danych oraz identyfikacja i odnajdywanie komputerów w tym mechanizmie opiera się na protokole SMB (***Server Message Block***). Jest to protokół siódmej, aplikacyjnej warstwy modelu OSI, którego głównym zadaniem jest zapewnienie dostępu do plików, drukarek i innych zasobów (również komunikacji międzyprocesowej) w sieciach lokalnych. Pomimo że sam protokół implementuje model komunikacji klient-serwer, to w sieci Microsoft Windows Netwok bardziej przypomina on model równy-z-równym (***peer to peer***) z racji tego, że każdy z komputerów jest jednocześnie serwerem i klientem. W pierwszych implementacjach (powstałych w IBM) protokół zapewniał operacje takie jak tworzenie, otwieranie i zamykanie plików oraz drukarek, tworzenie, otwieranie, przeszukiwanie i usuwanie katalogów, zmianę atrybutów plików i katalogów. W miarę popularyzacji systemu lista jego możliwości znacznie się zwiększyła (do ponad 100). Wraz z systemem Vista, Microsoft wprowadził duże zmiany mające na celu głównie poprawienie wydajności protokołu. Co więcej, specyfikacja SMB 2.0 użytego w Viście została opublikowana dzięki czemu w łatwy sposób powstały jego implementacje przeznaczone dla innych platform (np. Linuxa). Przed pojawieniem się Visty istniały oczywiście rozwiązania umożliwiające korzystanie z Otoczenia sieciowego Windows w Linuxie (tzw. pakiet oprogramowania Samba), jednak ich powstanie było trudne ze względu na konieczność odkrywania szczegółów implementacyjnych SMB w wydaniu Microsoft poprzez tzw. inżynierię odwrotną.

Interfejs programisty – API

Większość systemów operacyjnych, ale także zwykłych aplikacji i bibliotek jest wyposażana w tzw. interfejs programisty (***Application programming interface – API***). Jest to zbiór specyfikacji dostępnych funkcji i ich interfejsów, a także specyficznych struktur, klas, konwencji wywołań i protokołów, które umożliwiają nowym aplikacjom dostęp do funkcjonalności zaimplementowanych w oprogramowaniu z API.

Rolą interfejsu programisty jest z jednej strony ułatwienie pracy programistom, ponieważ udostępnione funkcje w API zwykle są dogłębnie udokumentowane. Z drugiej strony API pozwala na ukrycie szczegółów implementacyjnych, których ujawnienie często jest niekorzystne (np. oprogramowanie komercyjne rzadko jest dostępne w postaci kodów źródłowych) oraz może prowadzić do utrudnienia pracy programistom (konieczność „przedzierania się” przez kody źródłowe nieznanych programów, często bardzo rozbudowanych). Dodatkową korzyścią jest przenośność kodu tworzonego oprogramowania – jeżeli np. API systemu operacyjnego, z którego korzysta to oprogramowanie spełnia określone standardy, to możliwa jest jego kompilacja na innym systemie, który posiada API spełniające ten sam standard. Przykładowo standard, który musi spełniać interfejs programisty definiuje POSIX. Podobnie wszystkie współcześnie używane systemy Microsoft posiadają wspólny interfejs API, który nie dość że zapewnia znajome środowisko programistom, to dodatkowo umożliwia uruchomienie aplikacji projektowanych dla starszych wersji systemu na systemach nowych przy użyciu trybu zgodności. W praktyce nie działa to zawsze bezbłędnie, często wynika to jednak z błędów twórców aplikacji, nie producenta systemu.

W przypadku systemów operacyjnych zgodnych ze standardem POSIX, interfejs programisty jest zaprojektowany zgodnie z regułami określonymi w tym standardzie. Zapewnia to dużą wygodę programistom, ponieważ dla wszystkich tych systemów model (i zbiór) funkcji API jest taki sam, a więc przenoszenie aplikacji pomiędzy tymi systemami jest, o ile nie bezproblemowe, to przynajmniej dużo łatwiejsze. Do grona systemów POSIX, oprócz różnych odmian UNIXa i Linuxa dołączył system operacyjny kontrolujący komputery Apple, Mac OS X. Poprzedni system, Mac OS był niekompatybilny z POSIXem, a więc decyzja Apple'a, o zmianie standardu API wiązała się z koniecznością przepisania większości oprogramowania pracującego na komputerach Apple.

Jak wspomniano, interfejs programisty mogą posiadać przeróżne biblioteki i aplikacje, najczęściej jednak spotkać się można z tym pojęciem w przypadku systemów operacyjnych. W przypadku systemów Microsoft, API obejmuje takie zagadnienia jak:

- funkcje podstawowe – procesy i wątki, błędy, obsługa systemu plików i urządzeń
- funkcje dodatkowe – kontrola systemu operacyjnego (wyłączanie, ponowne uruchamianie, usypianie), obsługa rejestru systemowego, kontrola usług i użytkowników
- funkcje obsługi powłoki
- funkcje obsługi sieci – protokoły TCP/IP, UDP/IP, NetBIOS a także komunikacja RPC i DDE (NetDDE)
- funkcje obsługi graficznych urządzeń wyjściowych – obsługa grafiki (przeznaczonej do wyświetlania na monitorze oraz wydruku). Część ta dotyczy tylko wyświetlania 2D (przeznaczonego głównie do wolnego wyświetlania np. elementów interfejsu). Za wyświetlanie akcelerowanej grafiki 2D oraz 3D odpowiadają biblioteki DirectX, które posiadają własne API
- funkcje dotyczące interfejsu użytkownika – okna, kontrolki, suwaki, obsługa myszy i klawiatury, typowych okien dialogowych (otwarcia i zapisu pliku, wydruku itd.), pasków postępu, stanu itd.

Warto podkreślić, że najczęściej API jest pisane pod konkretny język programowania a więc programista może z niego korzystać jedynie jeżeli programuje w tym samym języku. Istnieje jednak możliwość tworzenia tzw. bibliotek opakowujących (***wrapper***), które pośredniczą między API a aplikacją tworzoną w języku przez nie nie obsługiwanym. Przykładem takich rozwiązań w systemie Windows jest np. .NET Framework, który to pakiet jest napisaną w .NET biblioteką obsługującą natywne Win32API. Występują jednak również API, które umożliwiają (najczęściej) dostęp do funkcji systemu poprzez usługi sieciowe, które z kolei mogą być obsługiwane przez różne systemy i języki programowania.

Standard POSIX

Jak wspomniano wcześniej przy okazji omawiania ( ) historii systemów operacyjnych, po powstaniu systemu Unix i wielu jego odmian postanowiono poddać standaryzacji niektóre, najbardziej istotne składniki i funkcje SO, tak aby ułatwić pracę programistom projektującym aplikacje dla tych systemów. W wyniku tego postanowienia komitet Institute of Electrical and Electronics Engineers (IEEE) w 1988 roku opracował dokument o symbolu IEEE Std 1003.1-1988, który uzyskał bardziej przystępną nazwę POSIX (akronim określenia definiującego zawartość tego dokumentu – Portable Operating System Interface - przenośny interfejs systemu operacyjnego), którego autorem jest jedna z najbardziej znanych postaci świata komputerowego – Richard Stallman, założyciel Free Software Foundation, współtwórca licencji GNU GPL i jeden z autorów kompilatora gcc.

Dokument ten przede wszystkim definiuje interfejs programisty aplikacji systemu operacyjnego (API). Dzięki temu, że wszystkie systemy oznaczone certyfikatem zgodności z POSIX mają identyczne API, aplikacje napisane na dowolny z nich są przenośne na każdy inny, pod warunkiem, że jest on zgodny z POSIX. Ważną częścią dokumentu była specyfikacja fragmentu API dotyczącego procesów i wątków. Standard 1003.1 określa również elementy interfejsu użytkownika takie jak linia poleceń, interfejs skryptowy oraz szereg narzędzi takich jak chociażby awk, uruchamiający programy napisane w języku programowania AWK, przeznaczonym do

przetwarzania danych tekstowych. W kolejnych wydaniach standardu został on uzupełniony o rozszerzenia dotyczące systemów czasu rzeczywistego.

Zgodnie z dokumentacją, standard POSIX podzielony jest na kilkanaście części, z których najważniejsze to:

POSIX.1, Usługi bazowe (IEEE Std 1003.1-1988)

- Tworzenie i kontrola procesów
- Sygnały
- Wyjątki w obliczeniach zmiennoprzecinkowych
- Naruszenia segmentacji
- Nieprawidłowe instrukcje
- Błędy szyn danych
- Zegary
- Operacje na plikach i katalogach
- *Pipe'y*
- Standard ANSI C
- Interfejs i kontrola portów wejścia / wyjścia
- Process Triggers

POSIX.1b, Rozszerzenia dla systemów czasu rzeczywistego (IEEE Std 1003.1b-1993)

- Zarządzanie priorytetowe
- Sygnały czasu rzeczywistego
- Zegary i liczniki
- Semafory
- Przekazywanie komunikatów
- Pamięć dzielona
- Operacje wejścia / wyjścia w odmianie synchronicznej i asynchronicznej
- Interfejs blokowania pamięci

POSIX.1c, Rozszerzenia dotyczące wątków (IEEE Std 1003.1c-1995)

- Tworzenie, kontrola i usuwanie wątków
- Zarządzanie wątkami
- Synchronizacja wątków
- Obsługa sygnałów

P1003.1e

- Rozszerzenia dotyczące bezpieczeństwa systemu spełniające kryteria Departamentu Obrony USA

POSIX.2, Powłoka i narzędzia (IEEE Std 1003.2-1992)

- Interpreter komend
- Programy użytkowe

W obecnej chwili prace nad POSIXem prowadzi Austin Group przy współpracy z komitetem normalizacyjnym ISO i The Open Group.

O zgodności danego systemu operacyjnego ze standardami POSIX orzeka jednostka certyfikująca po przejściu przez oprogramowanie tzw. zestawu testów PCTS czyli **POSIX Conformance Test Suite**.

Ze względu na to, że otrzymanie certyfikatu POSIX jest kosztowne, część systemów operacyjnych jest z nim zgodna (tzn. spełnia założenia standardu), ale nie posiada wspomnianego certyfikatu. Często zdarza się również, że SO w niewielkim stopniu odbiega od standardu, różniąc się mało istotnymi szczegółami, co nie pozwala jednak na uzyskanie certyfikatu zgodności. Takim systemem jest np. Linux.

Systemy czasu rzeczywistego

Poprzez system operacyjny czasu rzeczywistego (RTOS) rozumiany jest system, który zareaguje na pewien wejściowy sygnał i zwróci wynik operacji wykonanej na skutek tego sygnału w ściśle określonym czasie, bez przekroczenia granicznej wartości opóźnienia. Co ważne, długość tego czasu nie ma znaczenia z punktu widzenia definicji RTOS, istotne jest tylko to, że jest on nieprzekraczalny, choć oczywiście możliwie szybka reakcja jest zwykle pożądana. Jak wspomniano w rozdziale opisującym rodzaje systemów operacyjnych, systemy czasu rzeczywistego służą najczęściej do kontroli procesów przemysłowych, urządzeń sterujących (np. ogrzewaniem), multimedialnych (np. odtwarzacze CD) itp. Jak łatwo sobie wyobrazić, w takich zastosowaniach system o zachowaniach niedeterministycznych, czyli braku granicy czasu reakcji na bodziec zewnętrzny, jest niemożliwy do zaakceptowany.

Z drugiej strony jednak, sam system operacyjny czasu rzeczywistego nie gwarantuje determinizmu operacji we wszystkich zastosowaniach, ponieważ zależy ona również od oprogramowania na tym systemie uruchamianego. Jednakże, przy zachowaniu reguł i korzystania z mechanizmów RTOS, programista może osiągnąć deterministyczny styl pracy aplikacji i całego systemu, co nigdy nie jest możliwe w przypadku zastosowania zwykłego systemu operacyjnego.

Jak wspomniano wcześniej, istnieją dwa typy systemów operacyjnych czasu rzeczywistego – *hard* i *soft* RT. Systemy *hard* charakteryzują się tym, że każde zadanie musi zostać zakończone przed upływem określonego limitu czasu, żadne jego naruszenie nie jest akceptowalne ponieważ skutkowałoby np. zniszczeniem urządzenia lub inną awarią. Systemy *soft* zezwalają na przekroczenie limitu czasu przez zadanie, choć algorytmy zarządzające kolejkowaniem wiadomości dążą do tego, aby ów limit był spełniany.

Różnice pomiędzy systemami operacyjnymi ogólnego zastosowania a systemami czasu rzeczywistego przejawiają się głównie w następujących zagadnieniach [STALLING]:

- Determinizm zadań (czyli oznaczalność czasu po jakim zadanie zostanie wykonane)
- Czas reakcji na sygnały (czyli czas, który upływa od momentu wystąpienia sygnału do reakcji na niego). Razem z czasem, w jakim dane zadanie zostanie wykonane określa łączny czas odpowiedzi systemu na zdarzenie.
- Kontrola użytkownika (znacznie większa w stosunku do kontroli w systemach ogólnego zastosowania. W RTOS użytkownik ma bardzo duże możliwości ustalania priorytetów zadań, określania które z nich są krytyczne, a które nie dla działania systemu, często również może kontrolować zarządzanie pamięcią, specyfikować zadania, które nie mogą być przeniesione do pliku wymiany itd.
- Niezawodność (wymagania stabilności systemu operacyjnego RT są znacznie większe niż systemów ogólnego użycia)
- Odporność na błędy (w momencie wykrycia błędu, zatrzymanie awaryjne systemu, tak jak to ma miejsce w zwykłych systemach nie jest akceptowalnym rozwiązaniem, system powinien kontynuować pracę, zapewniając obsługę przynajmniej najbardziej krytycznych zadań. W wypadku awarii RTOS dąży również do zabezpieczenia danych przed uszkodzeniem)

Główną częścią systemu RT odróżniającą go od od systemów generalnego zastosowania jest mechanizm planowania zadań (*scheduler*), którego głównym celem jest zapewnienie wykonania zadań krytycznych (czyli niezbędnych do bezawaryjnego działania systemu) przed upływem

określonego czasu od ich uruchomienia. Algorytmy zarządzające szeregowaniem zadań dążą również do wykonania w stałych ramach czasowych zadań niekrytycznych, jednak zdarza się, że są one poświęcane aby zapewnić poprawne wykonanie zadań ważniejszych.

Odmierzanie czasu

Odmierzanie czasu w komputerach PC i podobnych zorganizowane jest najczęściej poprzez wykorzystanie programowalnego generatora impulsów, którym często jest układ Intel 8253 lub 8254. Układ ten generuje sygnały elektryczne, które są wprowadzane na linię przerwań systemowych (zwykle IRQ-0). Generator ten być zaprogramowany do pracy w sześciu różnych trybach, między innymi może być źródłem impulsu jednokrotnego, który kreowany jest po upływie ustalonego czasu, może też być generatorem okresowym, w którym impulsy generowane są powtarzalnie, co założony przez użytkownika okres. Choć na pierwszy rzut oka jednokrotne generowanie impulsu wydaje się nieprzydatne w systemie komputerowym, to w przypadku systemu RT, zwłaszcza w przypadku obsługiwania przezeń zadań okresowych (których okresy są znane) ma ono sens ponieważ umożliwia ograniczenie do minimum liczby generowanych przez zegar przerwań (których obsługa wprowadza zaburzenia w wykonaniu procesów). Z drugiej strony programowanie generatora trwa dość długo, zwłaszcza na jednoprocesorowych platformach x86. Najpopularniejszym jednak trybem pracy generatora impulsów jest praca okresowa.

Szeregowanie zadań

Podstawowy (najprostszy) mechanizm szeregowania zadań, w przypadku kiedy zadania te są jednakowe (tzn. ich wykonanie trwa tyle samo czasu, uruchamiane są synchronicznie w stałych odstępach czasowych) może wykorzystywać prosty algorytm kolejki typu *round-robin*. Jeżeli nie występuje potrzeba reakcji na asynchroniczne sygnały, to ten typ szeregowania sprawdza się doskonale, dodatkowo zapewniając praktycznie 100% determinizm zdarzeń. Nowe zadania są umieszczane na końcu kolejki, a do procesora przekazywane jest zadanie znajdujące się na jej początku. Po upłynięciu interwału przewidzianego na wykonywanie jednego procesu następuje przeniesienie bieżąco wykonywanego zadania na koniec kolejki, która jest przesuwana, a procesor przydzielany jest zadaniu, które znajduje się teraz na jej czele. Jeżeli wydajność obliczeniowa urządzenia z RTOS jest na tyle duża, że wykonanie pojedynczego zadania trwa krócej niż czas między zadawaniem nowych, to system ten pracuje w pełni deterministycznie. Niestety, tak prosty algorytm szeregowania może być stosowany tylko w bardzo podstawowych urządzeniach jak np. odtwarzacze CD.

W większości zastosowań systemów RT nie występują tak proste warunki pracy – zwykle uruchomionych jednocześnie jest kilka procesów, z których każdy ma inny termin wykonania i zakres pracy do wykonania (tzn. potrzebuje dostępu do procesora na inny okres).

Przed wyborem algorytmu szeregującego projektant systemu opartego o komputer z RTOS musi sprawdzić czy w ogóle możliwe jest wykonanie niezbędnych zadań w określonym czasie na danej platformie sprzętowej (ze względu na wydajność obliczeniową procesora). Sprawdzenie to polega na wyznaczeniu wartości parametru S, określonego poniższym wzorem:

$$S=\sum_{i=0}^{n}\frac{T_i}{O_i}$$

n – liczba uruchomionych zadań, T – czas przetwarzania zadania, O – okres powtarzalności zadania

Jeżeli wyznaczona z powyższego równania wartość S jest mniejsza niż 1, to teoretycznie możliwe jest wykonanie wszystkich zadań w określonych ramach czasowych i z określoną powtarzalnością na analizowanej platformie sprzętowej. Oczywistym jest jednak to, że system powinien posiadać pewien zapas mocy obliczeniowej i stosowanie takich konfiguracji, dla których wartość S jest bardzo bliska 1 jest niewskazane.

W ogólnym rozróżnieniu można wskazać dwa typy algorytmów szeregowania zadań w systemach RT: statyczne i dynamiczne. Metody statyczne szeregują zadania w kolejce w niezmiennej kolejności (tzn. ustalonej w momencie uruchomienia), najczęściej na podstawie ich priorytetów. Metody dynamiczne z kolei modyfikują kolejkę na bieżąco biorąc pod uwagę przeróżne właściwości procesów lub warunki wykonania operacji.

Metodą szeregowania zadań należącą do grupy metod statycznych jest algorytm zwany **Rate Monotonic Scheduling - RMS** (motoniczne szeregowanie ze względu na powtarzalność zadania). Głównym czynnikiem wpływającym na kolejność uruchamiania procesów jest okres ich powtarzalności. Zadania, które mają dużą częstotliwość powtarzania przydzielany mają wysoki priorytet, co powoduje że są wykonywane z uprzywilejowaniem w stosunku do zadań które są rzadko uruchamiane. Rozwiązanie to zapewnia dobre planowanie zadań w przypadku jeżeli ich oczekiwane czasy zakończenia są zbieżne z okresem powtarzalności. Co istotne, RMS nie jest algorytmem, który pozwala na maksymalne wykorzystanie procesora,w przypadku wielu zadań uruchomionych jednocześnie, wykorzystywać mogą one (teoretycznie) jedynie ok. 70% czasu procesora[35]. Niemniej, algorytm ten jest bardzo często wykorzystywany w szeregowaniu zadań w systemach czasu rzeczywistego ponieważ w normalnych okolicznościach można uzyskać większe wykorzystanie procesora, a ponadto pozostały, niewykorzystany czas może zostać użyty do przetwarzania zadań niekrytycznych.

Ważną cechą algorytmu szeregowania RMS jest stabilność sterowanego nim systemu – w przypadku wystąpienia przeciążenia systemu (zlecenia większej liczby zadań niż jest w stanie wykonać procesor) lub wystąpienia błędu powodującego zmniejszenie wydajności systemu, można zapewnić wykonywanie najważniejszych zadań krytycznych w terminie poprzez takie ich zaprojektowanie, aby cechowały się wysoką częstotliwością powtarzania. Wtedy będą one zawsze wykonywane w pierwszej kolejności co nawet przy zmniejszeniu zdolności przetwarzania systemu pozwoli na wykonanie ich w terminie. Jest to bardzo prosta, ale skuteczna metoda zapobiegania awariom całego systemu RT w przypadku błędów lub przeciążeń.

Kolejnym algorytmem, często używanym w systemach operacyjnych czasu rzeczywistego jest dynamiczny algorytm ***Earliest Deadline First – EDF*** (szeregowania zadań ze względu na czas pozostały do terminu w którym muszą być wykonane). Metoda ta działa na podstawie analizy listy procesów oczekujących na dostęp do CPU, przy czym lista ta jest układana według terminu zakończenia zadania, w ten sposób, że początkowe miejsca zajmują procesy, które muszą być zakończnone najszybciej. System uruchamia procesy w kolejności określonej w tej kolejce. W momencie kiedy zostaje uruchomione nowe zadanie, system analizuje termin jego zakończenia i na jego podstawie umieszcza ów proces w konkretnym miejscu kolejki. Jeżeli okaże się że termin zakończenia nowego zadania jest wcześniejszy niż bieżąco wykonywanego, to zadanie bieżące zostanie przerwane a rozpocznie się wykonywanie zadania priorytetowego.

