SYSTEMY OPERACYJNE

Architektura, funkcjonowanie i projektowanie

WILLIAM STALLINGS

WYDANIE IX

Helion

Tytuł oryginału: Operating Systems: Internals and Design Principles (9th Edition)

Tłumaczenie: Zdzisław Płoski

ISBN: 978-83-283-3759-6

Authorized translation from the English language edition, entitled: OPERATING SYSTEMS: INTERNALS AND DESIGN PRINCIPLES, Ninth Edition; ISBN 0134670957; by William Stallings; published by Pearson Education, Inc. Copyright © 2018, 2015, 2012, 2009 by Pearson Education, Inc., Hoboken, New Jersey 07030.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/syopa9
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

SPIS TREŚCI

SPIS TREŚCI ROZDZIAŁÓW ONLINE

Rozdział 2

Przegląd systemów operacyjnych

W TYM ROZDZIALE POZNASZ I ZROZUMIESZ:

- podsumowanie, na najogólniejszym poziomie, podstawowych funkcji SO;
- omówienie ewolucji systemów operacyjnych od wczesnych, prostych systemów wsadowych aż po nowoczesne, złożone systemy;
- krótkie wyjaśnienie każdego z głównych osiągnięć w badaniach SO, jak określono w podrozdziale 2.3;
- omówienie kluczowych obszarów projektowania, które przyczyniły się do rozwoju nowoczesnych systemów operacyjnych;
- zdefiniowanie i omówienie maszyn wirtualnych i wirtualizacji;
- zagadnienia projektowania SO wynikłe z pojawienia się organizacji wieloprocesorowych i wielordzeniowych;
- podstawową strukturę systemu Windows;
- istotne elementy tradycyjnego systemu UNIX;
- wyjaśnienie nowych właściwości występujących w nowoczesnych systemach uniksowych;
- omówienie Linuxa i jego związków z UNIX-em[1].

Nasz kurs systemów operacyjnych rozpoczynamy od krótkiego przeglądu historycznego. Jest to opowieść sama w sobie ciekawa, a ponadto służy jako przegląd podstaw SO. W pierwszym podrozdziale zajmiemy się przeznaczeniem i funkcjami systemów operacyjnych. Dalej przyjrzymy się ich ewolucji od prymitywnych systemów wsadowych do wyrafinowanych, wielozadaniowych systemów obsługujących jednocześnie wielu użytkowników. W pozostałej części rozdziału prześledzimy historię i przedstawimy ogólną charakterystykę dwóch systemów operacyjnych, które służą jako przykładowe w całej książce.

2.1. CELE I FUNKCJE SYSTEMU OPERACYJNEGO

System operacyjny (SO) jest programem, który nadzoruje wykonywanie aplikacji (programów użytkowych) i działa jak interfejs między aplikacjami a sprzętem komputerowym. Można go traktować jako mający do spełnienia trzy cele:

- **wygoda —** SO sprawia, że komputer jest wygodniejszy w użyciu;
- **sprawność —** SO umożliwia efektywne użytkowanie zasobów komputerowych;

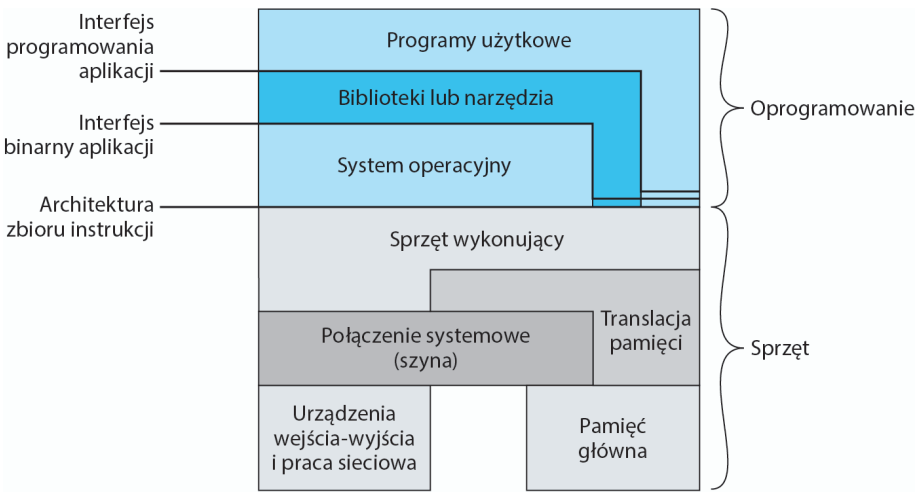
[1] Oraz podstawowych cech systemu Android, o czym tu nie wspomniano — *przyp. tłum.*

- **zdolność do ewolucji —** SO powinien być skonstruowany w taki sposób, aby można go było skutecznie rozwijać, testować i rozszerzać o nowe funkcje systemowe, nie zaburzając już świadczonych usług.

  Prześledźmy te trzy aspekty SO po kolei.

System operacyjny jako interfejs użytkownik – komputer

Sprzęt i oprogramowanie służące do udostępniania użytkownikowi aplikacji można rozpatrywać warstwowo, jak pokazano na rysunku 2.1. Użytkownika tych aplikacji (użytkownika docelowego) na ogół nie obchodzą szczegóły sprzętu komputerowego. Postrzega on system komputerowy w kategoriach zbioru aplikacji. Aplikację można wyrazić w języku programowania, opracowuje ją programista aplikacji[2]. Gdyby komuś przyszło napisać program użytkowy w formie zbioru instrukcji maszynowych, w pełni odpowiadający za sterowanie sprzętem komputera, okazałoby się to przedsięwzięciem złożonym ponad wszelką miarę. Aby ułatwić tę robotę, tworzy się zbiór programów systemowych. Niektóre z nich są nazywane programami narzędziowymi lub bibliotecznymi. Realizują one często używane funkcje, pomagające w budowaniu programów, zarządzaniu plikami i sterowaniu urządzeniami wejścia-wyjścia. Programista posłuży się nimi, tworząc aplikację, natomiast aplikacja w trakcie swojego działania będzie się uciekać do tych narzędzi, aby wykonać pewne funkcje. W skład SO wchodzi zestaw najważniejszych programów systemowych. SO ukrywa przed osobą programującą szczegóły sprzętowe, dostarczając jej wygodnego interfejsu do użytkowania systemu komputerowego. Działa jak mediator, ułatwiając programiście i programom użytkowym dostęp do narzędzi i usług oraz ich wykorzystanie.



Rysunek 2.1. Sprzęt komputerowy i struktura oprogramowania

[2] On lub ona — podkreślmy to tutaj, zaznaczając, że uwaga ta odnosi się do wszelkich zawężonych do rodzaju męskiego form w tym przekładzie — *przyp. tłum.*

W skrócie — SO zazwyczaj dostarcza usług w następujących obszarach:

- **Opracowywanie programów.** SO udostępnia rozmaitość narzędzi i usług, takich jak edytory i debugery (programy uruchomieniowe) pomagające programiście w tworzeniu programów. Na ogół usługi te przybierają postać programów narzędziowych, które — choć nie są nieodłączną częścią rdzenia SO — są dostarczane wraz z SO i określane jako narzędzia opracowywania programów użytkowych.
- **Wykonywanie programów.** Aby nadać bieg gotowemu programowi, trzeba wykonać kilka kroków. Rozkazy i dane należy załadować do pamięci, urządzenia zewnętrzne i pliki muszą być zainicjowane, a także trzeba przygotować inne zasoby. SO wywiązuje się wobec użytkownika z tych planowych obowiązków.
- **Dostęp do urządzeń wejścia-wyjścia.** Każde urządzenie zewnętrzne wymaga do działania specyficznego zbioru instrukcji lub sygnałów. SO dostarcza jednolitego interfejsu ukrywającego te szczegóły, dzięki czemu programiści mogą kontaktować się z takimi urządzeniami, używając po prostu operacji pisania i czytania.
- **Kontrolowany dostęp do plików.** Aby umożliwić dostęp do plików, SO musi szczegółowo rozeznawać nie tylko rodzaj urządzenia wejścia-wyjścia (napęd dysku czy taśmy), lecz także strukturę danych zawartych w plikach zmagazynowanych na nośniku danych. W przypadku systemu z wieloma użytkownikami SO może tworzyć mechanizmy ochronne w celu kontrolowania dostępu do plików.
- **Dostęp do systemu.** W przypadku systemów do wspólnego czy publicznego użytku SO kontroluje dostęp do systemu jako całości oraz do poszczególnych zasobów systemu. Funkcja dostępu musi zapewniać ochronę zasobów i danych przed nieupoważnionymi użytkownikami, a także rozwiązywać konflikty powstające na tle rywalizacji o zasoby.
- **Wykrywanie błędów i reagowanie.** Podczas działania systemu komputerowego mogą występować różnego rodzaju błędy. Są wśród nich wewnętrzne i zewnętrzne błędy sprzętowe (jak błąd pamięci lub wadliwe albo nie całkiem sprawne działanie urządzenia) oraz rozmaite błędy programowe (jak dzielenie przez zero, próba dostępu do zakazanego miejsca pamięci lub niezdolność SO do spełnienia żądania aplikacji). W każdym przypadku SO musi odpowiednio zareagować, usuwając skutki błędu w sposób możliwie najmniej inwazyjny dla wykonywanych aplikacji. Reakcja taka może być różnego rodzaju: od zakończenia programu, który spowodował błąd, do ponowienia operacji lub zwykłego powiadomienia aplikacji o błędzie.
- **Rozliczanie.** Dobry SO będzie gromadził statystykę wykorzystania różnych urządzeń i monitorował parametry działania, na przykład czas odpowiedzi. Tego rodzaju informacje są przydatne w każdym systemie, gdyż służą przewidywaniu konieczności przyszłych ulepszeń i strojeniu systemu w celu poprawienia jego wydajności. W systemie z wieloma użytkownikami informacje te mogą być użyte do wystawiania rachunków.

Na rysunku 2.1 zaznaczono również trzy podstawowe interfejsy typowego systemu komputerowego. Są to:

- **Architektura zbioru rozkazów** (ang. *instruction set architecture* — ISA)**.** ISA definiuje repertuar instrukcji maszynowych, którym dany komputer jest zdolny dać posłuch. Ten interfejs stanowi rozgraniczenie między sprzętem a oprogramowaniem. Zauważmy, że zarówno programy użytkowe, jak i narzędziowe mogą mieć bezpośredni dostęp do ISA. Dla tych programów wydziela

się podzbiór repertuaru instrukcji (ISA użytkownika). System operacyjny rozporządza dodatkowymi instrukcjami języka maszynowego, służącymi do zarządzania zasobami systemowymi (ISA systemu).

- **Interfejs binarny aplikacji** (ang. *application binary interface* — ABI)**.** ABI definiuje standard binarnej przenośności między programami. Określa też interfejs wywołań systemowych łączących z systemem operacyjnym oraz z zasobami sprzętowymi i usługami dostępnymi w systemie za pośrednictwem ISA użytkownika.
- **Interfejs programów użytkowych** (interfejs programowy aplikacji, ang. *application programming interface* — API[3])**.** API daje programowy dostęp do zasobów sprzętowych i usług osiągalnych w systemie za pośrednictwem ISA użytkownika, uzupełniony o wywołania biblioteczne w języku wysokiego poziomu (ang. *high-level language* — HLL). Dowolne wywołanie systemowe jest zwykle wykonywane przez biblioteki. Używanie API umożliwia łatwe przenoszenie oprogramowania użytkowego (przez rekompilację) do innych systemów udostępniających ten sam API.

System operacyjny jako zarządca zasobów

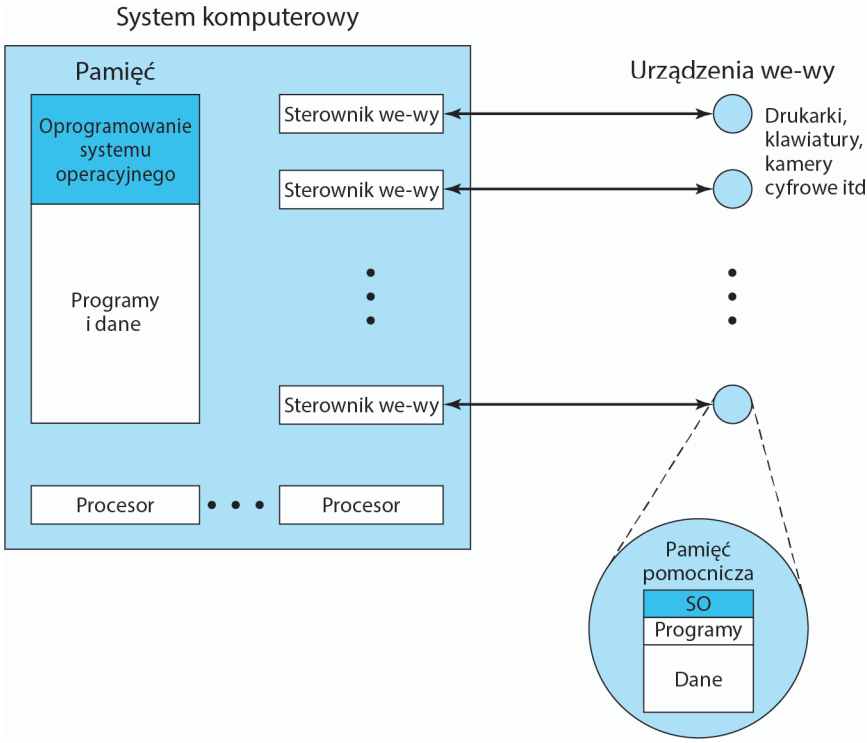
SO odpowiada za kontrolowanie użycia zasobów komputera, takich jak wejście-wyjście, pamięć główna i pomocnicza oraz czas działania procesora. Wszelako ten nadzór jest sprawowany w osobliwy sposób. Zwykle jesteśmy skłonni uważać mechanizm kontrolny za coś zewnętrznego w stosunku do tego, co jest kontrolowane, lub przynajmniej coś, co jest odrębną i wydzieloną częścią tego, co podlega kontroli. (Przykładem jest system centralnego ogrzewania, sterowany przez termostat — przyrząd odrębny względem źródła ciepła i kaloryferów). Rzecz ma się inaczej w przypadku SO, który jako mechanizm sterujący jest niezwykły z dwu powodów:

- SO działa w ten sam sposób jak zwykłe oprogramowanie komputera, to znaczy jest programem lub zestawem programów wykonywanych przez procesor.
- SO często wyzbywa się kontroli i musi polegać na procesorze, który umożliwia mu jej odzyskanie.

Jak inne programy komputerowe SO składa się z rozkazów wykonywanych przez procesor. W czasie działania SO decyduje o sposobie przydziału czasu procesora i o tym, które zasoby są dostępne do użycia. Aby jednak procesor mógł te decyzje realizować, musi zaprzestawać wykonywania programu SO i wykonywać inne programy. Tak więc SO zrzeka się sterowania, aby procesor mógł wykonać jakąś „pożyteczną" pracę, po czym wznawia kontrolę na czas potrzebny do przygotowania procesora do wykonania następnej pracy. Służące temu mechanizmy zostaną wyjaśnione w dalszej części rozdziału.

Na rysunku 2.2 naszkicowano główne zasoby zarządzane przez SO. Część systemu rezyduje w pamięci głównej. Obejmuje ona **jądro** (ang. *kernel* albo *nucleus*), które zawiera najczęściej używane w systemie funkcje i — w danym czasie — inne części SO, będące akurat w użyciu. Pozostała część pamięci głównej zawiera programy użytkowe i narzędziowe oraz dane. SO i sprzętowa jednostka zarządzania pamięcią w procesorze sprawują razem kontrolę nad przydzielaniem pamięci głównej, co jeszcze zobaczymy. SO decyduje, kiedy urządzenie zewnętrzne może być użyte przez wykonywany program, i kontroluje dostęp do plików i ich użycie. Sam procesor jest jednym z zasobów i SO musi określać, ile czasu procesora będzie poświęcone na wykonanie programu danego użytkownika.

[3] Akronim rozwijany również jako *application programmer interface*, czyli interfejs programisty aplikacji — *przyp. tłum.*



Rysunek 2.2. System operacyjny jako zarządca zasobów

Łatwość ewolucji systemu operacyjnego

Z biegiem czasu ważniejsze systemy operacyjne będą ewoluowały z kilku powodów:

- **Unowocześnienia sprzętowe i nowe rodzaje sprzętu.** Na przykład w pierwszych wersjach UNIX-a i macintoshowego Mac OS-a nie wykorzystywano mechanizmu stronicowania, ponieważ systemy te działały na procesorach bez sprzętowego zaplecza stronicowania[4]. Kolejne wersje tych systemów operacyjnych zostały zmodyfikowane tak, aby mogły wykorzystywać stronicowanie. Również zastosowanie terminali graficznych i terminali pracujących w trybie stronicowym zamiast działających w trybie przewijania wiersz po wierszu wpłynęło na projekt SO. Na przykład terminal graficzny z reguły umożliwia użytkownikowi oglądanie na ekranie kilku aplikacji w tym samym czasie za pośrednictwem „okien". To wymaga bardziej wyrafinowanego wsparcia ze strony SO.
- **Nowe usługi.** W odpowiedzi na zapotrzebowania użytkowników lub wymagania administratorów SO jest rozszerzany o nowe usługi. Jeśli na przykład okaże się, że za pomocą istniejących narzędzi trudno jest zapewnić użytkownikom dobrą wydajność, do SO mogą być dołączone nowe narzędzia pomiarowe i sterujące.
- **Poprawki.** Każdy SO ma wady. Z upływem czasu są one wykrywane i poprawiane. Oczywiście poprawka może spowodować powstanie nowych wad.

[4] O stronicowaniu powiemy pokrótce w dalszej części rozdziału, a dokładniej zajmiemy się nim w rozdziale 7.

Z koniecznością regularnego uaktualniania SO wiążą się pewne wymagania projektowe. Oczywistym postulatem jest zapewnienie modularności budowy systemu, z jasno zdefiniowanymi interfejsami między modułami, oraz to, że powinien on być dobrze udokumentowany. W wielkich programach — a takimi są typowe współczesne systemy operacyjne — to, co można by nazwać prostą modularyzacją, jest niewystarczające [DENN80a]. To znaczy trzeba zrobić znacznie więcej niż dokonać zwykłego podziału programu na moduły. Powrócimy do tego tematu w dalszej części rozdziału.

2.2. ROZWÓJ SYSTEMÓW OPERACYJNYCH

Próbując zrozumieć kluczowe wymagania dotyczące SO i znaczenie ważniejszych cech współczesnego SO, dobrze jest spojrzeć, jak wyglądał rozwój systemów operacyjnych na przestrzeni lat.

Przetwarzanie seryjne

W pierwszych komputerach z lat 40. i pierwszej połowy lat 50. XX wieku programista miał bezpośredni kontakt ze sprzętem komputerowym — nie było żadnego systemu operacyjnego. Komputery były uruchamiane z konsoli zawierającej lampki kontrolne i przełączniki stanowiące rodzaj urządzenia wejściowego, do tego dochodziła drukarka. Program w kodzie maszynowym był ładowany z urządzenia wejściowego (np. z czytnika kart). Jeśli program zatrzymał się z powodu błędu, powód błędu był wyświetlany za pomocą lampek. Gdy program zakończył działanie w normalny sposób, wyniki pojawiały się na drukarce. W tych pierwszych systemach występowały dwa główne problemy:

- **Planowanie.** W większości instalacji posługiwano się papierowymi arkuszami rezerwacji czasu komputera. Użytkownik zazwyczaj mógł zarezerwować porcję czasu wielkości pół godziny lub jej wielokrotność. Użytkownik mógł na przykład zapisać się na godzinę i skończyć pracę po trzech kwadransach, co skutkowało tym, że czas pracy komputera był marnowany. Z drugiej strony, zdarzało się, że użytkownik popadał w kłopoty, nie zdążył zakończyć obliczeń w przydzielonym czasie i musiał je przerwać przed uzyskaniem rozwiązania.
- **Czas przygotowania do pracy** (ang. *setup time*)**.** Pojedynczy program, zwany **zadaniem** (ang. *job*), mógł wymagać załadowania do pamięci kompilatora oraz programu w języku wysokiego poziomu (programu źródłowego), przechowania skompilowanego programu (programu wynikowego), a następnie załadowania i skonsolidowania programu wynikowego z typowymi funkcjami. Każdy z tych kroków mógł obejmować zakładanie i zdejmowanie taśm lub ładowanie pakietów kart do czytnika. Jeśli przytrafił się błąd, pechowy użytkownik musiał na ogół ciąg tych przygotowań zaczynać od nowa. Powodowało to znaczne straty czasu na samo przygotowanie programu do działania.

Ten tryb pracy można by nazwać *przetwarzaniem seryjnym*, zważywszy na fakt, że użytkownicy podchodzili do komputera raz po razie podczas jednej sesji. Z biegiem czasu opracowano różne systemowe narzędzia programowe w dążeniu do usprawnienia przetwarzania seryjnego. Należały do nich biblioteki typowych funkcji, konsolidatory, ładowacze, debugery i moduły (programy) sterujące urządzeniami zewnętrznymi, udostępniane jako typowy software wszystkim użytkownikom.

Proste systemy wsadowe

Pierwsze komputery były bardzo drogie, toteż maksymalizowanie wykorzystania procesora było niezwykle istotną kwestią. Marnowanie czasu wskutek godzinowych rezerwacji i czasu zużywanego na przygotowanie do pracy było nie do przyjęcia.

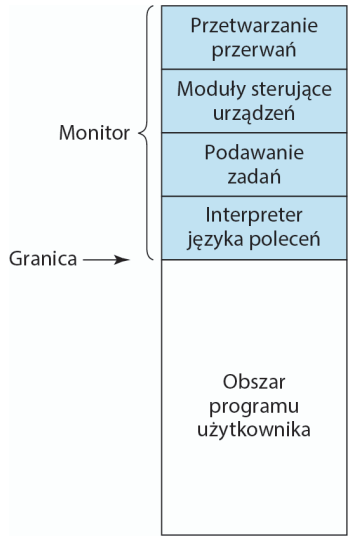
Aby poprawić wykorzystanie, opracowano koncepcję **wsadowego systemu operacyjnego** (ang. *batch OS*). Wydaje się, że pierwszy wsadowy SO (a zarazem pierwszy SO jakiegokolwiek typu) powstał w połowie lat 50. XX wieku w General Motors, opracowany na użytek komputera IBM 701 [WEIZ81]. Pomysł ten został następnie ulepszony i zrealizowany na komputerze IBM 704 przez wielu nabywców IBM-ów. We wczesnych latach 60. grupa dostawców opracowała wsadowe systemy operacyjne dla swoich systemów komputerowych. Szczególnie godny uwagi okazał się IBSYS, system operacyjny IBM-owskich komputerów 7090/7094, ze względu na jego szerokie oddziałanie na inne systemy.

Głównym pomysłem w prostym schemacie przetwarzania wsadowego jest zastosowanie oprogramowania zwanego **monitorem**. Użytkownik w systemie operacyjnym tego typu nie ma już bezpośredniego dostępu do procesora. Zamiast tego przekazuje zadanie na kartach lub taśmie operatorowi komputera, który grupuje zadania jedno po drugim w pakiet (wsad) i umieszcza go na urządzeniu wejściowym do użycia przez monitor. Każdy program jest skonstruowany w taki sposób, aby po zakończeniu działania następowało przejście z powrotem do monitora, a wówczas monitor automatycznie rozpoczyna ładowanie następnego programu.

Żeby zrozumieć działanie tego schematu, spójrzmy na niego z dwóch perspektyw: monitora i procesora.

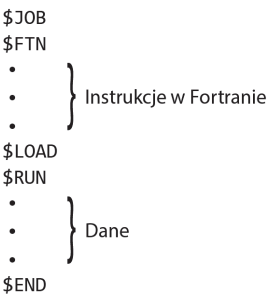
- **Perspektywa monitora.** Monitor nadzoruje ciąg zdarzeń. Aby to było możliwe, większa część monitora musi stale przebywać w pamięci głównej gotowa do działania (rysunek 2.3). Tę część nazywa się **monitorem rezydentnym** (ang. *resident monitor*). Reszta monitora składa się z narzędzi i typowych funkcji, ładowanych jako podprogramy do programu użytkownika na początku każdego zadania, które ich wymaga. Monitor czyta po jednym zadaniu z urządzenia wejściowego (zazwyczaj z czytnika kart lub taśmy magnetycznej). Przeczytane zadanie jest umieszczane w obszarze programu użytkownika i następuje przekazanie do niego sterowania. Gdy zadanie jest ukończone, zwraca sterowanie do monitora, który natychmiast czyta następne zadanie. Wyniki każdego zadania są przesyłane na urządzenie wyjściowe, takie jak drukarka, aby mogły być dostarczone użytkownikowi.
- **Perspektywa procesora.** W którymś momencie procesor wykonuje instrukcje z obszaru pamięci głównej zawierającego monitor. Powodują one przeczytanie następnego zadania do innego obszaru pamięci głównej. Po przeczytaniu zadania procesor napotyka w monitorze rozkaz skoku, polecający mu kontynuowanie działania, zaczynając od początku programu użytkownika. Procesor wykonuje więc instrukcje w programie użytkownika, aż napotka zakończenie lub błąd. Każde z tych zdarzeń powoduje, że następny rozkaz procesor pobiera z programu monitora. Tak więc wyrażenie „sterowanie jest przekazywane do zadania" oznacza po prostu, że procesor obecnie pobiera i wykonuje rozkazy w programie użytkownika, a „przekazanie sterowania z powrotem do monitora" oznacza, że procesor pobiera teraz i wykonuje rozkazy z programu monitora.

Monitor wykonuje funkcję planowania: pakiet zadań tworzy kolejkę, a zadania są wykonywane tak szybko jak to możliwe, bez przestojów między jednym a drugim. Monitor poprawia również **czas przygotowania zadania** (czas instalowania zadania, ang. *job setup time*). Każde zadanie



Rysunek 2.3. Rozplanowanie pamięci monitora rezydentnego

jest zaopatrzone w instrukcje wyrażone w dość prostym **języku sterowania zadaniami** (ang. *job control language* — JCL). Jest to specjalny rodzaj języka programowania używany do instruowania monitora. Prosty przykład to ten, w którym użytkownik przedkłada do wykonania program napisany w języku programowania Fortran i trochę danych do użycia w tym programie. Wszystkie instrukcje Fortranu i dane znajdują się na osobnych kartach perforowanych lub tworzą osobne rekordy na taśmie. Oprócz wierszy fortranowych i wierszy danych zadanie zawiera instrukcje sterujące zadaniem, oznaczane za pomocą początkowego znaku `$`. Ogólna postać takiego zadania wygląda następująco:



Aby wykonać to zadanie, monitor czyta wiersz `$FTN` i ładuje kompilator odpowiedniego języka ze swojej pamięci masowej (zwykle jest to taśma). Kompilator tłumaczy program użytkownika na kod wynikowy, który zostaje przechowany w pamięci głównej lub w pamięci masowej. Jeśli jest przechowany w pamięci głównej, operacja jest określana jako „kompiluj, ładuj i wykonaj”. Jeśli jest przechowany na taśmie, wymagana jest instrukcja `$LOAD`. Ta instrukcja jest czytana przez monitor, który odzyskuje sterowanie po operacji kompilowania. Monitor uruchamia ładowacz, a ten wprowadza program wynikowy do pamięci (na miejsce kompilatora) i przekazuje mu sterowanie. W ten sposób duży segment pamięci głównej może być dzielony między kilka różnych podsystemów, aczkolwiek tylko jeden podsystem może być wykonywany w danym czasie.

Podczas wykonywania programu użytkownika każda instrukcja wejścia powoduje przeczytanie jednego wiersza danych. Instrukcja wejścia w programie użytkownika powoduje wywołanie procedury wejścia będącej częścią SO. Procedura wejścia upewnia się, że program nie czyta przez przypadek wiersza JCL. Gdyby się tak zdarzyło, wystąpi błąd i sterowanie wraca do monitora. Po zakończeniu zadania użytkownika monitor przegląda wiersze wejściowe dopóty, dopóki nie napotka następnej instrukcji JCL. Tak więc system jest chroniony przed programem ze zbyt dużą lub zbyt małą liczbą wierszy danych[5].

Monitor, czyli wsadowy SO, jest po prostu programem komputerowym. Funkcjonuje dzięki zdolności procesora do pobierania rozkazów z różnych fragmentów pamięci głównej, aby na przemian przejmować i oddawać sterowanie. Pożądane są również pewne inne cechy sprzętu:

- **Ochrona pamięci.** Gdy działa program użytkownika, nie wolno mu zmienić obszaru pamięci zawierającej monitor. Jeśli dojdzie do takiej próby, sprzęt procesora powinien wykryć błąd i przekazać sterowanie monitorowi. Monitor może wtedy zaniechać zadania, wydrukować komunikat błędu i załadować następne zadanie.
- **Czasomierz** (ang. *timer*). Czasomierz jest używany do ochrony przed zmonopolizowaniem systemu przez jedno zadanie. Jest ustawiany na początku każdego zadania. Jeśli czas się wyczerpie, program użytkownika jest zatrzymywany i sterowanie wraca do monitora.
- **Instrukcje uprzywilejowane.** Pewne instrukcje z poziomu maszynowego są oznaczone jako uprzywilejowane i mogą być wykonywane tylko przez monitor. Jeśli procesor napotka taki rozkaz podczas wykonywania programu użytkownika, wystąpi błąd powodujący przekazanie sterowania do monitora. Do instrukcji uprzywilejowanych należą instrukcje wejścia-wyjścia, tak więc monitor zachowuje kontrolę nad wszystkimi urządzeniami wejścia-wyjścia. Chroni to na przykład przed niezamierzonym przeczytaniem przez program użytkownika instrukcji sterujących z następnego zadania. Jeśli program użytkownika chce wykonać operację wejścia-wyjścia, musi zgłosić zapotrzebowanie na jej wykonanie w monitorze.
- **Przerwania.** Wczesne modele komputerów nie miały tej zdolności. Ta cecha sprawia, że SO staje się bardziej elastyczny w przekazywaniu sterowania do programów użytkowych i jego (od nich) odzyskiwaniu.

Rozważania nad ochroną pamięci i instrukcjami uprzywilejowanymi naprowadziły na pomysł trybów działania. Program użytkownika pracuje w **trybie użytkownika** (ang. *user mode*), w którym pewne obszary pamięci są chronione przed wykorzystaniem przez użytkownika, a pewne instrukcje są zabronione. Monitor działa w trybie systemu lub — jak to się zwykło nazywać — w **trybie jądra** (ang. *kernel mode*), w którym instrukcje uprzywilejowane mogą być wykonywane, a dostęp do chronionych obszarów pamięci jest dozwolony.

Oczywiście system operacyjny można zbudować bez tych właściwości. Jednak dostawcy komputerów szybko pojęli, że w rezultacie powstawał chaos, więc nawet stosunkowo proste systemy wsadowe były wyposażane w te cechy sprzętowe.

We wsadowym SO czas procesora jest naprzemiennie przydzielany na wykonywanie programów użytkowników i działanie monitora. Odbywa się to za cenę dwóch wyrzeczeń. Pewna partia

[5] Chyba że użytkownik lub operator zapomniał wstawić kartę `$JCL`. Aby ustrzec się przed „połykaniem" następnego zadania w charakterze „danych" przez poprzedni program, w używanym w Polsce na maszynach ODRA 1305 ICL-owskim systemie GEORGE 3 stosowano dodatkową kartę z tak zwanym ogranicznikiem uniwersalnym, której dodanie na końcu każdego zadania było rutynowym obowiązkiem operatora — *przyp. tłum.*

pamięci głównej jest obecnie przekazana we władanie monitorowi i część czasu procesora jest konsumowana przez monitor. I jedno, i drugie stanowi swoistą daninę. Mimo to proste systemy wsadowe poprawiają wykorzystanie komputera.

Wieloprogramowe systemy wsadowe

Nawet z automatycznym podawaniem zadań umożliwianym przez prosty, wsadowy SO procesor jest często bezczynny. Problemem są urządzenia wejścia-wyjścia, które są wolniejsze w porównaniu z procesorem. Na rysunku 2.4 przedstawiono szczegóły reprezentatywnego obliczenia. Dotyczy ono programu, który przetwarza plik rekordów i wykonuje średnio 100 instrukcji maszynowych na rekord. W tym przykładzie komputer spędza ponad 96% czasu w oczekiwaniu na zakończenie przez urządzenia wejścia-wyjścia przesyłania danych do i z pliku. Na rysunku 2.5a przedstawiono tę sytuację: mamy tu do czynienia z jednym programem, co jest określane jako **jednoprogramowość** (ang. *uniprogramming*). Procesor działa przez pewien czas, aż do napotkania operacji wejścia-wyjścia. Wówczas musi zaczekać, aż instrukcja wejścia-wyjścia zakończy działanie, i dopiero wtedy może kontynuować pracę.

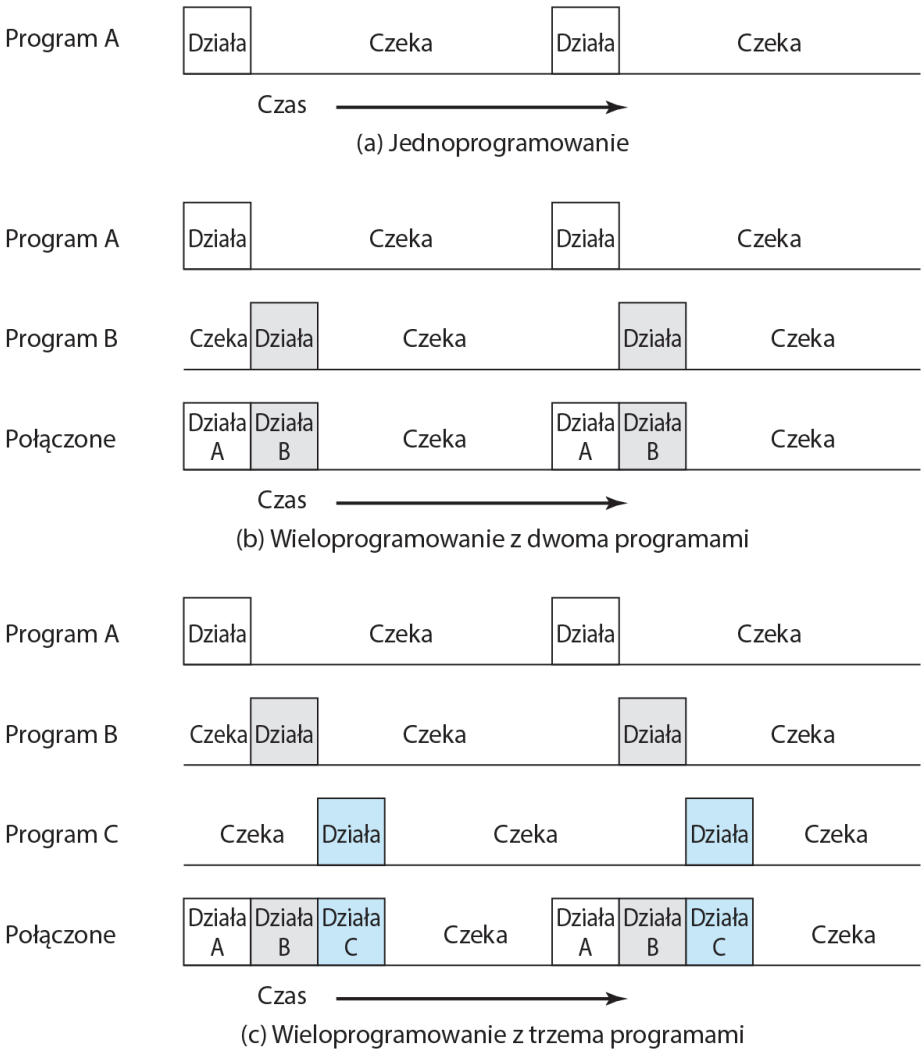
Czytanie jednego rekordu z pliku	15 µs
Wykonanie 100 instrukcji	1 µs
Zapisanie jednego rekordu do pliku	15 µs
Ogółem	31 µs

$$\text{Procent wykorzystania CPU} = \frac{1}{31} = 0{,}032 = 3{,}2\%$$

Rysunek 2.4. Przykład wykorzystania systemu

Tej nieefektywności można uniknąć. Wiemy, że pamięci musi wystarczyć na utrzymywanie SO (monitora rezydentnego) i jednego programu użytkownika. Załóżmy, że mamy miejsce na SO i dwa programy użytkownika. Kiedy jedno zadanie musi czekać na wejście-wyjście, procesor może się przełączyć do drugiego zadania, które być może nie czeka na wejście-wyjście (zob. rysunek 2.5b). Mało tego — możemy powiększyć pamięć, aby pomieściła trzy, cztery lub więcej programów, i dokonywać przełączeń między nimi wszystkimi (zob. rysunek 2.5c). To podejście jest nazywane **wieloprogramowością** (ang. *multiprogramming*) lub **wielozadaniowością** (ang. *multitasking*). Stanowi ono centralny motyw nowoczesnych systemów operacyjnych.

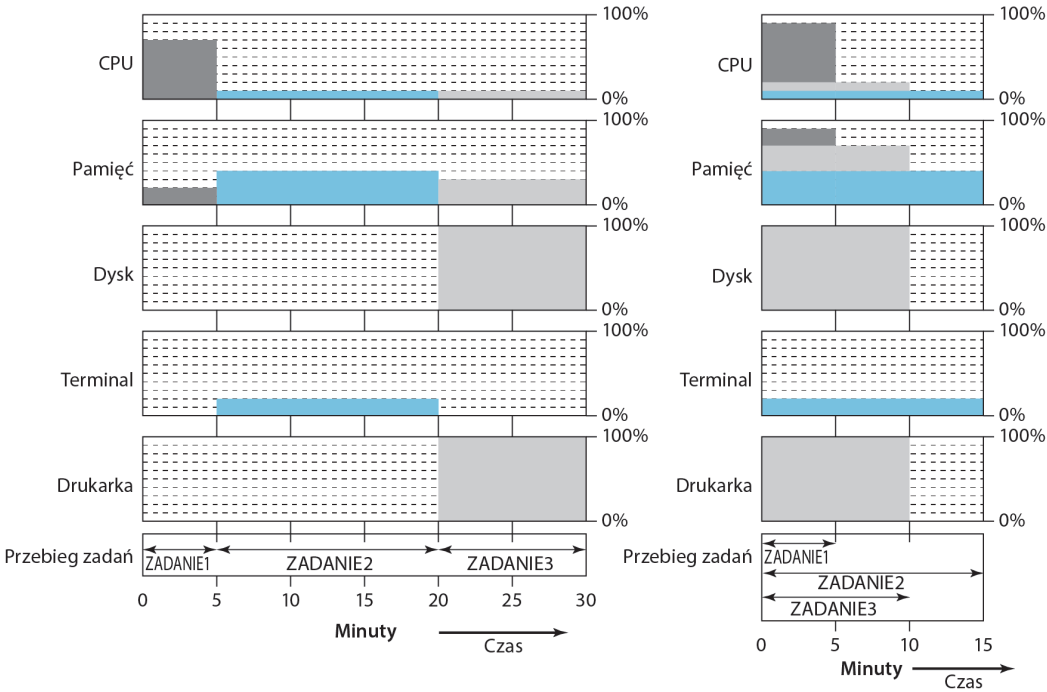
Aby zilustrować korzyści płynące z wieloprogramowości, podamy prosty przykład. Rozważmy komputer mający 250 MB dostępnej pamięci (nieużywanej przez system operacyjny), dysk, terminal i drukarkę. W tej samej chwili do wykonania trafiają trzy programy: ZADANIE1, ZADANIE2 i ZADANIE3 o charakterystykach jak w tabeli 2.1. Zakładamy minimalne zapotrzebowanie na procesor w zadaniach ZADANIE1 i ZADANIE3 i stałe używanie dysku i drukarki przez ZADANIE3. W prostym środowisku wsadowym zadania te będą wykonywane po kolei. Wobec tego ZADANIE1 zakończy się po 5 minutach, ZADANIE2 musi czekać przez 5 minut i potem będzie działać przez 15 minut. ZADANIE3 rozpocznie się po 20 minutach i zakończy w 30 minucie od chwili przedłożenia do wykonania. Średnie wykorzystanie zasobów, przepustowość i czas odpowiedzi pokazano w kolumnie jednoprogramowości tabeli 2.2. Wykorzystanie urządzeń przedstawiono na rysunku 2.6a. Widać wyraźnie duże niewykorzystanie wszystkich zasobów uśrednione w wymaganym 30-minutowym odcinku czasu.



Rysunek 2.5. Przykład wieloprogramowości

Tabela 2.1. Przykładowe charakterystyki wykonania programów

	ZADANIE1	ZADANIE2	ZADANIE3
Rodzaj zadania	Dużo obliczeń	Dużo we-wy	Dużo we-wy
Czas trwania	5 minut	15 minut	10 minut
Zapotrzebowanie na pamięć	50 MB	100 MB	75 MB
Dysk potrzebny?	Nie	Nie	Tak
Terminal potrzebny?	Nie	Tak	Nie
Drukarka potrzebna?	Nie	Nie	Tak



Rysunek 2.6. Histogramy wykorzystania

Tabela 2.2. Wpływ wieloprogramowości na wykorzystanie zasobów

	Jednoprogramowość	Wieloprogramowość
Użycie procesora	20%	40%
Użycie pamięci	33%	67%
Użycie dysku	33%	67%
Użycie drukarki	33%	67%
Zużyty czas	30 minut	15 minut
Przepustowość	6 zadań/godzinę	12 zadań/godzinę
Średni czas odpowiedzi	18 minut	10 minut

Załóżmy teraz, że zadania są wykonywane współbieżnie w systemie wieloprogramowym. Ponieważ między zadaniami jest mało rywalizacji o zasoby, wszystkie trzy mogą działać prawie w minimalnym czasie, koegzystując w komputerze z innymi (przyjmując, że ZADANIE2 i ZADANIE3 mają przydzielone dość czasu procesora, aby utrzymać w ruchu swoje operacje wejścia i wyjścia). ZADANIE1 nadal będzie wymagało 5 minut do zakończenia, lecz pod koniec tego czasu ZADANIE2 będzie już w jednej trzeciej wykonane, a ZADANIE3 będzie wykonane do połowy. Wszystkie trzy zadania zakończą się po 15 minutach. Poprawa jest ewidentna, gdy przyjrzymy się kolumnie wieloprogramowości w tabeli 2.2, otrzymanej z histogramu przedstawionego na rysunku 2.6b.

Podobnie jak w prostym systemie wsadowym, wieloprogramowy system wsadowy musi mieć wsparcie ze strony pewnych właściwości sprzętu. Najważniejszą dodatkową cechą przydatną w wieloprogramowości jest sprzęt umożliwiający przerwania pochodzące z wejścia-wyjścia i od DMA (bezpośredniego dostępu do pamięci). Mając sterowane przerwaniami operacje wejścia-wyjścia lub DMA, procesor może wydać polecenie wejścia-wyjścia w jednym zadaniu i kontynuować wykonywanie innego zadania w czasie wykonywania wejścia-wyjścia przez sterownik urządzenia. Gdy operacja wejścia-wyjścia dobiegnie końca, działanie procesora jest przerywane i sterowanie przekazuje się do programu obsługi przerwania w SO. Po obsłużeniu przerwania SO przekaże sterowanie innemu zadaniu.

Wieloprogramowe systemy operacyjne są dosyć skomplikowane w porównaniu z systemami wykonującymi jeden program, czyli **jednoprogramowymi**. Aby mieć pewną liczbę zadań gotowych do działania, trzeba je utrzymywać w pamięci, co wymaga jakiejś formy **zarządzania pamięcią**. Ponadto jeśli kilka zadań jest gotowych do działania, procesor musi decydować, które ma być wykonywane, a ta decyzja wymaga jakiegoś algorytmu planowania. Te zagadnienia będą omówione dalej w tym rozdziale.

Systemy z podziałem czasu

Z zastosowaniem wieloprogramowości **przetwarzanie wsadowe** (ang. *batch processing*) może być całkiem sprawne. Jednak w wielu zadaniach jest pożądany tryb, w którym użytkownik pozostaje w bezpośredniej interakcji z komputerem. Rzeczywiście, w niektórych zadaniach, jak na przykład przetwarzanie transakcji, tryb interakcyjny ma duże znaczenie.

Obecnie wymaganie na obliczenia interaktywne można zaspokoić — i często tak się dzieje — stosując wydzielony komputer osobisty lub stację roboczą. W latach 60. nie było tej możliwości. Komputery były wówczas w większości wielkie (gabarytowo — *przyp. tłum.*) i kosztowne. Wymyślono więc podział czasu.

Skoro wieloprogramowość umożliwia obsługiwanie przez procesor wielu zadań wsadowych, postępując analogicznie, można ją zaprząc do obsługiwania wielu zadań interakcyjnych. W tym drugim przypadku nazywa się tę technikę **podziałem czasu** (ang. *time sharing*), ponieważ czas procesora jest dzielony między wielu użytkowników. Do systemu z podziałem czasu ma jednoczesny dostęp wielu użytkowników za pośrednictwem terminali, a SO przeplata wykonywanie programów każdego z nich w krótkotrwałych dostępach do procesora, czyli kwantach obliczeń. Jeśli więc w danym czasie n użytkowników domaga się obsługi, to każdy z nich będzie miał do dyspozycji średnio tylko $1/n$ efektywnej mocy obliczeniowej komputera, nie wliczając nakładów wymaganych przez SO. Zważywszy jednak na stosunkowo wolny czas reakcji człowieka, czas odpowiedzi w dobrze zaprojektowanym systemie powinien być podobny jak w przypadku użycia wydzielonego komputera.

Zarówno w przetwarzaniu wsadowym, jak i w podziale czasu jest wykorzystywana wieloprogramowość. Zasadnicze różnice zestawiono w tabeli 2.3.

Tabela 2.3. Porównanie wieloprogramowania wsadowego z podziałem czasu

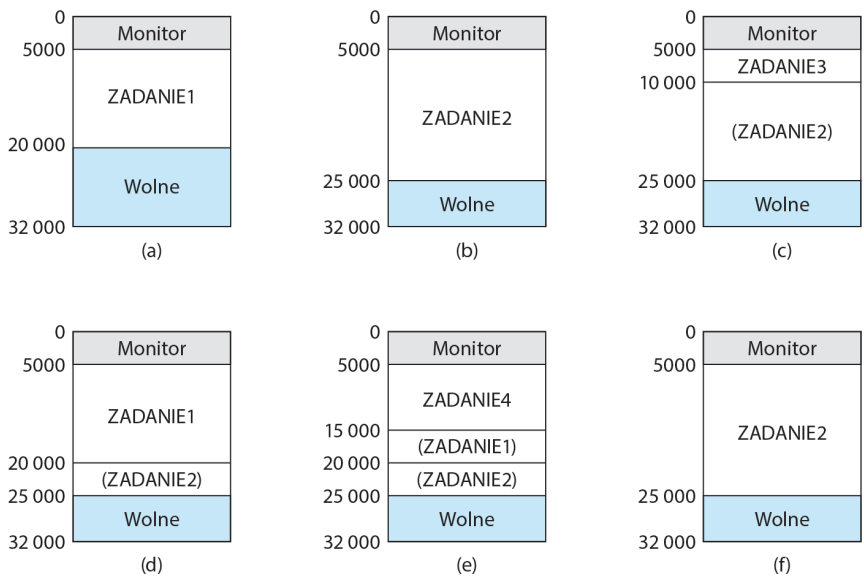
	Wieloprogramowość wsadowa	Podział czasu
Główny cel	Maksymalizacja wykorzystania procesora	Minimalizacja czasu odpowiedzi
Źródło dyrektyw dla systemu operacyjnego	Polecenia w języku sterowania zadaniami podane w opisie zadania	Polecenia wprowadzane z terminala

Jednym z pierwszych skonstruowanych systemów operacyjnych z podziałem czasu był Compatible Time-Sharing System (CTSS) [CORB62], opracowany w MIT przez grupę o nazwie Project MAC (Machine-Aided Cognition, z ang. percepcja wspomagana maszynowo, albo Multiple-Access Computers, z ang. komputery z wielodostępem). Początkowo, w 1961 roku, system ten został utworzony dla komputerów IBM 709, a potem przeniesiono go na maszyny IBM 7094.

W porównaniu z późniejszymi systemami CTSS był prymitywny. Działał na komputerze z 32 000 słów 36-bitowych w pamięci głównej, z rezydentnym monitorem zajmującym 5000 z nich. Kiedy miało nastąpić przekazanie sterowania do interakcyjnego użytkownika, program i dane użytkownika były ładowane do pozostałych 27 000 słów pamięci głównej. Program był zawsze ładowany tak, aby zaczynał się od słowa 5000. Upraszczało to zarówno monitor, jak i zarządzanie pamięcią. Zegar systemowy generował przerwania w przybliżeniu co 0,2 sekundy. Przy każdym przerwaniu zegarowym SO przejmował sterowanie i mógł przydzielić procesor innemu użytkownikowi. To postępowanie nazywa się **kwantowaniem czasu** (dosł. plastrowaniem czasu, ang. *time slicing*). Dzięki temu w regularnych odstępach czasu bieżący użytkownik mógł być wywłaszczony, a inny użytkownik załadowany. Aby zachować stan programu starego użytkownika do późniejszego wznowienia, przed wczytaniem programów i danych nowego użytkownika programy i dane starego użytkownika były zapisywane na dysku. Gdy później ten program otrzymywał następną rundę, kod i dane starego użytkownika sprowadzano z powrotem do pamięci głównej.

Aby zminimalizować ruch na dysku, pamięć użytkownika była wypisywana tylko wówczas, gdy wchodzący program mógł ją zapisać na nowo. Zasadę tę ilustruje rysunek 2.7. Załóżmy, że mamy czterech użytkowników interakcyjnych z następującymi zapotrzebowaniami na pamięć (w słowach):

- ZADANIE1 — 15 000;
- ZADANIE2 — 20 000;
- ZADANIE3 — 5000;
- ZADANIE4 — 10 000.



Rysunek 2.7. Działanie CTSS

Na początku monitor ładuje ZD1[6] i przekazuje mu sterowanie (rysunek 2.7a). Później monitor postanawia przekazać sterowanie do ZD2. Ponieważ ZD2 wymaga więcej pamięci niż ZD1, ZD1 musi być najpierw wypisane, dopiero potem można załadować ZD2 (rysunek 7.2b). Dalej jest ładowane do wykonania ZD3. Ponieważ jednak ZD3 jest mniejsze niż ZD2, część ZD2 może pozostać w pamięci, zmniejszając czas zapisu na dysku (rysunek 2.7c). Następnie monitor decyduje się oddać sterowanie z powrotem ZD1. Na okoliczność ponownego ładowania ZD1 trzeba uprzednio wypisać dodatkową porcję ZD2 (rysunek 2.7d). Przy ładowaniu ZD4 część pozostającego w pamięci ZD1 i fragment ZD2 pozostają zachowane (rysunek 7.2e). Jeśli teraz zostanie uaktywnione ZD1 lub ZD2, będzie potrzebne tylko częściowe ładowanie. W tym przykładzie jako następne działa ZD2. Wymaga to przepisania na dysk ZD4 i pozostałej porcji ZD1, po czym będzie można wczytać brakującą porcję ZD2 (rysunek 2.7f).

Podejście CTSS jest prymitywne w porównaniu ze współczesnym podziałem czasu, lecz było skuteczne. Dzięki jego skrajnej prostocie można było zminimalizować rozmiar monitora. Ponieważ zadanie ładowano zawsze na to samo miejsce w pamięci, nie trzeba było stosować technik relokacji podczas ładowania (co jest omówione dalej). Technika wypisywania tylko tego, co konieczne, minimalizowała aktywność dysku. Wykonywany na komputerze 7094 system CTSS obsługiwał maksymalnie 32 użytkowników.

Podział czasu i wieloprogramowość rodzą w SO mnóstwo nowych problemów. Jeśli wiele zadań znajduje się w pamięci, to trzeba je chronić przed wzajemnym oddziaływaniem, na przykład przed zmianą danych jednego przez drugie. W obecności wielu użytkowników należało chronić system plików, aby dostęp do konkretnego pliku mieli tylko upoważnieni użytkownicy. Należało uwzględnić rywalizację o zasoby, takie jak drukarki i urządzenia pamięci masowej. Te i inne problemy z możliwymi rozwiązaniami będą się pojawiać w tym podręczniku.

2.3. NAJWAŻNIEJSZE OSIĄGNIĘCIA

Systemy operacyjne są jednymi z najbardziej skomplikowanych rodzajów oprogramowania, jakie kiedykolwiek zbudowano. Odzwierciedla to dążenie do zaspokojenia trudnych i niekiedy przeciwstawnych celów wygody, wydajności i zdolności ewolucyjnej. W pracy [DENN80a] wysunięto cztery zasadnicze, teoretyczne kierunki rozwoju systemów operacyjnych:

- procesy,
- zarządzanie pamięcią,
- ochrona informacji i bezpieczeństwo,
- planowanie i zarządzanie zasobami.