Największą zaletą algorytmu EDF jest jego zdolność do działania w przypadku zdarzeń asynchronicznych, co nie jest możliwe w przypadku użycia metody planowania RMS. Dodatkowo, wykorzystanie procesora niezależnie od liczby zadań może sięgać nawet 97.5%, co nie powoduje niepotrzebnych strat mocy obliczeniowej jak w wypadku RMS. Wadą algorytmu EDF jest z kolei narzut obliczeniowy związany z wyznaczeniem oczekiwanych terminów zakończenia procesów oraz ich sortowaniem w kolejce zadań. Zapewnienie wykonania najważniejszych zadań w przypadku przeciążenia lub awarii nie jest tak łatwe jak w przypadku *Rate Monotonic Scheduling*

35 Analiza teoretycznej granicy wykorzystania procesora w zależności od liczby uruchomionych w danej chwili zadań znajduje się na przykład w [STALLING]

ponieważ priorytet każdego zadania w EDF jest modyfikowany dynamicznie.

Mechanizm przerwań oraz synchronizacji zadań.

Mechanizm obsługi przerwań w systemach czasu rzeczywistego różni się nieco od tego stosowanego w systemach operacyjnych ogólnego zastosowania. Z racji tego, że (w każdym) systemie nadchodzące przerwanie zaburza tok wykonywanych procesów przesuwając termin ich zakończenia, w systemach RT aby zminimalizować czas, o który przesuwane jest wykonanie zadań krytycznych, funkcje obsługujące przerwania są projektowane tak, aby wykonywały się jak najkrócej, zwykle tylko potwierdzając przyjęcie przerwania lub jego zablokowanie do momentu zakończenia procesu przezeń wywołanego. W dalszej kolejności uruchamiany jest proces przetwarzający, często poprzez „obudzenie" procesu sterownika danego urządzenia, które wywoływane jest najczęściej przez odblokowanie określonego semafora.

Jak wspomniano w rozdziale **2**, w przypadku kiedy wykonywane jednocześnie procesy wykorzystują te same zasoby, konieczne jest wprowadzenie mechanizmu synchronizacji i wzajemnego blokowania. Najczęściej jest to realizowane poprzez *mutexy* i semafory. Z wzajemnym blokowaniem jak wcześniej wspomniano związany jest problem zakleszczania procesów. W systemach czasu rzeczywistego najczęściej powoduje ono powstanie zjawiska zwanego odwróceniem priorytetów (***priority inversion***) czyli sytuacji, w której wykonanie zadania krytycznego jest wstrzymywane, ponieważ dowolne z zadań o niższym priorytecie dokonało zablokowania współdzielonego zadania. W momencie, kiedy uruchomione są tylko te zadania, problem ten wiąże się z opóźnieniem zadania krytycznego, niemniej jednak nie z kompletnym zatrzymaniem jego obsługi. W sytuacji, kiedy oprócz tych dwóch procesów uruchomiony jest dowolny inny o priorytecie mieszczącym się pomiędzy priorytetami zadań poprzednich, to proces ten wstrzymuje wykonanie zadania mało istotnego (bo jest istotniejszy), co pociąga za sobą dalsze wstrzymywanie procesu krytycznego, który wciąż czeka na odblokowanie współdzielonego zasobu. Opracowano kilka sposobów przezwyciężenia sytuacji, z których najczęściej używaną metodą jest dziedziczenie priorytetów polegające na chwilowym przypisaniu priorytetu zadania krytycznego zadaniu mało istotnemu jeżeli blokuje ono zasób potrzebny procesowi RT. W ten sposób rozwiązano problem odwrócenia priorytetów, który wystąpił w systemie VxWorks sterującym pracą Mars Pathfinder.

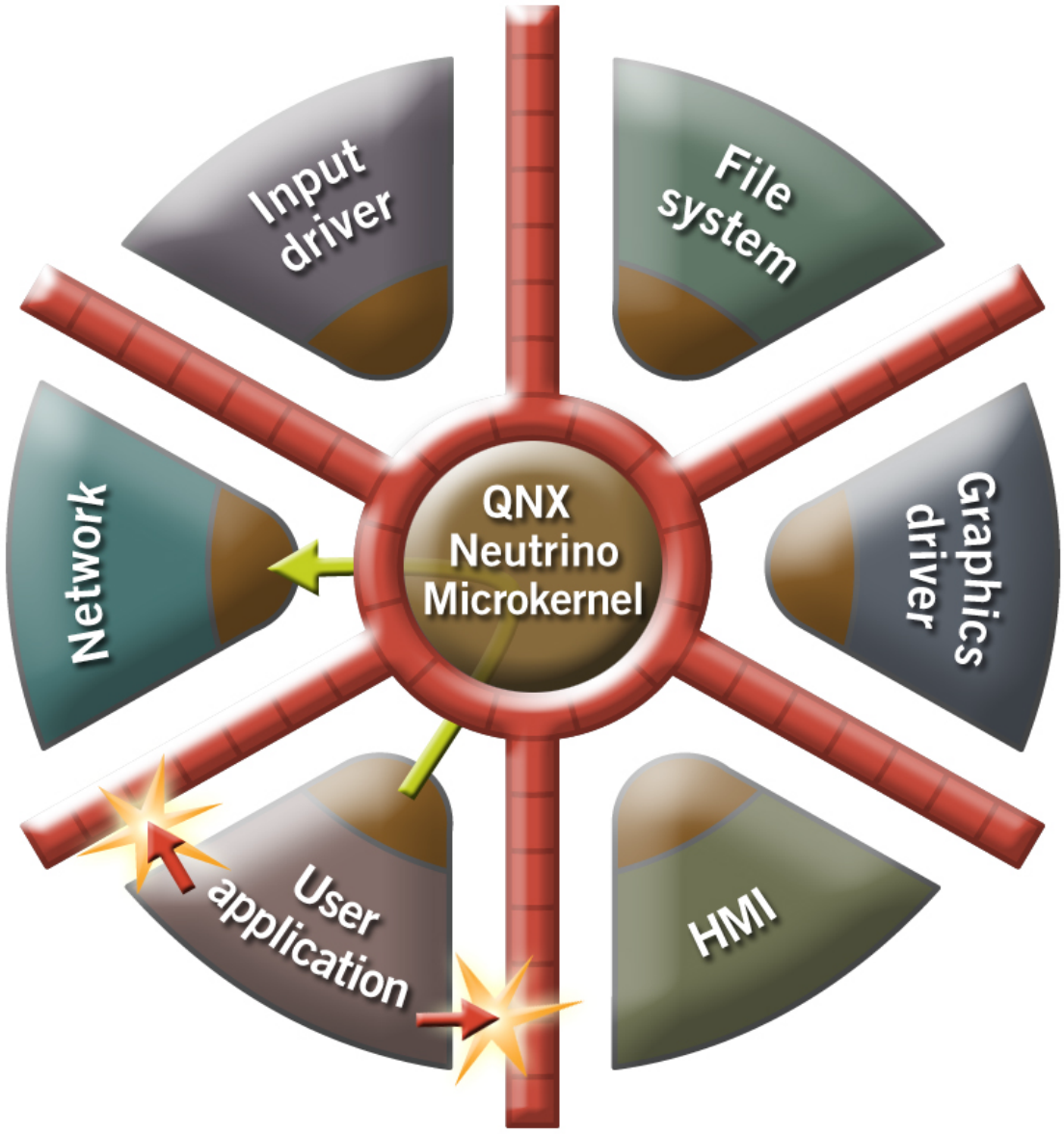
Przykładowe systemy czasu rzeczywistego - QNX oraz RTLinux

Istnieje wiele rozmaitych systemów operacyjnych czasu rzeczywistego, spośród nich można wymienić np. VxWorks (kontrolujący m.in. pracę robota badającego Marsa w misji Pathfinder z 1994 roku), Windows CE (kontrolujący urządzenia przenośne typu palmtop), LynxOS, QNX oraz RTLinux. Te dwa ostatnie systemy są dostępne w wersjach komercyjnych (zamkniętych) oraz z otwartym kodem, do zastosowań niekomercyjnych. Oba systemy charakteryzują się interesującymi rozwiązaniami niektórych problemów związanych z pracą w czasie rzeczywistym, które zostaną przedstawione poniżej.

QNX

System QNX jest zbudowany w oparciu o mikrojądro, które zapewnia bazowe funkcjonalności OS, takie jak zarządzanie przydziałem procesów, obsługa komunikacji międzyprocesowej, zegarów oraz

przerwań. Wszystkie inne funkcjonalności, nawet zarządzanie pamięcią operacyjną znajdują się poza jądrem, podobnie jak sterowniki urządzeń. Do uruchomienia tak zaprojektowanego systemu niezbędny jest specjalny program ładujący, który poza jądrem systemu potrafi załadować dowolny program użytkownika.



Architektura systemu QNX (źródło – www.qnx.com)

Wszystkie procesy systemu operacyjnego pracujące poza jądrem, traktowane są jako miniserwery usług co pozwala na łatwą minimalizację rozmiaru systemu operacyjnego – dodawane i uruchamiane są tylko te procesy, które są niezbędne, nie wymaga to rekompilacji jądra jak w klasycznych konstrukcjach OS z jądrem monolitycznym. Mały rozmiar jądra i niezbędnych usług pozwala na zastosowanie tego systemu operacyjnego w wielu specyficznych rozwiązaniach o ograniczonych zasobach pamięci (systemach wbudowanych) jak np. samochodowe komputery pokładowe.

Istotną częścią QNX jest implementacja komunikacji międzyprocesowej. Jak wspomniano powyżej, znajduje się ona w jądrze systemu, przekazywanie wiadomości między procesami obsługiwane jest przez kernel poprzez kopiowanie określonego obszaru pamięci procesu

wysyłającego do przestrzeni adresowej odbiorcy. Co istotne, w przypadku gdy odbiorca oczekuje na wiadomość, to wraz z nią przydzielany jest mu dostęp do procesora (z pominięciem *schedulera*) co pozwala na sprawniejsze zarządzanie procesami, które wymieniają informacje między sobą. Komunikacja międzyprocesowa jest podstawą działania wszystkich operacji wejścia / wyjścia w QNX (łącznie z obsługą dysków i sieci). Zarządzanie kolejnością przekazywania komunikatów (co wiąże się również ze zmianą kolejności wykonywanych zadań) odbywa się w oparciu o priorytety procesów przekazujących wiadomości, a więc procesy krytyczne otrzymują dostęp do urządzeń wejścia / wyjścia przed procesami o małym priorytecie.

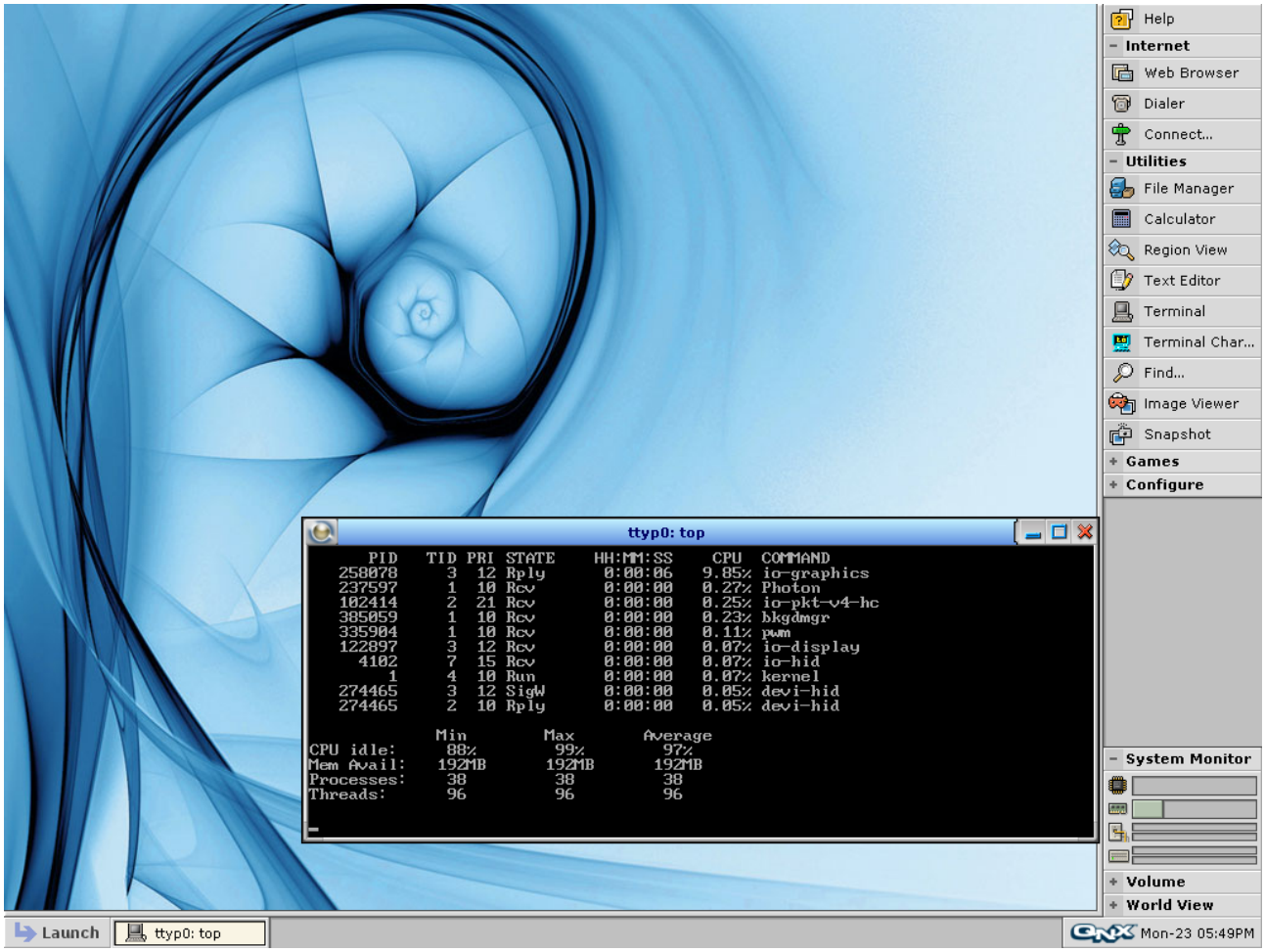
W systemie QNX procedura wyznaczenia następnego procesu, któremu zostanie przydzielony procesor, wykonywana jest w sytuacji gdy proces o priorytecie wyższym niż bieżąco uruchomiony zostanie odblokowany a także wtedy, kiedy bieżący proces zostanie wywłaszczony oraz w momencie kiedy przydzielony procesowi czas dostępu do procesora wyczerpie się. Należy pamiętać, że decyzja o przydzieleniu procesora dotyczy tylko i wyłącznie procesów w stanie gotowości (***ready***), podczas gdy procesy zablokowane (***blocked***) nie są brane pod uwagę. Głównym czynnikiem warunkującym uruchomienie danego procesu jest jego priorytet (QNX 4 posiada 32 poziomy przywilejów, przy czym 0 oznacza najniższy. QNX Neutrino rozszerza ten zakres do 64 wartości). Uruchamiany jest zawsze proces o najwyższym priorytecie. Na poszczególnych poziomach przywilejów decyzja szeregowania procesów jest podejmowana według jednego z trzech algorytmów (wybieranych przez użytkownika):

- FIFO (procesy uruchamiane w kolejności ich umieszczenia w kolejce)
- *Round-robin* (procesom przydzielany jest określony interwał, po którym następuje wywłaszczenie bieżącego procesu i uruchomienie następnego w kolejce. Wywłaszczony proces umieszczany jest na końcu kolejki)
- Szeregowania adaptacyjnego (który, w przypadku nie wystąpienia przeciążeń procesora pracuje jak scheduler *round-robin* rezerwując jednak pulę procesów, dla których na stałe przypisana jest określona część zasobów systemowych co pozwala na ich wykonanie nawet w przypadku wystąpienia nadmiernego obciążenia systemu. Procesy spoza wskazanej puli mogą być w tym wypadku pomijane. Algorytm ten obecny jest tylko w systemie QNX 4, w Neutrino używane są tylko dwa pierwsze.

QNX niemalże od początku istnienia jest zgodny ze standardem POSIX, a więc możliwe jest przenoszenie aplikacji z innych POSIXowych systemów na tą platformę. Co ciekawe, wśród tych aplikacji jest również graficzny interfejs użytkownika X Window System, obecny zarówno w Unixie jak i Linuxie. QNX dostarcza również swoją wersję GUI o nazwie PhotonMicroGUI.

System operacyjny QNX pracuje na takich platformach sprzętowych jak procesory x86, MIPS, PowerPC, ARM, StrongARM, Xscale i SH-4. Obecny jest w wielu zastosowaniach, najciekawsze z nich to komputery pokładowe samochodów (w obecnej chwili ponad 200 modeli) oraz system operacyjny urządzeń sieciowych Cisco (routerów i zarządzalnych switchy).

Jak wspomniano, system ten jest dostępny za darmo dla użytkowników prywatnych, można go pobrać ze strony producenta (http://www.qnx.com).



System QNX Neutrino z graficznym interfejsem użytkownika Photon

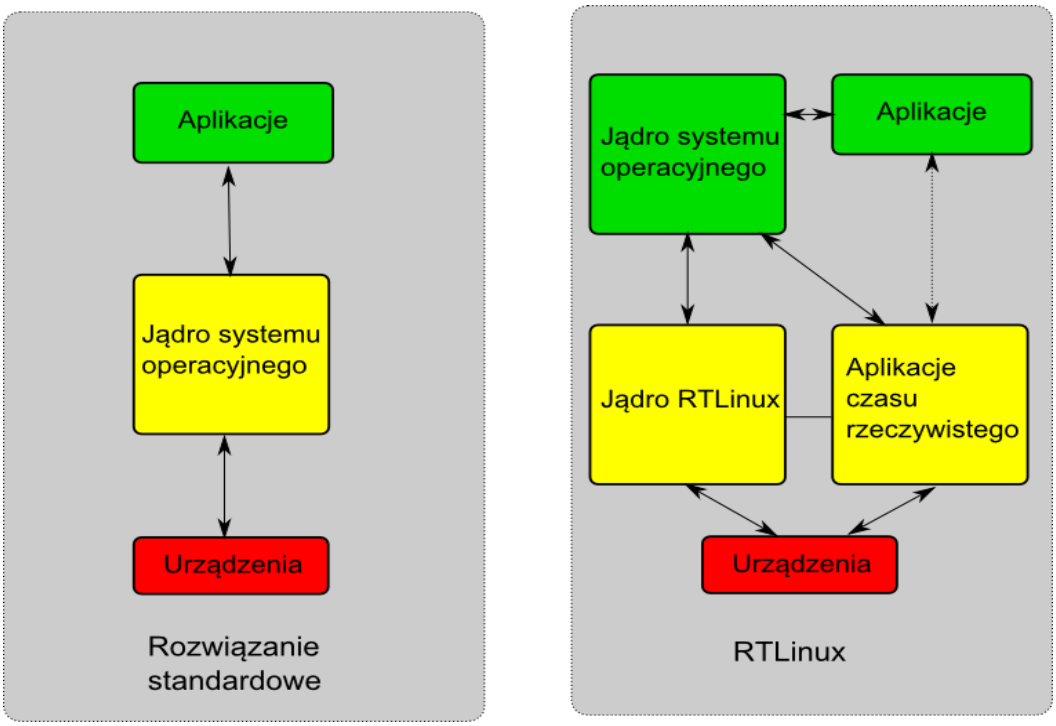
RTLinux

Konstrukcja systemu operacyjnego RTLinux ma swoje początki w opracowanym w Bell Labs pod koniec lat 80 eksperymentalnym systemie o nazwie MERT. Głównym celem projektowym tego systemu była możliwość jednoczesnego uruchamiania procesów czasu rzeczywistego oraz procesów ogólnych, takich jak np. aplikacje użytkownika. Zostało to zrealizowane poprzez podział głównego OS na dwie części – czasu rzeczywistego i ogólnego użycia. W systemie RTLinux, powstałym w Institute for Mining and Technology of New Mexico, zastosowano podobne rozwiązanie, mianowicie jądro systemu RT, będące główną częścią systemu uruchamia zwykłe jądro Linuxa, ale jako usługę o najniższym priorytecie, która podlega normalnej procedurze szeregowania zadań. W ten sposób, zadania krytyczne, są wykonywane z pierwszeństwem a wszystkie zadania związane z normalnym działaniem systemu (np. interfejs użytkownika, aplikacje niekrytyczne) mają automatycznie niski priorytet i wykonywane są wtedy, kiedy nie zakłócają pracy zadań krytycznych.

Architektura ta pozwala na uzyskanie bardzo dobrej wydajności zaawansowanych usług i programów pracujących pod kontrolą zwykłego systemu Linux przy gwarancji obsługi najważniejszych zadań w sposób deterministyczny.

Porównanie architektur standardowego systemu operacyjnego i architektury RTLinuxa znajduje się na rysunku (RYSUNEK). Jak opisano w rozdziale poświęconym budowie systemów operacyjnych, jądro systemu pośredniczy pomiędzy urządzeniami fizycznymi a procesami systemowymi, zarządzając ich dostępem do tych urządzeń. W przypadku systemu RTLinux, główne

jądro komunikuje się z urządzeniami, jednak pomiędzy procesami systemowymi (niekrytycznymi), a jądrem RT pośredniczy zwykłe jądro Linuxa. Co ważne, jądro to odwołuje się do urządzeń w ten sam sposób co w standardowym użyciu ponieważ kernel RTLinux symuluje ich obecność poprzez tzw. wirtualny mechanizm przerwań, który jest jednym z głównych elementów architektury tego systemu. Jądro systemu przechwytuje wszystkie przerwania sprzętowe i analizuje je pod kątem tego, czy istnieje jakakolwiek procedura czasu rzeczywistego, która obsługuje przychodzące przerwanie. W przypadku, kiedy taka procedura istnieje, to zostaje wywołana, w przeciwnym razie informacja o przerwaniu przekazywana jest do zwykłego jądra Linuxa. Informacja ta jest przekazywana natychmiastowo, jeżeli nie jest właśnie wykonywane jakiekolwiek zadanie czasu rzeczywistego, inaczej jest ona przekazywana po jego zakończeniu.



Porównanie organizacji systemu operacyjnego ogólnego zastosowania (z lewej) i RTLinux (z prawej)

Interesującym rozwiązaniem w RTLinux jest umieszczenie zadań krytycznych (czasu rzeczywistego) w przestrzeni adresowej jądra RT, bez ochrony pamięci. Wynikiem tego jest duży zysk czasowy związany z szybkim przełączaniem kontekstu (czyli zmianie wykonywanego procesu) nie obarczonego koniecznością zmiany rejestrów bazowych MMU związanych z przełączaniem przestrzeni adresowych oraz zdolność do przekazywania danych między procesami RT bez konieczności użycia komunikacji międzyprocesowej. Niestety, rozwiązanie to jest zupełnie nieodporne na błędy programistyczne, nadpisanie pamięci może się wiązać (i najczęściej tak jest) z awarią całego systemu. Procesy RT ładowane są przy uruchomieniu systemu jako moduły jądra. Mogą one być okresowe, jednorazowe oraz uśpione do momentu wystąpienia jakiegoś zdarzenia, najczęściej przerwania systemowego.

Kolejnym interesującym rozwiązaniem RTLinuxa, czyniącym go bardzo elastycznym systemem RT jest to, że *scheduler* jest tutaj również ładowany jako zewnętrzny moduł jądra systemowego, a więc może on być podmieniony na dowolny, stworzony przez użytkownika. Standardowo używanym algorytmem szeregowania zadań jest tzw. priorytetowe wywłaszczanie (***priority preempting***), w którym uruchomione zadanie jest wykonywane przez procesor do

momentu jego naturalnego zakończenia (lub samodzielnego oddania kontroli nad procesorem) lub do chwili, w której zostanie uruchomione inne zadanie o wyższym priorytecie.

Szeregowanie zadań

Program szeregujący jest zaimplementowany jako ładowalny moduł jądra RTCore, więc użytkownik może napisać własny, lepiej dostosowany do konkretnej implementacji systemu. Algorytmy szeregowania stosowane w systemach czasu rzeczywistego opierają się zwykle na priorytetowym wywłaszczaniu tzn. zadanie krytyczne jest wykonywane dopóki dobrowolnie nie odda procesora albo w kolejce zadań gotowych nie pojawi się zadanie o wyższym priorytecie. Domyślnym w RTLinuxie algorytmem jest przedstawiony wcześniej w bieżącym rozdziale algorytm Rate Monotonic Scheduling. Od pewnego dostarczany jest również *scheduler* pracujący według metody Earliest Deadline First. Metody zarządzania przydzielaniem procesów zawierają również implementację algorytmów Slot Shifting oraz Stack Stealing, co pozwala na poprawę obsługi zadań asynchronicznych, uruchamianych w trakcie wykonywania zadań okresowych.

Ważnym zadaniem związanym ze współpracą jądra systemu RT (i programów czasu rzeczywistego) z zaawansowanymi usługami zwykłego systemu operacyjnego jest komunikacja międzyprocesowa. Jest to jedyny sposób umożliwiający wymianę danych pomiędzy programami RT a aplikacjami i funkcjami systemowymi zwykłego systemu – procesy krytyczne nie mogą bowiem wywoływać funkcji systemowych jądra zwykłego systemu Linux. W RTLinux istnieją dwie metody przekazywania komunikatów między procesami – poprzez pamięć dzieloną oraz kolejki czasu rzeczywistego. Dostęp do pamięci dzielonej jest zgodny ze standardowymi metodami określonymi w standardzie POSIX. Kolejka RT działaja na zasadzie podobnej do *named-pipe*, zadania uruchomione w przestrzeni jądra RT mogą tego typu kolejki tworzyć i usuwać oraz oczywiście zapisywać i odczytywać z nich informacje. Z kolei w zwykłym systemie Linux są one widoczne jako urządzenia znakowe (o lokalizacji /dev/rtfx, gdzie x to numer kolejki) a dostęp do nich odbywa się poprzez funkcje określone w standardzie POSIX.

Jak wspomniano powyżej, obsługa komunikacji międzyprocesowej odbywa się w sposób zgodny z POSIX, to nie jedyny punkt zgodności z tym standardem, RTLinux jest zgodny z ustaleniami zawartymi w POSIX 1003.13 (określającym minimalny interfejs systemów czasu rzeczywistego).

System RTLinux jest dostępny w postaci wolnej (OpenRTLinux) oraz komercyjnej Wind River Real-Time Core, dystrybuowanej przez WindRiver.

Przykładowym zastosowaniem omawianego systemu jest demonstracyjna wersja robota humanoidalnego HOAP-3 firmy Fujitsu. System operacyjny kontroluje pracę robota (którego węzły w sumie mają 28 stopni swobody) w oparciu o platformę sprzętową zawierającą procesor Pentium 1.1 GHz.

Robot Fujitsu HOAP3 kontrolowany przez system operacyjny czasu rzeczywistego RTLinux

Administracja systemem Microsoft Windows oraz Linux

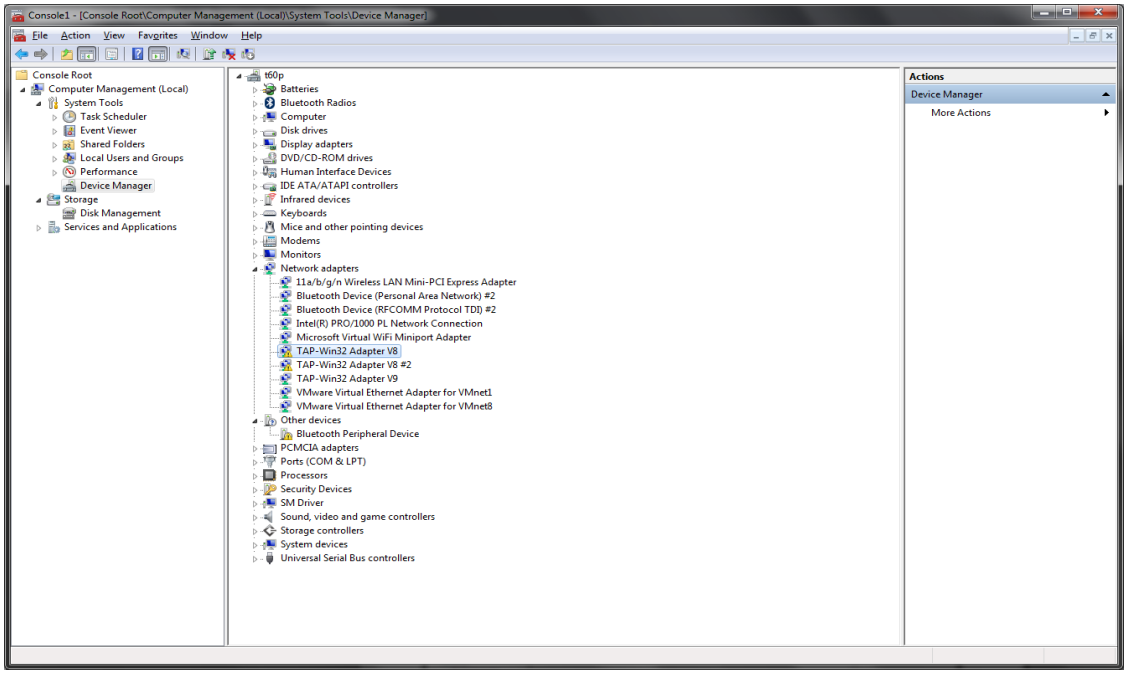
W obecnych czasach najpopularniejszymi systemami operacyjnymi ogólnego zastosowania są systemy Microsoft Windows (w wersji osobistej Vista lub 7 oraz serwerowej 2008 Server) oraz różne dystrybucje Linuxa. W bieżącym rozdziale omówione zostaną podstawowe narzędzia i czynności służące administracji tymi systemami. Podane metody konfiguracyjne (o ile nie podano inaczej) dotyczą systemu Microsoft Windows 7 Professional oraz Ubuntu Linux 8.04 LTS.

Podstawowe narzędzia i czynności administracyjne w systemach operacyjnych

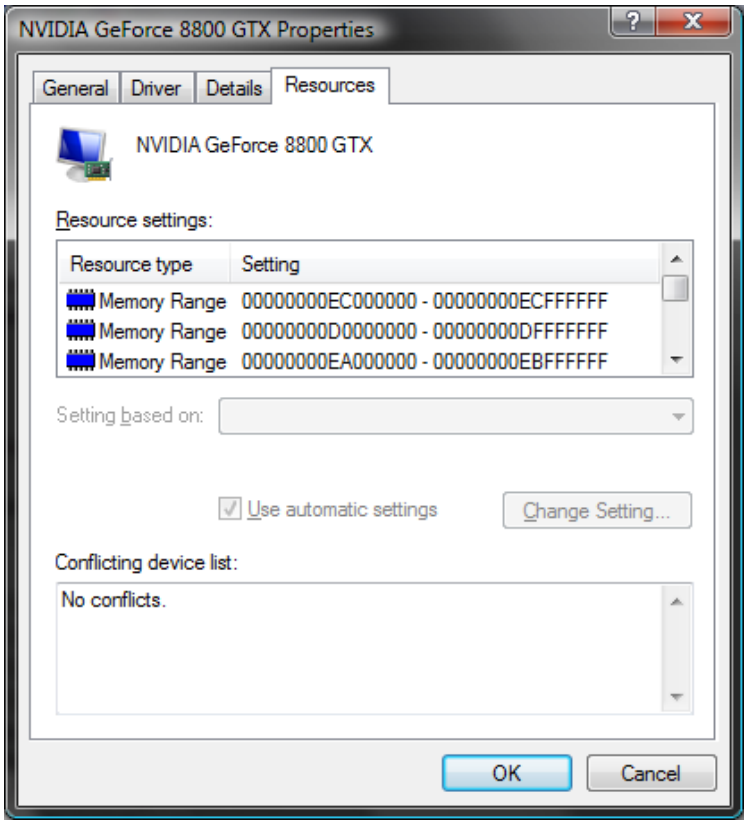
W zależności od zastosowania systemu operacyjnego (komputer osobisty, serwer WWW, router, serwer poczty, serwer ftp), zakres czynności administracyjnych jest w oczywisty sposób różny. Niemniej jednak, w zasadzie każdy użytkownik komputera wykonuje mniej lub bardziej świadomie pewien zestaw operacji, które są takie same bez względu na zastosowanie jego komputera.

Zarządzanie zasobami systemowymi

Podczas konfigurowania komputera do pracy, zwłaszcza po instalacji systemu lub przy dodaniu nowego urządzenia niezbędne jest wyświetlenie wykrytych przez system urządzeń. W systemie Microsoft Windows jest to wykonywane poprzez otwarcie konsoli zarządzania MMC, rozwinięcie w niej opcji Zarządzanie komputerem (***Computer Management***) i wybranie opcji Manager Urządzeń (***Device Manager***). Lista obecnych w systemie urządzeń wyświetlana jest w oknie MMC, przy każdym z tych urządzeń widnieje informacja o tym czy pracuje ono poprawnie czy nie (na poniższym rysunku system wskazuje że karta sieciowa TAP-Win32 Adapter V8 i V8#2 nie pracuje poprawnie). Wybranie dowolnego urządzenia poprzez podwójne kliknięcie myszą otwiera dodatkowe okno, w którym użytkownik może sprawdzić jakich zasobów używa dane urządzenie a także nakazać aktualizację sterownika bądź jego odinstalowanie.



Okno przedstawiające listę urządzeń zidentyfikowanych przez system



Szczegóły konfiguracji urządzenia (karty graficznej)

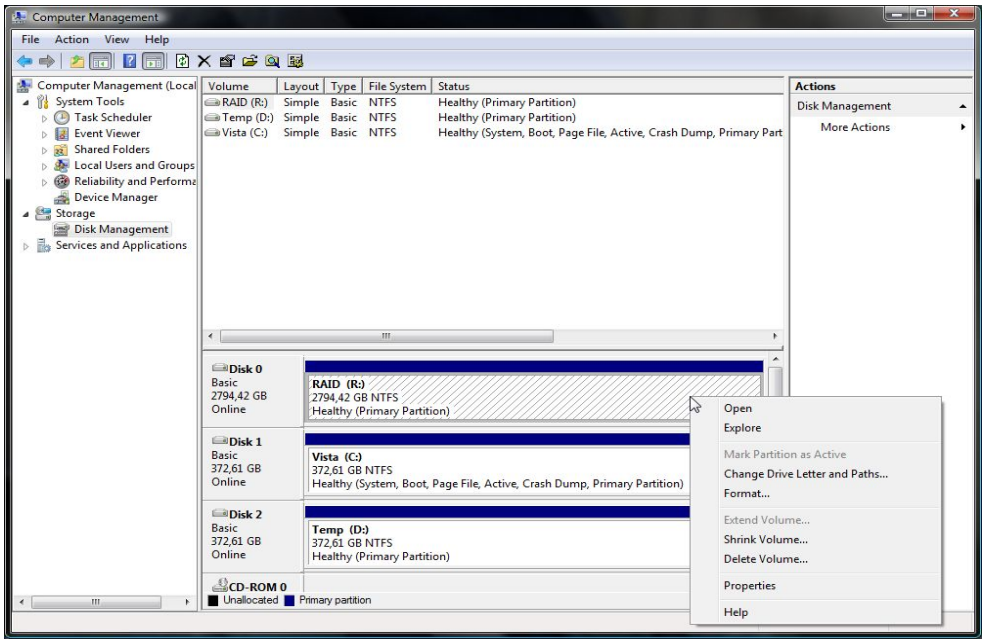
W systemie Linux wszystkie informacje na temat podłączonych do komputera urządzeniach można znaleźć przeglądając specjalny katalog /proc zawierający wirtualny system plików, które zawierają szczegółowy opis zasobów i opcji konfiguracyjnych urządzeń. Przykładowo, szczegóły dotyczących urządzeń wpiętych w gniazda PCI płyty głównej mogą być odnalezione w podkatalogu /proc/pci, procesora w /proc/cpuinfo itd:

Szczegóły urządzeń PCI	Szczegóły procesora

[negative@zto-serw ide]$ cat /proc/pci
PCI devices found:
  Bus 0, device   0, function 0:
    Class 0600: PCI device 8086:3590 (rev 10).
  Bus 0, device   1, function 0:
    Class 0880: PCI device 8086:3594 (rev 10).
      Non-prefetchable 32 bit memory at 0x40000000 [0x40000fff].
  Bus 0, device   2, function 0:
    Class 0604: PCI device 8086:3595 (rev 10).
      IRQ 169.
      Master Capable. No bursts. Min Gnt=4.
  Bus 0, device   3, function 0:
    Class 0604: PCI device 8086:3596 (rev 10).
      IRQ 169.
      Master Capable. No bursts. Min Gnt=4.
  Bus 0, device   4, function 0:
    Class 0604: PCI device 8086:3597 (rev 10).
      IRQ 169.
      Master Capable. No bursts. Min Gnt=4.
  Bus 0, device   5, function 0:
    Class 0604: PCI device 8086:3598 (rev 10).
      IRQ 169.
  Bus 0, device   6, function 0:
    Class 0604: PCI device 8086:3599 (rev 10).
      IRQ 169.
      Master Capable. No bursts. Min Gnt=4.
	[negative@zto-serw ~]$ cat /proc/cpuinfo
	processor : 0
	vendor_id : GenuineIntel
	cpu family : 15
	model : 3
	model name : Intel(R) Xeon(TM) CPU 3.20GHz
	stepping : 4
	cpu MHz : 3201.727
	cache size : 1024 KB
	physical id : 0
	siblings : 2
	core id : 0
	cpu cores : 1
	fdiv_bug : no
	hlt_bug : no
	f00f_bug : no
	coma_bug : no
	fpu : yes
	fpu_exception : yes
	cpuid level : 5
	wp : yes
	flags : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca
	cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe lm pni
	monitor ds_cpl cid xtpr
	bogomips : 6324.22

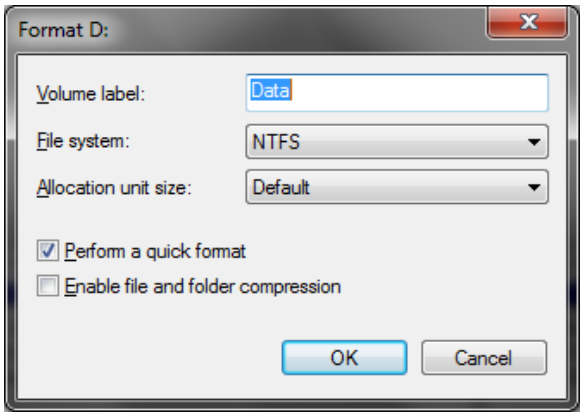
Instalacja nowych urządzeń w systemach Windows jest bardzo prosta i najczęściej wiąże się z uruchomieniem programu instalacyjnego dostarczanego przez producenta sprzętu. W przypadku systemu Linux proces ten jest często bardziej skomplikowany, nierzadko wiążący się z koniecznością samodzielnej kompilacji kodów źródłowych sterownika (szczególnie jeżeli ten nie jest udostępniany przez producenta urządzenia, a jest dziełem entuzjastów) i załadowaniem skompilowanych modułów do jądra systemu. Należy jednak podkreślić że w środowisku pracującym nad rozwojem Linuxa poświęca się tej problematyce wiele pracy i coraz mniej urządzeń wymaga aż tak skomplikowanych działań.

Często wykonywanym przez użytkowników zadaniem administracyjnym jest formatowanie dysków (zwłaszcza przenośnych) oraz usuwanie i tworzenie na nich partycji. W systemach Microsoft Windows wszystkie wymienione operacje dostępne są z poziomu konsoli zarządzania MMC (w dziale Zarządzanie komputerem – Computer Management, w części Magazyn – ***Storage***). Możliwe do wykonania operacje (dla dysków i partycji) są widoczne jako opcje w podręcznych menu pojawiających się po kliknięciu prawym przyciskiem na wybrany dysk.