Każdy kierunek jest scharakteryzowany przez wytyczne, lub abstrakcje, wykształcone w celu rozwiązania trudnych problemów praktycznych. Te cztery obszary rozpatrywane łącznie obejmują wiele kluczowych zagadnień projektowych i implementacyjnych nowoczesnych systemów operacyjnych. Przedstawiony w tym podrozdziale krótki przegląd tych czterech obszarów zarysowuje znaczną część zagadnień poruszonych w podręczniku.

[6] Dla wygody skracamy dalej ZADANIE do ZD — *przyp. tłum.*

Proces

Centralną koncepcją projektowania systemów operacyjnych jest **proces** (ang. *process*). Po raz pierwszy terminu tego użyli projektanci systemu Multics w latach 60. XX wieku [DALE68]. Jest on nieco ogólniejszy niż *zadanie* (ang. *job*). Sformułowano wiele definicji terminu *proces*, między innymi takie:

- program w trakcie wykonywania;
- egzemplarz (ang. *instance*) programu działającego na komputerze;
- obiekt (pewna całość, ang. *entity*), któremu można przydzielić procesor w celu wykonywania;
- jednostka (ang. *unit*) aktywności charakteryzująca się pojedynczym, sekwencyjnym wątkiem wykonania, bieżącym stanem i przydzielonym zbiorem zasobów systemu.

Pojęcie to powinno stać się jaśniejsze w miarę kontynuowania przez nas tematu.

Trzy główne fronty rozwoju systemów komputerowych doprowadziły do powstania problemów koordynacji i synchronizacji, które przyczyniły się do ukształtowania koncepcji procesu: wieloprogramowość działań wsadowych, podział czasu i systemy transakcji przebiegających w czasie rzeczywistym. Jak widzieliśmy, wieloprogramowość zaprojektowano po to, aby utrzymywać procesor i urządzenia wejścia-wyjścia (w tym urządzenia pamięci masowej) w jednoczesnym działaniu w celu uzyskania maksymalnej wydajności. Zasadniczy mechanizm przedstawia się następująco: w reakcji na sygnały zakończenia operacji wejścia-wyjścia procesor jest przełączany między różnymi programami przebywającymi w pamięci głównej.

Drugim frontem rozwoju był uniwersalny podział czasu. Tutaj podstawowym celem projektowym jest żywe reagowanie na potrzeby indywidualnego użytkownika i dodatkowo — ze względów ekonomicznych — zdolność jednoczesnego obsługiwania wielu użytkowników. Te cele są ze sobą zgodne dzięki stosunkowo długim czasom reakcji użytkownika. Typowy użytkownik potrzebuje na przykład średnio 2 sekundy czasu przetwarzania na minutę, więc blisko 30 takich użytkowników powinno móc wspólnie korzystać z tego samego systemu bez zauważalnych wzajemnych utrudnień. Oczywiście w takich kalkulacjach trzeba uwzględnić nakład pracy ze strony SO[7].

Trzecią ważną linią rozwoju były systemy przetwarzania transakcji w czasie rzeczywistym. W tym przypadku pewna liczba użytkowników wprowadza zapytania lub uaktualnienia do bazy danych. Przykładem jest system rezerwacji linii lotniczych. Zasadniczą różnicą między systemem przetwarzania transakcji a systemem z podziałem czasu jest ograniczenie tego pierwszego do jednego lub kilku zastosowań, podczas gdy użytkownicy systemu z podziałem czasu mogą się angażować w opracowywanie programów, wykonywanie zadań i korzystanie z różnych aplikacji. W obu przypadkach czas reakcji systemu ma kapitalne znaczenie.

Podstawowym narzędziem programistów systemowych opracowujących wczesne interakcyjne, wieloprogramowe i dostępne dla wielu użytkowników systemy było przerwanie. Działanie każdego zadania mogło być zawieszone wskutek wystąpienia określonego zdarzenia w rodzaju zakończenia operacji wejścia-wyjścia. Procesor mógł przechować pewien rodzaj kontekstu (np. licznik programu i inne rejestry) i przejść do procedury obsługi przerwania, która potrafiła określić naturę przerwania i przetworzyć je, po czym następowało wznowienie przerwanego zadania użytkownika lub podjęcie innego zadania.

[7] I są to kalkulacje w aspekcie liczbowym historyczne, na miarę mocy obliczeniowej komputerów z lat 60. — *przyp. tłum.*

Zaprojektowanie oprogramowania systemowego koordynującego te różne czynności okazało się nadzwyczaj trudne. Wobec wielu zadań w toku realizacji, z których każde wymagało wykonania wielu kroków po kolei, przeanalizowanie wszystkich potencjalnych kombinacji ciągów zdarzeń stało się niemożliwe. Wskutek braku systematycznych środków koordynacji i kooperacji aktywności programiści uciekali się do doraźnych metod opartych na ich rozumieniu środowiska, które SO miał kontrolować. Wysiłki te były narażone na subtelne błędy programowania, których skutki dawały się zauważyć tylko w stosunkowo rzadkich układach zdarzeń. Były to błędy trudne do diagnozowania, gdyż należało je odróżnić od błędów w oprogramowaniu aplikacji i błędów sprzętowych. Nawet w wypadku wykrycia błędu niełatwo było ustalić jego przyczynę z powodu wielkich trudności w precyzyjnym odtworzeniu warunków jego powstania. Najogólniej biorąc, są cztery główne przyczyny takich błędów [DENN80a]:

- **Niewłaściwa synchronizacja.** Niejednokrotnie zdarza się, że procedura musi być zawieszona, aby zaczekać na zdarzenie w innej części systemu. Na przykład program, który zapoczątkowuje na wejściu-wyjściu operację czytania, musi czekać przed podjęciem dalszego działania, aż dane staną się dostępne w buforze. W takich razach jest potrzebny sygnał z innej procedury. Niewłaściwe zaprojektowanie mechanizmu sygnalizacji może prowadzić do utraty sygnałów lub ich podwajania.
- **Wadliwe wzajemne wykluczanie.** Często się zdarza, że więcej niż jeden użytkownik lub program będzie usiłował skorzystać ze wspólnego zasobu w tym samym czasie. Na przykład dwóch użytkowników mogłoby próbować jednocześnie redagować ten sam plik. Musi istnieć jakiś rodzaj mechanizmu wzajemnego wykluczania, który pozwala w danej chwili uaktualniać plik tylko jednemu programowi. Poprawność realizacji takiego wzajemnego wykluczania jest trudna do zweryfikowania z uwzględnieniem wszystkich możliwych ciągów zdarzeń.
- **Nieokreślone działanie programu.** Wyniki konkretnego programu powinny w normalnych okolicznościach zależeć tylko od danych wejściowych tego programu, nie zaś od działań innych programów we wspólnie użytkowanym systemie. Jednak gdy programy wspólnie korzystają z pamięci i ich wykonanie jest przeplatane przez procesor, mogłyby oddziaływać na siebie, zapisując wspólne obszary pamięci w nieokreślony sposób. Tak więc kolejność planowania poszczególnych programów mogłaby wpłynąć na wyniki konkretnego programu.
- **Zakleszczenia.** Może się zdarzyć, że dwa lub więcej programów zawiesi się, oczekując na siebie wzajemnie. Na przykład dwa programy mogą potrzebować dwóch urządzeń zewnętrznych do wykonania pewnej operacji (np. do kopiowania z dysku na taśmę). Jeden z programów przejmuje kontrolę nad jednym z urządzeń, a drugi program rozporządza drugim urządzeniem. Każdy z programów czeka, aby drugi zwolnił potrzebne urządzenie. Takie zakleszczenie może zależeć od przypadkowej kolejności przydziału i zwalniania zasobów.

Aby poradzić sobie z tymi problemami, należy systematycznie nadzorować poszczególne programy wykonywane przez procesor i odpowiednio nimi sterować. Podstawą takich metod jest pojęcie procesu. Możemy przyjąć, że na proces składają się trzy elementy:

1. Wykonywalny program.
2. Związane z programem potrzebne mu dane (zmienne, obszar roboczy, bufory itp.).
3. Kontekst wykonywania programu.

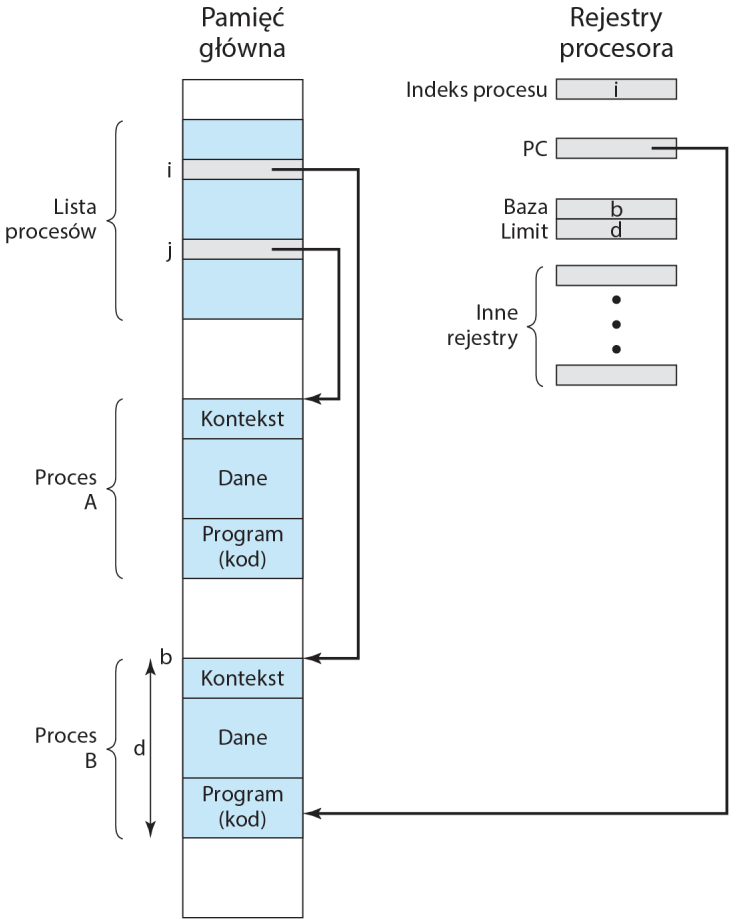
Ostatni element ma istotne znaczenie. **Kontekst wykonywania** (ang. *execution context*), inaczej: **stan procesu** (ang. *process state*), stanowią wewnętrzne dane, dzięki którym SO potrafi nadzorować proces i nim sterować. Te wewnętrzne informacje są oddzielone od procesu, ponieważ jako informacje systemowe są mu zabronione. Kontekst obejmuje wszystkie informacje potrzebne systemowi do zarządzania procesem, a procesorowi do właściwego wykonywania procesu. Do kontekstu należy zawartość różnych rejestrów procesora, takich jak licznik programu i rejestry danych. Zawiera on również informacje wykorzystywane przez SO, jak priorytet procesu oraz to, czy proces czeka na zakończenie jakiejś operacji wejścia-wyjścia.

Na rysunku 2.8 pokazano sposób, w jaki można zarządzać procesami. Dwa procesy, A i B, rezydują w dwu obszarach pamięci głównej, to znaczy każdemu z nich jest przydzielony blok pamięci, który zawiera program, dane i informacje dotyczące kontekstu. Każdy proces jest zapisany na liście procesów budowanej i utrzymywanej przez SO. Lista procesów ma po jednym wpisie dla każdego procesu, zawierającym wskaźnik do położenia bloku pamięci z procesem. Wpis ten może również zawierać część lub całość kontekstu wykonywania procesu. Pozostała część kontekstu wykonywania jest pamiętana gdzie indziej, być może w samym procesie (jak pokazano na rysunku 2.8) lub często w osobnym regionie pamięci. Rejestr indeksu procesu zawiera indeks położenia na liście procesów wpisu procesu aktualnie sterującego procesorem. Licznik programu (PC) wskazuje następny rozkaz do wykonania w danym procesie. Rejestry bazowy i graniczny określają obszar pamięci zajmowany przez proces. Rejestr bazowy zawiera adres początkowy tego obszaru, a rejestr graniczny jego rozmiar (w bajtach lub słowach). Licznik programu i wszystkie odniesienia do danych są interpretowane względem rejestru bazowego i nie mogą przekroczyć wartości w rejestrze granicznym. Zapobiega to wzajemnemu zaburzaniu się procesów.

Na rysunku 2.8 indeks procesu (a bezpośrednio licznik programu, PC) wskazuje, że wykonywany jest proces B. Proces A był wykonywany wcześniej, lecz chwilowo został przerwany. Zawartości wszystkich rejestrów w momencie przerwania A zostały zapisane w jego kontekście wykonywania. System operacyjny może później przełączyć procesy i wznowić wykonywanie procesu A. Przełączenie procesów składa się z przechowania kontekstu B i odtworzenia kontekstu A. Gdy w liczniku programu zostanie umieszczona wartość wskazująca miejsce w obszarze programu A, proces A automatycznie podejmie działanie.

Tak więc proces jest urzeczywistniany w postaci struktury danych. Proces może być wykonywany albo może czekać na wykonanie. W każdym momencie cały **stan** procesu jest zawarty w jego kontekście. Ta struktura umożliwia opracowywanie skutecznych sposobów zapewniania koordynacji i kooperacji między procesami. Można projektować i wprowadzać do systemu nowe właściwości (np. priorytety), rozszerzając kontekst o dowolne nowe informacje niezbędne do ich uzyskania. Wielokrotnie w tej książce napotkamy przykłady wykorzystania struktury procesu do rozwiązywania problemów wynikających z wieloprogramowości i dzielenia zasobów.

Na koniec zajmiemy się tutaj krótko koncepcją **wątku** (ang. *thread*). Pojedynczy proces, który ma przydzielone pewne zasoby, można w istocie podzielić na wiele współbieżnych wątków, które kooperatywnie wykonują pracę procesu. To prowadzi do nowego poziomu równoległych działań, którymi trzeba zarządzać za pomocą metod sprzętowych i programowych.



Rysunek 2.8. Typowa implementacja procesu

Zarządzanie pamięcią

Potrzeby użytkowników mogą być zaspokajane najlepiej przez środowisko obliczeniowe umożliwiające programowanie modularne i elastyczne użytkowanie danych. Administratorzy systemów potrzebują sprawnego i uporządkowanego zarządzania przydzielaniem pamięci. Aby spełnić te wymagania, na systemie operacyjnym spoczywa pięć podstawowych powinności dotyczących zarządzania pamięcią:

1. **Izolowanie procesów.** System operacyjny musi powstrzymywać niezależne procesy przed ingerencją w obszary pamięci należące do innych procesów, zarówno w odniesieniu do instrukcji, jak i danych.
2. **Automatyczny przydział i zarządzanie.** Programy powinny być lokowane w hierarchii pamięci dynamicznie, stosownie do potrzeb. Przydział powinien być przezroczysty dla osoby programującej. Programista jest więc uwalniany od problemów związanych z ograniczeniami pamięci, a SO może zyskiwać na efektywności dzięki przydzielaniu pamięci zadaniom tylko stosownie do zapotrzebowań.

3. **Umożliwianie programowania modularnego.** Programiści powinni móc definiować moduły programu, dynamicznie je tworzyć i likwidować oraz zmieniać ich rozmiary.
4. **Ochrona i kontrolowanie dostępu.** Współużytkowanie pamięci na każdym poziomie hierarchii pamięci daje możliwość zaadresowania przez program pamięci w obrębie innego programu. Jest to pożądane, gdy pewne aplikacje wymagają dzielenia pamięci w sensie wspólnego korzystania z tego samego jej obszaru. W innych przypadkach rodzi to zagrożenie integralności programów, a nawet samego SO. System musi zatem udostępniać porcje pamięci w różny sposób różnym użytkownikom.
5. **Magazynowanie długoterminowe.** Wiele programów użytkowych wymaga środków długotrwałego przechowywania informacji po wyłączeniu zasilania komputera.

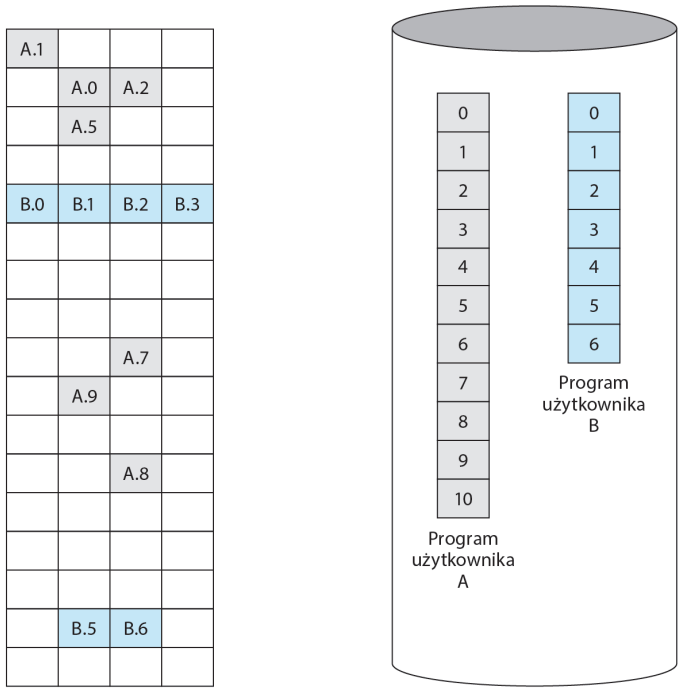
Systemy operacyjne na ogół spełniają te wymagania za pomocą pamięci wirtualnej i udogodnień zawartych w systemie plików. System plików urzeczywistnia pamięć długoterminową, w której informacje są przechowywane w ponazywanych obiektach, zwanych plikami. Plik stanowi wygodną koncepcję dla programisty i jest użyteczną jednostką kontrolowania dostępu i ochrony w SO.

Pamięć wirtualna (ang. *virtual memory*) jest udogodnieniem, które umożliwia programom adresowanie pamięci w aspekcie logicznym, bez zważania na ilość pamięci głównej dostępnej fizycznie. Pamięć wirtualną wymyślono po to, aby spełnić wymaganie jednoczesnej obecności w pamięci głównej zadań wielu użytkowników, co umożliwia unikanie przerw między wykonaniami kolejnych procesów, gdy jeden proces jest zapisywany w pamięci drugorzędnej, a następny jest wczytywany. Ponieważ procesy różnią się rozmiarami, ich ścisłe upakowywanie w pamięci głównej sprawia trudności, gdy procesor jest przełączany od jednego procesu do drugiego. Wprowadzono więc systemy stronicowania, które umożliwiają rozmieszczenie procesów w pewnej liczbie stałej długości bloków, zwanych **stronami**. Program odwołuje się do słowa za pomocą **adresu wirtualnego** składającego się z numeru strony i odległości na stronie. Każda strona procesu może być ulokowana gdziekolwiek w pamięci głównej[8]. System stronicowania realizuje dynamiczne odwzorowanie między adresem wirtualnym używanym w programie a **adresem rzeczywistym**, czyli fizycznym adresem w pamięci głównej.

W obecności sprzętu realizującego dynamiczne odwzorowywanie następnym logicznym krokiem było wyeliminowanie wymagania, aby wszystkie strony procesu jednocześnie rezydowały w pamięci. Wszystkie strony procesu są utrzymywane na dysku. Podczas działania procesu w pamięci głównej znajdują się niektóre z jego stron. Jeśli wystąpi odwołanie do strony nieobecnej w pamięci głównej, sprzęt zarządzający pamięcią wykrywa to i w uzgodnieniu z SO przechodzi do załadowania brakującej strony. Taki schemat jest określany mianem **pamięci wirtualnej**. Przedstawiono go na rysunku 2.9.

Sprzęt procesora wraz z systemem operacyjnym dostarcza użytkownikowi „procesora wirtualnego", który ma dostęp do pamięci wirtualnej. Ta pamięć może być liniową przestrzenią adresów lub kolekcją segmentów będących różnej długości blokami kolejnych adresów. W obu przypadkach instrukcje języka programowania mogą się odnosić do miejsc w programie i do danych w obszarze pamięci wirtualnej. Izolowanie procesów można osiągnąć przez danie każdemu procesowi jednoznacznie określonej, niezachodzącej na inną pamięci wirtualnej. Dzielenie pamięci można uzyskać przez zachodzące na siebie porcje dwu pamięci wirtualnych. Pliki są utrzymywane w pamięci

[8] A dokładniej: musi trafić do którejś z tak zwanych ramek, na które pamięć stronicowana jest sprzętowo dzielona — *przyp. tłum.*

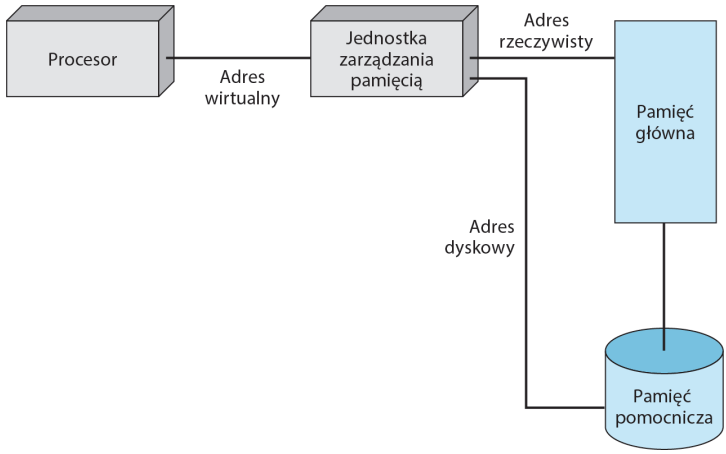


Rysunek 2.9. Zasady pamięci wirtualnej

długoterminowej. Pliki i fragmenty plików można kopiować do pamięci wirtualnej, aby można było nimi manipulować w programach.

Na rysunku 2.10 uwidoczniono kwestie adresowania w schemacie pamięci wirtualnej. Cała pamięć składa się z bezpośrednio adresowalnej (za pomocą instrukcji maszynowych) pamięci głównej i wolniej działającej pamięci pomocniczej, dostępnej pośrednio przez ładowanie bloków do pamięci głównej. Między procesor a pamięć jest wstawiony sprzęt tłumaczący adresy (jednostka zarządzania pamięcią). Odwołania do pamięci w programach są wyrażone za pomocą adresów wirtualnych odwzorowywanych na rzeczywiste adresy pamięci głównej. Jeśli wystąpi odwołanie do adresu wirtualnego nieobecnego w pamięci głównej, porcja zawartości rzeczywistej pamięci jest wyprowadzana do pamięci pomocniczej, a potrzebny blok danych jest sprowadzany do pamięci głównej[9]. W tym czasie proces, który wygenerował odwołanie do takiego adresu, musi być zawieszony. Osoba projektująca SO musi opracować mechanizm tłumaczenia adresów, który wprowadza nieduży nakład dodatkowy i politykę przydziału pamięci minimalizującą ruch między poziomami pamięci.

[9] Taka obustronna wymiana następuje tylko wówczas, gdy w pamięci głównej nie ma wolnej ramki na brakującą stronę — *przyp. tłum.*



Rysunek 2.10. Adresowanie pamięci wirtualnej

Ochrona informacji i bezpieczeństwo

Wzrost zastosowań systemów z podziałem czasu oraz — w czasach bliższych współczesności — sieci komputerowych spowodował zwiększenie troski o ochronę informacji. Natura zagrożeń, którym poświęcają uwagę instytucje i przedsiębiorstwa, zależy od okoliczności. Istnieją jednak pewne ogólne narzędzia, nadające się do wbudowania w komputery i systemy operacyjne, które umożliwiają różnorodne mechanizmy ochrony i bezpieczeństwa. Ogólnie biorąc, interesuje nas problem kontrolowania dostępu do systemów komputerowych i przechowywanych w nich informacji.

Większość prac dotyczących ochrony i bezpieczeństwa w odniesieniu do systemów operacyjnych można z grubsza zaliczyć do następujących kategorii:

1. **Dostępność** (ang. *availability*)**.** Dotyczy ochrony systemu przed przerwami w pracy.
2. **Poufność** (ang. *confidentiality*)**.** Zapewnienie, że użytkownicy nie mogą czytać danych, jeśli nie zostali upoważnieni, aby po nie sięgać.
3. **Nienaruszalność danych** (ang. *data integrity*)**.** Ochrona danych przed modyfikacją bez upoważnienia.
4. **Uwierzytelnianie** (ang. *authenticity*)**.** Dotyczy odpowiedniego sprawdzania tożsamości użytkowników i ważności (wiarygodności) komunikatów lub danych.

Planowanie operowania zasobami

Podstawowym obowiązkiem systemu operacyjnego jest zarządzanie rozmaitymi zasobami, którymi rozporządza (obszarem pamięci głównej, urządzeniami zewnętrznymi, procesorami), oraz planowanie ich użycia przez różne aktywne procesy. Przydział każdego zasobu i polityka planowania muszą uwzględniać trzy czynniki:

1. **Uczciwość.** Zwykle dążymy do tego, aby wszystkie procesy rywalizujące o możliwość użycia pewnego zasobu otrzymywały do niego dostęp w przybliżeniu na równych prawach i sprawiedliwie. Jest tak szczególnie w przypadku zadań tej samej klasy, to znaczy zadań o podobnych wymaganiach.

2. **Zróżnicowana reakcja.** Z drugiej strony, może istnieć konieczność rozróżniania przez SO klas zadań o różnych wymaganiach dotyczących obsługi. System operacyjny powinien starać się podejmować decyzje o przydziale i planowaniu w taki sposób, aby spełniać cały zbiór wymagań. Powinien również podejmować te decyzje dynamicznie. Jeśli na przykład proces czeka na użycie urządzenia wejścia-wyjścia, to SO może dążyć do zaplanowania tego procesu do wykonywania najszybciej jak to możliwe. Proces może wtedy natychmiast skorzystać z urządzenia i zwolnić je, oddając do dyspozycji innych procesów, którym będzie ono potrzebne później.
3. **Sprawność.** SO powinien starać się maksymalizować przepustowość, minimalizować czas odpowiedzi i — w przypadku podziału czasu — przyjąć tylu użytkowników, ilu się da. Te kryteria są wzajemnie sprzeczne; znalezienie właściwej równowagi w konkretnej sytuacji jest nieustającym problemem w badaniach systemów operacyjnych.

Planowanie zasobów i zarządzanie nimi są w istocie problemami badań operacyjnych, jednak matematyczne wyniki osiągnięte w tej dyscyplinie nadają się do zastosowania również w systemach operacyjnych. Ważne jest ponadto mierzenie aktywności systemu, aby można było monitorować i regulować jego działanie.