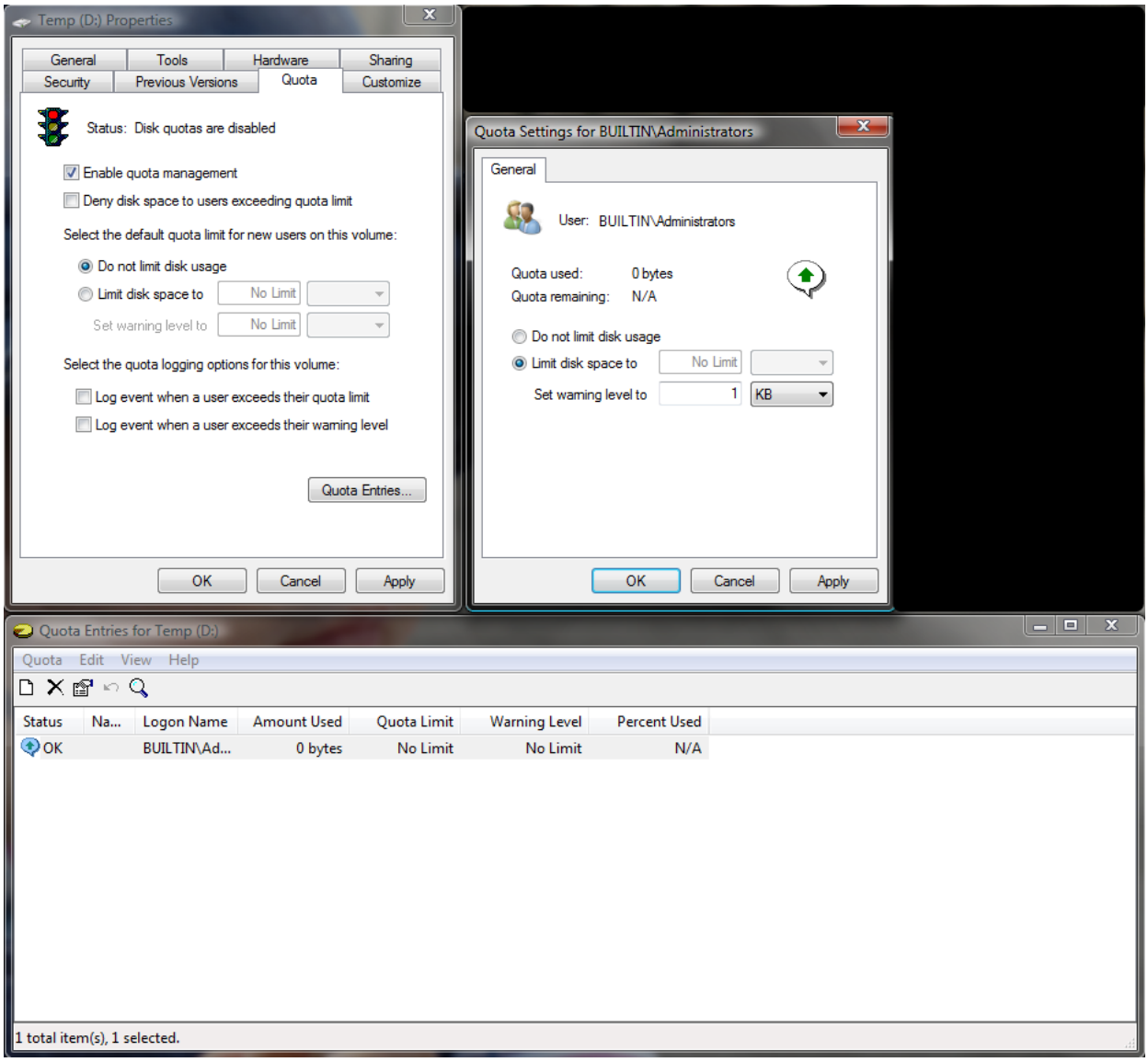
Okno managera dysków Microsoft Windows

Zakładanie partycji oraz jej formatowanie jest tutaj zrealizowane intuicyjnie i polega na zaznaczeniu kilku opcji konfiguracyjnych oraz podaniu wielkości tworzonej partycji, nazwy woluminu oraz typu systemu plików i wielkości jednostki alokacji przy formatowaniu. Wskazywana jest również litera dysku, która ma być przypisana nowej partycji (punkt montowania systemu plików). Proces formatowania wymaga podania nazwy woluminu, określenia typu systemu plików oraz wielkości jednostki alokacji (patrz rysunek poniżej).



Okno parametrów formatowania systemu plików na partycji D:

Z poziomu konsoli MMC można również ustawić przydział miejsca na dysku dla użytkowników. Odbywa się to poprzez zakładkę ***Quota***:



Okno zmiany parametrów przydzielania przestrzeni dyskowej użytkownikom

Operacje na dyskach twardych w systemie Linux można wykonać w trybie tekstowym. Do tworzenia i edycji partycji na dyskach służy program fdisk. Za jego pomocą można wyświetlić listę

istniejących na dysku partycji, usunąć dowolną z nich, utworzyć nową określając jej rozmiar oraz typ systemu plików, oznaczyć partycję jako zawierającą kod uruchomiający system operacyjny (ustawiając tzw. ***boot flag***). Przykład użycia tego programu znajduje się poniżej:

Wyświetlenie listy partycji na dysku SDA (pierwszym dysku podłączonym do kontrolera SCSI/SATA)

```
[root@zto-serw ide]# /sbin/fdisk /dev/sda

The number of cylinders for this disk is set to 14593.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): p

Disk /dev/sda: 120.0 GB, 120034123776 bytes
255 heads, 63 sectors/track, 14593 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1   *           1          32      257008+  83  Linux
/dev/sda2              33        1307    10241437+  83  Linux
/dev/sda3            1308        1568     2096482+  82  Linux swap / Solaris
/dev/sda4            1569       14593   104623312+   5  Extended
/dev/sda5            1569       14593   104623281   83  Linux
```

Po utworzeniu nowej partycji założenie na niej systemu plików odbywa się poprzez wywołanie polecenia ***mkfs.***

```
negative@negative-desktop:/etc$ sudo mkfs /dev/sdb1
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
62259200 inodes, 249035613 blocks
12451780 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=0
7600 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
        4096000, 7962624, 11239424, 20480000, 23887872, 71663616, 78675968,
        102400000, 214990848

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 32 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
negative@negative-desktop:/etc$
```

Po utworzeniu systemu plików na nowej partycji należy go zamontować w ogólnym systemie plików. Wykonuje się to poprzez polecenie mount:

sudo mount /dev/sda3 /mnt/nowy_dysk – powoduje zaczepienie systemu plików nowej partycji w /mnt/nowy dysk głównego systemu plików

Usunięcie systemu plików odbywa się poprzez wywołanie komendy umount:

sudo umount /dev/sda3

lub

sudo umount /mnt/nowy_dysk

Polecenia te montują system plików tymczasowo, do momentu restartu systemu operacyjnego. Aby zapewnić automatyczne montowanie partycji należy dopisać informację o niej do pliku /etc/fstab:

/dev/sda3 /mnt/nowy_dysk ext2 defaults 1 1

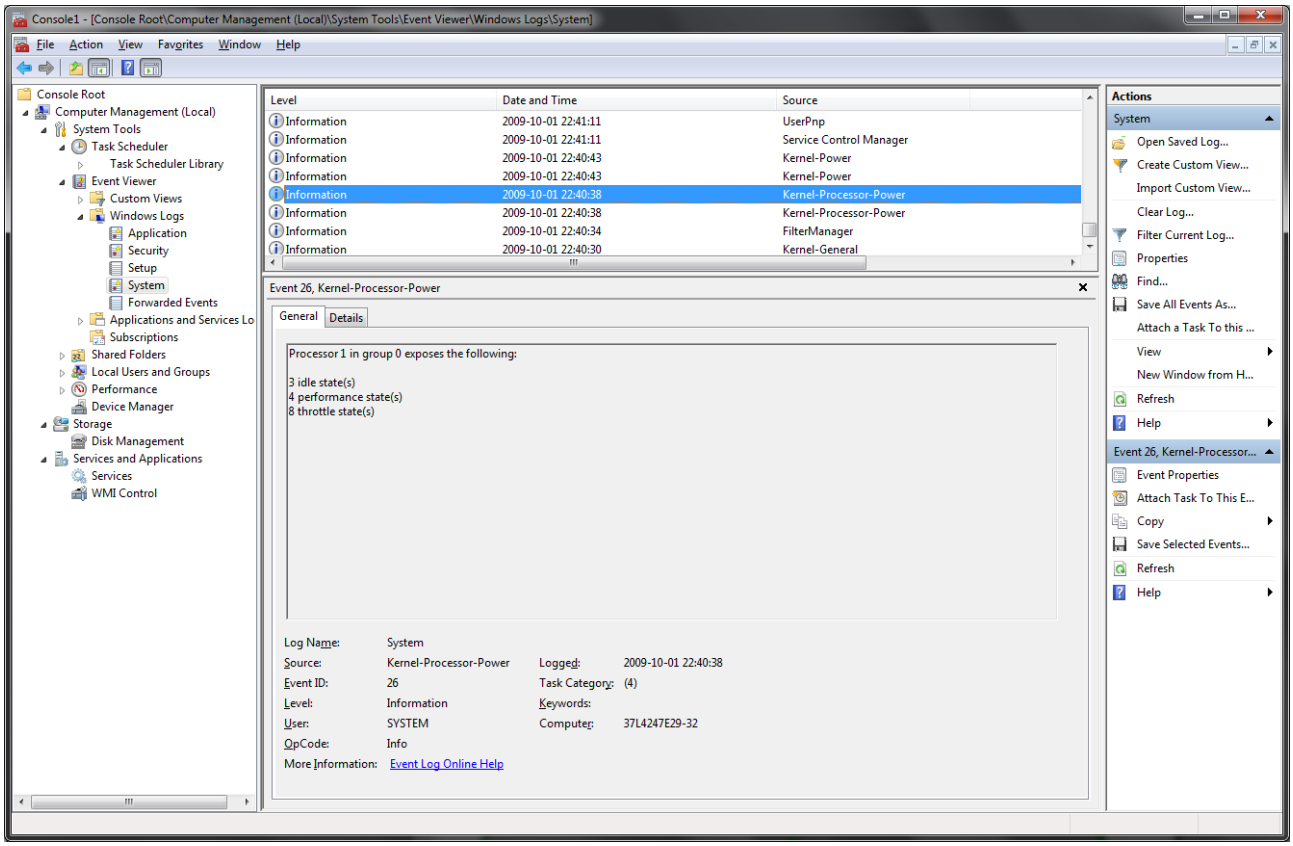
Składnia konfiguracji pliku fstab jest następująca: najpierw podawana jest nazwa urządzenia, którego opis dotyczy (/dev/sda3), potem punkt montowania (/mnt/nowy_dysk), następnie typ systemu plików (ext2) oraz dodatkowe opcje, na przykład rw (tylko do odczytu), user (możliwość montowania i odmontowania systemu plików przez użytkowników nie będących administratorami). Podanie w tym miejscu słowa *defaults* automatycznie określa parametry na (*rw, suid, dev, exec, auto, nouser, and async*). Po opcjach montowania podany jest parametr określający czy dla systemu plików ma być wykonywana kopia zapasowa, a ostatni parametr określa czy, i jeżeli tak, to w jakiej kolejności system ma być sprawdzany pod kątem błędów.

Przydział miejsca na dysku użytkownikom w systemie Linux załącza się poprzez modyfikację plików konfigurujących montowanie partycji w systemie plików. Zmianę wielkości przydziałów wykonuje się wykonuje się poprzez polecenie ***edquota*** według schematu zamieszczonego poniżej:

Włączenie przydziałów na określonej partycji	Edycja przydziałów
Modyfikacja pliku /etc/fstab: /dev/hda2 /limit ext3 defaults,**usrquota,grpquota** 1 1 dopisanie parametrów oznaczonych pogrubieniem powoduje włączenie przydziałów dyskowych na partycji hda2 dla użykowników i grup Kolejnym etapem jest utworzenie w głównym katalogu zamontowanej partycji plików quota.user i quota group oraz ustawienie im możliwości edycji i czytania tylko przez administratora Ostatnim etapem jest dodanie skryptu inicjującego obsługę przydziałów dyskowych przy starcie systemu (/etc/rc.d/rc.local): # Sprawdź i włącz przydziały if [-x /usr/sbin/quotacheck] then echo "Checking quota" /usr/sbin/quotacheck -avug echo "Quota checking done."	sudo edquota -u testuser otwiera edytor z określeniami przydziału: Quotas for user testuser: /dev/hdb1: blocks in use: 12033, limits (soft = 15000, hard = 20000) inodes in use: 163, limits (soft = 4500, hard = 5000) Edytując podane liczby można dokonywać zmian przydziałów dyskowych dla użytkownika W ten sam sposób określane są przydziały dla grup, przy czym polecenie edquota musi zostać uruchomione z przełącznikiem -g: sudo edquota -g testgroup

```
fi
if [ -x /usr/sbin/quotaon ]
then
    echo "Turning quota on"
    /usr/sbin/quotaon -avug
fi
```

Ważną czynnością administracyjną jest przeglądanie dzienników systemowych czyli plików, w których zapisywana jest informacja o zdarzeniach występujących w czasie pracy komputera (najczęściej awariach i błędach). System Microsoft Windows udostępnia dzienniki w konsoli zarządzania MMC, dzieląc je automatycznie na grupy (dziennik aplikacji, systemu, związany z bezpieczeństwem). Przykładowy widok dziennika i zapisanej w nim informacji dotyczącej wykrytego w systemie procesora znajduje się na rysunku poniżej:



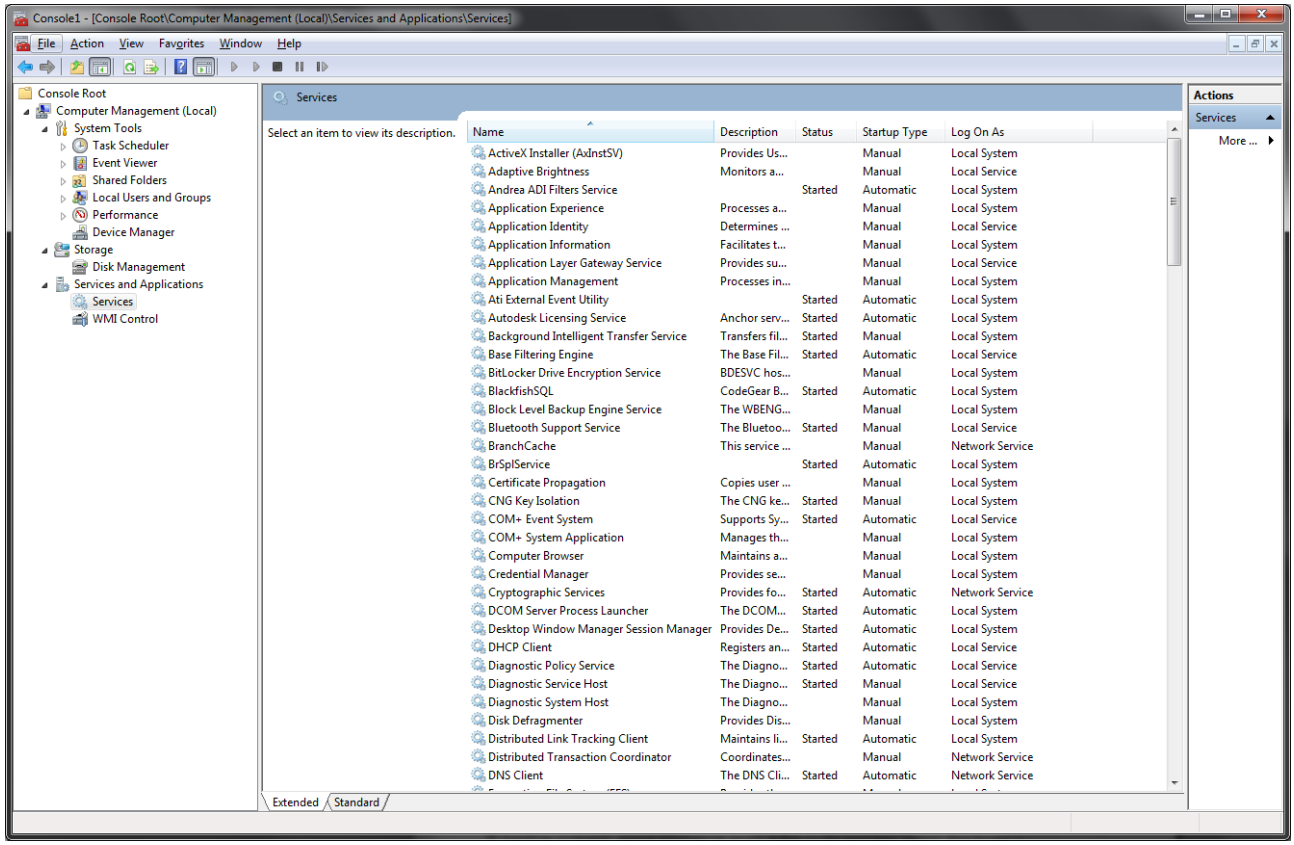
Dziennik systemowy Microsoft Windows

W systemie Linux dzienniki systemowe umieszczone są domyślnie w katalogu /var/log. Podzielone są one również na grupy. Najważniejsze z nich to:

- /var/log/message: ogólne wiadomości systemowe
- /var/log/auth.log: dziennik zawierający informacje o próbach uwierzytelnienia się użytkowników
- /var/log/cron.log: zawiera informacje dotyczące wykonania zadań zaplanowanych

- /var/log/dmesg: zawiera informacje zwracane przez jądro systemu operacyjnego (najczęściej związane z ładowaniem sterowników urządzeń)
- /var/log/maillog: zawiera informacje serwera poczty wewnętrznej
- /var/log/httpd/: zawiera informacje dotyczące serwera http jeżeli taki jest uruchomiony
- /var/log/boot.log : przechowuje informacje zapisane podczas uruchamiania systemu operacyjnego
- /var/log/samba : przechowuje informacje dotyczące współpracy z siecią Microsoft Windows Network

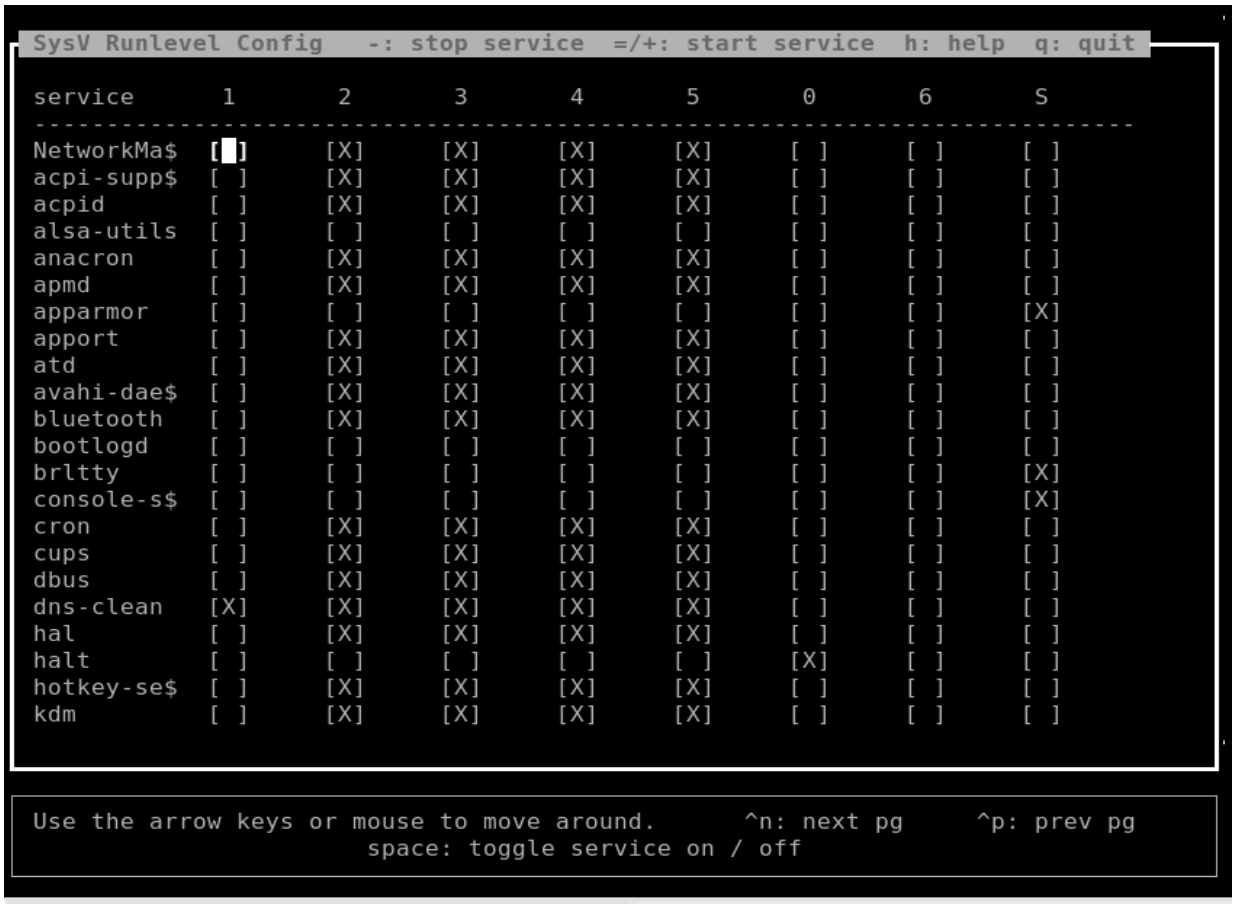
Kolejną czynnością administracyjną jest konfiguracja działających na komputerze usług, czyli programów uruchamianych w tle, nie wymagających współpracy użytkownika. Programy te wykonują różnorodne funkcje (np. serwera sieciowego, wyszukiwania komputerów i zasobów w podsieci, serwerów licencji aplikacji itd.). W systemie Microsoft Windows informacja o dostępnych usługach i ich stanie możliwa jest do uzyskania z poziomu konsoli zarządzania MMC w dziale Usługi i Aplikacje (***Services and Applications***). W oknie tym użytkownik może konfigurować uruchamianie określonych usług wraz z systemem a także specyfikować parametry ich uruchomienia. W systemach Vista i 7 istnieje również możliwość skonfigurowania usług jako uruchamianych z opóźnieniem (po pewnym czasie od włączenia systemu operacyjnego co przyspiesza jego uruchomienie).