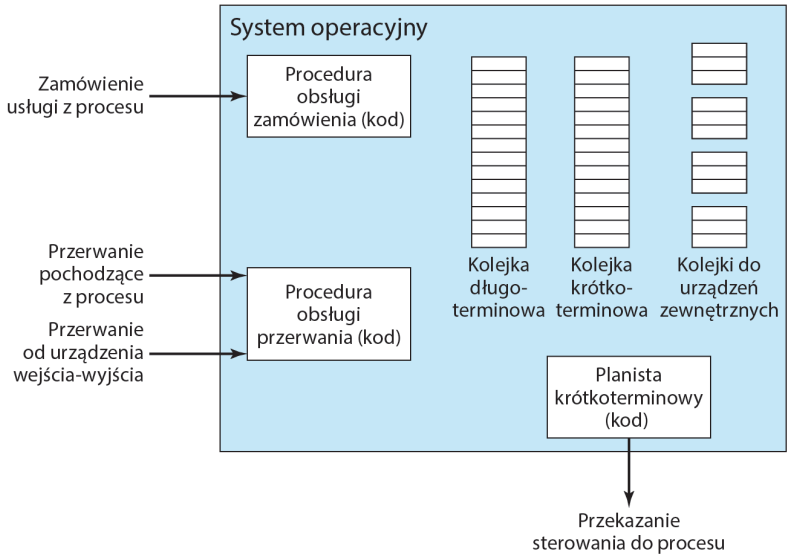
Na rysunku 2.11 naszkicowano główne elementy SO zaangażowane w procesy planowania i przydziału zasobów w środowisku wieloprogramowym. System operacyjny utrzymuje pewną liczbę kolejek będących po prostu listami procesów oczekujących na jakiś zasób. Kolejka krótkoterminowa składa się z procesów, które znajdują się w pamięci głównej (lub przynajmniej znajduje się tam istotna porcja każdego z nich) i są gotowe do działania, gdy tylko procesor stanie się dostępny[10]. Każdy z tych procesów mógłby natychmiast użyć procesora. Od planisty krótkoterminowego, czyli ekspedytora (dyspozytora, ang. *dispatcher*), zależy, który zostanie wybrany[11]. Typowa strategia polega na dawaniu każdemu procesowi w kolejce pewnej porcji czasu po kolei; określa się to mianem techniki **rotacyjnej** (ang. *round-robin*). Skutkuje ona traktowaniem kolejki jako cyklicznej. Inna strategia polega na przydzieleniu poszczególnym procesom poziomów priorytetów. Planista wybiera wtedy procesy w porządku priorytetów.

Kolejka długoterminowa jest listą nowych zadań czekających na użycie procesora. System operacyjny dodaje nowe zadania do wykonania przez przeniesienie procesu z kolejki długoterminowej do kolejki krótkoterminowej. W tym czasie nowo przybyłemu procesowi musi być przydzielona porcja pamięci głównej. SO musi zatem mieć pewność, że nie przekroczy pamięci lub możliwości przetwarzania przez przyjęcie zbyt wielu procesów. Z każdym urządzeniem zewnętrznym jest związana kolejka wejścia-wyjścia. To samo urządzenie może być zamówione przez więcej niż jeden proces. Wszystkie procesy czekające na użycie dowolnego z urządzeń są ustawiane jeden za drugim w kolejce do odpowiedniego urządzenia. Tu również SO musi rozstrzygać, któremu procesowi przydzielić dostępne urządzenie.

Jeśli wystąpi przerwanie, SO uzyskuje kontrolę nad procesorem w procedurze obsługi przerwania. Proces może określać, o którą usługę mu chodzi (np. o pobudzenie do działania procedury obsługi urządzenia wejścia-wyjścia), za pomocą wywoływania usług. W tej sytuacji procedura obsługi danego zamówienia jest punktem wejścia do SO. W każdym przypadku po obsłużeniu przerwania lub wywołanej usługi uruchamiany jest planista krótkoterminowy, aby wybrać proces do wykonania.

[10] Opisana sytuacja sprawia, że inni autorzy używają nazwy kolejka gotowych (kolejka procesów gotowych do działania, ang. *ready queue*) — *przyp. tłum.*

[11] Inni autorzy nie utożsamiają tych funkcji: planista krótkoterminowy wybiera proces do wykonywania, po czym ekspedytor ładuje kontekst tego procesu do rejestrów sprzętowych — *przyp. tłum.*



Rysunek 2.11. Podstawowe elementy systemu operacyjnego służące do organizowania wieloprogramowości

To, co tu powiedziano, jest opisem funkcjonalnym; szczegóły i modularny projekt tej części SO będą się różniły w różnych systemach. Znaczna część badań i prac rozwojowych w obszarze systemów operacyjnych była ukierunkowana na znalezienie algorytmów i struktur danych umożliwiających planowanie operowania zasobami, które zapewniałyby uczciwość, różnicowanie reakcji i efektywność działania.

2.4. DROGA DO WSPÓŁCZESNYCH SYSTEMÓW OPERACYJNYCH

Z biegiem czasu następowała stopniowa ewolucja struktury i zdolności SO. Jednakże w ostatnich latach zarówno do projektów nowych systemów operacyjnych, jak i do nowych wydań systemów istniejących wprowadzono liczne nowe elementy, które spowodowały poważne zmiany w ich naturze. Nowoczesne systemy operacyjne są odpowiedzią na postęp w rozwoju sprzętu, nowe zastosowania i nowe zagrożenia bezpieczeństwa. Do kluczowych osiągnięć w sprzęcie należą systemy wieloprocesorowe istotnie zwiększające szybkość procesora, osprzęt sieci dużych prędkości i rosnąca pojemność różnego rodzaju urządzeń magazynowania informacji. Na polu zastosowań wpływ na projekt SO wywarły aplikacje multimedialne, dostęp do Internetu i usługi WWW oraz model klient-serwer. Jeśli chodzi o bezpieczeństwo, dostęp z Internetu do komputerów spowodował znaczny wzrost potencjalnych zagrożeń, a rosnąca liczba wymyślnych ataków (jak wirusy, robaki i techniki hakerskie) odcisnęła poważne piętno na projektowaniu SO.

Tempo zmian w oczekiwaniach stawianych przed systemami operacyjnymi wymaga już nie tylko modyfikacji i ulepszeń w istniejących architekturach, lecz nowych sposobów organizacji SO. Wypróbowano szeroki zakres różnych podejść i elementów projektowych, tak w eksperymentalnych, jak i komercyjnych systemach operacyjnych, jednak znaczna część tych działań mieści się w następujących kategoriach:

- architektura mikrojądra,
- wielowątkowość,

- przetwarzanie symetryczne,
- rozproszone systemy operacyjne,
- projektowanie obiektowe.

Do niedawna większość systemów operacyjnych przybierała postać wielkiego **monolitycznego jądra** (ang. *monolithic kernel*). Większość tego, co pojmuje się jako funkcjonalność SO, mieści się w tych wielkich jądrach: planowanie, system plików, praca sieciowa, moduły sterujące urządzeń, zarządzanie pamięcią — i to jeszcze nie wszystko. Jądro monolityczne jest zazwyczaj implementowane w postaci pojedynczego procesu, którego wszystkie elementy dzielą tę samą przestrzeń adresową. W architekturze **mikrojądra** (ang. *microkernel*) jądru przypada w udziale tylko kilka istotnych funkcji, w tym zarządzanie przestrzenią adresową, komunikacja międzyprocesowa (ang. *interprocess communication* — IPC) i podstawowe planowanie. Inne usługi SO są świadczone przez procesy, niekiedy zwane serwerami, pracujące w trybie użytkownika i traktowane przez mikrojądro na równi z innymi aplikacjami. Koncepcja mikrojądra upraszcza implementację, zapewnia elastyczność i dobrze pasuje do środowiska rozproszonego. Reasumując, mikrojądro współdziała z lokalnymi i zdalnymi procesami serwerów w jednakowy sposób, ułatwiając konstruowanie systemów rozproszonych.

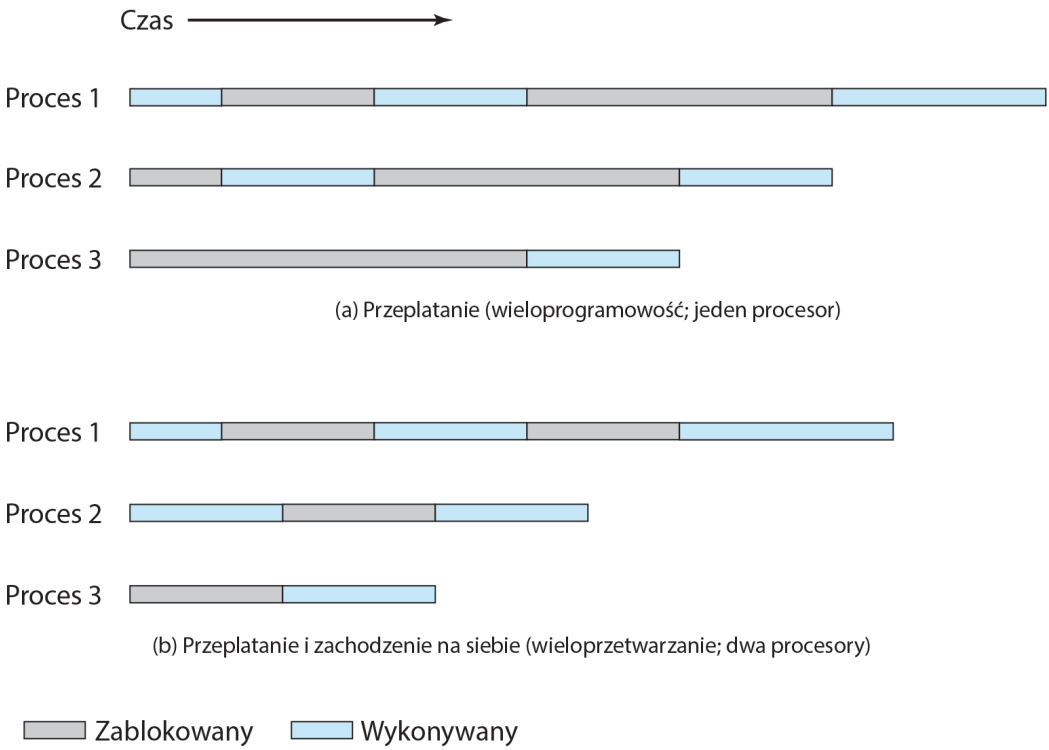
Wielowątkowość (ang. *multithreading*) jest techniką polegającą na tym, że proces wykonujący aplikację jest dzielony na wątki, które mogą działać współbieżnie. Możemy dokonać następującego rozróżnienia:

- **Wątek** (ang. *thread*) jest ekspediowalną jednostką pracy. Zawiera kontekst procesora (m.in. licznik programu i wskaźnik stosu) i własny obszar danych na stos (aby umożliwić wywoływanie podprogramów). Wątek działa sekwencyjnie i jest przerywalny, więc procesor może przejść do innego wątku.
- **Proces** to zbiór złożony z jednego lub więcej wątków i skojarzonych z nimi zasobów systemowych (takich jak pamięć zawierająca kod i dane, otwarte pliki i urządzenia). Jest to bliskie koncepcji programu w trakcie wykonywania. Rozbijając aplikację na wiele wątków, programista zyskuje dużą kontrolę nad modularnością aplikacji i przebiegiem związanych z nią zdarzeń.

Wielowątkowość jest pożyteczna w aplikacjach wykonujących wiele w gruncie rzeczy niezależnych **prac** (ang. *tasks*), których nie trzeba szeregować. Przykładem jest będący na nasłuchu serwer bazy danych przetwarzający zamówienia licznych klientów. W procesie z wieloma wątkami przełączanie między jednym wątkiem a innym (i z powrotem) zużywa mniej czasu procesora niż poważniejsze w realizacji przełączanie między różnymi procesami. Wątki są również przydatne do strukturalizacji procesów będących częścią jądra SO, co zostanie opisane w następnych rozdziałach.

Wieloprzetwarzanie symetryczne (ang. *symmetric multiprocessing* — SMP) jest pojęciem odnoszącym się do architektury sprzętowej komputera (opisanej w rozdziale 1.), a także do zachowania SO, który korzysta z tej architektury. System operacyjny realizujący SMP planuje procesy lub wątki na wszystkich procesorach. SMP ma wiele potencjalnych przewag nad architekturą jednoprocesorową, w tym:

- **Wydajność.** Jeśli pracę do wykonania na komputerze można zorganizować tak, aby pewne jej fragmenty można było wykonać równolegle, to system z wieloma procesorami będzie działał efektywniej niż z jednym procesorem tego samego typu. Zilustrowano to na rysunku 2.12.



Rysunek 2.12. Wieloprogramowość i wieloprzetwarzanie

W wieloprogramowości tylko jeden proces może pracować w danej chwili, wszystkie inne procesy czekają na procesor. W wieloprzetwarzaniu może działać jednocześnie więcej procesów — każdy na innym procesorze.

- **Dostępność.** Ponieważ wszystkie procesory wieloprocesora symetrycznego mogą wykonywać te same funkcje, awaria jednego procesora nie zatrzymuje systemu; system może kontynuować działanie ze zmniejszoną sprawnością.
- **Stopniowy wzrost.** Użytkownik może poprawić działanie systemu przez dodanie kolejnego procesora.
- **Skalowanie.** Dostawcy mogą oferować szerszy asortyment wyrobów o różnych cenach i charakterystykach działania, bazując na liczbie procesorów skonfigurowanych w systemie.

Należy zauważyć, że są to korzyści raczej potencjalne niż gwarantowane. System operacyjny musi udostępniać narzędzia i funkcje do wykorzystania równoległości w systemie SMP.

Wielowątkowość i przetwarzanie SMP są często omawiane razem, lecz są to dwa niezależne udogodnienia. Nawet w systemie jednoprocesorowym wielowątkowość jest przydatna do strukturalizacji procesów użytkowych i jądra. System SMP jest pożyteczny nawet w wypadku procesów bez wątkowości, gdyż kilka procesów może działać równolegle. Wszakże oba rozwiązania uzupełniają się wzajemnie i można z nich skutecznie korzystać razem.

Atrakcyjną cechą SMP jest to, że istnienie wielu procesorów jest dla użytkownika przezroczyste. Planowaniem wątków lub procesów na poszczególnych procesorach i synchronizacją między nimi zajmuje się system operacyjny. W tej książce omawiamy mechanizmy planowania i synchronizacji

stosowane do wywierania na użytkowniku wrażenia jednego systemu. Odmiennym problemem jest tworzenie wrażenia jednego systemu w przypadku grona osobnych komputerów — systemu wielokomputerowego. W tym wypadku mamy do czynienia ze zbiorem komputerów: każdy z nich ma własną pamięć główną, pamięć drugorzędną i inne moduły wejścia-wyjścia. **Rozproszony system operacyjny** (ang. *distributed operating system*) dostarcza iluzji jednej przestrzeni pamięci głównej i jednej przestrzeni pamięci pomocniczej oraz innych ujednoliconych udogodnień dostępu, takich jak rozproszony system plików. Chociaż grona (klastry) coraz bardziej zyskują na popularności i na rynku jest wiele produktów tego typu, stan sztuki, jeśli chodzi o rozproszone systemy operacyjne, zostaje w tyle za systemami operacyjnymi jednoprocesorowymi i SMP. W części VIII książki przyjrzymy się takim systemom.

Innowacją w projektowaniu SO jest również zastosowanie technologii obiektowych. **Projektowanie obiektowe** (ang. *object-oriented design*) pomaga dyscyplinować dodawanie modularnych rozszerzeń do małego jądra. Na poziomie SO struktura oparta na obiektach umożliwia osobom programującym dokonywanie przeróbek SO bez naruszania całości systemu. Obiektowość ułatwia również opracowywanie narzędzi rozproszonych i rozwiniętych rozproszonych systemów operacyjnych.

2.5. TOLEROWANIE AWARII

Tolerowanie awarii oznacza zdolność systemu lub komponentu do kontynuowania normalnego działania mimo występowania wad w sprzęcie lub oprogramowaniu. Zazwyczaj wymaga to zastosowania pewnego rodzaju nadmiarowości. Tolerowanie awarii ma służyć zwiększaniu niezawodności systemu. Polepszeniu zdolności do tolerowania awarii (a więc i zwiększeniu niezawodności) na ogół towarzyszy wzrost kosztów rozumianych w kategoriach finansowych albo wydajnościowych (albo w obu). Dlatego jest konieczne ustalenie miar zakresu stosowania tolerowania awarii wedle tego, do jakiego stopnia dany zasób ma krytyczne znaczenie w systemie.

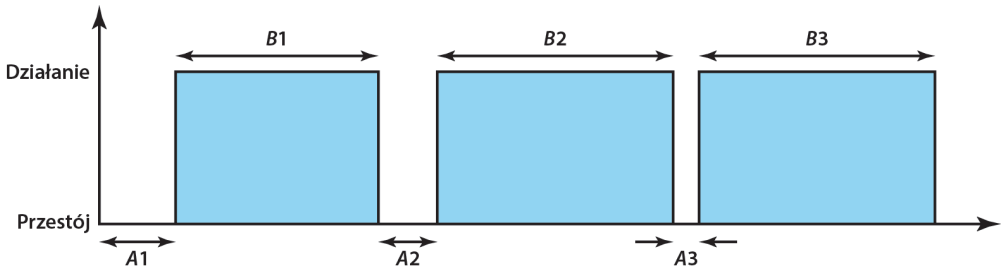
Podstawowe pojęcia

Z tolerowaniem awarii wiążą się trzy podstawowe miary jakości działania systemu: niezawodność, średni czas do wystąpienia awarii (MTTF) oraz dostępność. Te pojęcia powstały ze szczególnym odniesieniem do wad sprzętowych, lecz stosują się ogólniej zarówno do wad sprzętowych, jak i programowych.

Niezawodność (ang. *reliability*) $R(t)$ systemu jest definiowana jako prawdopodobieństwo jego poprawnego działania do chwili t przy założeniu, że system działał poprawnie w chwili $t = 0$. W odniesieniu do systemów komputerowych i systemów operacyjnych termin *poprawne działanie* oznacza poprawne wykonywanie zbioru programów i ochronę danych przed niepożądanymi modyfikacjami. **Średni czas do wystąpienia awarii** (ang. *mean time to failure* — MTTF) jest definiowany jako:

$$\mathrm{MTTF} = \int_0^\infty \mathrm{R}(\mathrm{t})$$

Średni czas naprawy (ang. *mean time to repair* — MTTR) jest średnim czasem zużywanym na zreperowanie lub wymianę wadliwego elementu. Na rysunku 2.13 pokazano związek między MTTF i MTTR.



$$MTTF = \frac{B1 + B2 + B3}{3} \qquad MTTR = \frac{A1 + A2 + A3}{3}$$

Rysunek 2.13. Stany operacyjne systemu

Dostępność (ang. *availability*) systemu lub usługi jest ułamkiem czasu, w którym system jest zdolny obsługiwać żądania użytkowników. Równoważnie dostępność oznacza prawdopodobieństwo, że jednostka działa poprawnie, spełniając zadane warunki przez określony czas. Czas, w którym system nie jest dostępny, jest nazywany **przestojem** (ang. *downtime*). Czas, w którym system jest dostępny, określany jest jako **czas pracy** (ang. *uptime*). Dostępność systemu można wyrazić wzorem:

```
$JOB
$FTN
 •
 •          } Instrukcje w Fortranie
 •
$LOAD
$RUN
 •
 •          } Dane
 •
$END
```

W tabeli 2.4 pokazano niektóre powszechnie rozróżniane poziomy dostępności i odpowiednie okresy przestoju w skali roku.

Tabela 2.4. Klasy dostępności

Klasa	Dostępność	Przestój w skali roku
Ciągła	1,0	0
Tolerująca awarie	0,99999	5 minut
Znosząca awarie	0,9999	53 minuty
Wysoka dostępność	0,999	8,3 godziny
Zwykła dostępność	0,99 – 0,995	44 – 87 godzin

Często lepszym niż dostępność wskaźnikiem jest MTTF, czyli średni czas pracy. Połączenie krótkotrwałych przestojów z krótkimi czasami pracy mogłoby dać wysoką miarę dostępności, lecz użytkownicy nie byliby w stanie skorzystać z żadnej usługi, gdyby czas pracy był mniejszy niż czas potrzebny do jej wykonania.

Wady

Słownik standardów IEEE definiuje **wadę** (ang. *fault*) jako błędny (zafałszowany, ang. *erroneous*) stan sprzętu lub oprogramowania wynikający z wady komponentu, pomyłki operatora, fizycznego wpływu środowiska, błędu projektowego, błędu programu lub błędu w strukturze danych. Standard głosi również, że wada objawia się 1) defektem urządzenia lub komponentu sprzętowego, na przykład krótkim spięciem lub zerwanym przewodem, lub 2) niepoprawnym krokiem, procesem lub definicją danych w programie komputerowym.

Wady możemy podzielić na następujące kategorie:

- **Trwała** (ang. *permanent*)**.** Wada, która pojawia się i nie ustępuje. Wada trwa dopóty, dopóki wadliwa składowa nie zostanie wymieniona lub naprawiona. Przykładem może być awaria głowicy dyskowej, błędy w oprogramowaniu lub przepalony komponent komunikacyjny.
- **Chwilowa** (ang. *temporary*)**.** Wada, która nie jest obecna nieustannie, w każdych warunkach działania. Wady chwilowe można dalej sklasyfikować następująco:
  - **Przejściowa** (ang. *transient*). Wada występująca tylko jeden raz. Do przykładów należą błędy transmisji bitu spowodowane impulsowym szumem, zakłóceniami zasilania lub promieniowaniem, które zmienia bit pamięci.
  - **Nieciągła** (ang. *intermittent*)**.** Wada pojawiająca się wielokrotnie w nieokreślonych momentach. Przykładem wady przejściowej jest luźny styk w łączówce.

Ogólnie biorąc, tolerowanie awarii jest wbudowywane do systemu przez dodanie nadmiarowości. Do rodzajów nadmiarowości należą:

- **Nadmiarowość przestrzenna (fizyczna).** Fizyczna nadmiarowość polega na zastosowaniu wielu komponentów, które albo wykonują tę samą funkcję jednocześnie, albo są tak skonfigurowane, że jeden komponent jest dostępny jako zapasowy w wypadku awarii drugiego. Przykładem pierwszego rozwiązania jest zastosowanie wielu równoległych obwodów, w których za wynik wyjściowy uznaje się rezultat większościowy. Przykładem drugiej możliwości jest zapasowy serwer nazw w Internecie.
- **Nadmiarowość chwilowa.** Nadmiarowość chwilowa polega na powtórzeniu funkcji lub operacji w sytuacji wykrycia błędu. To podejście jest skuteczne w wypadku wad chwilowych, lecz nieprzydatne przy wadach trwałych. Przykładem jest powtórzenie transmisji bloku danych po wykryciu błędu, jak to się robi w protokołach łącza danych.
- **Nadmiarowość informacji.** Nadmiarowość informacji umożliwia tolerowanie awarii przez zwielokrotnienie lub zakodowanie danych w taki sposób, że błędy bitów można zarówno wykryć, jak i skorygować. Przykładami są obwód kodujący z kontrolą błędów, używany w systemach pamięci, i techniki korygowania błędów w dyskach RAID, o czym powiemy w następnych rozdziałach.

Mechanizmy systemu operacyjnego

Oprogramowanie SO może zawierać wiele mechanizmów wspierających tolerowanie awarii. W książce napotkamy sporo tego rodzaju przykładów. Tu podajemy niektóre:

- **Izolowanie procesu.** Jak już wspominano w tym rozdziale, procesy są z zasady izolowane od siebie w rozumieniu miejsca ich pobytu w pamięci głównej, dostępu do plików i przebiegu sterowania. Struktura dostarczana przez SO na użytek administrowania procesami umożliwia pewien stopień ochrony innych procesów przed procesem powodującym błąd.
- **Sterowanie współbieżnością.** W rozdziałach 5. i 6. będą omówione niektóre trudności i wady mogące powstawać podczas komunikacji i współpracy procesów. W tych rozdziałach zostaną również omówione sposoby stosowane do zapewniania poprawnego działania i wychodzenia z sytuacji wadliwych, w rodzaju zakleszczenia.
- **Maszyny wirtualne.** Maszyny wirtualne, o czym powiemy w rozdziale 14., umożliwiają większy stopień izolowania aplikacji, więc również tolerowania awarii. Można je także zastosować do tworzenia nadmiarowości na tej zasadzie, że jedna maszyna wirtualna służy jako zapasowa dla innej.
- **Punkty kontrolne i wycofania.** Punkt kontrolny (ang. *checkpoint*) jest kopią stanu aplikacji zmagazynowaną w jakiejś pamięci odpornej na przewidywane awarie. Wycofanie polega na wznowieniu wykonywania od wcześniej przechowanego punktu kontrolnego. Gdy dojdzie do awarii, aplikację cofa się do stanu z poprzedniego punktu kontrolnego i od tego miejsca podejmuje się jej wykonywanie. Tej techniki można użyć do wychodzenia z przejściowych, a także trwałych awarii oraz z pewnych rodzajów awarii oprogramowania. Rozwiązania takie są zazwyczaj wbudowane w systemy baz danych i systemy transakcyjne.

Na uwagę zasługiwałoby znacznie więcej technik, lecz pełne omówienie tolerowania awarii w systemach operacyjnych wykracza poza ramy, które tutaj sobie nakreśliliśmy.

2.6. PROBLEMY PROJEKTOWANIA SYSTEMÓW OPERACYJNYCH WIELOPROCESORÓW I KOMPUTERÓW WIELORDZENIOWYCH

Rozważania dotyczące wieloprocesorowego symetrycznego SO

W systemie SMP (wieloprocesorowego przetwarzania symetrycznego) jądro może działać na każdym procesorze i na ogół każdy procesor sam planuje procesy lub wątki z dostępnej puli. Jądro może być skonstruowane z wielu procesów lub wątków, co umożliwia równoległe wykonywanie jego części. Podejście SMP komplikuje system operacyjny. Osoba projektująca SO musi poradzić sobie ze złożonością wynikającą z dzielenia zasobów (np. struktur danych) i działań koordynujących (np. związanych z dostępem do urządzeń) pochodzących z wielu części SO wykonywanych w tym samym czasie. Trzeba stosować odpowiednie techniki zamawiania i synchronizowania zasobów.

System operacyjny SMP zarządza procesorem i innymi zasobami komputera tak, aby użytkownik postrzegał go jako wieloprogramowy system jednoprocesorowy. Użytkownik może budować aplikacje wieloprocesowe lub z wieloma wątkami w procesach, nie troszcząc się o to, czy ma do dyspozycji jeden, czy wiele procesorów. Tak więc wieloprocesorowy SO musi zapewnić całą funkcjonalność systemu wieloprogramowego, a ponadto dodatkowe właściwości umożliwiające korzystanie z wielu procesorów. Do podstawowych zagadnień projektowych należą tu:

- **Jednoczesne współbieżne procesy lub wątki.** Procedury jądra muszą być wznawialne, aby umożliwiać jednoczesne wykonywanie na kilku procesorach tego samego kodu jądra. Jeśli wiele procesorów wykonuje te same lub różne części jądra, to zarządzanie tablicami i strukturami jądrowymi musi odbywać się tak, aby uniknąć uszkodzenia danych lub niedozwolonych operacji.
- **Planowanie.** Dowolny procesor może realizować planowanie, co komplikuje zadanie wymuszania polityki planowania i zapewnianie, że nie dojdzie do naruszenia struktur danych planisty. Jeśli na poziomie jądra stosuje się wielowątkowość, to istnieje możliwość planowania wielu wątków tego samego procesu jednocześnie na wielu procesorach. Planowanie wieloprocesorów będzie omówione w rozdziale 10.
- **Synchronizacja.** Jeśli wiele aktywnych procesów ma potencjalny dostęp do wspólnych przestrzeni adresowych lub dzielonych zasobów wejścia-wyjścia, należy zadbać o skuteczną synchronizację. Synchronizacja wymusza wzajemne wykluczanie i porządkowanie zdarzeń. Typowym mechanizmem synchronizacji, używanym w wieloprocesorowych systemach operacyjnych, są blokady, które opiszemy w rozdziale 5.
- **Zarządzanie pamięcią.** Zarządzanie pamięcią wieloprocesora musi uwzględniać wszystkie problemy występujące na komputerach jednoprocesorowych i zostanie przedstawione w części III książki. System operacyjny musi ponadto wykorzystywać istniejącą w sprzęcie równoległość, aby osiągać najlepsze działanie. Mechanizmy stronicowania na poszczególnych procesorach muszą być koordynowane, aby wymuszać spójność w sytuacjach, gdy ileś procesorów dzieli stronę lub segment, i rozstrzygać o zastępowaniu stron. Ponowne użycie stron fizycznych stanowi największy problem: trzeba zagwarantować, że strona fizyczna ze starą zawartością nie będzie dostępna, zanim nie zostanie jej wyznaczone zastosowanie.
- **Niezawodność i tolerowanie awarii.** W wypadku awarii procesora system operacyjny powinien zapewniać łagodną degradację. Planista i inne części SO muszą rozpoznawać utratę procesora i odpowiednio restrukturyzować tablice zarządzania.

Ponieważ projektowanie wieloprocesorowego SO w zasadzie polega na rozszerzeniu rozwiązań problemów projektowania wieloprogramowego jednoprocesora, nie traktujemy wieloprocesorowych systemów operacyjnych oddzielnie. Zamiast tego specyficzne zagadnienia dotyczące wieloprocesorów są omawiane w książce w odpowiednich miejscach.

Rozważania dotyczące wielordzeniowych SO

Wszystkie zagadnienia projektowe dotyczące systemów SMP, omówione dotychczas w tym podrozdziale, odnoszą się również do systemów wielordzeniowych. Niemniej występują tu również inne problemy. Jednym z nich jest skala potencjalnej równoległości. Obecni dostawcy komputerów mają do zaoferowania dziesięć lub więcej rdzeni na jednym chipie. W każdej następnej technologii kolejnych generacji procesorów liczba rdzeni i ilość wspólnej i wydzielonej pamięci podręcznej rośnie, wchodzimy więc w erę systemów „licznych rdzeni”.

Niemałym wyzwaniem w projektowaniu systemów wielordzeniowych o wielkiej liczbie rdzeni jest efektywne zaprzęgnięcie do roboty mocy obliczeniowej licznych rdzeni i inteligentne zarządzanie sporymi zasobami pomieszczonymi w jednym układzie. W centrum zainteresowań znajduje się pytanie, jak dopasować wewnętrzny paralelizm systemu mnogich rdzeni do zapotrzebowań wydajnościowych aplikacji. Potencjalna równoległość istnieje w istocie na trzech poziomach współcze-

snego systemu wielordzeniowego. Po pierwsze, wewnątrz każdego procesora rdzeniowego istnieje równoległość sprzętowa, znana jako równoległość na poziomie rozkazów, z której mogą, lub nie, skorzystać programiści aplikacji i kompilatory. Po drugie, możliwość wykonywania wieloprogramowego i wielowątkowego można realizować w każdym procesorze. I na koniec potencjalnie każda aplikacja z osobna może być wykonana we współbieżnych procesach lub wątkach na wielu rdzeniach. Bez mocnego i skutecznego wsparcia ze strony SO dwu ostatnich rodzajów równoległości zasoby sprzętowe nie będą wykorzystane efektywnie.

Z nastaniem technologii wielordzeniowej projektanci SO zmagają się w istocie z problemem możliwie najlepszego wydobycia równoległości z prac, którymi są obciążane komputery. Sprawdza się rozmaite podejścia do budowy systemów operacyjnych nowej generacji. W tym podrozdziale przedstawimy dwa ogólne sposoby postępowania, o innych szczegółach opowiemy w dalszych rozdziałach.

RÓWNOLEGŁOŚĆ W OBRĘBIE APLIKACJI

Większość aplikacji można w zasadzie podzielić na wiele prac, które mogą być wykonywane równolegle przez realizowanie tych prac w postaci wielu procesów, być może wielowątkowych. Trudność polega na tym, że o podziale pracy aplikacji na niezależnie wykonywalne zadania musi decydować jej twórca. To znaczy musi on zdecydować, które fragmenty mogą lub powinny być wykonane asynchronicznie lub równolegle. W procesie projektowania programowania równoległego przede wszystkim pomagają właściwości kompilatora i języka programowania. Jednak SO może wspomagać ten proces projektowania przynajmniej przez skuteczne przydzielanie zasobów równoległym zadaniom zdefiniowanym przez twórcę aplikacji.

Jedną z najskuteczniejszych inicjatyw mających wesprzeć budowniczych jest Grand Central Dispatch (GCD, z ang. Wielka Centralna Ekspedycja) zrealizowana w ostatnich wydaniach opartych na UNIX-ie systemów Mac OS X oraz iOS. GCD jest środkiem wspomagania wielordzeniowości. Nie pomaga twórcy oprogramowania w rozstrzyganiu, jak podzielić zadanie lub aplikację na osobne współbieżne części. Jednak gdy budowniczy zidentyfikuje coś, z czego można wydzielić osobne zadanie, GCD umożliwia urzeczywistnienie tego zamiaru w łatwy i nieinwazyjny sposób.

W gruncie rzeczy GCD jest mechanizmem puli wątków, w którym SO odwzorowuje prace na wątki reprezentujące osiągalny stopień współbieżności (plus wątki blokujące się na wejściu-wyjściu). System Windows ma również mechanizm puli wątków (od 2000 roku), a w aplikacjach serwerowych pule wątków są używane od lat. Nowością w GCD jest to, że stanowi rozszerzenie wobec języków programowania, umożliwiające stosowanie funkcji anonimowych (zwanych blokami) jako sposób specyfikowania prac. Wielka Centralna Ekspedycja nie jest zatem poważnym krokiem ewolucyjnym, niemniej stanowi nowe i wartościowe narzędzie wykorzystywania możliwej równoległości w systemie wielordzeniowym.

Jeden ze sloganów rodem z Apple'a głosi, że GCD stanowi „wyspy serializacji na morzu współbieżności". Ujmuje to praktyczny aspekt coraz większego uwspółbieżniania przeciętnych aplikacji komputerów biurkowych. Owe wyspy są tym, co izoluje twórców od drażliwych problemów jednoczesnego dostępu do danych, zakleszczeń i innych pułapek wielowątkowości. Budowniczowie aplikacji są zachęcani do identyfikowania w swoich wyrobach funkcji, które mogłyby być lepiej wykonywane poza głównym wątkiem — nawet jeśli składają się z pewnej liczby sekwencyjnych lub w inny sposób częściowo współzależnych prac. GCD ułatwia podzielenie całej jednostki pracy z utrzymaniem istniejącego porządku i zależności między podzadaniami. W dalszych rozdziałach przyjrzymy się niektórym detalom GCD.

PODEJŚCIE OPARTE NA MASZYNIE WIRTUALNEJ

Rozwiązanie alternatywne polega na uświadomieniu sobie, że wraz z nieustannie rosnącą liczbą rdzeni w układzie dążenie do wieloprogramowania poszczególnych rdzeni w celu obsługiwania wielu aplikacji może być mylnym użyciem zasobów [JACK10]. Jeśli zamiast tego pozwolimy na przypisanie jednego lub więcej rdzeni do konkretnego procesu i zostawimy procesor do jego wyłącznej dyspozycji, to unikniemy większości nakładów związanych z przełączaniem prac i decyzjami planistycznymi. Wielordzeniowy SO mógłby się wtedy przekształcić w hipernadzorcę podejmującego wysokopoziomowe decyzje o przydziale rdzeni do aplikacji, czyniąc niewiele ponad to w kwestii sposobu przydzielania zasobów.

Za takim rozwiązaniem przemawiają następujące argumenty. W zaraniu komputeryzacji jeden program działał na jednym procesorze. Z nadejściem wieloprogramowości każdej aplikacji stworzono iluzję działania na osobnym procesorze. Wieloprogramowość opiera się na koncepcji procesu będącego abstrakcją środowiska wykonywania. Aby zarządzać procesami, SO wymaga chronionej przestrzeni, wolnej od wpływu użytkownika i programu. W tym celu wprowadzono rozróżnienie między trybem jądra a trybem użytkownika. W rezultacie tryby jądra i użytkownika wyabstrahowały procesor w dwa procesory. W obecności tych procesorów przyszło jednak zmagać się z problemem, któremu z nich powierzyć prawdziwy procesor. Nakłady na przełączanie między tymi procesorami zaczęły rosnąć, aż osiągnęły punkt, w którym zaczęło się to negatywnie odbijać na gotowości do reagowania, szczególnie gdy wprowadzono wiele rdzeni. Wszakże w systemach z licznymi rdzeniami możemy dać sobie spokój z rozróżnianiem między trybami jądra i użytkownika. System operacyjny działa w tym podejściu jak swoisty hipernadzorca. Wiele obowiązków dotyczących zarządzania zasobami przechodzi na same programy. SO przydziela aplikację, procesor i jakieś zasoby, a program, korzystając z metadanych wygenerowanych przez kompilator, będzie sam najlepiej wiedział, jak z tych zasobów korzystać.

2.7. PRZEGLĄD SYSTEMU MICROSOFT WINDOWS

Rodowód

Microsoft początkowo (w 1985 roku) użył nazwy Windows na określenie rozszerzenia środowiska operacyjnego nader elementarnego systemu MS-DOS, który jako system operacyjny wczesnych komputerów osobistych okazał się bardzo udany. Kombinacja Windows/MS-DOS została w końcu zastąpiona nową wersją systemu Windows, znaną jako Windows NT, której pierwsze wydanie ukazało się w 1993 roku z przeznaczeniem do laptopów i systemów biurkowych. Choć podstawowa architektura wewnętrzna pozostawała z grubsza ta sama od czasów Windows NT, system był rozwijany dalej, obrastając w funkcje i właściwości. Ostatnie wydanie, pochodzące z czasu pisania tej książki, to Windows 10. Windows 10 łączy cechy poprzednich wydań biurkowo-laptopowych, wersji Windows 8.1, a także wersji systemu Windows przeznaczonych dla urządzeń mobilnych w Internecie rzeczy (ang. *Internet of things* — IoT). W Windows 10 wcielono również oprogramowanie z systemu Xbox One. Wynikowy, ujednolicony Windows 10 obsługuje komputery biurkowe, laptopy, smartfony, tablety i Xbox One.

Architektura

Na rysunku 2.14 przedstawiono ogólną budowę systemu Windows. Jak niemal wszystkie systemy, Windows oddziela oprogramowanie użytkowe od oprogramowania stanowiącego rdzeń SO. Ten ostatni, zawierający egzekutora, jądro, moduły sterujące i warstwę abstrakcji sprzętu, działa w trybie jądra. Oprogramowanie z trybu jądra ma dostęp do danych systemowych i do sprzętu. Pozostałe oprogramowanie, działające w trybie użytkownika, ma ograniczony dostęp do danych systemu.

ORGANIZACJA SYSTEMU OPERACYJNEGO

Windows ma wysoce modularną architekturę. Każda funkcja systemowa jest zarządzana przez tylko jedną składową SO. Reszta SO i wszystkie aplikacje uzyskują dostęp do tej funkcji za pośrednictwem odpowiedzialnej za nią składowej, korzystając ze standardowych interfejsów. Kluczowe dane systemu są dostępne tylko za pośrednictwem odpowiedniej funkcji. W zasadzie każdy moduł może być uaktualniony lub zastąpiony bez przepisywania całego systemu lub jego standardowych interfejsów programów użytkowych (ang. *application program interfaces* — API[12]). Oto składowe systemu Windows działające w trybie jądra:

- **Egzekutor** (ang. *Executive*[13])**.** Zawiera podstawowe usługi SO, takie jak zarządzanie pamięcią, zarządzanie procesami, bezpieczeństwo, wejście-wyjście i komunikacja międzyprocesowa.
- **Jądro** (ang. *Kernel*)**.** Steruje działaniem procesorów. Jądro zarządza planowaniem wątków, przełączaniem procesów, obsługą wyjątków i przerwań i synchronizacją wieloprocesora. W odróżnieniu od reszty[14] egzekutora i poziomów użytkowych kod jądra nie działa w wątkach.
- **Warstwa abstrakcji sprzętu** (ang. *hardware abstraction layer* — HAL)**.** Dokonuje odwzorowań między ogólnymi poleceniami i odpowiedziami sprzętowymi a specyficznymi dla danej platformy. Izoluje SO od różnic sprzętowych zależnych od platformy. HAL sprawia, że szyna systemowa każdego komputera, sterownik bezpośredniego dostępu do pamięci (DMA), sterownik przerwań, czasomierze systemowe i sterownik pamięci od strony egzekutora i składowych jądrowych wyglądają tak samo. Dostarcza również zaplecza potrzebnego do SMP (komunikacji międzyprocesowej), co zostanie wyjaśnione później.
- **Moduły sterujące urządzeń** (ang. *device drivers*)[15]**.** Są to biblioteki dynamiczne, które rozszerzają funkcjonalność egzekutora. Należą do nich moduły sterujące urządzeń sprzętowych, tłumaczące wywołania wejścia-wyjścia użytkownika na żądania wejścia-wyjścia specyficzne dla danego sprzętu, oraz komponenty programowe do implementowania systemu plików, protokoły sieciowe i wszelkie inne rozszerzenia systemu, które muszą działać w trybie jądra.
- **System okien i grafiki** (ang. *windowing and graphics system*)**.** Realizuje funkcje GUI, takie jak operowanie oknami, kontrolkami interfejsu użytkownika i rysowaniem.

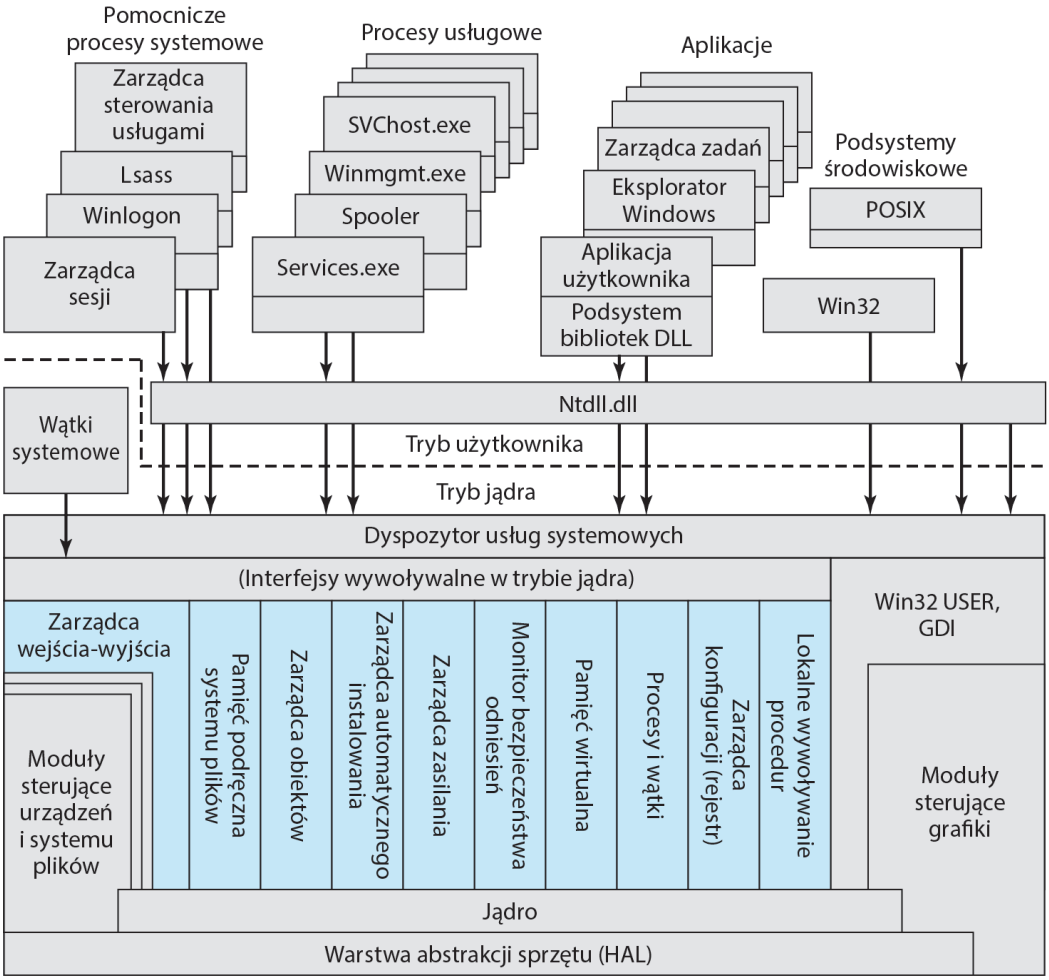
---

[12] Jak widać, również w języku angielskim akronim API jest rozwijany różnie, por. 2.1 — *przyp. tłum.*

[13] Zachowujemy za oryginałem rozpoczynanie nazw niektórych składowych Windows dużą literą, choć nie jest to powszechnie przyjęte w innych opisach tego systemu — *przyp. tłum.*

[14] Reszty w tym sensie, że również wywołania egzekutora są realizowane w trybie jądra (por. rysunek 2.14) — *przyp. tłum.*

[15] Potocznie nazywane „drajwerami" lub sterownikami. Należy zaznaczyć, że sterownikami (ang. *controller*) nazywamy sprzętowe elementy systemu komputerowego; moduły sterujące są elementami oprogramowania; jeszcze inna spotykana nazwa: programy obsługi — *przyp. tłum.*



Lsass = serwer uwierzytelnień bezpieczeństwa lokalnego (ang. *local security authentication server*)
POSIX = interfejs przenośnego systemu operacyjnego (ang. *portable operating system interface*[16])
GDI = interfejs urządzeń graficznych (ang. *graphics device interface*)
DLL = biblioteka konsolidowana dynamicznie (ang. *dynamic link library*)

Obszarami zabarwionymi zaznaczono egzekutora

Rysunek 2.14. Wewnętrzna architektura systemu Windows [RUSS11]

Egzekutor systemu Windows zawiera składowe dotyczące specyficznych funkcji systemowych i dostarcza API oprogramowaniu działającemu w trybie użytkownika. Poniżej podajemy krótki opis poszczególnych modułów egzekutora:

- **Zarządca wejścia-wyjścia.** Tworzy ramę udostępniania aplikacjom urządzeń wejścia-wyjścia i odpowiada za ekspediowanie do odpowiednich modułów sterujących urządzeń w celu dalszego przetwarzania. Zarządca wejścia-wyjścia realizuje wszystkie API wejścia-wyjścia w systemie Windows i wymusza bezpieczeństwo oraz nazewnictwo urządzeń, protokołów sieciowych i systemów plików (używając zarządcy obiektów). Wejście-wyjście w systemie Windows będzie omówione w rozdziale 11.

- **Zarządca pamięci podręcznej.** Poprawia działanie wejścia-wyjścia opartego na plikach, umieszczając w pamięci głównej w celu szybkiego dostępu dane plików, do których następowały niedawno odniesienia, oraz opóźniając zapisy na dysk przez krótkotrwałe utrzymywanie uaktualnień w pamięci, nim zostaną wysłane na dysk w wydajniejszych partiach.
- **Zarządca obiektów.** Tworzy obiekty egzekutora systemu Windows, administruje nimi oraz je usuwa; obiekty te są używane do reprezentowania zasobów takich jak procesy, wątki i obiekty synchronizacji. Wymusza jednolite reguły zachowywania, nazywania i określania bezpieczeństwa obiektów. Zarządca obiektów tworzy również wpisy w tablicy uchwytów procesowych, zawierające informacje służące do kontroli dostępu i wskaźnik do obiektu. Obiekty systemu Windows będą omówione w dalszej części tego podrozdziału.
- **Zarządca automatycznego instalowania** (ang. *plug-and-play manager*)**.** Określa, które moduły sterujące są potrzebne do obsługi danego urządzenia, i ładuje te moduły.
- **Zarządca zasilania** (ang. *power manager*)**.** Koordynuje zarządzanie zasilaniem różnych urządzeń i może być konfigurowany w celu zmniejszenia zużycia energii przez wyłączanie bezczynnych urządzeń, usypianie procesora, a nawet przepisywanie całej pamięci operacyjnej na dysk i wyłączanie zasilania całego systemu.
- **Monitor bezpieczeństwa odniesień** (ang. *security reference monitor*)**.** Wymusza reguły kontrolowania prawomocności dostępu i generowania audytów. Model obiektowy systemu Windows umożliwia spójne i ujednolicone ujęcie bezpieczeństwa aż po elementarne jednostki tworzące egzekutora. Tak więc Windows stosuje te same procedury sprawdzania legalności dostępu i kontroli w odniesieniu do wszystkich chronionych obiektów, włączając w to pliki, procesy, przestrzenie adresowe i urządzenia zewnętrzne. Bezpieczeństwo w systemie Windows będzie omówione w rozdziale 15.
- **Zarządca pamięci wirtualnej** (ang. *virtual memory manager*)**.** Administruje adresami wirtualnymi, pamięcią fizyczną i plikami stronicowania na dysku. Nadzoruje sprzęt zarządzający pamięcią i struktury danych odwzorowujące wirtualne adresy z przestrzeni adresowej procesu na strony fizyczne w pamięci komputera. Zarządzanie pamięcią wirtualną w systemie Windows będzie omówione w rozdziale 8.
- **Zarządca procesów lub wątków** (ang. *process/thread manager*)**.** Tworzy i usuwa obiekty procesów i wątków oraz administruje nimi. Zarządzanie procesami i wątkami będzie opisane w rozdziale 4.
- **Zarządca konfiguracji** (ang. *configuration manager*)**.** Odpowiada za zarządzanie rejestrem systemu, będącym magazynem ustawień różnych parametrów, zarówno ogólnosystemowych, jak i poszczególnych użytkowników.
- **Udogodnienie ulepszonego wywołania procedury lokalnej** (ang. *advanced local procedure call facility* — ALPC)**.** Realizuje sprawny mechanizm międzyprocesowych wywołań procedur między lokalnymi procesami implementującymi usługi i podsystemy. Jest podobne do zdalnego wywołania procedury (ang. *remote procedure call* — RPC) używanego w przetwarzaniu rozproszonym.

PROCESY TRYBU UŻYTKOWNIKA

Windows tworzy zaplecze czterech podstawowych rodzajów procesów działających w trybie użytkownika:

1. **Specjalne procesy systemowe.** Usługi trybu użytkownika potrzebne do zarządzania systemem, takie jak zarządca sesji, podsystem uwierzytelniania, zarządca usług i proces rozpoczynania sesji (logowania, *logon*).
2. **Procesy usługowe.** Spooler drukarki, rejestrator zdarzeń, składowe trybu użytkownika współpracujące z modułami obsługi urządzeń, różnorodne usługi sieciowe i wiele innych. Z usług korzystają zarówno twórcy oprogramowania w Microsofcie, jak i konstruktorzy zewnętrzni, aby rozszerzać funkcjonalność systemu, jako że są one w systemie Windows jedynym sposobem wykonywania drugoplanowych działań w trybie użytkownika.
3. **Podsystemy środowiskowe.** Udostępniają różne osobiste prezencje (środowiska) SO. Dostępne są podsystemy Win32 i POSIX. Każdy podsystem środowiskowy zawiera proces podsystemu dzielony między wszystkie aplikacje korzystające z danego podsystemu i dynamicznie konsolidowane biblioteki (DLL), które zamieniają wywołania aplikacji użytkownika na wywołania ALPC procesu podsystemu i (lub) rodzime wywołania Windows.
4. **Aplikacje użytkownika.** Pliki wykonywalne (EXE-ki) i DLL-e reprezentujące działania wykonywane przez użytkowników w celu zastosowania systemu. Pliki EXE i DLL są na ogół nakierowane na konkretny podsystem środowiskowy, aczkolwiek niektóre programy dostarczane jako część SO używają rodzimych interfejsów systemu (NT API). Istnieje też zaplecze wykonywania programów 32-bitowych w systemach 64-bitowych.

Strukturalizacja systemu Windows ma umożliwiać użytkowanie aplikacji pisanych dla wielu prezencji („osobowości", ang. *personalities*) SO. Windows tworzy te możliwości za pomocą wspólnego zbioru komponentów trybu jądra, które leżą u podłoża podsystemów środowiskowych SO. Implementacja każdego podsystemu środowiskowego zawiera osobny proces, który mieści dzielone struktury danych, przywileje i uchwyty do obiektów egzekutora potrzebne do realizacji danej prezencji. Proces ów jest rozpoczynany przez zarządcę sesji Windows, kiedy rusza pierwsza aplikacja danego typu. Proces podsystemu działa jako użytkownik systemowy, więc egzekutor będzie chronił jego przestrzeń adresową przed procesami wykonywanymi przez zwykłych użytkowników.

Podsystem środowiskowy realizuje graficzny interfejs użytkownika lub interfejs poleceń pisanych, dający użytkownikowi poczucie określonego SO. Ponadto każdy podsystem dostarcza API konkretnego środowiska. Oznacza to, że aplikacje utworzone pod kątem konkretnego środowiska operacyjnego wystarczy ponownie skompilować, aby mogły działać pod systemem Windows. Ponieważ interfejs systemu operacyjnego oglądany przez aplikacje jest taki sam jak ten, dla którego zostały napisane, nie trzeba modyfikować kodu źródłowego.