Lista usług systemowych Microsoft Windows

W systemie Ubuntu Linux lista uruchomionych usług systemowych dostępna jest dzięki programowi sysv-rc-config. Umożliwia on określenie na których poziomach uruchomienia

(***runlevel***[36]) dana usługa ma być uruchomiona, a na jakich nie:

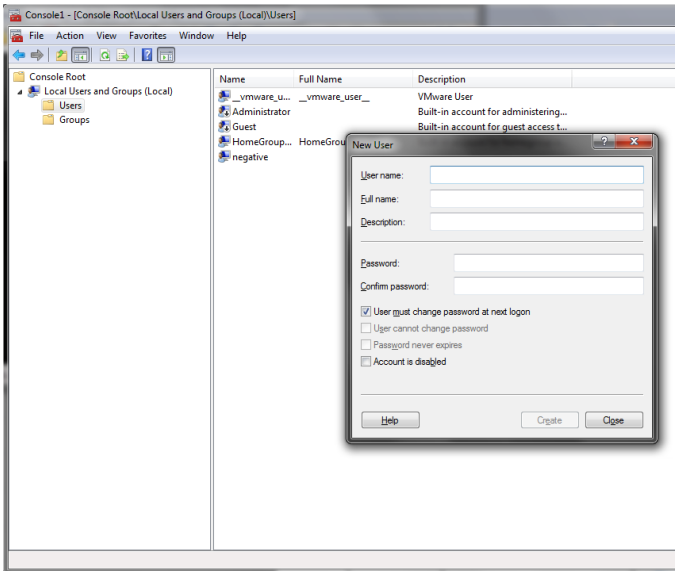
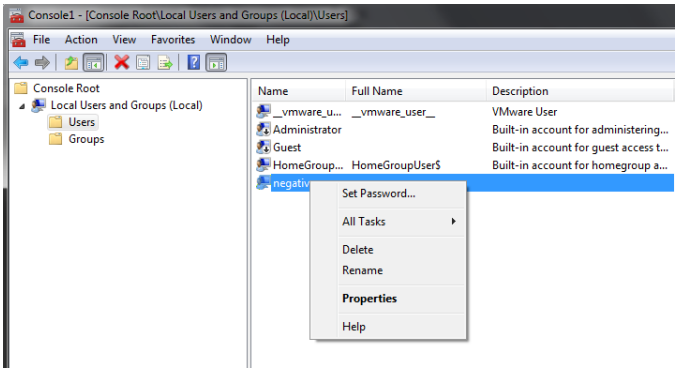


Tworzenie grup roboczych i kont użytkowników, konfiguracja dostępu do plików i katalogów

Na pierwszy rzut oka wydaje się, że tworzenie kont użytkowników ma sens tylko w wypadku komputerów, do których dostęp ma mieć więcej niż jedna osoba. Podejście to jest nieprawidłowe i, niestety, pozostaje aktywne w świadomości użytkowników dzięki systemom firmy Microsoft, a nawet nie tyle samym systemom, a wadliwie napisanym aplikacjom pracującym pod ich kontrolą. W każdym systemie, nawet używanym przez jedną osobę powinny istnieć dwa konta – administratora i użytkownika, przy czym to drugie powinno mieć ograniczone prawa dostępu do różnych zasobów. Pozwala to na zabezpieczenie systemu przed działaniem wirusów oraz innych szkodliwych programów, ponieważ najczęściej dostają się one do systemu poprzez działania użytkownika (np. otworzenie załącznika poczty elektronicznej z wirusem). Jeżeli działanie to jest wykonane przez osobę z ograniczonymi prawami, to zarażona zostanie tylko ta część danych, do których ma on dostęp, a sam system operacyjny pozostanie nietknięty. Jeżeli natomiast użytkownik pracuje na koncie administratora, co jest bardzo częstą sytuacją w systemach Windows, to zainfekowane zostaną przede wszystkim pliki systemu operacyjnego. Podobnie, szkodliwe programy często instalują się jako usługi systemowe co nie jest możliwe do wykonania z poziomu uprawnień zwykłego użytkownika. Warto jednak podkreślić, że Microsoft w swoich systemach proponuje już przy instalacji możliwość stworzenia konta zwykłego użytkownika, jest zatem nadzieja aby stało się to standardem, a nie wyjątkiem.

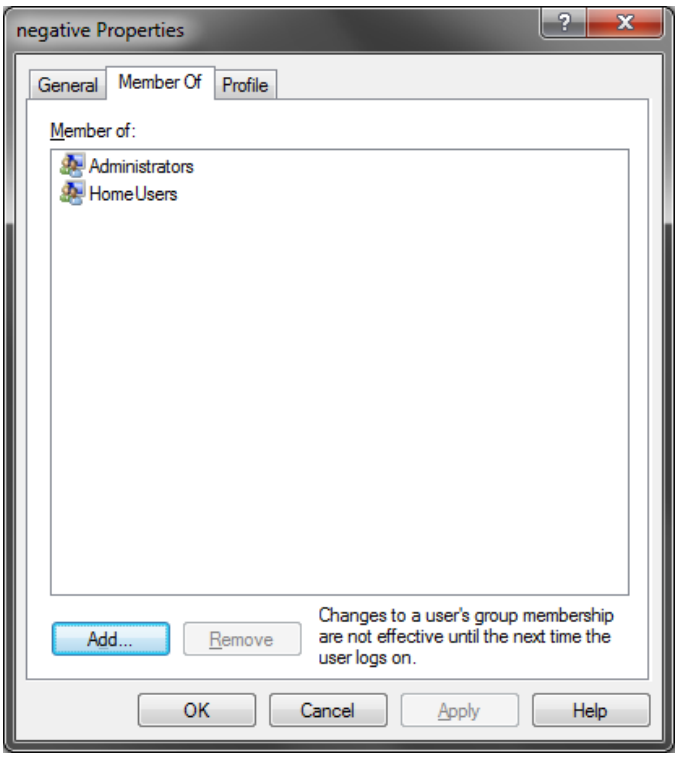
36 *Runlevel* – tryb pracy komputera określony wg implementacji modelu inicjalizacyjnego Unix SystemV. W Linuxie istnieje zwykle 6 trybów pracy: 0 – wyłączenie systemu, 1 – tryb jednego użytkownika, 2 – zastrzeżony, 3 – tryb tekstowy dla wielu użytkowników, 4 – zastrzeżony, 5 – tryb graficzny dla wielu użytkowników, 6 – restart systemu

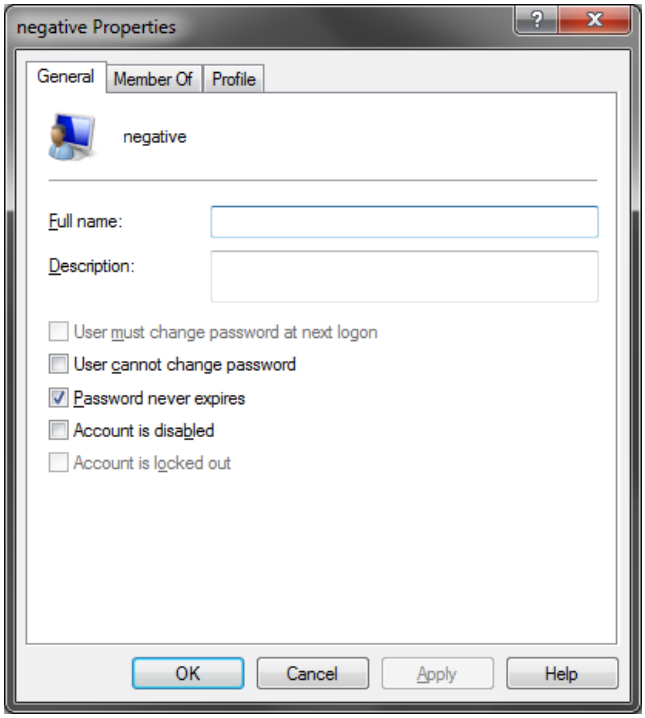
W procesie instalacji współczesnych systemów operacyjnych zwykle wymagane jest podanie identyfikatora użytkownika i hasła, a także często hasła administratora. Automatycznie tworzone wtedy są dwa konta użytkowników. Oczywiście konta można dodawać w dowolnej chwili już po zainstalowaniu systemu operacyjnego. Służą do tego narzędzia administracyjne, w Linuxie najczęściej obsługiwane z poziomu terminala (czyli bez graficznego interfejsu użytkownika). W systemach Microsoft służą do tego odpowiednie przystawki konsoli zarządzania systemem (***Microsoft Management Console – MMC***) aczkolwiek istnieją też narzędzia tekstowe, takie jak na przykład AccessChk. W poniższej tabeli umieszczono przykłady najczęściej wykonywanych operacji administracyjnych związanych z kontami użytkownika:

Operacja	Linux (Ubuntu 8.04 LTS)[37]	Microsoft Windows 7 Professional[38]
Dodanie konta użytkownika	sudo /usr/sbin/useradd testuser	 W konsoli MMC należy rozwinąć drzewko Local Users and Groups (Lokalni użytkownicy i grupy), wybrać folder Users (Użytkownicy), z menu podręcznego wybrać New User (Nowy użytkownik)
Utworzenie hasła użytkownika	sudo passwd testuser Enter new UNIX password: Retype new UNIX password: passwd: password updated successfully	 Wybrać użytkownika z listy i z menu podręcznego wybrać polecenie Set Password (Ustaw hasło)
Zmiana hasła użytkownika	1. Zmiana swojego hasła: passwd Changing password for	1. Zmiana swojego hasła Po wciśnięciu Ctrl+Alt+Delete należy wybrać opcję Change Password (Zmień hasło) i podać kolejno

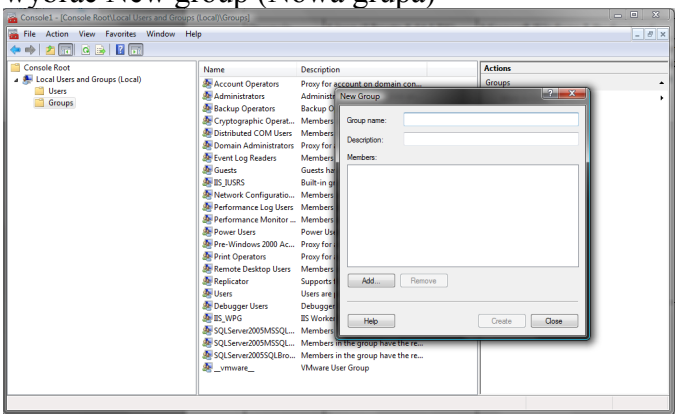
37 Wszystkie komendy administracyjne w systemie Ubuntu muszą być poprzedzone poleceniem sudo, które powoduje wywołanie ich z poziomem uprzywilejowania administratora. Polecenie sudo wymaga podania hasła dostępu

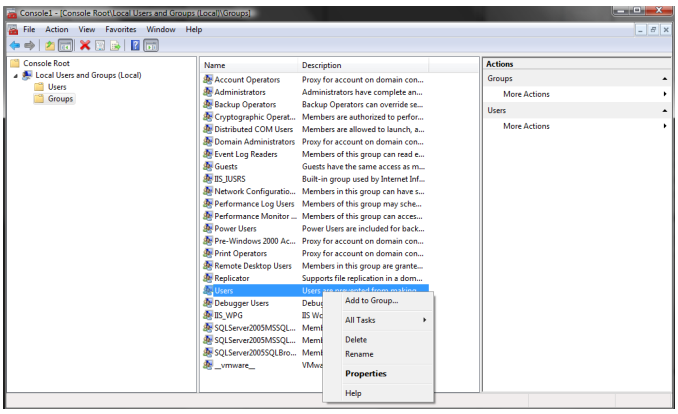
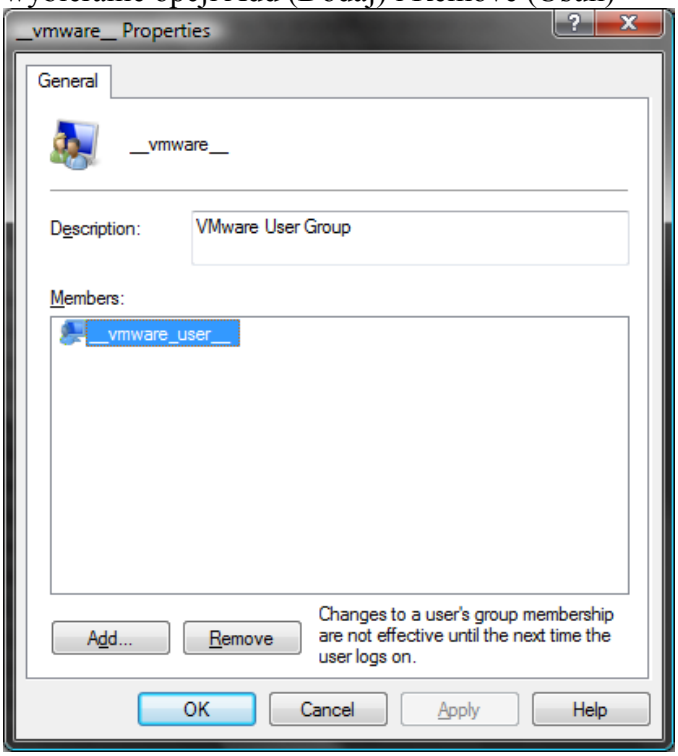
38 Demonstrowane operacje mogą być wykonane tylko przez użytkownika mającego uprawnienia administratora (z wyjątkiem zmiany hasła bieżącego użytkownika)

	negative. (current) UNIX password: Enter new UNIX password: Retype new UNIX password: passwd: password updated successfully 2. Zmiana hasła innego użytkownika – tak jak tworzenie nowego	obecne i dwukrotnie nowe hasło. 2. Zmiana hasła innego użytkownika – tak jak tworzenie nowego
Zmiana przynależności do grupy	sudo gpasswd -a testuser testgroup -dodanie użytkownika do grupy testgroup sudo gpasswd -d testuser testgroup -usunięcie użytkownika do grupy testgroup	 W konsoli MMC należy wskazać użytkownika, wybrać z menu podręcznego polecenie Properties (Właściwości). Ukazane zostanie okno przynależności użytkownika do grup. Przyciski Add i Remove służą do zmiany tych przynależności.
Usunięcie konta użytkownika	sudo /usr/sbin/userdel testuser	Wybrać użytkownika z listy i z menu podręcznego wybrać polecenie Delete (Usuń).
Blokowanie i odblokowanie konta użytkownika	sudo passwd testuser -l sudo passwd testuser -u	Zaznaczenie lub odznaczenie flagi przy opcji „Account is disabled” (Konto zablokowane):

Poniższa tabela ukazuje podstawowe operacje wykonywane na grupach użytkowników w systemie operacyjnym:

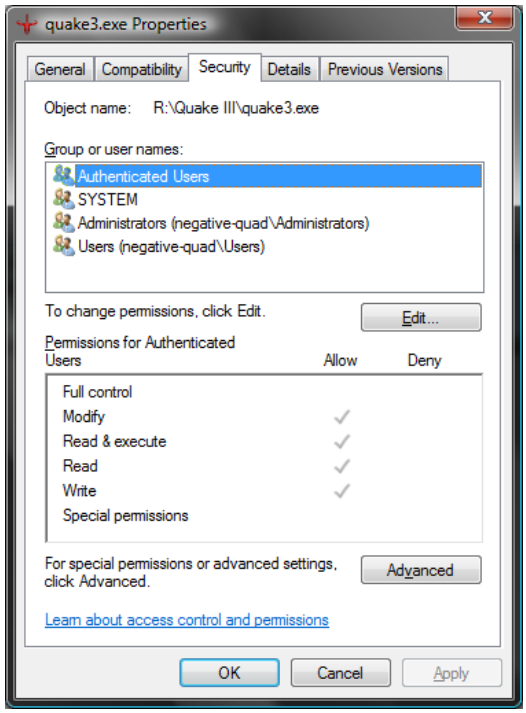
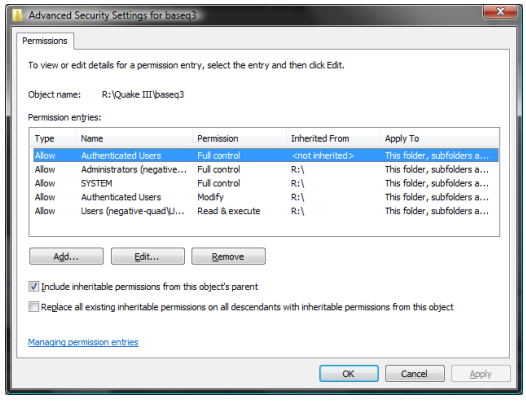
Operacja	Linux (Ubuntu 8.04 LTS)	Microsoft Windows 7 Professional
Dodanie grupy	sudo groupadd testgroup - utworzenie grupy testgroup	W konsoli MMC należy rozwinąć drzewko Local Users and Groups (Lokalni użytkownicy i grupy), wybrać folder Groups (Grupy) z menu podręcznego wybrać New group (Nowa grupa) 
Usunięcie grupy	sudo groupdel testgroup - usunięcie grupy testgroup	W konsoli MMC należy rozwinąć drzewko Local Users and Groups (Lokalni użytkownicy i grupy), wybrać folder Groups (Grupy) z menu podręcznego wybrać Delete (Usuń)

		
Dodanie i usunięcie użytkowników do grupy	sudo gpasswd -a testuser testgroup -dodanie użytkownika do grupy testgroup sudo gpasswd -d testuser testgroup -usunięcie użytkownika do grupy testgroup sudo gpasswd -M testuser1, testuser2, testuser3 testgroup -określenie listy użytkowników (testuser2, testuser2, testuser3) należących do grupy testgroup	W konsoli MMC należy rozwinąć drzewko Local Users and Groups (Lokalni użytkownicy i grupy), wybrać folder Groups (Grupy), wskazać grupę do edycji i wybrać Add to group (Dodaj do grupy). W wyświetlonym oknie dodawać i usuwać użytkowników ze wskazanej grupy można poprzez wybieranie opcji Add (Dodaj) i Remove (Usuń) 

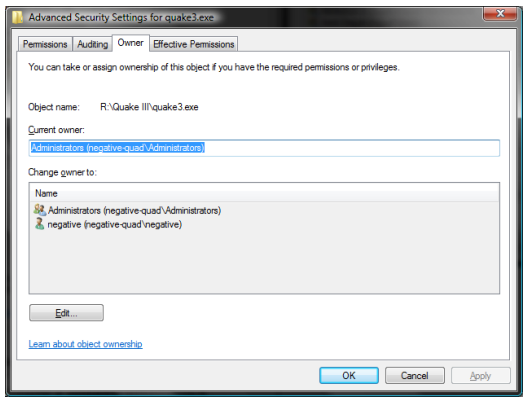
W systemach operacyjnych zwykle istnieje kilka zdefiniowanych grup, które określają charakterystyczne uprawnienia ich członków do pewnych operacji w ramach systemu. I tak, grupa użytkowników administracyjnych ma dostęp do praktycznie wszystkich opcji systemu. Nieco ograniczone uprawnienia przewyższające jednak te, które domyślnie posiadają zwykli użytkownicy ma grupa Power Users. Są też specjalne grupy mające np. dostęp tylko do dzienników systemowych lub mogące wykonywać tylko operacje backupu. Różne procesy mogą być wykonywane jako przynależne do danych grup co pozwala w łatwy sposób ograniczyć ich możliwości dostępu do chronionych danych.

Szczególnie często wykorzystywane jest ograniczenie dostępu do plików znajdujących się

na dysku komputera użytkownikom, którzy nie powinni znać lub modyfikować ich zawartości. W poniższej tabeli przedstawiono sposoby ustawiania najczęściej spotykanych ograniczeń.