Model klient-serwer

Usługi SO Windows, podsystemy środowiskowe i aplikacje tworzą strukturę z wykorzystaniem modelu obliczeniowego klient-serwer, popularnego w obliczeniach rozproszonych — będzie on omówiony w części VI książki. Tę samą architekturę można zaadaptować na wewnętrzny użytek jednego systemu, jak w przypadku SO Windows.

Rodzimy interfejs NT API jest zbiorem usług opartych na jądrze, które realizują podstawowe abstrakcje używane przez system: procesy, wątki, pamięć wirtualną, wejście-wyjście i komunikację. Dzięki zastosowaniu modelu klient-serwer do realizacji poszczególnych funkcji w procesach trybu użytkownika Windows dostarcza znacznie bogatszego zbioru usług. Zarówno podsystemy środowiskowe, jak i usługi Windows w trybie użytkownika są implementowane w postaci procesów komunikujących się z klientami za pośrednictwem RPC (zdalnych wywołań procedur). Każdy proces serwera czeka na zamówienie przez klienta którejś z jego usług (np. usług dotyczących pamięci, tworzenia procesów lub sieci). Klient, który może być programem użytkowym lub programem innego serwera, zamawia usługę, wysyłając komunikat. Komunikat jest kierowany przez egzekutora do odpowiedniego serwera. Serwer wykonuje żądaną operację i zwraca wyniki lub informację o stanie za pomocą innego komunikatu, który — kierowany przez egzekutora — dociera do klienta.

Do zalet architektury klient-serwer należą:

- **Uproszczenie egzekutora.** Jest możliwe skonstruowanie wielu interfejsów API zrealizowanych w serwerach trybu użytkownika bez konfliktów lub podwojeń w egzekutorze. Nowe API można łatwo dodawać.
- **Poprawianie niezawodności.** Każdy nowy serwer działa na zewnątrz jądra, z własną częścią pamięci, chroniony przed innymi serwerami. Pojedynczy serwer może ulec awarii bez załamywania lub naruszania reszty SO.
- **Dostarczanie jednolitych środków komunikowania się aplikacji z usługami za pośrednictwem RPC bez ograniczania elastyczności.** Proces przekazywania komunikatów jest ukryty przed aplikacjami klienta za pomocą namiastek funkcji, będących niewielkimi fragmentami kodu opakowującego wywołanie RPC. Gdy aplikacja wywołuje za pomocą API podsystem środowiskowy lub usługę, namiastka w aplikacji klienta pakuje parametry wywołania i przesyła je w komunikacie do procesu serwera, który realizuje wywołanie.
- **Tworzenie bazy odpowiedniej do obliczeń rozproszonych.** W obliczeniach rozproszonych z reguły używa się modelu klient-serwer ze zdalnymi wywołaniami procedur implementowanymi za pomocą rozproszonych modułów klientów i serwerów i wymiany komunikatów między klientami i serwerami. W systemie Windows lokalny serwer może przekazywać zdalnemu serwerowi komunikaty z zamówieniami przetwarzania pochodzącymi od lokalnych aplikacji klientów. Klienci nie muszą wiedzieć, czy zamówienie jest obsługiwane lokalnie, czy zdalnie. W rzeczywistości to, czy zamówienie jest obsługiwane lokalnie, czy zdalnie, może się zmieniać dynamicznie, wedle bieżących obciążeń i stosownie do dynamicznych zmian konfiguracji.

Wątki i SMP

Ważną cechą systemu Windows jest tworzenie zaplecza wątkowości i wieloprzetwarzania symetrycznego (SMP); obie techniki przedstawiliśmy w podrozdziale 2.4. W pracy [RUSS11] wymieniono następujące cechy systemu Windows umożliwiające realizację wątków i SMP:

- Podprogramy SO mogą działać na dowolnym z dostępnych procesorów, a różne podprogramy mogą być wykonywane jednocześnie na różnych procesorach.
- Windows umożliwia użycie wielu wątków wykonania w jednym procesie, przy czym poszczególne wątki mogą działać jednocześnie na różnych procesorach.

- Procesy serwera mogą stosować wiele wątków do jednoczesnego przetwarzania zamówień wielu klientów.
- Windows udostępnia mechanizmy dzielenia danych i zasobów między procesami oraz elastyczne możliwości komunikacji międzyprocesowej.

Obiekty systemu Windows

Chociaż rdzeń Windows jest napisany w języku C, przyjęte zasady projektowe są mocno osadzone w paradygmacie projektowania obiektowego. Takie podejście ułatwia współużytkowanie zasobów i danych przez procesy oraz ochronę zasobów przed nieuprawnionym dostępem. Do najważniejszych koncepcji obiektowych zastosowanych w systemie Windows należą:

- **Obudowywanie** (ang. *encapsulation*)**.** Obiekt składa się z jednej lub więcej jednostek danych, zwanych *atrybutami*, i jednej lub więcej procedur, które mogą być wykonywane na tych danych, zwanych *usługami*. Jedynym sposobem dostępu do danych obiektu jest wywołanie którejś z jego usług. Dzięki temu dane w obiekcie można łatwo chronić przed nieupoważnionym lub niewłaściwym użyciem (np. przed próbą wykonania niewykonywalnego fragmentu danych).
- **Klasa obiektów i konkret obiektu** (ang. *object class and instance*)**.** Klasa obiektów jest szablonem (ang. *template*)[16], w którym wymieniono atrybuty i usługi obiektu i zdefiniowano pewne cechy charakterystyczne obiektu. System operacyjny może tworzyć konkretne obiekty („instancje") danej klasy stosownie do potrzeb. Mamy na przykład jedną klasę obiektu procesu i po jednym jej obiekcie dla każdego aktualnie czynnego procesu. To podejście ułatwia tworzenie obiektów i zarządzanie nimi.
- **Dziedziczenie** (ang. *inheritance*)**.** Choć implementacja jest kodowana ręcznie, egzekutor korzysta z dziedziczenia, aby rozszerzać klasy obiektów przez dodawanie nowych właściwości. Każda klasa egzekutora wywodzi się z klasy bazowej, która określa metody wirtualne tworzenia, nazywania, zabezpieczania i usuwania obiektów. Obiekty dyspozytora są obiektami egzekutora dziedziczącymi własności obiektu zdarzeń, mogą więc używać wspólnych metod synchronizacji. Inne swoiste typy obiektów, jak te z klasy urządzeń, umożliwiają klasom specyficznych urządzeń dziedziczenie z klasy bazowej i dokładanie dodatkowych danych i metod.
- **Wielopostaciowość** (ang. *polymorphism*)**.** Windows utrzymuje typowy zbiór funkcji API do manipulowania obiektami dowolnego typu; jest to własność polimorfizmu, jak zdefiniowano w dodatku D. Jednak Windows nie jest w pełni polimorficzny, gdyż istnieje wiele interfejsów API specyficznych dla obiektu określonego typu.

  Czytelnik niezaznajomiony z zasadami obiektowości powinien zajrzeć do dodatku D.

Nie wszystkie elementy Windows są obiektami. Obiekty są używane tam, gdzie przewiduje się korzystanie z danych w trybie użytkownika, lub wówczas, gdy dostęp do danych odbywa się wspólnie lub jest ograniczany. Wśród jednostek reprezentowanych w postaci obiektów znajdują się pliki, procesy, wątki, semafory, czasomierze i okna graficzne. Windows tworzy i kontroluje wszystkie typy obiektów w sposób jednolity, przy udziale zarządcy obiektów. Zarządca obiektów odpowiada za tworzenie i likwidowanie obiektów na życzenie aplikacji oraz za udzielanie dostępu do usług i danych obiektu.

[16] Użycie słowa szablon jest tu trochę niefortunne, zważywszy na jego znaczenie w rozumieniu klasy parametrycznej w języku C++; klasa jest matrycą obiektu, przepisem, według którego obiekt jest konkretyzowany — *przyp. tłum.*

Każdy obiekt w egzekutorze, niekiedy nazywany obiektem jądra (w celu odróżnienia od obiektów poziomu użytkownika, niemających związku z egzekutorem), istnieje w postaci bloku pamięci przydzielonego przez jądro i jest bezpośrednio osiągalny tylko przez komponenty trybu jądra. Niektóre elementy tej struktury danych są wspólne dla obiektów wszystkich typów (np.: nazwa obiektu, parametry bezpieczeństwa, konto użycia), inne są swoiste dla konkretnego typu obiektu (np. priorytet obiektu wątku). Ponieważ struktury danych tych obiektów znajdują się w części przestrzeni adresowej każdego procesu dostępnej tylko dla jądra, aplikacja nie może się do nich odnieść bezpośrednio, aby je czytać lub zapisywać. Zamiast tego aplikacje manipulują obiektami pośrednio, za pomocą zbioru funkcji działań na obiekcie udostępnianych przez egzekutora. Przy tworzeniu obiektu aplikacja zamawiająca tworzenie otrzymuje w odpowiedzi uchwyt do obiektu. Najkrócej mówiąc, **uchwyt** (ang. *handle*) określa pozycję w procesowej tablicy egzekutora zawierającą wskaźnik do potrzebnego obiektu. Ten uchwyt może być potem używany przez dowolny wątek tego samego procesu do wywoływania funkcji Win32 działających na obiektach lub jego kopię można przekazać do innych procesów.

Z obiektami mogą być kojarzone informacje dotyczące ich bezpieczeństwa w postaci **deskryptora bezpieczeństwa** (ang. *security descriptor* — SD). Z tych informacji można korzystać, aby ograniczać dostęp do obiektu oparty na **dowodzie** (ang. *token*) obiektu opisującym konkretnego użytkownika. Na przykład proces mógłby utworzyć nazwany obiekt semafora z zamiarem, aby tylko pewni użytkownicy mogli ten semafor otwierać i z niego korzystać. Deskryptor SD obiektu takiego semafora może wymieniać użytkowników, którym wolno (lub jest zabronione) sięgać po obiekt semafora, wraz z wykazem dozwolonych praw dostępu (czytanie, pisanie, zmiana itp.).

Obiekty w systemie Windows mogą być nazwane lub nienazwane. Jeśli proces tworzy obiekt nienazwany, zarządca obiektu zwraca uchwyt do tego obiektu i ten uchwyt jest jedynym sposobem odwołania się do niego. Uchwyty mogą być dziedziczone przez procesy potomne lub podwajane między procesami. Obiekty nazwane mają również nadawaną nazwę, za pomocą której inne, postronne procesy mogą uzyskać uchwyt do obiektu. Gdyby na przykład proces A życzył sobie synchronizacji z procesem B, mógłby utworzyć nazwany obiekt zdarzenia i przekazać nazwę zdarzenia do B. Proces B mógłby wówczas otworzyć ten obiekt zdarzenia i go używać. Gdyby jednak proces A chciał po prostu posłużyć się zdarzeniem do synchronizacji dwu swoich wątków, mógłby utworzyć nienazwany obiekt zdarzenia, ponieważ nie ma potrzeby, aby inne procesy mogły z tego zdarzenia robić użytek.

Istnieją dwie kategorie obiektów stosowanych przez Windows do synchronizacji użycia procesora:

- **Obiekty dyspozytora** (ang. *dispatcher objects*)**.** Podzbiór obiektów egzekutora, których wątki mogą czekać, aby posterować ekspediowaniem i synchronizacją operacji opartych na systemie wątków. Zostanie to opisane w rozdziale 6.
- **Obiekty sterujące** (ang. *control objects*)**.** Używane przez komponent jądra do zarządzania działaniem procesora w obszarach niepodlegających normalnemu planowaniu wątków. W tabeli 2.5 ujęto obiekty sterujące jądra.

Windows nie jest w pełni obiektowym systemem operacyjnym. Nie jest zrealizowany w języku obiektowym. Struktury danych pozostające w całości w jednej składowej egzekutora nie są reprezentowane jako obiekty. Niemniej Windows stanowi ilustrację potencjału technologii obiektowej i reprezentuje wzmożony trend do stosowania tej technologii w projektowaniu SO.

Tabela 2.5. Obiekty sterujące jądra Windows

Asynchroniczne wywołanie procedury	Używane do wkraczania w wykonywanie określonego wątku i powodowania wywołania procedury w określonym trybie procesora
Odroczone wywołanie procedury	Używane do odwlekania przetworzenia przerwania w celu uniknięcia opóźniania przerwań sprzętowych. Stosowane również do realizacji czasomierzy i do komunikacji międzyprocesorowej
Przerwanie	Używane do połączenia źródła przerwania z procedurą obsługi przerwania za pomocą wpisu w tablicy rozdzielczej przerwań (ang. *interrupt dispatch table* — IDT). Każdy procesor ma IDT, która jest używana do ekspediowania występujących w nim przerwań
Proces	Reprezentuje przestrzeń adresów wirtualnych i informacje kontrolne niezbędne do wykonywania zbioru obiektów wątków. Proces zawiera wskaźnik do mapy adresów, listę wątków gotowych do działania zawierającą obiekty wątków, listę wątków należących do procesu, całkowity nagromadzony czas wszystkich wątków działających w procesie i priorytet bazowy
Wątek	Reprezentuje obiekty wątków, zawiera priorytet planowania i przyznany kwant czasu oraz wykaz procesorów, na których dany wątek może działać
Profil	Używany do mierzenia dystrybucji czasu wykonywania w obrębie bloku kodu. Profilować można zarówno kod użytkownika, jak i kod systemu

2.8. TRADYCYJNE SYSTEMY UNIKSOWE

Historia

UNIX powstał w Bell Labs i wszedł do eksploatacji na maszynie PDP-7 w 1970 roku. Prace nad UNIX-em w Bell Labs i później, w wielu różnych ośrodkach, doprowadziły do wytworzenia całej serii odmian UNIX-a. Pierwszym godnym odnotowania kamieniem milowym okazało się przeniesienie systemu UNIX z maszyny PDP-7 na PDP-11. To była pierwsza oznaka, że UNIX mógłby stać się systemem dla wszystkich komputerów[17]. Następnym ważnym wydarzeniem było przepisanie UNIX-a w języku programowania C. W owym czasie była to nieznana nikomu strategia. W powszechnym odczuciu rzecz tak złożona jak system operacyjny, mający do czynienia ze zdarzeniami, w których czas ma znaczenie przesądzające, powinna była być pisana wyłącznie w asemblerze. Sąd ten uzasadniano następująco:

- Pamięć (zarówno RAM, jak i pomocnicza) była mała i kosztowna jak na dzisiejsze standardy, więc efektywne jej użycie miało pierwszorzędne znaczenie. Uwzględniano przy tym rozmaite techniki pamięci nakładkowej, zawierającej różny kod i różne segmenty danych, w tym również kod dokonujący samomodyfikacji.
- Mimo że kompilatory były w użyciu od lat 50 XX wieku, przemysł komputerowy odnosił się na ogół z rezerwą do jakości kodu generowanego automatycznie. Mała pojemność zasobów sprawiała, że wydajny kod, zarówno pod względem czasu, jak i przestrzeni, był nieodzowny.

[17] Maszyna PDP-11 miała odpowiednik produkowany w ZSRR pod nazwą SM-4. Również na nią dotarła taśma z UNIX-em, co ciekawe, jeszcze w 1981 roku, oficjalną przesyłką — *przyp. tłum.*

- Szybkości procesora i szyny były stosunkowo małe, więc oszczędzanie cykli zegarowych mogło mieć istotny wpływ na czas działania.

Realizacja w C wykazała przewagę zastosowania języka wysokiego poziomu w odniesieniu do większości (jeśli nie do całego) kodu systemu. Dziś w zasadzie wszystkie implementacje UNIX-a są pisane w języku C.

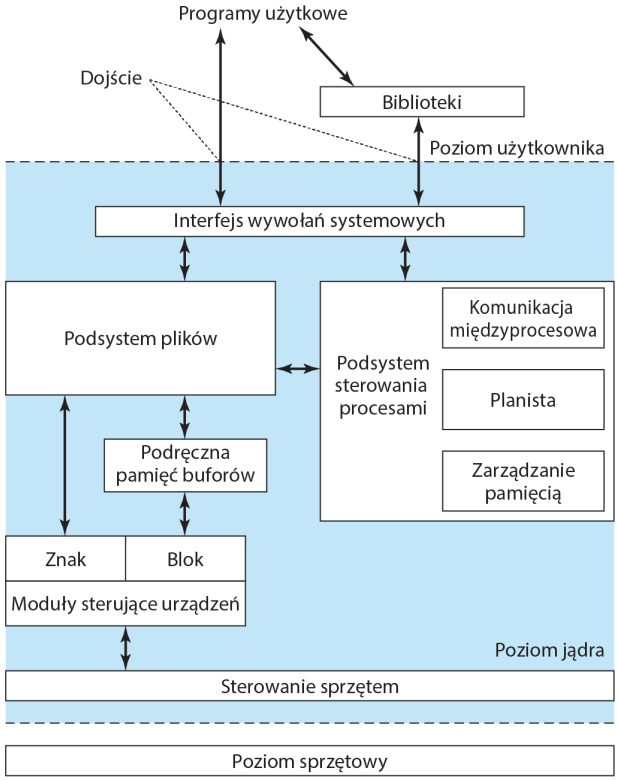
Owe wczesne wersje UNIX-a były popularne wewnątrz Bell Labs. W 1974 roku system UNIX został po raz pierwszy opisany w czasopiśmie technicznym [RITC74]. Wzbudziło to wielkie zainteresowanie tym systemem. Licencje na UNIX-a były udzielane instytucjom komercyjnym, a także uniwersytetom. Pierwszą wersją szeroko upowszechnioną poza Bell Labs była Version 6 z 1976 roku. Następna, Version 7, wydana w roku 1978, jest protoplastą większości współczesnych systemów uniksowych. Najważniejszy z systemów opracowanych poza AT&T[18] był wykonany na Uniwersytecie Kalifornijskim w Berkeley. Opatrzony nazwą UNIX BSD (Berkeley Software Distribution), pracował najpierw na PDP, a potem na komputerach VAX. W AT&T dalej rozwijano i ulepszano system. W 1982 roku w Bell Labs połączono kilka wariantów UNIX-a AT&T w jeden system, któremu nadano handlowe oznaczenie UNIX System III. Później dodano do tego systemu kolejne właściwości i tak powstał UNIX System V.

Opis

Typową architekturę UNIX-a można zobrazować w trzech poziomach: sprzętu, jądra i użytkownika. SO jest często nazywany jądrem systemu, lub po prostu jądrem, aby zaakcentować jego odizolowanie od użytkownika i aplikacji. Jądro współpracuje bezpośrednio ze sprzętem. To właśnie na jądrze UNIX-a będziemy się koncentrować w naszych odwołaniach do UNIX-a jako do przykładowego systemu w tej książce. UNIX bywa również wyposażony w sporo usług i interfejsów użytkownika, uważanych za część systemu. Można je pogrupować w **powłokę** (ang. *shell*) dostarczającą wywołań systemowych z aplikacji, oprogramowanie innych interfejsów i składowe kompilatora C (kompilator, asembler, ładowacz). Poziom ponad tym stanowią aplikacje użytkownika i interfejs użytkownika do kompilatora C.

Zarys jądra przedstawiono na rysunku 2.15. Programy użytkownika mogą wywoływać usługi SO bezpośrednio lub za pomocą programów systemowych. Interfejs wywołań systemowych graniczy z użytkownikiem i umożliwia oprogramowaniu wyższego poziomu uzyskiwanie dostępu do poszczególnych funkcji jądra. Na drugim końcu SO zawiera elementarne procedury, które współpracują bezpośrednio ze sprzętem. Między tymi dwoma interfejsami system dzieli się na dwie główne części: jedna dotyczy sterowania procesami, druga zarządzania plikami i wejściem-wyjściem. Podsystem sterowania procesami odpowiada za zarządzanie pamięcią, planowanie i ekspediowanie procesów oraz za synchronizację i komunikację międzyprocesową. System plików wymienia dane między pamięcią a urządzeniami zewnętrznymi za pomocą strumieni znaków lub bloków. Aby to osiągnąć, stosuje się rozmaite moduły sterujące urządzeń. W przypadku przesyłania blokowego znajduje zastosowanie dyskowa pamięć podręczna: między przestrzenią adresową użytkownika a urządzeniem zewnętrznym lokowany jest w pamięci głównej bufor systemowy.

[18] Zakłady Bell Labs podlegały wówczas konsorcjum AT&T — *przyp. tłum.*

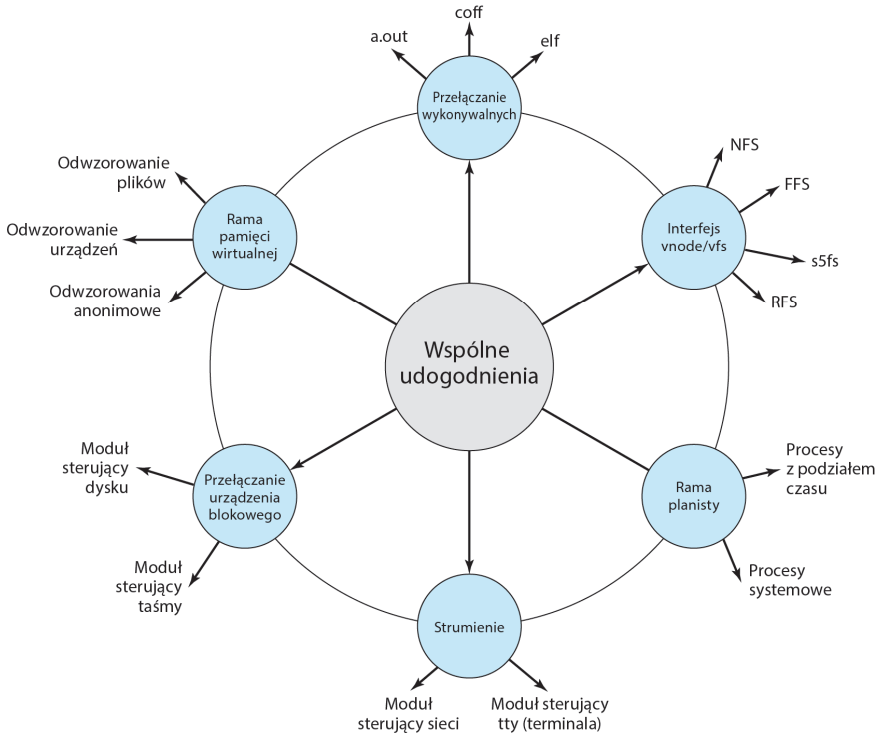


Rysunek 2.15. Architektura tradycyjnego systemu UNIX

To, co opisujemy w tym rozdziale, odnosi się do tak zwanych *tradycyjnych systemów uniksowych*; w [VAHA96] użyto tego terminu w odniesieniu do wersji System V Release 3 (SVR3), 4.3BSD i wcześniejszych. O tradycyjnym (konwencjonalnym) systemie UNIX można wypowiedzieć następujące stwierdzenia. Jest zaprojektowany do działania na pojedynczym procesorze i brakuje mu zdolności chronienia swoich struktur danych przed współbieżnym dostępem wielu procesorów. Jego jądro nie jest zbyt uniwersalne, realizując jeden typ systemu plików, politykę planowania procesów i format pliku wykonywalnego. Tradycyjne jądro UNIX-a nie jest zaprojektowane z myślą o rozszerzaniu i ma niewiele udogodnień, jeśli chodzi o wtórne wykorzystanie kodu. W rezultacie w miarę dodawania nowych cech do różnych wersji UNIX-a trzeba było dodawać dużo nowego kodu, co doprowadziło do przerośniętego i niemodularnego jądra.

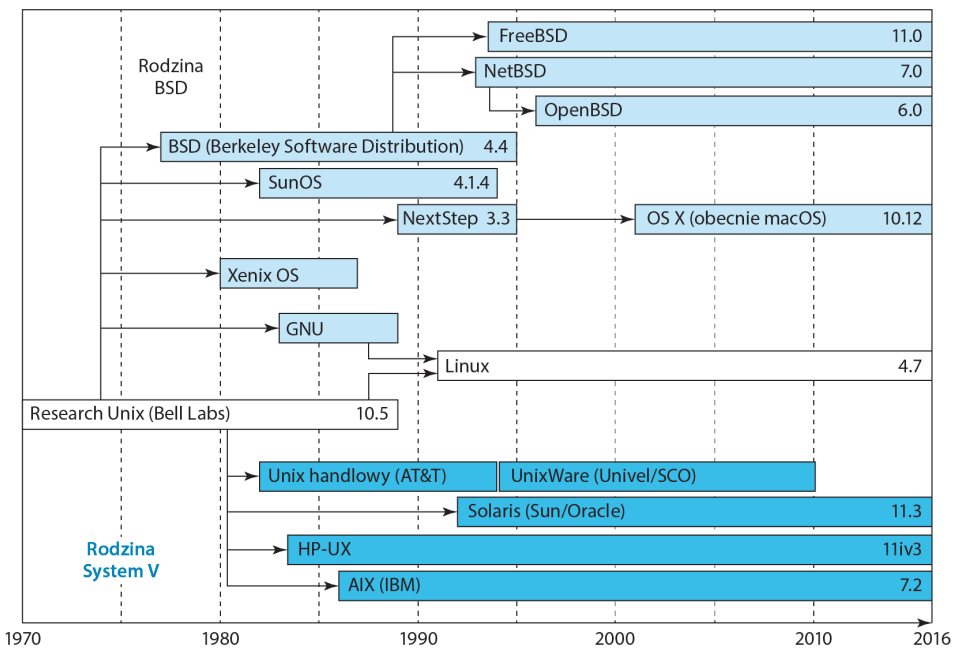
2.9. NOWOCZESNE SYSTEMY UNIKSOWE

W miarę ewolucji UNIX-a szybko mnożyły się liczne i różne jego implementacje, każda z pewnymi użytecznymi właściwościami. Pojawiła się potrzeba wytworzenia nowej implementacji, w której ujednolicono by wiele ważnych innowacji, dodano cechy projektowe innych nowoczesnych SO i utworzono bardziej modularną architekturę. Typową architekturę współczesnego jądra UNIX-a przedstawia rysunek 2.16. Mamy tu niewiele rdzennych właściwości, zapisanych modularnie, stanowiących funkcje i usługi wymagane przez pewną liczbę procesów SO. Każde z zewnętrznych kół reprezentuje funkcje i interfejs, który można zrealizować na różne sposoby.



Rysunek 2.16. Jądro nowoczesnego UNIX-a

Przejdziemy teraz do kilku przykładów nowoczesnych systemów uniksowych (rysunek 2.17).