Operacja	Linux (Ubuntu 8.04 LTS)	Microsoft Windows 7 Professional
Ustawienie praw odczytu, zapisu i wykonania dla plików / katalogów	chmod sabc nazwa_pliku s – liczba określająca prawa specjalne a – liczba określająca prawa dostępu dla właściciela pliku b – liczba określająca prawa dostępu dla grupy, do której należy plik c – liczba określająca prawa dostępu dla wszystkich użytkowników Pominięcie któregoś z argumentów powoduje że chmod traktuje go jako 0. Liczby te obliczane są wg następującego schematu: a = x + y + z, gdzie: x = 0 jeżeli brak prawa odczytu, x = 4 jeżeli jest prawo odczytu; y = 0 jeżeli brak prawa zapisu, y = 2 jeżeli jest prawo zapisu; z = 0 jeżeli brak prawa wykonania z = 1 jeżeli jest prawo wykonania przykładowo: chmod 721 test oznacza że plik test może być zapisany, odczytany i uruchomiony przez właściciela, tylko zapisany przez członków grupy do której jest przypisany oraz tylko wykonany przez innych użytkowników Inną notacją jest określenie praw dostępu dla pliku symbolicznie, według schematu: chmod [ugoa][+-=][rwxXstugo] nazwa pliku gdzie: u – określa prawa dla właściciela g – określa prawa dla grupy o – określa prawa dla użytkowników spoza grupy a – określa prawa dla wszystkich użytkowników	W programie wyświetlającym zawartość katalogu (np. Windows Explorer) należy zażądać wyświetlenia okna z określeniem Właściwości pliku (***Properties***). W oknie prezentującym te właściwości należy wybrać kartę Zabezpieczenia (***Security***). Wszystkie prawa określane są poprzez zaznaczenie lub odznaczenie odpowiedniej opcji na liście:  Zaawansowane prawa można ustawiać poprzez wyświetlenie dodatkowego okna po naciśnięciu przycisku Zaawansowane (***Advanced***): 

	+ - oznacza że określone prawa zostaną dodane do istniejących - - oznacza że określone prawa zostaną usunięte z zestawu istniejących = - oznacza że określone prawa nadpiszą istniejący zestaw r – prawo odczytu w – prawo zapisu x – prawo wykonania X – prawo wykonania tylko jeżeli plik jest katalogiem lub posiada prawo wykonania dla innego użytkownika s – zmiana identyfikatora użytkownika przy wykonaniu pliku t - zachowanie tekstu programu na urządzeniu wymiany u - prawa które posiada właściciel pliku g - prawa innego użytkownika z grupy posiadającej plik o - prawa innych użytkowników Parametry Xstugo odpowiadają parametrom specjalny określanym jako pierwsze pole w pierwszej zaprezentowanej notacji opcji podawania listy argumentów polecenia chmod W przypadku kiedy określane są prawa dostępu do katalogów prawo odczytu rozumiane jest jako możliwość wejścia do katalogu, zapisu – dodania do niego plików lub katalogów, wykonania – dostęp do plików znajdujących się w katalogu	
Zmiana użytkownika pliku / katalogu	chown nazwa_użytkownika:nazwa_grupy nazwa_pliku zmienia przynależność podanego pliku na określonego użytkownika i grupę.	W karcie Zabezpieczenia (patrz powyżej) należy wybrać przycisk Zaawansowane (***Advanced***) i w nowym oknie – zakładkę Właściciel (***Owner***). Zmiana użytkownika następuje poprzez wskazanie nowego właściciela i zatwierdzenie decyzji przyciskiem OK.



Zarówno zmiany praw dostępu jak i właścicieli katalogów mogą być wykonane tylko dla wskazanego katalogu lub dla niego i całej jego zawartości. W systemie Linux wymaga to dodania przełącznika -R do wyżej wymienionych poleceń. W systemie Windows należy zaznaczyć opcję *Zamień wpisy uprawnień na wszystkich obiektach podrzędnych na wpisy tutaj pokazane, stosowane do obiektów podrzędnych* w karcie uprawnień okna Zaawansowane.

Konfiguracja sieci

W dzisiejszych systemach komputerowych rzadko występują komputery osobiste lub serwery, które nie są podłączone do żadnej sieci, czy to przewodowej, czy bezprzewodowej. Najczęściej sieć ta oparta jest o protokół IP (w wersji 4), w dawniejszych zastosowaniach były to również sieci IPX, TokenRing itp[39]. W bieżącym rozdziale przedstawione zostaną podstawowe metody konfiguracji sieci IP w Linuxie oraz Microsoft Windows.

Adres IP, maska podsieci i brama

Adres IP jest swojego rodzaju identyfikatorem, który skojarzony jest z danym komputerem dzięki któremu inne urządzenia w sieci mogą się z nim komunikować. Maska podsieci określa rozległość sieci czyli liczbę unikalnych adresów (warunkującą liczbę możliwych urządzeń sieciowych) identyfikowalnych w sieci. Brama jest adresem IP komputera (lub innego urządzenia sieciowego), który jest punktem styku sieci bieżącej z innymi sieciami (np. Internetem). W sieciach lokalnych małego zasięgu (np. domowych) popularną klasą adresową[40] jest klasa C o adresach typu 192.168.x.x[41] z maską sieciową 255.255.255.0. W poniższych przykładach założono że sieć do której będzie podłączony komputer jest siecią 192.168.1.0. Konfiguracja powyższych parametrów dotyczy zarówno sieci przewodowej jak i bezprzewodowej.

W systemie Linux, konfiguracja sieci może odbywać się poprzez wywołanie z poziomu powłoki systemowej polecenia ifconfig wraz z podaniem parametrów takich, jak numer karty sieciowej która jest konfigurowana, adresu IP który ma jej zostać nadany, maski i adresu rozgłaszania (***broadcast address***)[42] i bramy:

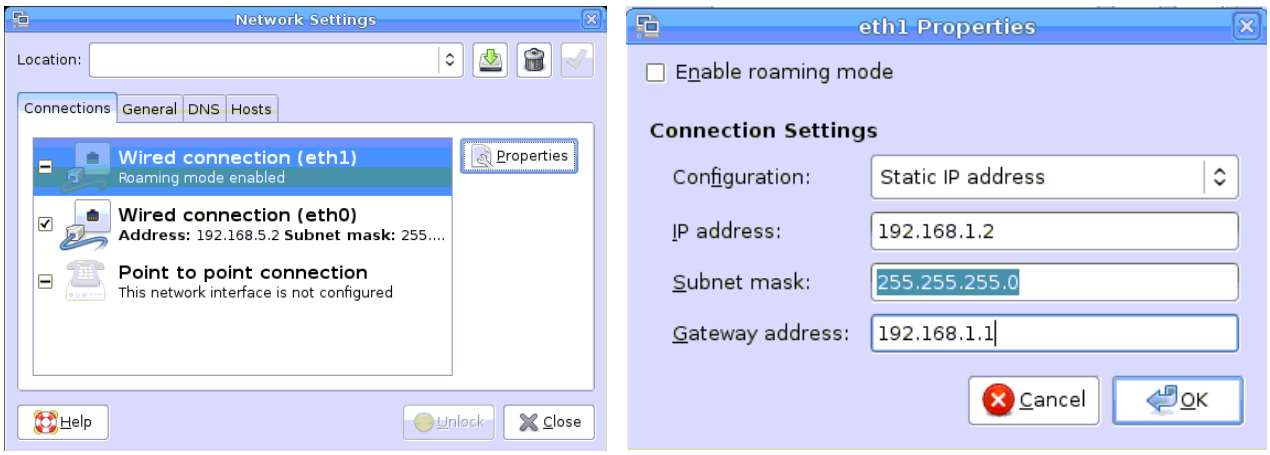
```
sudo  /sbin/ifconfig eth0 192.168.1.2 netmask 255.255.255.0 broadcast 192.168.1.255 gateway 192.168.1.1
sudo ifconfig eth0 up
```

39 Więcej informacji nt rodzajów protokołów sieciowych można znaleźć np. w [CASTELLI]
40 Więcej informacji nt rodzajów protokołów sieciowych można znaleźć np. w [BENVENUTI]
41 x może być dowolną liczbą całkowitą z przedziału 0 - 255
42 Wysłanie danych na adres rozgłaszania skutkuje automatycznym rozesłaniem go do wszystkich adresów IP podsieci

Należy pamiętać, że ustawienia wprowadzone przez ifconfig nie są zapisywane w systemie i obowiązują do momentu ponownego jego uruchomienia. Aby te parametry były wpisywane w momencie uruchamiania można wywołanie ifconfig umieścić w pliku określającym programy uruchamiane na starcie (/etc/rc.d/rc.local). Innym sposobem jest edycja plików przechowujących stałe ustawienia sieciowe (w omawianym systemie Ubuntu 8.04 - /etc/network/interfaces):

```
auto eth0 // automatyczne uruchamianie karty eth0
iface eth0 inet static // statycznie podany adres IP
address 192.168.1.2 // adres
netmask 255.255.255.0 // maska sieciowa
broadcast 192.168.1.1 // adres rozgłaszania
network 192.168.1.0 // początek podsieci
gateway 192.168.1.1 // brama
```

Istnieją również programy narzędziowe do konfiguracji połączeń, zarówno pracujące w trybie tekstowym jak i graficznym. Poniżej przedstawiono wprowadzanie danych do programu *The Network Administrator Tool*:



Narzędzie GUI do konfiguracji sieci w Ubuntu Linux

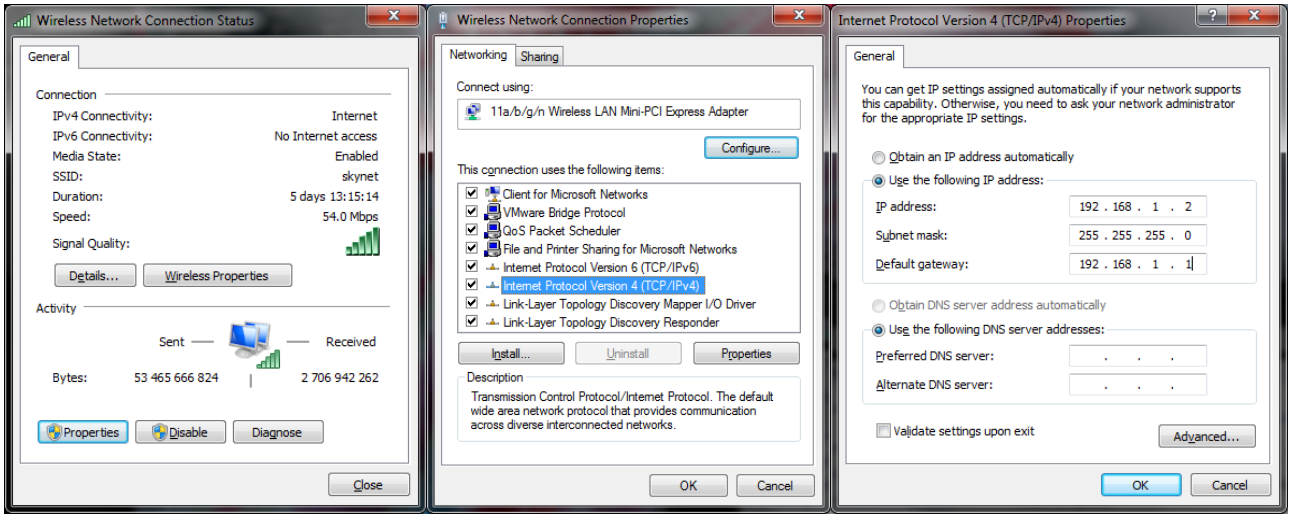
W systemie Microsoft Windows parametry sieciowe można również skonfigurować na dwa sposoby – z poziomu konsoli oraz graficznie. Tryb tekstowy wykorzystuje program narzędziowy netsh.exe, będący składnikiem systemu:

netsh interface ip set address name="Local Area Connection" static 192.168.1.2 255.255.255.0 192.168.1.1 1

w tym przykładzie karcie o nazwie „Local Area Connection” nadawany jest adres IP 192.168.1.2, z maską podsieci 255.255.255.0 i bramą domyślną 192.168.1.1. Metryka bramki[43] tego interfejsu została ustawiona na 1.

Oczywiście parametry te można również ustawić z poziomu interfejsu graficznego:

43 Metryka bramki określa jej priorytet, bardziej prawdopodobne jest wybranie przez system tej o niższym numerze



Okna konfiguracji sieci w Microsoft Windows 7

W przypadku kiedy konfigurowany komputer ma więcej niż jedną kartę sieciową, należy pamiętać o tym, że adres bramy domyślnej musi być podany tylko dla tego interfejsu, który należy do sieci, która ma styczność z innymi sieciami. Innymi słowy, jeżeli jedna z kart sieciowych jest podłączona do routera, który ma łączność z siecią Internet, a druga karta służy do komunikacji z komputerami w sieci lokalnej (nie podłączonymi do routera), to adres bramy powinien być zdefiniowany tylko dla pierwszej karty sieciowej. Podanie go dla karty obsługującej sieć lokalną spowoduje niepoprawne działanie usług sieciowych.

Adresy serwerów nazw

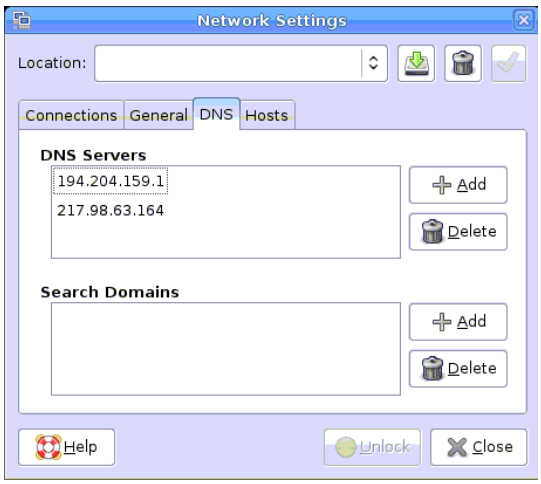
W przypadku bardziej rozbudowanej sieci, często występuje w niej urządzenie oferujące usługę rozwiązywania nazw domen (***Domain Name Server - DNS***). Ogólnie rzecz ujmując usługa ta działa jak baza danych wiążąca adresy IP sieci z nazwami (np. www.mchtr.pw.edu.pl). Kiedy użytkownik wpisuje w okno przeglądarki adres http, zwykle w postaci tekstowej wysyłane jest zapytanie do serwera nazw, który odsyła adres IP komputera, na którym uruchomiony jest serwer www obsługujący stronę www.mchtr.pw.edu.pl. Bez usługi rozwiązywania nazw zapamiętywanie adresów interesujących stron byłoby zdecydowanie trudniejsze (zamiast wspomnianego adresu tekstowego należałoby wpisywać 194.29.140.8).

W systemie linux adresy serwerów DNS podane są w pliku tekstowym /etc/resolv.conf. Każdy adres umieszczany jest w oddzielnej linii (można w ten sposób zdefiniować serwery redundantne):

nameserver 194.204.159.1
nameserver 194.204.152.34

Znajdująca się powyżej zawartość przykładowego pliku /etc/resolv.conf definiuje adresy dwóch serwerów nazw (podane adresy są rzeczywistymi serwerami DNS dla sieci Telekomunikacji Polskiej S.A dla usługi Neostrada).

Graficzne narzędzie do konfiguracji sieci w Ubuntu również pozwala na konfigurację serwerów DNS:

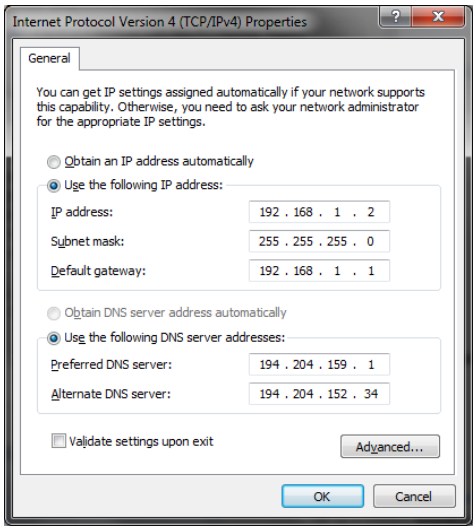


Konfiguracja adresów serwerów DNS w narzędziu graficznym Ubuntu Linux

W systemie Microsoft Windows konfiguracja serwerów DNS może być przeprowadzona przy użyciu polecenia netsh (poniższy przykład przedstawia wprowadzenie adresu pierwszego serwera DNS):

netsh interface ip set dns "Local Area Connection" static 194.204.159.1

W trybie graficznym, informacja o adresach serwerów nazw wprowadzana jest w tej samej zakładce, co adres IP, maska i brama:



Konfiguracja adresów serwerów DHCP w Microsoft Windows 7

Często występuje sytuacja że w sieci do której podłączony jest komputer istnieje urządzenie udostępniające usługę automatycznej konfiguracji podstawowych parametrów protokołu IP (serwer DHCP). W tym wypadku należy zrezygnować z ręcznego ustawiania tych parametrów i załączyć automatyczną konfigurację. Można to wykonać w systemie Linux z wykorzystaniem konsoli:

sudo ifconfig eth0 up
sudo dhclient eth0

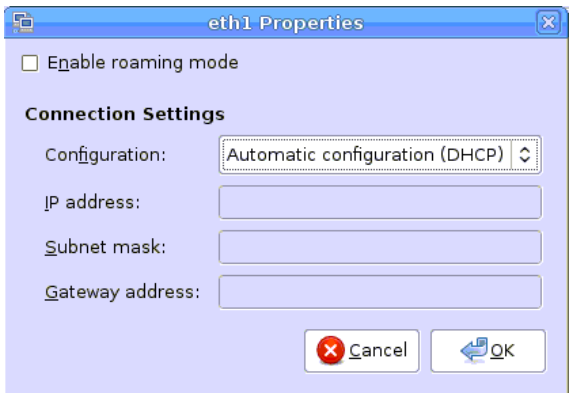
Podany przykład uruchamia kartę sieciową eth0 i startuje dla niej program *dhclient*, którego rolą jest kontakt z serwerem DHCP i skonfigurowanie wskazanej karty (eth0) według parametrów, które zwróci serwer. Podobnie jak konfiguracja statycznego adresu IP przy użyciu *ifconfig*, konfiguracja karty przez *dhclient* jest nietrwała (obowiązująca tylko do momentu ponownego uruchomienia systemu). Aby automatycznie (przy starcie systemu) karta uzyskiwała adres z serwera DHCP należy

przeprowadzić edycję pliku */etc/network/interfaces* umieszczając w nim linię:

iface eth0 inet dhcp

która wskazuje że automatycznie uruchamiana przy starcie systemu karta eth0 ma adres konfigurowany przez serwer DHCP.

Ustawienie automatycznego pobierania parametrów sieci można również wykonać przy użyciu *The Network Administrator Tool:*

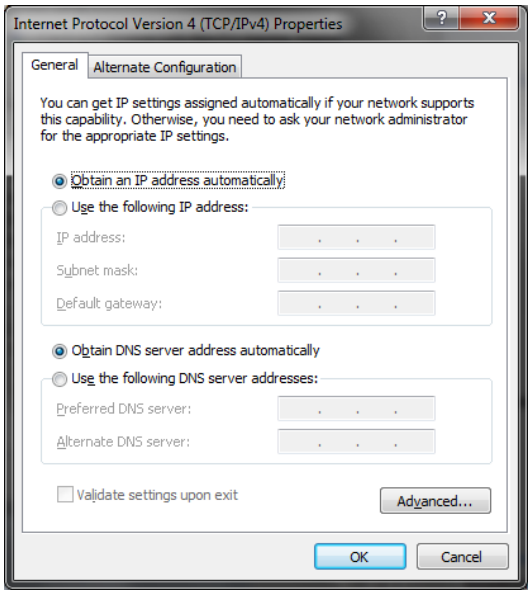


Ustawienie automatycznej konfiguracji parametrów połączenia sieciowego w Ubuntu Linux

W systemie Microsoft Windows automatyczne pobieranie adresu dla karty sieciowej można wywołać z użyciem polecenia netsh:

netsh interface ip set dns "Local Area Connection" dhcp

Z poziomu interfejsu graficznego ustawianie trybu automatycznego odbywa się poprzez wybranie odpowiednich opcji w oknie, w którym można dokonać ręcznego wprowadzenia parametrów karty:



Ustawienie automatycznej konfiguracji parametrów połączenia sieciowego w Microsoft Windows 7

Najczęściej serwer DHCP oprócz adresu IP, maski i bramy połączenia zwraca również informację o dostępnych w sieci serwerach DNS, tak więc przy automatycznej konfiguracji parametrów połączenia nie zachodzi potrzeba ręcznego podawania adresów tych serwerów.

Administracja zdalna

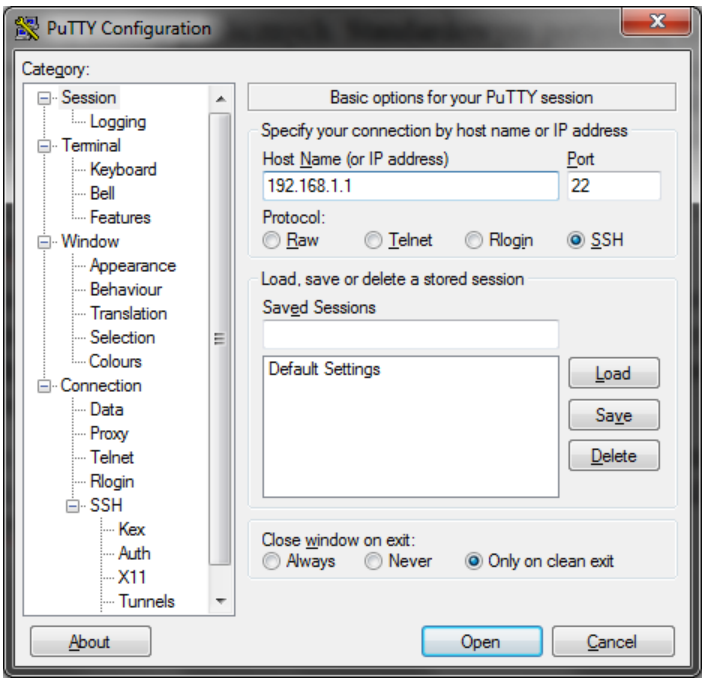
Czasem, szczególnie w wypadku komputerów pracujących jako serwery, ale również np. maszyn obliczeniowych czy po prostu umieszczonych w lokacjach, które nie są łatwo dostępne zachodzi potrzeba zdalnego dostępu administracyjnego. Ogólnie rzecz biorąc, dostęp zdalny polega na uzyskaniu możliwości sterowania pracą danego komputera za pomocą innego komputera. Istnieje wiele metod takiego zdalnego dostępu, wybór optymalnej zależy od wielu czynników (np. czynności jakie użytkownik chce wykonywać na zdalnym systemie, przepustowości sieci łączącej komputer zdalny z administracyjnym, wymaganego poziomu bezpieczeństwa i ochrony danych transmitowanych itd.). Generalnie można jednak podzielić metody dostępu w taki sam sposób jak powłokę systemową – na metody tekstowe i graficzne. W ramach tego rozdziału omówione zostaną dwie metody administracji zdalnej z interfejsem tekstowym oraz dwie metody z interfejsem graficznym.

Interfejsy tekstowe

Podstawowym sposobem administracji zdalnej, w zasadzie od momentu pojawienia się sieci komputerowych jest protokół telnet (***teletype network***), służący do zapewnienia zdalnego dostępu do interfejsu tekstowego na zarządzanym komputerze. Protokół ten jest od dawna ustandaryzowany i w zasadzie każda platforma systemowa ze stosem TCP/IP posiada jego implementację. Domyślnym portem telnetu jest port 23 TCP. Niestety, z racji swojej prostoty protokół ten nie posiada szyfrowania komunikatów ani żadnej innej metody ochrony przekazywanych danych (także haseł), które mogą być przechwycone przy pomocy dowolnego oprogramowania zapisującego pakiety TCP wędrujące przez sieć. Z tego powodu protokół ten został wyparty przez protokół ssh (***Secure Shell***). Jak wskazuje jego nazwa, odróżnia się od telnetu przede wszystkim zabezpieczeniem danych transmitowanych przez sieć przy użyciu szyfrowania wiadomości i użyciu algorytmów sprawdzania tożsamości klienta i serwera. Sprawdzanie tożsamości jest oparte o model kryptograficzny z wymianą kluczy publicznych. Standardowym portem tej usługi jest port 22 TCP. Przy użyciu *ssh* można, oprócz uzyskania dostępu do linii poleceń serwera, uzyskać również do wymiany plików (tzw. SFTP), *forwardowania* portów komputera zdalnego do klienta, przekazywania interfejsu graficznego X11. SSH stanowi też podstawę transportową w programowej implementacji tzw. wirtualnych sieci prywatnych (***Virtual private network – VPN***). Większość systemów operacyjnych oferuje obsługę serwera i klienta ssh (Unix, Linux, Mac OS X, Solaris, FreeBSD, także OpenSSH dla Microsoft Windows).

Oba rozwiązania (telnet i ssh) umożliwiają dostęp zdalny do powłoki systemowej (w wersji tekstowej) systemu operacyjnego. Pomijając pewne zjawiska wynikające z transmisji danych przez sieć komputerową, administracja zdalnego systemu odbywa się w ten sam sposób jakby użytkownik miał do niego bezpośredni dostęp (klawiaturę i monitor).

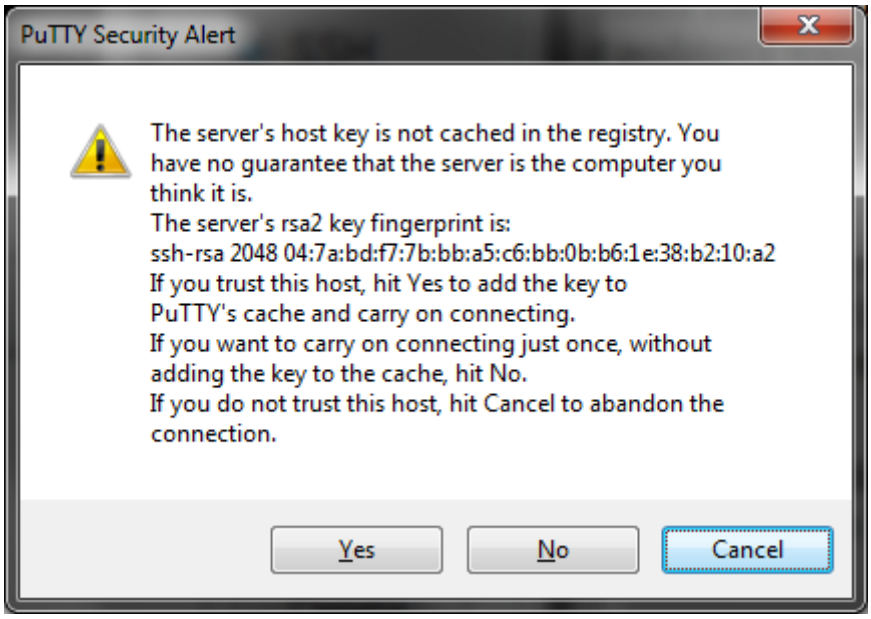
Przykładowe operacje związane z zestawianiem połączenia przez ssh przedstawione są na poniższych rysunkach (połączenie zestawiane poprzez najpopularniejszego klienta ssh dla systemu Microsoft Windows, Putty):



Główne okno programu komunikacyjnego putty

Po uruchomieniu program wyświetla główne okno konfiguracji, w którym można podać adres serwera oraz wybrać protokół połączenia (telnet, ssh, rlogin, raw). Główny ekran aplikacji i pole wprowadzania adresu IP oraz portu serwera ssh.

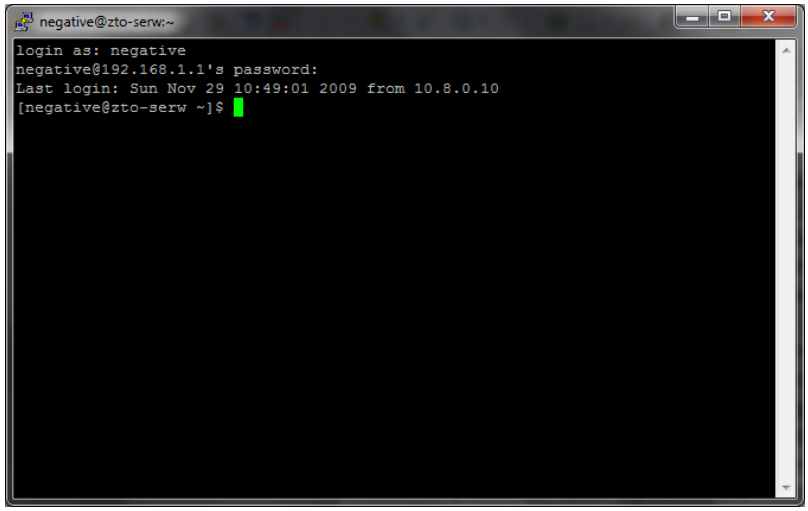
Po wprowadzeniu danych i naciśnięciu przycisku *Open* następuje nawiązanie połączenia z serwerem i wymiana kluczy publicznych między serwerem i klientem. W przypadku, kiedy połączenie nawiązywane jest po raz pierwszy system wyświetla ostrzeżenie o nieznanym kluczu serwera i podaje jego „odcisk", który użytkownik powinien porównać z odciskiem przekazanym mu przez administratora systemu zdalnego. Jeżeli odciski są takie same można zaakceptować dodanie klucza do zbioru.



Ostrzeżenie o nowym kluczu identyfikującym serwer

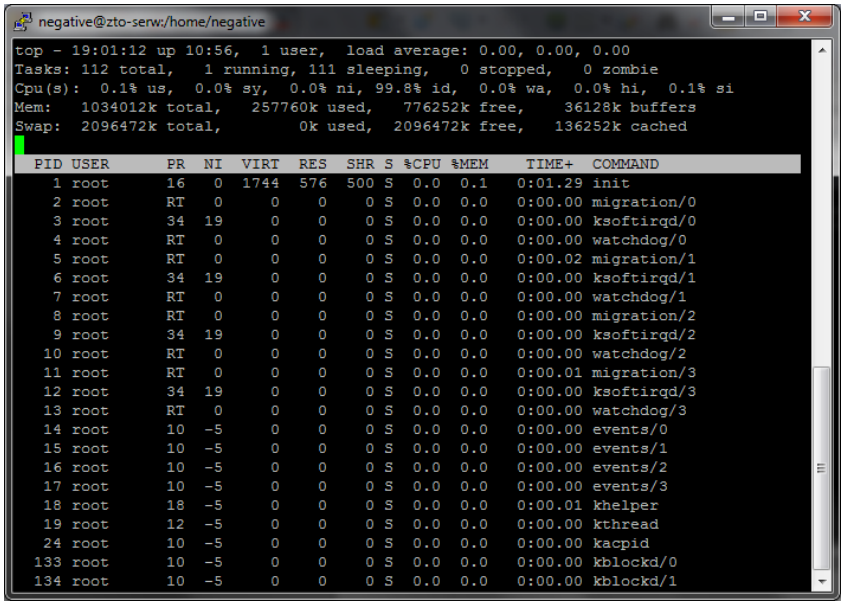
Po wymianie kluczy wyświetlane jest okno logowania, w dokładnie taki sam sposób jak ma to miejsce w przypadku logowania do systemu na miejscu. System żąda podania nazwy użytkownika i

hasła (użytkownik może również identyfikować się swoim kluczem, jednak wymaga to nieco innej konfiguracji serwera i klienta). Po poprawnym zalogowaniu wyświetlany jest znak zachęty i powłoka systemu zdalnego gotowa jest do pracy:



Logowanie do zdalnego systemu poprzez ssh

Mając dostęp do powłoki można użytkownik może administrować komputerem zdalnym tak jakby znajdował się bezpośrednio przy nim:

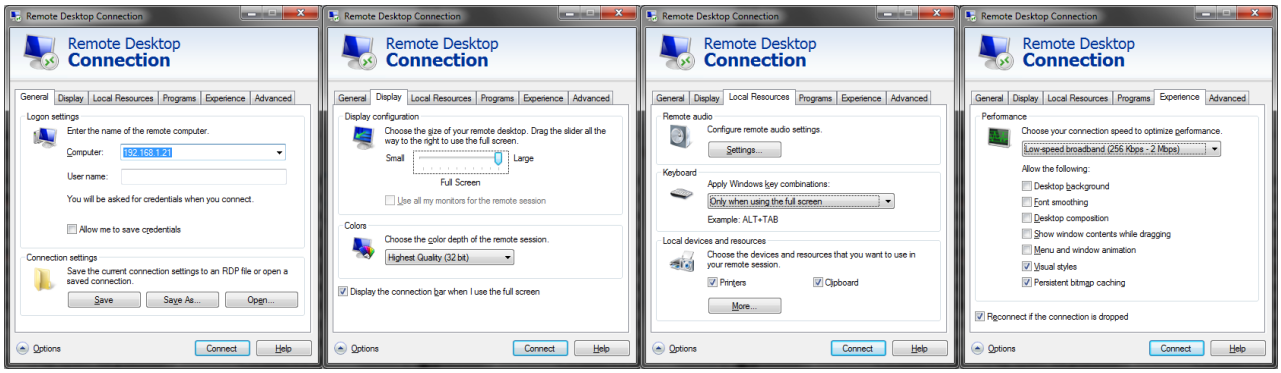


Wyświetlenie listy uruchomionych procesów w systemie zdalnym

Dostęp zdalny poprzez terminale tekstowe ma te same wady i zalety, które mają tekstowe powłoki systemowe – niektóre operacje mogą być wykonane szybciej niż w trybie graficznym, jednak użycie interfejsów tekstowych nie jest tak proste i wygodne jak graficznych. W przypadku połączeń zdalnych ich główną zaletą są małe wymagania co do przepustowości sieci łączącej klienta z serwerem, w przypadku dostępu tekstowego z powodzeniem można administrować systemem poprzez łącze GPRS o szybkości maksymalnej 114 kbit/s i opóźnieniach rzędu 600 – 1000 ms, które jest zdecydowanie niewystarczające do dostępu graficznego.

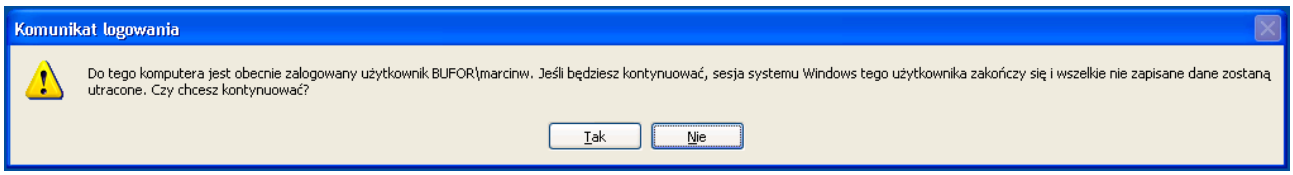
W dobie błyskawicznie rozwijającego się szybkiego dostępu do Internetu, ograniczenia wnoszone przez wolne łącza przestają być istotne i coraz większa część systemów operacyjnych oferuje możliwość zdalnej administracji poprzez graficzny interfejs użytkownika. Możliwość ta, czasem wbudowana w system, czasem dostępna po instalacji określonego narzędzia, dostępna jest zarówno na systemach Microsoft Windows jak i Linux. Poniżej omówione zostaną dwie metody udostępniania GUI systemu operacyjnego – Remote Desktop Sharing (RDS, charakterystyczna dla Microsoft Windows) oraz Virtual Network Computing (VNC, możliwa do uruchomienia na wielu systemach operacyjnych).

W systemach Microsoft Windows wbudowano mechanizm zdalnego udostępniania pulpitu (**Remote Desktop Sharing**). Usługa ta pozwala na dostęp do graficznego interfejsu użytkownika (GUI) systemu operacyjnego zdalnego komputera poprzez komputer kliencki w ten sam sposób jak gdyby użytkownik znajdował się fizycznie przed systemem zdalnym. Oprócz oczywistej funkcji prezentowania GUI, Remote Desktop Sharing umożliwia wykorzystanie niektórych zasobów lokalnego komputera przez aplikacje uruchomione na systemie zdalnym. Jest to tzw. przekierowanie zasobów, a zasobami, które mogą być poddane przekierowaniu są klawiatura, mysz, porty komunikacyjne, podłączone drukarki oraz karty dźwiękowe. Umożliwia to na przykład wydrukowanie na lokalnej drukarce dokumentu zamieszczonego na zdalnym serwerze otwartego w programie, który nie jest zainstalowany lokalnie bez konieczności udostępniania drukarki w sieci. Serwer usług RDS jest dostępny tylko na platformy Microsoft Windows (a więc można administrować tylko komputerami w takie systemy wyposażonymi), aplikacje klienckie natomiast są dostępne zarówno dla Microsoft Windows (zwykły i CE) jak i Apple Mac OS X. Istnieją również otwarte implementacje klientów RDC (**remote desktop client**) dla systemów Linux i Unix. Ilustracje obrazujące użycie usług RDS znajdują się poniżej:



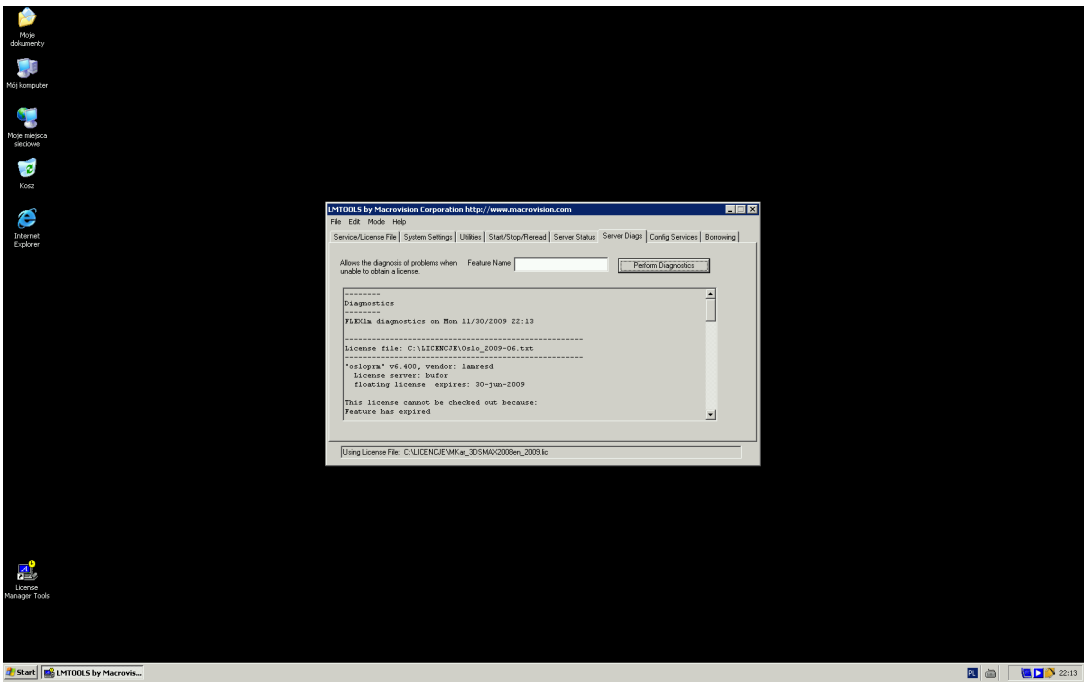
Okno konfiguracji połączenia. Widoczne opcje konfiguracyjne: adres IP, nazwa użytkownika, rozdzielczość ekranu i głębia kolorów, współdzielone zasoby, parametry przepustowości łącza

RDS nie umożliwia współdzielenia pulpitu przez wielu użytkowników jednocześnie, próba zalogowania się na system używany przez inną osobę wywołuje komunikat o konieczności zdalnego wylogowania tej osoby:



Komunikat o innym użytkowniku zalogowanym na komputerze zdalnym

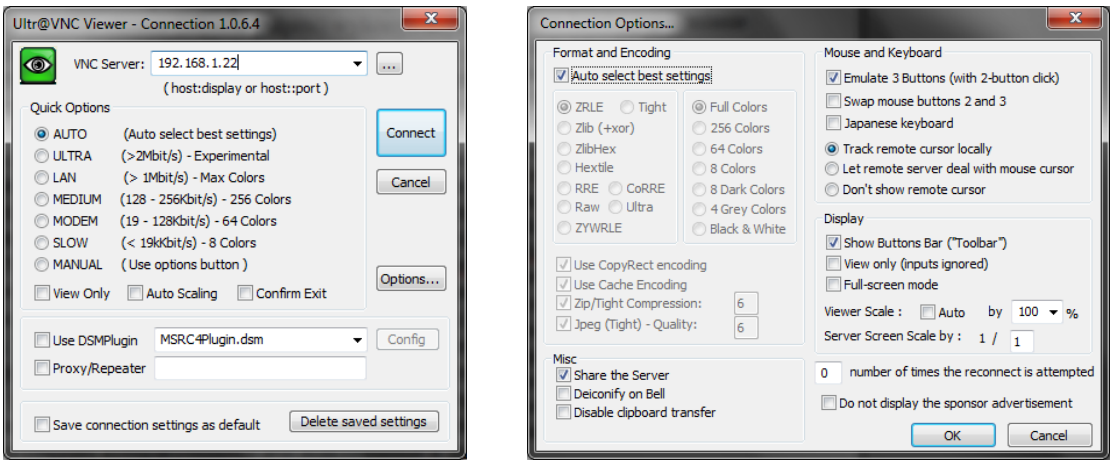
Po poprawnym zalogowaniu się, użytkownik widzi pulpit zdalnego systemu operacyjnego:



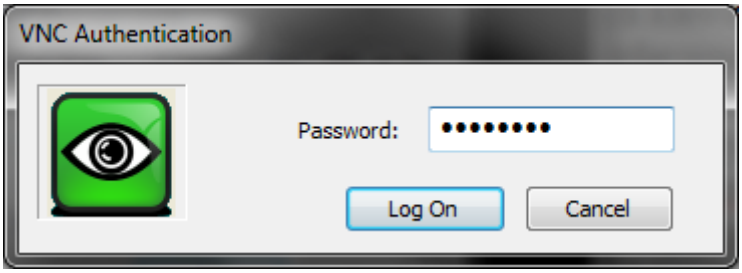
Podgląd pulpitu zdalnego (RDC)

Alternatywną metodą współdzielenia graficznego interfejsu użytkownika, która możliwa jest do zastosowania na większości systemów operacyjnych jest użycie systemu VNC. System ten działa na zasadzie wykonywania zrzutów ekranu (zdjęcia pulpitu) wykonywanych po każdej jego zmianie (czyli przerysowania interfejsu użytkownika). Pobrane obrazy przesyłane są do klienta, który wyświetla je na komputerze lokalnym. Obrazy te przesyłane są po poddaniu ich kompresji mającej na celu oszczędność przepustowości sieci. Często też przesyłane dane są różnicowe, tzn. transmitowane jest informacja o tych obszarach obrazu, które uległy zmianie od ostatniego odświeżenia. Ruchy myszy i informacja o naciśniętych klawiszach klawiatury przesyłana jest z kolei z maszyny klienckiej do systemu zdalnego gdzie symuluje zdarzenia systemowe wywołane przez mysz i klawiaturę rzeczywistą. Niektóre implementacje VNC (system ten jest otwarty, jego kody źródłowe są dostępne w Internecie) pozwalają na szereg dodatkowych czynności takich jak wymiana plików (RYSUNEK) lub rozmowa z innymi klientami współdzielącymi pulpit (RYSUNEK). Standardowym portem na którym działa usługa VNC jest port 5900 TCP, przy czym w systemach Linux używa ona często portów 5901 i wzwyż, w zależności od tego dla którego ekranu jest uruchamiana. Tak jak serwery VNC, tak również aplikacje klienckie dostępne są na wiele systemów operacyjnych. Istnieje także możliwość korzystania z przeglądarki internetowej obsługującej aplety Javy jako aplikacji klienckiej (rysunek). Z racji niskiego poziomu bezpieczeństwa zapewnianego przez ten protokół (dostęp do systemu zdalnego jest chroniony tylko hasłem, które często dla wstecznej kompatybilności jest maksymalnie 8 znakowe), jest on tunelowany z wykorzystaniem SSH.