Rysunek 2.17. Drzewo genealogiczne UNIX-a

System V Release 4 (SVR4)

SVR4 (z ang. System V, Wydanie 4), opracowany przez AT&T razem z Sun Microsystems, łączy cechy systemów SVR3, 4.3BSD, Microsoft Xenix System V i SunOS. Było to niemal całkowicie od nowa napisane jądro Systemu V, co doprowadziło do powstania czystej, mimo że złożonej, implementacji. Do nowych cech w tym wydaniu należą: przetwarzanie w czasie rzeczywistym, klasy planowania procesów, dynamiczny przydział struktur danych, zarządzanie pamięcią wirtualną, wirtualny system plików i wywłaszczalne jądro.

System SVR4 czerpie z wysiłków zarówno projektantów komercyjnych, jak i akademickich. Opracowano go w celu dostarczenia ujednoliconej platformy rozwoju handlowego UNIX-a. Pod tym względem odniósł sukces i jest być może najważniejszą odmianą UNIX-a. Łączy większość ważnych cech, jakie kiedykolwiek opracowano w którymkolwiek z systemów UNIX, przy czym robi to w sposób zintegrowany i opłacalny z handlowego punktu widzenia. SVR4 działa na szerokim asortymencie procesorów: od 32-bitowych aż po superkomputery.

System BSD

Ciąg wydań UNIX-a określanych mianem Berkeley Software Distribution (BSD) odegrał kluczową rolę w rozwoju teorii projektowania systemów operacyjnych. 4.xBSD jest szeroko stosowany w instalacjach akademickich i posłużył jako podstawa wielu handlowych produktów UNIX-a. Można zapewne bezpiecznie rzec, że UNIX swą popularność zawdzięcza po większej części systemowi BSD oraz że najwięcej ulepszeń w UNIX-ie ujrzało światło dzienne w wersjach BSD.

System 4.4BSD był ostateczną wersją BSD ukazującą się w Berkeley, której projekt i organizacja implementacji przeniknęły dalej. Jest on istotnym uaktualnieniem wersji 4.3BSD i zawiera nowy system pamięci wirtualnej, zmiany w strukturze jądra oraz długą listę innych ulepszeń.

Istnieje kilka szeroko rozpowszechnionych otwartych wersji BSD. FreeBSD jest popularny w serwerach internetowych i zaporach sieciowych, jest także używany w wielu systemach wbudowanych. NetBSD jest dostępny na wielu platformach, w tym w wielkoskalowych systemach serwerów, systemach biurkowych i urządzeniach ręcznych, znajduje też liczne zastosowania w systemach wbudowanych. OpenBSD jest otwartą wersją SO, w której specjalny nacisk położono na bezpieczeństwo.

Najnowsza wersja SO dla komputerów Macintosh, pierwotnie nazywana OS X, obecnie zaś macOS, opiera się na FreeBSD 5.0 i mikrojądrze systemu Mach 3.0.

Solaris 11

Solaris jest wydaniem UNIX-a pochodzącym z Oracle'a[19] i opartym na SVR4, jego ostatnia wersja ma numer 11. Solaris udostępnia wszystkie własności systemu SVR4, ma także liczne bardziej zaawansowane właściwości: w pełni wywłaszczalne, wielowątkowe jądro, kompletne zaplecze przetwarzania symetrycznego (SMP) i obiektowy interfejs do systemu plików. Solaris jest jedną z najszerzej stosowanych i najbardziej udanych realizacji systemu UNIX.

[19] Solaris jako taki pochodzi z Sun Microsystems, wchłoniętej później przez Oracle — *przyp. tłum.*

2.10. LINUX

Historia

Linux wystartował jako odmiana UNIX-a dla architektury IBM PC (Intel 80386). Pierwszą jego wersję napisał Linus Torvalds, fiński student informatyki. Torvalds zamieścił wczesną wersję Linuxa w Internecie w 1991 roku. Od tego czasu wiele osób, współpracując przez Internet, wniosło wkład w rozwój Linuxa, a wszystko to pod kontrolą Torvaldsa. Ponieważ Linux jest bezpłatny, a jego kod źródłowy dostępny, od początku istnienia system ten stał się alternatywną możliwością wobec innych uniksowych stacji roboczych, jak te oferowane przez Sun Microsystems i IBM. Dziś jest pełnowartościowym systemem uniksowym, działającym na niemal wszystkich platformach.

Kluczem do sukcesu Linuxa była dostępność bezpłatnych pakietów oprogramowania upowszechnianych pod auspicjami Free Software Foundation (FSF, z ang. Fundacja Wolnego Oprogramowania). Celem FSF jest stabilne oprogramowanie, niezależne od platformy, bezpłatne, wysokiej jakości i cenione przez społeczność użytkowników. Projekt GNU[20] dostarcza narzędzi twórcom oprogramowania, a licencja GNU GPL (ang. GNU Public Licence) jest udzielaną przez FSF gwarancją jakości. Torvalds, pracując nad jądrem, używał narzędzi GNU i opublikował je wtedy na licencji GPL. Tak więc dystrybucje Linuxa, które oglądacie dzisiaj, są rezultatem projektu GNU fundacji FSF, indywidualnych starań Torvaldsa i wysiłków wielu uczestników tego przedsięwzięcia na całym świecie.

Prócz wykorzystania przez wielu indywidualnych twórców Linux znacząco przeniknął obecnie do świata korporacji. Jest to efektem nie tylko bezpłatności oprogramowania, lecz również jakości jądra Linuxa. Wielu utalentowanych budowniczych dołożyło się do bieżącej wersji, tworząc produkt robiący wrażenie swoją fachowością. Co więcej, Linux jest wysoce modularny i łatwy do konfigurowania. To ułatwia „wyciskanie" optymalnego działania z rozmaitych platform sprzętowych. Zważywszy jeszcze na dostępność kodu źródłowego, dostawcy mogą podrasowywać aplikacje i narzędzia tak, aby spełniały specyficzne wymagania. Są też przedsiębiorstwa, jak Red Hat lub Canonical, które na długi czas zapewniają wysoce profesjonalne i niezawodne wsparcie swoich opartych na Linuxie dystrybucji. W różnych miejscach książki przedstawimy szczegóły wewnętrznych rozwiązań jądra Linuxa, opierając się na jądrze 4.7 upublicznionym w 2016 roku.

System operacyjny Linux dużą część sukcesu zawdzięcza swojemu modelowi rozwojowemu. Nowo wnoszony kod trafia na jedną główną listę pocztową, zwaną LKML (ang. *Linux Kernel Mailing List*). Prócz tego jest wiele innych list pocztowych przeznaczonych poszczególnym podsystemom jądra Linuxa (jak *netdev* — lista pocztowa dotycząca pracy sieciowej, *linux-pci* — podsystem PCI, *linux-acpi* — podsystem ACPI i wiele, wiele innych). Łaty wysyłane na te listy powinny spełniać ścisłe reguły (przede wszystkim konwencje stylistyczne kodowania jądra Linuxa); są one przeglądane przez fachowców z całego świata, subskrybujących wiadomości z danych list. Pod adresem tych list łaty może wysłać każdy; statystyki (np. publikowane od czasu do czasu w witrynie *lwn.net*) wykazują, że wiele łat jest przesyłanych przez deweloperów ze znanych firm handlowych, między innymi: Intel, Red Hat, Google i Samsung. Wielu opiekunów (pielęgnatorów, ang. *maintener*) również pochodzi z firm komercyjnych (np. David Miller, konserwator sieci zatrudniony w Red Hat).

[20] GNU jest rekurencyjnym akronimem pochodzącym od *GNU's Not Unix* (z ang. GNU nie jest Unixem). Projekt GNU jest zbiorem będących w wolnym obiegu pakietów oprogramowania i narzędzi służących do budowy uniksopodobnego systemu operacyjnego; jest często używany w powiązaniu z jądrem Linuxa.

Niejednokrotnie łaty takie są poprawiane dzięki wzajemnym kontaktom i dyskusjom na listach pocztowych, po czym są wysyłane ponownie i po raz kolejny sprawdzane. Dany opiekun decyduje w końcu, czy przyjąć łaty, czy je odrzucić, a ponadto każdy opiekun podsystemu od czasu do czasu wysyła zamówienie ściągnięcia swojego drzewa do głównego drzewa jądra, utrzymywanego przez Linusa Torvaldsa. Sam Linus publikuje nową wersję jądra co około 7 – 10 tygodni, a każde takie wydanie ma około 5 – 8 wersji wydań kandydujących (ang. *Release Candidate* — RC)[21].

Przy okazji warto się zastanowić, dlaczego inne systemy operacyjne o otwartym kodzie, jak różne odmiany BSD lub OpenSolaris, nie osiągnęły sukcesu i popularności, które stały się udziałem Linuxa. Powodów może być wiele, natomiast ponad wszelką wątpliwość przyczyniła się do tego otwartość modelu rozwojowego Linuxa. Jednak ten temat wybiega poza zakres naszej książki.

Struktura modularna

Większość jąder UNIX-a jest monolityczna. Przypomnijmy, co zostało już powiedziane w tym rozdziale, że jądro monolityczne zawiera w zasadzie całą funkcjonalność SO w jednym wielkim bloku kodu, działającym jako jeden proces w jednej przestrzeni adresowej. Wszystkie składowe funkcjonalne jądra mają dostęp do wszystkich jego wewnętrznych struktur danych i podprogramów. Jeśli zmiana następuje w jednym fragmencie typowego monolitycznego SO, to wszystkie moduły i podprogramy muszą być na nowo skonsolidowane, ponownie zainstalowane, po czym system musi zostać uruchomiony od początku. Dopiero wtedy skutki zmiany staną się widoczne. W rezultacie każda modyfikacja, jak dodanie modułu obsługi nowego urządzenia lub funkcji systemu plików, staje się trudna. Ten problem jest szczególnie dokuczliwy w Linuxie, którego rozwój ma skalę globalną i przebiega z udziałem luźno powiązanych grup niezależnych twórców.

Choć w Linuxie nie przyjęto koncepcji mikrojądra, osiągnięto w nim wiele potencjalnych zalet tego podejścia, dzięki jego drobiazgowo modularnej architekturze. Linux jest ustrukturyzowany w postaci kolekcji modułów, z których część może być automatycznie ładowana i rozładowywana na żądanie. Te względnie niezależne bloki są określane jako **moduły ładowalne** (ang. *loadable modules*) [GOYE99]. Zasadniczo moduł jest plikiem wynikowym, którego kod można skonsolidować z jądrem lub od niego odłączyć w trakcie działania systemu. Moduł zazwyczaj implementuje pewną specyficzną funkcję: system plików, oprogramowanie obsługi urządzenia lub jakąś inną własność górnej warstwy jądra. Moduł nie działa na zasadzie samoistnego procesu lub wątku, chociaż stosownie do potrzeb może tworzyć w różnych celach wątki jądrowe. Zamiast tego jest wykonywany w trybie jądra na zlecenie bieżącego procesu.

W ten sposób, choć Linux może być traktowany jako monolityczny, jego modularna struktura pokonuje pewne trudności związane z rozwijaniem i ewoluowaniem jądra. Ładowalne moduły jądra wyróżniają dwie ważne cechy:

1. **Dynamiczna konsolidacja** (łączenie, ang. *linking*)**.** Moduł jądra można załadować i skonsolidować z jądrem, kiedy jądro jest już w pamięci i działa. Moduł można też od jądra odłączyć i pozbyć się go z pamięci w dowolnej chwili.
2. **Stosowość modułów** (ang. *stackable modules*)**.** Moduły są zorganizowane w hierarchię. Poszczególne moduły służą jako biblioteki, jeśli odwołują się do nich moduły klienta położone wyżej w hierarchii, a gdy odwołują się do modułów położonych niżej, są traktowane jak klienci.

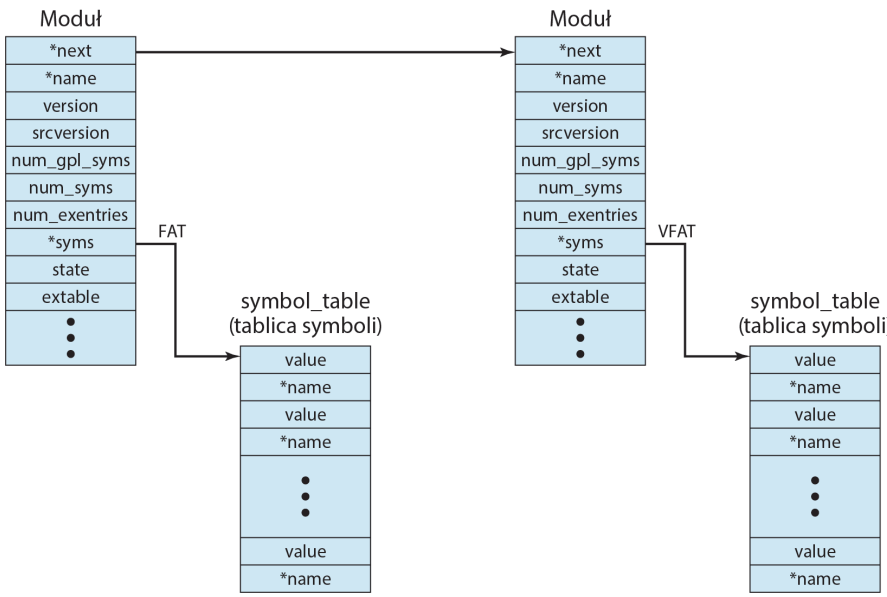
[21] Dodajmy jeszcze, że wszystko to przebiega w rozproszonym systemie kontroli wersji Git, którego autorem jest również Linus Torvalds; por. J. Loeliger, M. McCullough, *Kontrola wersji z systemem Git. Narzędzia i techniki programistów*, Helion 2014 — *przyp. tłum.*

Dynamiczna konsolidacja ułatwia konfigurowanie i oszczędza pamięć jądra [FRAN97]. W Linuxie program użytkownika lub użytkownik może jawnie ładować i rozładowywać moduły jądra, korzystając z poleceń `insmod` lub `modprobe` i `rmmod`. Jądro samo monitoruje zapotrzebowanie na poszczególne funkcje i może wprowadzać i usuwać moduły zależnie od potrzeb. Dzięki stosowości modułów można definiować zależności między modułami. Ma to dwie zalety:

1. Kod wspólny dla zbioru podobnych modułów (np. obsługujących podobny sprzęt) można przenieść do odrębnego modułu, zmniejszając zwielokrotnienie.
2. Jądro może zapewnić obecność niezbędnych modułów, powstrzymując się od rozładowania modułu, od którego zależą inne moduły, i ładując wszelkie dodatkowo wymagane moduły podczas ładowania nowego modułu.

Na rysunku 2.18 podano przykład ilustrujący struktury używane przez Linuxa do zarządzania modułami. Rysunek ukazuje listę modułów jądra po załadowaniu tylko dwóch modułów: FAT i VFAT. Każdy moduł jest zdefiniowany przez dwie tablice: tablicę modułu i tablicę symboli[22]. Tablica modułu zawiera następujące elementy:

- ***name** — nazwa modułu.
- **refcnt** — licznik modułu; licznik jest zwiększany, gdy rozpoczyna się operacja zawierająca funkcje modułu, i zmniejszany, gdy ta operacja się kończy.
- **num_syms** — liczba eksportowanych symboli.
- ***syms** — wskaźnik do tablicy symboli modułu.



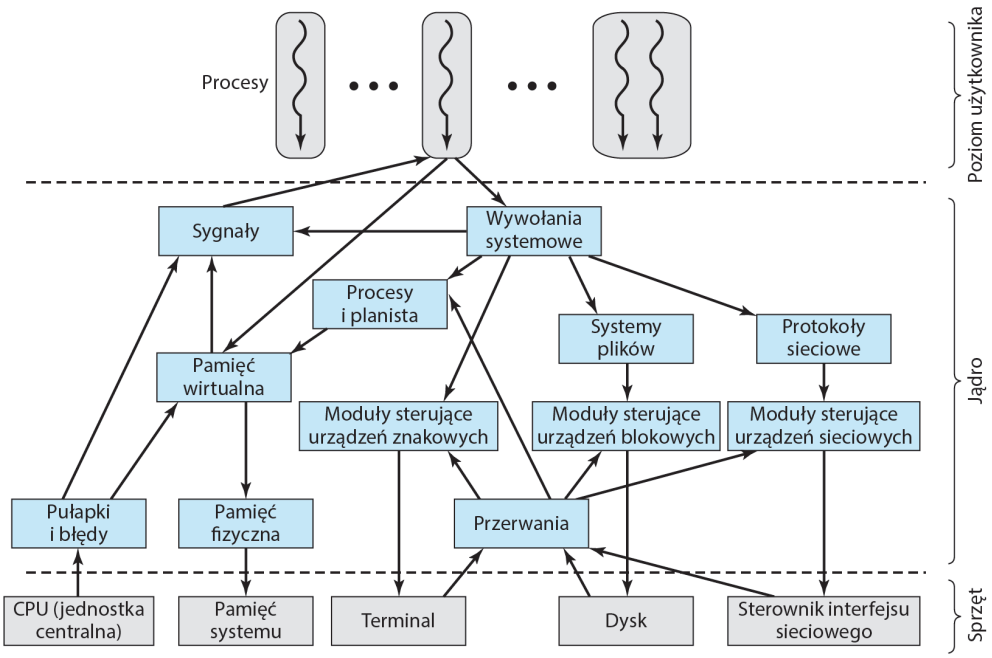
Rysunek 2.18. Przykładowa lista modułów jądra Linuxa

Tablica symboli zawiera wykaz symboli zdefiniowanych w danym module i używanych gdziekolwiek.

[22] Do czego dochodzi jeszcze tablica symboli jądra (ang. *kernel symbol table*) rejestrująca wszystkie symbole eksportowane przez ładowany moduł — *przyp. tłum.*

Składowe jądra

Na rysunku 2.19, zaczerpniętym z [MOSB02], przedstawiono główne komponenty typowej implementacji jądra Linuxa. Pokazano na nim kilka procesów działających w szczytowej partii jądra. Każda ramka symbolizuje osobny proces, natomiast linia falista ze strzałką reprezentuje wykonanie wątku. Jądro składa się ze zbioru współpracujących składowych (strzałkami zaznaczono główne interakcje). Usytuowany poniżej sprzęt jest również wyobrażony w postaci zbioru komponentów ze strzałkami wskazującymi, które z nich używają innych lub nimi sterują. Wszystkie komponenty jądra działają oczywiście na procesorze. Tych związków dla prostoty nie pokazano.



Rysunek 2.19. Składowe jądra Linuxa

W skrócie — do podstawowych składowych jądra należą:

- **Sygnały.** Jądro używa sygnałów do wkraczania do procesu. Sygnały są w szczególności wykorzystywane do powiadamiania procesu o pewnych błędach, takich jak dzielenie przez zero. W tabeli 2.6 podano kilka przykładów sygnałów.
- **Wywołania systemowe.** Wywołanie systemowe służy procesowi do zamawiania w jądrze jakiejś usługi. Wywołań systemowych jest kilkaset, można je z grubsza pogrupować w sześć kategorii: dotyczące systemu plików, procesów, planowania, komunikacji międzyprocesowej, gniazd (praca w sieci) i inne. W tabeli 2.7 zdefiniowano po kilka przykładów z każdej kategorii.
- **Procesy i planista.** Tworzy procesy oraz zajmuje się ich planowaniem i zarządzaniem nimi.
- **Pamięć wirtualna.** Przydziela pamięć wirtualną na użytek procesów i zarządza nią.
- **Systemy plików.** Dostarczają globalnej hierarchicznej przestrzeni nazw plików, katalogów i innych obiektów związanych z plikami oraz udostępniają funkcje systemu plików.
- **Protokoły sieciowe.** Realizują interfejs gniazd dla użytkowników zestawu protokołów TCP/IP.

Tabela 2.6. Niektóre sygnały Linuxa

SIGHUP	Zawieszenie terminala	SIGCONT	Kontynuuj
SIGQUIT	Zaniechanie z klawiatury	SIGTSTP	Stop z klawiatury
SIGTRAP	Pułapka śladu	SIGTTOU	Pisanie na terminalu
SIGBUS	Błąd szyny	SIGXCPU	Przekroczony limit CPU
SIGKILL	Sygnał likwidacji	SIGVTALARM	Wirtualny alarm zegarowy
SIGSEGV	Naruszenie segmentacji	SIGWINCH	Rozmiar okna niezmieniony
SIGPIPT	Przerwany potok	SIGPWR	Awaria zasilania
SIGTERM	Zakończenie	SIGRTMIN	Pierwszy sygnał czasu rzeczywistego
SIGCHLD	Stan potomka niezmieniony	SIGRTMAX	Ostatni sygnał czasu rzeczywistego

Tabela 2.7. Niektóre wywołania systemowe Linuxa

Odnoszące się do systemu plików	
`close`	Zamknij deskryptor pliku
`link`	Utwórz nową nazwę pliku
`open`	Otwórz, a być może utwórz, plik lub urządzenie
`read`	Czytaj z deskryptora pliku
`write`	Pisz do deskryptora pliku
Odnoszące się do procesów	
`execve`	Wykonaj program
`exit`	Zakończ wywołujący proces
`getpid`	Pobierz identyfikator procesu
`setuid`	Ustaw tożsamość użytkownika bieżącego procesu
`ptrace`	Zapewnij możliwość obserwowania i kontrolowania przez proces rodzicielski wykonania innego procesu oraz sprawdzania i zmiany jego obrazu pamięci i rejestrów
Odnoszące się do planowania	
`sched_getparam`	Ustaw parametry planowania związane z zasadami planowania procesu identyfikowanego przez `pid`
`sched_get_priority_max`	Zwróć maksymalną wartość priorytetu, który może być użyty w algorytmie planowania identyfikowanym przez `policy`
`sched_setscheduler`	Ustaw zarówno politykę planowania (np. FIFO), jak i związane z nią parametry w odniesieniu do procesu `pid`
`sched_rr_get_interval`	Zapisz w strukturze `timespec` wskazanej przez parametr kwant czasu algorytmu rotacyjnego w odniesieniu do procesu `pid`
`sched_yield`	Za pośrednictwem tego wywołania systemowego proces może zwolnić procesor dobrowolnie, bez blokowania. Proces może potem być przeniesiony na koniec kolejki z uwagi na jego statyczny priorytet, a działanie rozpocznie nowy proces

Tabela 2.7. Niektóre wywołania systemowe Linuxa — ciąg dalszy

Odnoszące się do komunikacji międzyprocesowej (IPC)	
`msgrev`	Przydziel strukturę bufora komunikatu do odbioru komunikatu. Wywołanie systemowe czyta wówczas komunikat z kolejki komunikatów określonej przez `msgid` do nowo utworzonego bufora
`semctl`	Wykonaj operację sterującą określoną przez `cmd` na zbiorze semaforów `semid`
`semop`	Wykonaj operacje na wybranych elementach zbioru semaforów `semid`
`shmat`	Przydziel segment pamięci dzielonej identyfikowany przez `semid` do segmentu danych procesu wywołującego
`shmctl`	Zezwól użytkownikowi na odebranie informacji z segmentu pamięci dzielonej, ustaw właściciela, grupę i pozwolenia dotyczące segmentu pamięci dzielonej albo zlikwiduj segment
Odnoszące się do gniazd (pracy sieciowej)	
`bind`	Przydziel gniazdu lokalny adres IP i port. Zwróć 0 w przypadku powodzenia lub 1, gdy wystąpił błąd
`connect`	Nawiąż połączenie między danym gniazdem a gniazdem zdalnym skojarzonym z adresem gniazda (`sockaddr`)
`gethostname`	Zwróć lokalną nazwę komputera sieciowego (hosta)
`send`	Wyślij danym gniazdem bajty zawarte w buforze wskazywanym przez `*msg`
`setsockopt`	Ustaw opcje gniazda
Inne	
`fsync`	Skopiuj na dysk wszystkie części pliku przebywające w pamięci i czekaj, aż urządzenie zaraportuje, że wszystkie części są w pamięci trwałej
`time`	Zwróć czas w sekundach odliczony od 1 stycznia 1970 roku
`vhangup`	Zasymuluj zahamowanie na aktualnym terminalu. To wywołanie pozwala innym użytkownikom mieć „czysty” terminal (tty) w czasie rozpoczynania sesji (logowania)

- **Moduły sterujące urządzeń znakowych.** Zarządzają urządzeniami, które wymagają od jądra przesyłania do lub z urządzenia po jednym bajcie jednorazowo, takimi jak terminale, modemy i drukarki.
- **Moduły sterujące urządzeń blokowych.** Zarządzają urządzeniami czytającymi i zapisującymi dane blokami; zaliczają się do nich różne odmiany pamięci pomocniczych (dyski magnetyczne, CD-ROM-y itd.).
- **Moduły sterujące urządzeń sieciowych.** Zarządzają kartami interfejsów sieciowych i portami komunikacyjnymi łączącymi z urządzeniami sieci, takimi jak mosty i rutery.
- **Pułapki i błędy.** Obsługują pułapki i błędy generowane przez procesor, takie jak błąd pamięci.
- **Pamięć fizyczna.** Zarządza pulą ramek stron w rzeczywistej pamięci i rozmieszcza strony pamięci wirtualnej.
- **Przerwania.** Ta składowa zajmuje się obsługą przerwań pochodzących od urządzeń zewnętrznych.

2.11. ANDROID

Oparty na Linuxie system operacyjny Android pierwotnie był zaprojektowany do telefonów przenośnych. Jest najpopularniejszym mobilnym SO, i to z dużą przewagą: sprzedaż telefonicznych zestawów Androida przewyższyła iPhone'y Apple'a w stosunku 4 do 1 [MORR16]. To tylko jeden aspekt rosnącej dominacji Androida. W coraz większym stopniu staje się on systemem operacyjnym prawie wszystkich urządzeń z układem komputera innych niż serwery i PC-ty. Android jest szeroko używanym SO w Internecie rzeczy.

Pierwsze prace nad SO Android wykonano w firmie Android Inc., kupionej w 2005 roku przez Google'a. Pierwsza komercyjna wersja, Android 1.0, ukazała się w 2008 roku. W czasie pisania tych słów najnowszą wersją jest Android 7.0 (Nougat). Android ma aktywną społeczność twórców i entuzjastów używających kodu źródłowego Android Open Source Project (AOSP) do opracowywania i rozpowszechniania własnych, zmodyfikowanych wersji systemu operacyjnego. Otwartość kodu źródłowego Androida stała się kluczem do jego popularności.

Architektura oprogramowania Androida

Android jest zdefiniowany w postaci stosu oprogramowania, które zawiera zmodyfikowaną wersję Linuxa, jądro, warstwę pośrednią i podstawowe aplikacje. Na rysunku 2.20 przedstawiono architekturę oprogramowania Androida nieco szczegółowiej. Na Androida należy zatem spoglądać jak na kompletny zestaw oprogramowania, nie tylko sam SO.