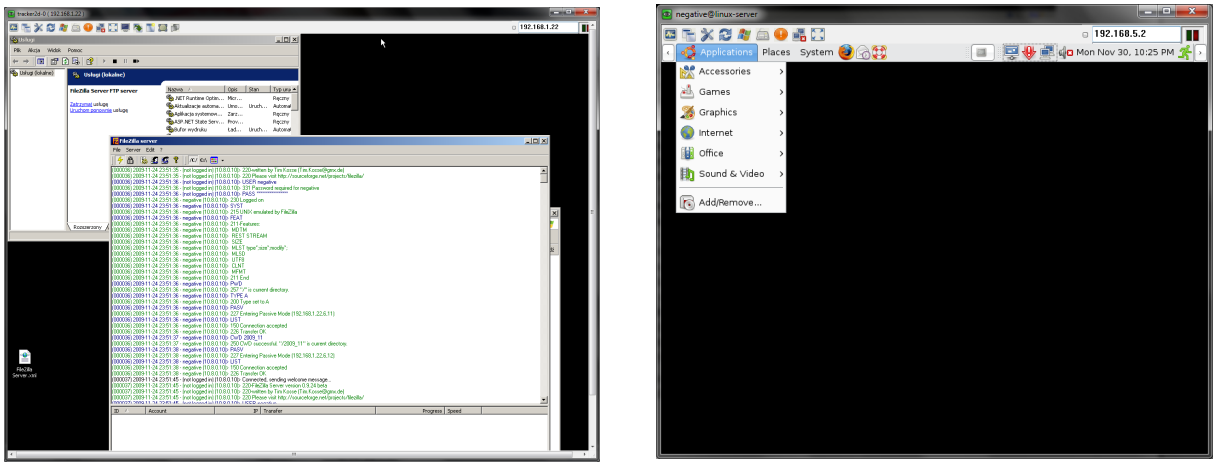
Przykładowe ilustracje obrazujące użycie systemu VNC (w prezentowanej wersji – UltraVNC) znajdują się poniżej:



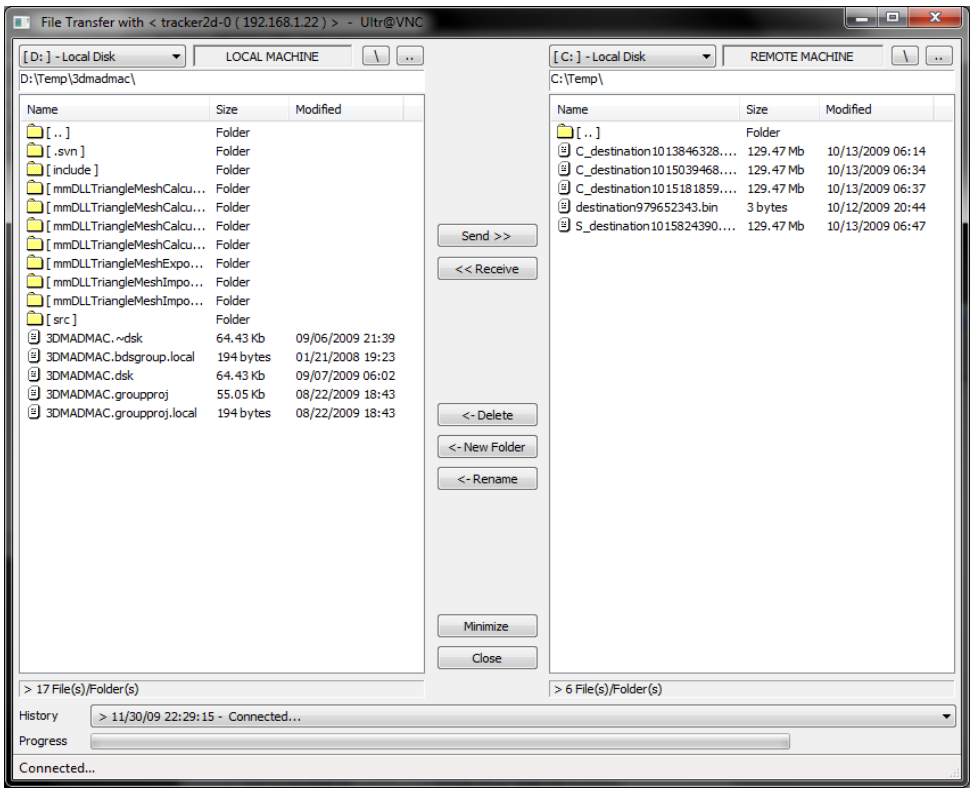
Okno konfiguracyjne klienta UltraVNC



Okno wprowadzania hasła uwierzytelniającego użytkownika



Podgląd pulpitów zdalnych Microsoft Windows XP i Ubuntu Linux 8.04 LTS udostępnionych poprzez VNC



Okno transferu plików w systemie UltraVNC

Tworzenie skryptów administracyjnych

Czynności związane z administracją systemu komputerowego często wymagają powtarzających się operacji, które mogą być wykonane automatycznie. Do tego celu służą tak zwane skrypty, czyli krótkie programy napisane w języku, który jest interpretowany przy uruchomieniu programu (nie kompilowany do postaci kodu maszynowego jak w przypadku klasycznych języków programowania jak np. c++). Kosztem tego rozwiązania jest jego mała wydajność w stosunku do tradycyjnych programów, jednakże prostota uruchamiania jest ogromną zaletą przeważającą tą wadę. Języki skryptowe najczęściej mają mniejsze możliwości niż języki pełnowymiarowe, jednakże doskonale sprawdzają się w automatyzacji zadań administracyjnych. Języki te najczęściej posiadają podstawowe funkcjonalności takie jak:

- posiadają zmienne bez określonych typów (zadeklarowana zmienna nie musi przez cały okres istnienia być np. liczbą, może być wykorzystywana do przechowywania np. tekstu)
- umożliwiają tworzenie zmiennych globalnych i lokalnych (istniejących tylko w określonych fragmentach programu)
- umożliwiają tworzenie pętli (typu for / while)
- zawierają instrukcje warunkowe (if / else / case)
- mogą w łatwy sposób uruchamiać polecenia systemowe i korzystać ze zmiennych środowiskowych

Programy skryptowe mogą, oprócz automatyzacji zadań administracyjnych służyć również do szybkiego prototypowania aplikacji, ponieważ tworzenie programów skryptowych jest dużo szybsze niż tradycyjnych, co pozwala na przetestowanie różnych rozwiązań przed ich

implementacją (długotrwałą) w zaawansowanym języku programowania. Składnia języków skryptowych jest nieskomplikowana, zawierają one też stosunkowo małą liczbę specyficznych dla siebie operatorów i komend.

Najszerzej znanym językiem skryptowym w systemach Unix / Linux jest ***bash,*** następca popularnego języka ***sh***. W tym miejscu należy podkreślić, że zarówno *sh* jak i *bash* są nazwami zarówno powłok systemowych w systemach wywodzących się z UNIXa jak i nazwami języków skryptowych umożliwiających wykonanie skryptów z poziomu powłoki *sh/bash*. Skrypty języka *bash* są najczęściej pojedynczymi plikami tekstowymi, rozpoczynającymi się od specjalnego nagłówka ***#!/bin/bash***. Nagłówek ten wskazuje interpretorowi jaki program powinien zostać użyty do uruchomienia skryptu. Najprostszy skrypt wypisujący tekst w konsoli wygląda następująco:

```
#!/bin/bash
echo Hello World
```

Po zapisaniu pliku zawierającego powyższe linie (np. jako skrypt.sh) należy nadać mu atrybut wykonalności. Można to wykonać np. poprzez wywołanie polecenia chmod +x skrypt.sh. Uruchomienie skryptu wykonuje się poprzez wpisanie w konsoli ./skrypt.sh i wciśnięcie Enter.

Jak wspomniano we wstępie, języki skryptowe umożliwiają tworzenie pętli oraz instrukcji warunkowych. Poniżej zamieszczono przykładowe skrypty drukujące na ekranie kolejne liczby (od 0 do 100), korzystające z pętli różnych rodzajów:

Typ pętli	Przykład
While	#!/bin/bash v_iCounter = 0 while [$ v_iCounter -lt 100]; do echo $ v_iCounter let v_iCounter=v_iCounter+1 done
For	#!/bin/bash for ((v_iCounter=1; v_iCounter<=100; v_iCounter++)) do echo $ v_iCounter done
Until	#!/bin/bash v_iCounter = 100 until [$ v_iCounter -lt 0]; do echo $ v_iCounter let v_iCounter-=1 done

Podobnie, *bash* umożliwia korzystanie z instrukcji warunkowych:

```
#!/bin/bash
```

```
v_sText1 = "tekst1" // deklaracja zmiennych
v_sText2 = "tekst2"

if [ "$v_sText1" = "$v_sText2" ]; then
   echo Tekst1 i Tekst2 są identyczne
else
   echo Tekst1 i Tekst2 są różne
fi
```

Podobnie jak w bardziej zaawansowanych językach programowania możliwe jest dzielenie skryptu na fragmenty zwane funkcjami. Funkcja pozwala na zamknięcie często powtarzanego fragmentu programu w bloku o określonej nazwie i wywoływaniu go poprzez tą nazwę co umożliwia minimalizację powtarzających się fragmentów kodu. Przykład deklaracji i użycia funkcji znajduje się poniżej:

Deklaracja funkcji

```
#!/bin/bash

function wypisz_tekst
{
  for ((v_iCounter2=1; v_iCounter2<=$2;
v_iCounter2++))
  do
      echo $1
  done
}

#$1 oznacza pierwszy argument podany do
funkcji
```

Wywołanie funkcji

```
v_iCounter=0
while [  $v_iCounter -lt 10 ];
do
     wypisz_tekst $v_iCounter $v_iCounter*2
     let v_iCounter=v_iCounter+1
done
```

Przykładowy skrypt administracyjny, służący do usuwania zamieszczonych przez użytkowników danych w katalogach nie nazywanych wg określonego klucza (dla_IN, gdzie I – pierwsza litera imienia, N – nazwiska użytkownika) lub katalogu public, został zamieszczony poniżej. Skrypt ten przeszukuje nazwy znajdujących się w określonym katalogu podkatalogów sprawdzając czy są one zgodne z kluczem. W pierwszej części skryptu usuwane są wszystkie katalogi (i ich zawartość) jeżeli ich nazwa nie zawiera słowa „dla” oraz nie jest równa „public”. W drugiej części skryptu usuwane są wszystkie katalogi z zawartością, których nazwa zawiera słowo „dla” ale jest dłuższa niż 6 liter (nie pasujące do klucza dla_IN):

```
#!/bin/bash

cd /home/bufor
IFS=$'\n'
for NOT_USER_FILES in $(ls | grep -v dla)
do
   if [ "$NOT_USER_FILES" != "public" ]; then
         rm -rf "$NOT_USER_FILES"
         rm "$NOT_USER_FILES"
  fi
```

```
done

for USER_DIRS in $(ls | grep dla)
do
   NAME_LENGTH=$(expr length $USER_DIRS)
   if [ "$NAME_LENGTH" != "6" ]; then
         rm -rf "$USER_DIRS"
   fi
done
```

Innym skryptem, którego zadaniem jest zapisywanie do dziennika adresów, z którymi łączą się komputery sieci lokalnej znajdujące się za routerem jest skrypt przedstawiony poniżej. Skrypt ten wywołuje polecenie logujące ruch sieciowy (netstat-nat), przekierowując zwracane przezeń informacje do pliku tymczasowego log.txt (w pętli co dziesięć sekund). Po zakończeniu pętli (minięciu godziny) plik ten jest pakowany archiwizatorem tar, a do nazwy archiwum dodawany jest tekst określający datę utworzenia pliku. Archiwum jest kopiowane do katalogu użytkownika ze zmienionym ID właściciela (z administratora na użytkownika negative):

```
#!/bin/bash
DATE=`date +'%m-%d-%Y-%k%M'`

cd /root
#mkdir log
cd log
#cp /etc/rc.d/rc.local /root/log_$DATE

for i in `seq 1 358`;
do
   date >> log.txt
   netstat-nat -n >> log.txt
   sleep 10;
done

cd /root
tar -czvf log_$DATE.tgz log
chown negative.negative log_$DATE.tgz
mv log_$DATE.tgz /home/negative/log
#cd /root/log_$DATE
#rm * -rf
#cd ..
#rmdir log_$DATE
```

Jak widać, przy pomocy skryptów bash w łatwy sposób można zautomatyzować wiele czynności, które byłyby praktycznie niewykonalne manualnie (np. logowanie ruchu sieciowego wymagałoby ciągłego uruchamiania przez administratora wielu komend) co w znacznym stopniu zwiększa wygodę zarządzania danym serwerem czy nawet komputerem osobistym (przykładowo można stworzyć skrypt automatycznie tworzący kopię zapasową wybranych danych w określonych momentach).

Systemy Microsoft Windows również posiadają możliwość uruchamiania skryptów. Różnią

się one nieco od skryptów *bash*, jednakże zachowują większość ich cech. Począwszy od systemu Microsoft Windows 98 w systemie wbudowany jest mechanizm ***Windows Script Host*** (WSH) umożliwiający uruchamianie programów skryptowych napisanych w różnych językach, między innymi JScript (pliki JS i JSE), VBScript (pliki VBS, VBE). Dodatkowo *ScriptHost* można rozszerzać instalując interpretery innych języków skryptowych jak np. Perl. Dodatkowo można tworzyć programy łączące różne języki (pliki WSF – *WindowsScript File*).

Zgodnie z dokumentacją WSH przy użyciu języków skryptowych w systemie Microsoft Windows można między innymi:

- wypisywać komunikaty dla użytkownika
- odczytywać i zmieniać ustawienia systemowe
- odczytywać i modyfikować rejestr systemowy
- mapować dyski sieciowe
- łączyć się z drukarkami

Uruchamianie skryptów odbywa się tak samo jak plików wykonywalnych aplikacji – poprzez dwukrotne kliknięcie na ich ikonie (w Eksploratorze Windows) lub wpisaniu nazwy w linii poleceń. Domyślnym interpretatorem skryptów jest moduł wscript i on jest uruchamiany przy włączaniu skryptów w sposób opisany powyżej. Istnieje jednak drugi interpreter cscript, który różni się od wscript tym, że komunikaty wypisywane poprzez polecenie Echo kierowane są do konsoli, a nie okien dialogowych.

Przykładowy skrypt (VBScript) wypisujący podany tekst na ekranie znajduje się poniżej:

```
WScript.Echo "Hello World"
WScript.Quit
```

Podobnie jak w *bash*, WSH umożliwia tworzenie pętli w programach:

```
For v_iCounter = 1 to 100
   Wscript.echo v_iCounter
Next
```

Możliwe jest również tworzenie instrukcji warunkowych:

```
v_iA = 10
v_iB = 100

If v_iA < v_iB Then
        Wscript.echo "A jest mniejsze od B"
Else
        Wscript.echo "A nie jest mniejsze od B"
End If
```

Przykładowym, bardziej zaawansowanym skryptem administracyjnym używającym VBScript jest program służący do sprawdzenia czy dana usługa (lub program) jest uruchomiony w systemie, Kod skryptu został umieszczony poniżej. Skrypt ten pobiera listę procesów uruchomionych na

komputerze i kopiuje do struktury v_sColProcesses nazwy procesów identyczne z podanym ciągiem znaków (w tym przykładzie vmware-usbarbitrator.exe”). Jeżeli struktura v_sColProcesses zawiera przynajmniej jeden obiekt, to znaczy że usługa ta została uruchomiona. Odpowiedni komunikat wypisywany jest na ekranie.

```
Set v_sObjWMIService = GetObject("winmgmts:\\.\root\cimv2")
Set v_sColProcesses = v_sObjWMIService.ExecQuery ("Select * from Win32_Process Where
Name = 'vmware-usbarbitrator.exe'")

If v_sColProcesses.Count = 0 Then
   wSCRIPT.Echo "Usługa vmware-usbarbitrator nie została uruchomiona."
Else
   wSCRIPT.Echo "Usługa vmware-usbarbitrator została uruchomiona"
End If
```

Kolejny skrypt umożliwia sprawdzenie czy użytkownik ma włączoną opcję wygaszania ekranu po określonym czasie bezczynności. Jeżeli nie, skrypt oferuje włączenie wygaszacza (skrypt pobrany ze strony http://www.pctools.com/guides/scripting):

```
' Begin code for rewrite.vbs
' Downloaded from the Scripting Guide for Windows
' http://www.pctools.com/guides/scripting/
' Version: 1.0 (June 8, 2005)

Option Explicit

Dim WSHShell, RegKey, ScreenSaver, Result
Set WSHShell = CreateObject("WScript.Shell")
RegKey = "HKCU\Control Panel\Desktop\"

ScreenSaver = WSHShell.RegRead (regkey & "ScreenSaveActive")
If ScreenSaver = 1 Then 'Screen Saver is Enabled
  Result = MsgBox("Your screen saver is currently active." & _
      vbNewLine & "Would you like to disable it?", 36)
  If Result = 6 Then 'clicked yes
    WSHShell.RegWrite regkey & "ScreenSaveActive", 0
  End If
Else 'Screen Saver is Disabled
  Result = MsgBox("Your screen saver is currently disabled." & _
      vbNewLine & "Would you like to enable it?", 36)
  If Result = 6 Then 'clicked yes
    WSHShell.RegWrite regkey & "ScreenSaveActive", 1
  End If
End If

' End code
```

Automatyczne podłączenie określonych zasobów sieciowych po zalogowaniu się użytkownika może być wykonane poprzez umieszczenie poniższego skryptu w folderze Autostart:

```
set WshNetwork=CreateObject ("Wscript.Network")
WshNetwork.AddWindowsPrinterConnection "lpt1", "\\servername\lasername"
WshNetwork.MapNetworkDrive "z:", "\\servername\sharename"
WScript.Echo "You are now connected to the network printer and file share"
```

Ogromna kolekcja przykładowych skryptów WindowsScriptingHost znajduje się na przykład stronach Microsoft Technet. Duży zbiór programów rozwiązujących wiele problemów administracyjnych umieszczono również na http://www.ericphelps.com/.

Bibliografia

SILBERSCHATZ: A. Silberschatz, G. Cagne, P. Baer, Operating system concepts with Java (Sixth Edition ed.), 2004
APS-QNX: QNX Software Systems, Adaptive Partitioning - Scheduler, , http://community.qnx.com/sf/wiki/do/viewPage/projects.core_os/wiki/Adaptive_Partitioning_Scheduler
TANNENBAUM: A. S. Tannenbaum, Modern Operating Systems,
DIJKSTRA-DPP: E. W. Dijkstra, Hierarchical ordering of sequential processes, 1971
WALDSPURGER: C. A. Waldspurger, W. E. Weihl, Lottery Scheduling: Flexible Proportional-Share Resource Management, 1994
STALLING: W. Stallings, Operating systems: internals and design principles, 2004
MREBOOT: G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, A. Fox, Microreboot &\#8212; A technique for cheap recovery, 2004
FTH: T. Anderson, Windows 7's dirty secrets revealed, 2009
RFC2828: , RFC 2828:Internet Security Glossary,
MSBLAST: Microsoft, Alert dotyczący wirusa typu worm Blaster i jego odmian, 2003
PGP: Simson Garfinkel, PGP: Pretty Good Privacy, 1991
GPG: Werner Koch, GnuPG 2.0.13 released, 2009, http://lists.gnupg.org/pipermail/gnupg-announce/2009q3/000294.html
DISCLOG: D.R. Stinson, Cryptography: Theory and Practice, 2002
CRYPTOGRAPHY: A. Menezes, P.C. van Oorschot, S. A. Vanstone, Handbook of Applied Cryptography, 1996
W3C: The World Wide Web Consortium (W3C) , HTTP - Hypertext Transfer Protocol, , http://www.w3.org/Protocols/
SSL: S. A. Thomas, SSL and TLS essentials securing the Web, 2000
CASTELLI: Mathew Castelli, Network Consultants Handbook, 2002
BENVENUTI: Christian Benvenuti, Understanding Linux Network Internals, 2005