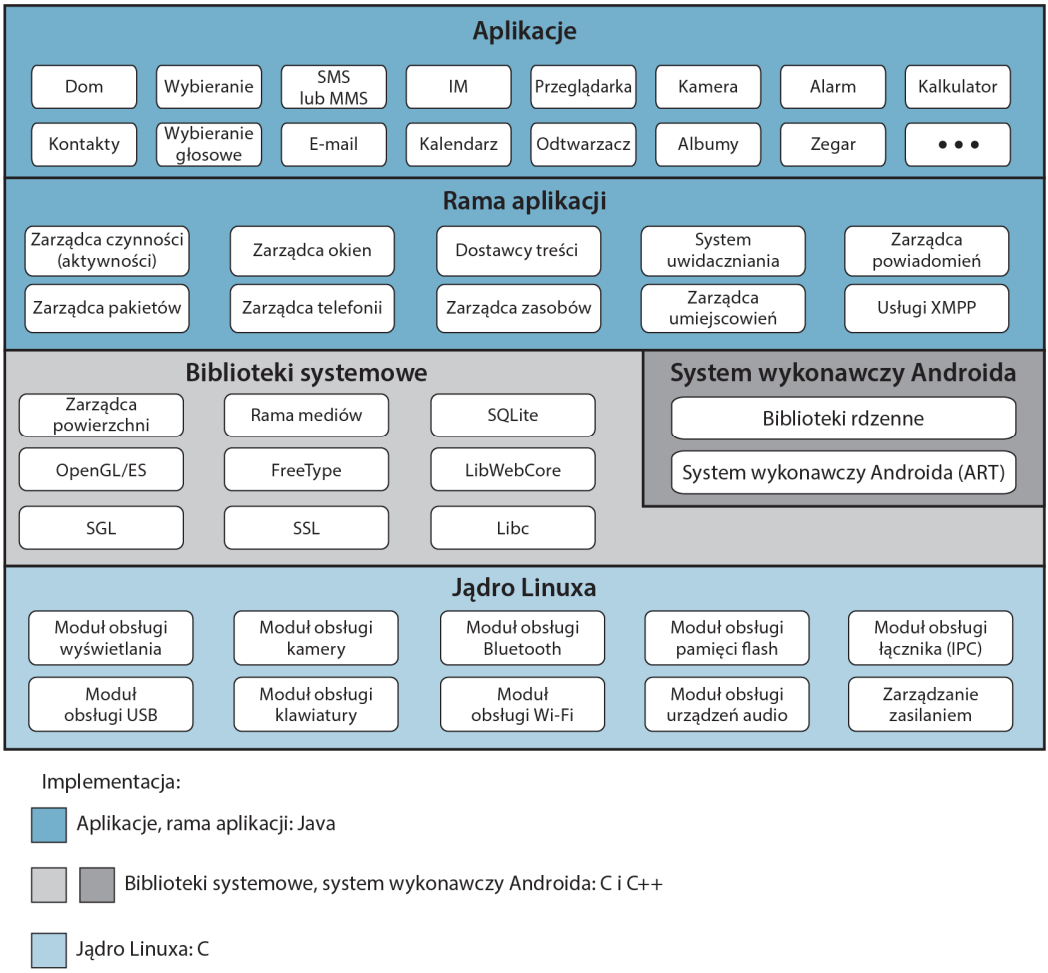
APLIKACJE

Wszystkie aplikacje, z którymi użytkownik ma bezpośredni kontakt, są częścią warstwy zastosowań. Mamy tutaj rdzenny zestaw aplikacji ogólnego przeznaczenia: klienta poczty elektronicznej, program SMS, kalendarz, mapy, przeglądarkę, kontakty i inne aplikacje będące standardowo na wyposażeniu każdego mobilnego telefonu. Aplikacje są z reguły zrealizowane w Javie. Zasadniczym celem otwartości architektury kodu źródłowego Androida jest ułatwienie budowniczym realizowania nowych aplikacji dla specyficznych urządzeń i specyficznych wymagań użytkownika docelowego. Zastosowanie Javy uwalnia budowniczych od zagadnień związanych ze sprzętem i jego osobliwościami oraz umożliwia szybkie robienie użytku z właściwości zawartych w Javie, języku wyższego poziomu, takich jak predefiniowane klasy. Na rysunku 2.20 podano przykłady podstawowych rodzajów aplikacji występujących na platformie Android.

RAMA APLIKACJI

Warstwa **ramy aplikacji** (ang. *Application Framework*) dostarcza wysokopoziomowych bloków konstrukcyjnych dostępnych za pośrednictwem ustandaryzowanych API, z których osoby programujące tworzą nowe aplikacje. Architektura jest zaprojektowana tak, aby uprościć ponowne użycie komponentów. Oto niektóre podstawowe komponenty ramy aplikacji:

- **Zarządca czynności.** Zarządza cyklem eksploatacyjnym aplikacji. Odpowiada za uruchamianie, pauzowanie i wznawianie różnych aplikacji.
- **Zarządca okien.** Jest to abstrakcja Javy usytuowana niżej zarządcy powierzchni. **Zarządca powierzchni** (ang. *Surface Manager*) manipuluje interakcją bufora ramek i niskopoziomowym rysowaniem, natomiast zarządca okien tworzy warstwę ponad nim, umożliwiając aplikacjom deklarowanie obszarów swoich klientów i używanie właściwości takich jak pasek stanu.



Rysunek 2.20. Architektura oprogramowania Androida

- **Zarządca pakietów.** Instaluje i usuwa aplikacje.
- **Zarządca telefonii.** Umożliwia interakcję z telefonem, usługi SMS-owe i MMS-owe.
- **Dostawcy treści.** Te funkcje obudowują dane aplikacji, które muszą być dzielone między aplikacjami, takie jak kontakty.
- **Zarządca zasobów.** Zarządza zasobami aplikacji, takimi jak zlokalizowane napisy (ang. *strings*) i bitmapy.
- **System uwidaczniania.** Udostępnia elementy interfejsu użytkownika (ang. *user interface* — UI) w rodzaju przycisków, wykazów, terminarzy i innych kontrolek, a także zdarzenia UI (ang. *UI Events*), takie jak dotknięcia i gesty.
- **Zarządca umiejscowienia.** Umożliwia budowniczym wprowadzanie do komputera usług opartych na lokalizacji: czy to przez GPS, czy przez identyfikatory wież telefonów komórkowych, czy też za pomocą lokalnych baz danych Wi-Fi (rozpoznane hotspoty Wi-Fi i ich stan).

- **Zarządca powiadomień.** Zarządza zdarzeniami w rodzaju nadejścia komunikatów lub terminów.
- **XMPP.** Udostępnia standaryzowane funkcje przesyłania komunikatów (także czat) między aplikacjami.

BIBLIOTEKI SYSTEMOWE

Warstwa poniżej ramy aplikacji składa się z dwu części: bibliotek systemowych i **systemu wykonawczego Androida** (ang. *Android Runtime*). Składowa **biblioteki systemowe** (ang. *System Libraries*) jest zbiorem przydatnych funkcji systemowych napisanych w C lub C++ i używanych przez różne komponenty systemu Android. Są one wywoływane z ramy aplikacji i z samych aplikacji za pośrednictwem interfejsu Javy. Te właściwości są uwidaczniane przed budowniczymi w ramie aplikacji Androida. Do niektórych z podstawowych bibliotek systemowych należą:

- **Zarządca powierzchni** (ang. *Surface Manager*)**.** Android używa kompozycyjnego zarządcy okien, podobnego do Visty lub Compiza, lecz znacznie prostszego. Zamiast rysowania bezpośrednio w buforze ekranu, polecenia rysowania trafiają do pozaekranowych bitmap, łączonych następnie z innymi bitmapami w celu utworzenia zawartości ekranu oglądanej przez użytkownika. Umożliwia to systemowi tworzenie najróżniejszych ciekawych efektów, jak widzenie poprzez okna i fantazyjne przejścia.
- **OpenGL.** OpenGL (*Open Graphics Library*, z ang. otwarta biblioteka graficzna) jest międzyjęzykowym, wieloplatformowym API do renderowania dwu- i trójwymiarowej grafiki komputerowej. OpenGL/ES (OpenGL do systemów wbudowanych) jest podzbiorem OpenGL zaprojektowanym z przeznaczeniem dla systemów wbudowanych.
- **Rama mediów.** Rama mediów (ang. *Media Framework*) wspomaga wideozapis i odtwarzanie w różnych formatach, w tym ACC, AVC (H.264), H.263, MP3 i MPEG-4.
- **Baza danych SQL.** Android zawiera lekki mechanizm[23] bazy danych SQLite do przechowywania trwałych danych. SQLite jest omówiona w następnym podrozdziale.
- **Mechanizm przeglądarki** (ang. *Browser Engine*)**.** Do szybkiego wyświetlania treści w HTML-u Android używa biblioteki WebKit, będącej w istocie tą samą biblioteką co stosowana w przeglądarce Safari i iPhone'ach. Biblioteka ta była też używana w przeglądarce Google Chrome, dopóki Google nie przeszedł na Blink.
- **Bionic LibC.** Jest to zredukowana wersja standardowej biblioteki systemu C, dostrojona do urządzeń wbudowanych opartych na Linuxie. Interfejsem jest standardowy JNI (Java Native Interface, z ang. rodzimy interfejs Javy).

JĄDRO LINUXA

Jądro SO zastosowane w Androidzie jest podobne do dystrybucji standardowego jądra Linuxa. Godną odnotowania zmianą w jądrze Androida jest brak modułów sterujących urządzeniami nienadającymi się do środowisk mobilnych, dzięki czemu jądro jest mniejsze. Ponadto Android wzbogaca jądro Linuxa o cechy przykrojone na miarę środowiska mobilnego i na ogół niezbyt przydatne na platformach biurkowych czy w laptopach.

[23] Ang. *engine*. Powszechnie określa się to mianem „silnik", jednak z uwagi na ochronę środowiska proponujemy ogólniejsze określenie (by nie używać terminu „maszyneria") — *przyp. tłum.*

Android bazuje na swoim linuksowym jądrze, jeśli chodzi o rdzenne usługi systemowe: bezpieczeństwo, zarządzanie pamięcią, zarządzanie procesami, stos sieciowy i model programów obsługi urządzeń. Jądro działa również jako warstwa abstrakcji pomiędzy sprzętem a pozostałym stosem oprogramowania i umożliwia Androidowi korzystanie z szerokiego asortymentu udostępnianych w Linuxie modułów sterujących sprzętem.

Środowisko wykonawcze Androida

Większość systemów operacyjnych używanych w urządzeniach mobilnych, jak iOS lub Windows, stosuje oprogramowanie kompilowane bezpośrednio na konkretną platformę sprzętową. W Androidzie większość oprogramowania jest natomiast odwzorowywana na format bajtokodu, następnie przekształcany na rodzime instrukcje danego urządzenia. We wcześniejszych wydaniach Androida stosowano schemat o nazwie Dalvik. Jednak Dalvik ma sporo ograniczeń, jeśli chodzi o skalowanie na większe pamięci i architektury wielordzeniowe, toteż nowsze wydania Androida bazują na schemacie zwanym Android Runtime (ART). ART jest w pełni zgodny z istniejącym formatem bajtokodowym Dalvika (dex, od Dalvik Executable), więc twórcy aplikacji nie muszą zmieniać swojego kodu, aby działał on pod ART-em. Przyjrzymy się najpierw Dalvikowi, a potem spojrzymy na ART.

MASZYNA WIRTUALNA DALVIK

Dalvik VM (DVM) wykonuje pliki w formacie *.dex*, optymalizowanym pod kątem wydajnego magazynowania i wykonywania odwzorowywalnego w pamięci. VM może wykonywać klasy skompilowane przez kompilator języka Java, przekształcone na jego rodzimy format z zastosowaniem dołączonego narzędzia „dx". VM działa powyżej jądra Linuxa, wykorzystując jego funkcjonalność (np. wątkowość i niskopoziomowe zarządzanie pamięcią). Rdzenna biblioteka klas Dalvika ma w założeniu dostarczać znanej bazy rozwojowej tym, którzy przywykli do programowania w Java Standard Edition, lecz jest nastawiona specjalnie na wymagania małych urządzeń przenośnych.

Każda aplikacja Androida działa we własnym procesie, z własnym egzemplarzem maszyny wirtualnej Dalvik. Dalvika napisano w taki sposób, aby urządzenie mogło efektywnie wykonywać wiele maszyn wirtualnych.

FORMAT PLIKU DEX

Maszyna wirtualna DVM wykonuje aplikacje i kod napisany w Javie. Standardowy kompilator Javy przekształca kod źródłowy (zapisany w postaci pliku tekstowego) na bajtokod. Bajtokod jest następnie kompilowany na plik *.dex*, nadający się do czytania i użytku przez DVM. W istocie pliki klas są zamieniane na pliki *.dex* (na podobieństwo pliku *.jar*, gdyby używano standardowej maszyny wirtualnej Javy — JVM), po czym są czytane i wykonywane przez DVM. Powielone dane używane w plikach klas są włączane do pliku *.dex* tylko raz, co oszczędza pamięć i jest mniej kosztowne. Pliki wykonywalne można ponownie modyfikować podczas instalowania aplikacji, aby jeszcze bardziej optymalizować je pod kątem urządzenia mobilnego.

KONCEPCJE ŚRODOWISKA WYKONAWCZEGO ANDROIDA

ART jest aktualnym środowiskiem wykonawczym aplikacji używanym przez Androida, wprowadzonym w wersji Android 4.4 (KitKat). Pierwszy projekt Androida zakładał, że system ten będzie pracował na jednym rdzeniu (z minimalnym wsparciem sprzętowym wielowątkowości) i z urzą-

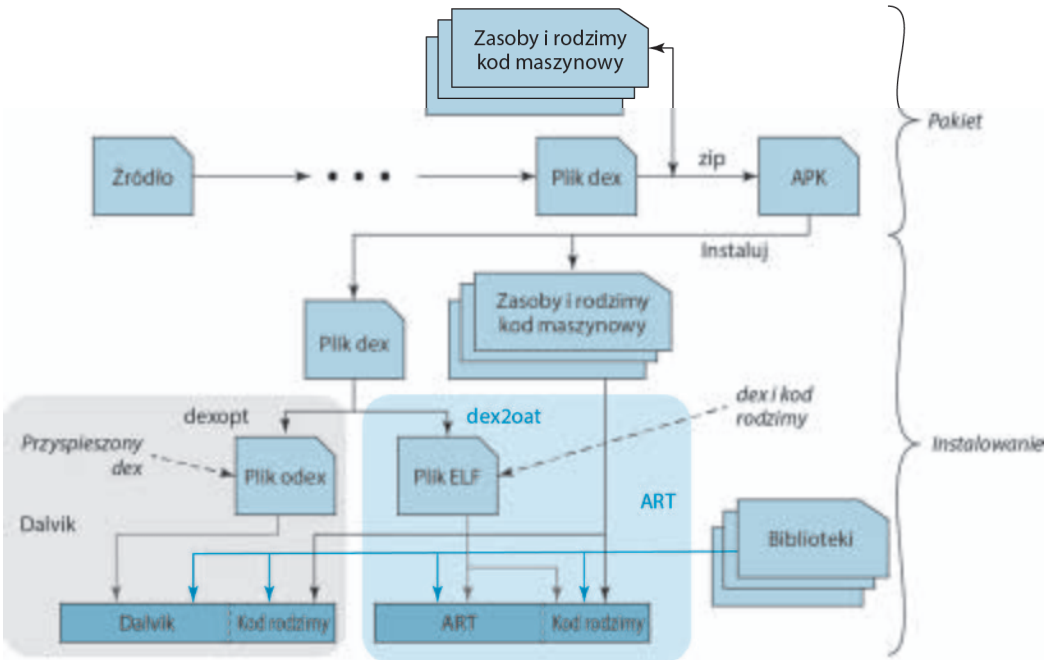
dzeniami o małych pamięciach, dla których Dalvik wydawał się odpowiednim środowiskiem wykonawczym. Jednak ostatnimi czasy urządzenia, na których działa Android, mają procesory wielordzeniowe i więcej pamięci (o względnie umiarkowanej cenie), co spowodowało, że w Google'u powtórnie przemyślano projekt środowiska wykonawczego, aby dostarczyć budowniczym aplikacji i użytkownikom więcej wrażeń przez umożliwienie wykorzystania własności będącego w obiegu sprzętu wysokiej klasy.

Zarówno na użytek Dalvika, jak i ART-a aplikacje pisane w Javie są kompilowane na bajtokod dex. Jako że Dalvik używa formatu bajtokodu dex ze względu na przenośność, format ten musi być zamieniany (kompilowany) na kod maszynowy, aby mógł być naprawdę wykonany przez procesor. Środowisko wykonawcze Dalvik dokonuje konwersji z bajtokodu dex na rodzimy kod maszynowy podczas biegu aplikacji, a proces ten nazwano JIT (ang. *just-in time* — kompilacja terminowa)[24]. Ponieważ JIT kompiluje tylko część kodu, w mniejszym stopniu odciska się w pamięci i używa mniej fizycznej przestrzeni w urządzeniu. (W pamięci trwałej są przechowywane tylko pliki dex, w przeciwieństwie do rzeczywistego kodu maszynowego). Dalvik identyfikuje często wykonywaną sekcję kodu i przechowuje ją po jednorazowym skompilowaniu w pamięci podręcznej, dzięki czemu kolejne wykonania takiej sekcji są szybsze. Strony pamięci fizycznej przechowujące podręcznie kod nie są wymienialne ani nie podlegają stronicowaniu, co również odciska się nieco na pamięci, jeśli system jest już w takim stanie. Nawet z tymi optymalizacjami Dalvik musi wykonywać kompilację terminową przy każdym przebiegu aplikacji, co zabiera sporo zasobów procesora. Zauważmy, że procesor jest używany nie tylko do faktycznego wykonywania aplikacji, lecz także do konwersji bajtokodu dex na kod rodzimy, co jest przyczyną „drenowania" mocy. Takie użycie procesora było również powodem słabego odbioru interfejsu przez użytkowników w niektórych aplikacjach mocno obciążanych w fazie początkowej.

Aby obejść część tych problemów i zwiększyć efektywność zastosowania dostępnego sprzętu wysokiej klasy, w Androidzie wprowadzono ART. Środowisko ART również wykonuje bajtokod dex, lecz zamiast kompilowania bajtokodu w fazie wykonywania ART kompiluje bajtokod na kod maszynowy podczas instalowania aplikacji. Nazywa się to kompilacją zawczasu (ang. *ahead of time* — AOT). Do wykonywania tej kompilacji w fazie instalowania ART używa narzędzia `dex2oat`. Narzędzie to daje na wyjściu plik, który jest wykonywany podczas działania aplikacji.

Na rysunku 2.21 pokazano realizowany cykl życia APK, pakietu aplikacji przychodzącego od jej wykonawcy do użytkownika. Cykl rozpoczyna się od skompilowania kodu źródłowego na format *.dex* i połączenia go z odpowiednim kodem uzupełniającym w celu utworzenia APK. Po stronie użytkownika otrzymany APK jest rozpakowywany. Zasoby i rodzimy kod są na ogół instalowane wprost w katalogu aplikacji. Jednak kod *.dex* wymaga dalszego przetwarzania, zarówno w przypadku Dalvika, jak i ART-a. W Dalviku do wyprodukowania z pliku *.dex* zoptymalizowanej wersji formatu dex (odex), określanego jako przyspieszony dex, jest używana funkcja o nazwie `dexopt`. Chodzi o to, aby uczynić kod dex szybszy w interpretacji przez interpreter dexa. W ART funkcja `dex2oat` wykonuje optymalizację tego samego typu: również kompiluje kod dex na kod maszynowy docelowego urządzenia. Wynikiem funkcji `dex2out` jest plik typu *Executable and Linkable Format* (ELF, z ang. format wykonywalny i konsolidowalny), który działa bezpośrednio, bez interpretera.

[24] Czyli w samą porę, na czas; przed blisko dwiema dekadami tak samo nazwano ten sposób interpretacji kodu w Javie — *przyp. tłum.*



Rysunek 2.21. Cykl życia APK (pakietu aplikacji)

ZALETY I WADY

Korzyści z zastosowania ART-a są następujące:

- Zmniejsza się czas rozruchu aplikacji, ponieważ jest wykonywany bezpośrednio kod rodzimy (maszynowy).
- Poprawia się żywotność baterii, gdyż unika się używania procesora do kompilacji terminowej (JIT).
- Ślad wymagany w pamięci do działania aplikacji jest mniejszy (bo nie ma zapotrzebowania na pamięć podręczną JIT). Co więcej, skoro nie ma niestronicowalnego kodu JIT do podręcznego przechowywania, uelastycznia to wykorzystanie pamięci RAM w sytuacjach, gdy zaczyna jej brakować.
- Wraz z ART-em przybyło parę optymalizacji przy łączeniu nieużytków i ulepszeń dotyczących uruchamiania kodu (oczyszczania go z błędów).

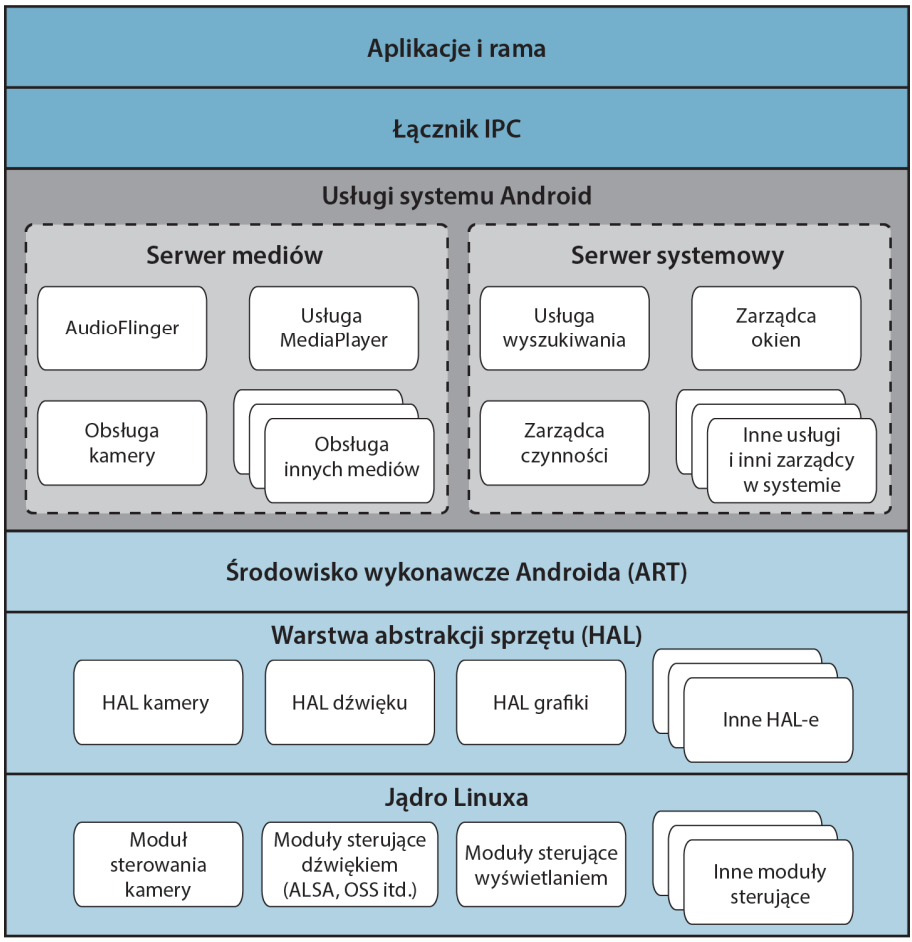
ART ma również kilka potencjalnych wad:

- Ponieważ zamiana bajtokodu na kod maszynowy jest wykonywana w czasie instalowania, instalowanie aplikacji zajmuje więcej czasu. Dla twórców aplikacji Androida, którzy ładują aplikację wiele razy podczas testowania, ten czas może być zauważalny.
- Podczas pierwszego, świeżego rozruchu po przywróceniu ustawień fabrycznych wszystkie aplikacje zainstalowane na urządzeniu są kompilowane na kod rodzimy za pomocą funkcji `dex2opt`. Dlatego w porównaniu z Dalvikiem pierwszy rozruch może trwać znacznie dłużej (3 – 5 sekund), nim pojawi się ekran powitalny (ang. *Home Screen*).

- Wygenerowany w ten sposób kod rodzimy zostaje przechowany w pamięci wewnętrznej, co wymaga znacznej dodatkowej jej ilości.

Architektura systemu Android

Pożyteczne jest przedstawienie Androida z perspektywy twórcy aplikacji, co ukazano na rysunku 2.22. Architektura tego systemu jest uproszczoną abstrakcją architektury oprogramowania z rysunku 2.20. Android oglądany w tym ujęciu składa się z następujących warstw:



Rysunek 2.22. Architektura systemu Android

- **Aplikacje i rama.** Twórcy aplikacji są skoncentrowani przede wszystkim na tej warstwie i na interfejsach API, które umożliwiają dostęp do usług niższej warstwy.
- **Łącznik IPC.** Mechanizm łącznika o nazwie Binder Inter-Process Communication umożliwia ramie aplikacji przekraczanie granic procesu i wywoływanie kodu usług systemu Android. To zasadniczo umożliwia ramowym interfejsom API wysokiego poziomu interakcję z usługami systemu Android.

- **Usługi systemu Android.** Większość funkcji uzewnętrznianych za pośrednictwem ramy interfejsów API wywołuje usługi systemu, które z kolei zwracają się do usytuowanych niżej funkcji sprzętowych i jądrowych. Usługi można postrzegać jako zorganizowane w dwie grupy: usługi medialne odnoszą się do odtwarzania i nagrywania mediów, a usługi systemowe realizują funkcje na poziomie systemu, takie jak zarządzanie energią, położeniem (lokalizacją) i powiadamianiem.
- **Warstwa abstrakcji sprzętu (HAL).** HAL (ang. *hardware abstraction layer*) realizuje standardowy interfejs do modułów sterujących urządzeń na poziomie jądra, więc kod górnej warstwy nie musi uwzględniać szczegółów implementacji specyficznych modułów sterujących i sprzętu. Warstwa HAL w zasadzie nie różni się od występującej w standardowej dystrybucji Linuxa. Służy do abstrahowania możliwości poszczególnych urządzeń (udostępnianych przez sprzęt i eksponowanych przez jądro) z przestrzeni użytkownika. Przestrzeń użytkownika może zawierać usługi Androida lub aplikacje. Celem warstwy HAL jest utrzymywanie przestrzeni użytkownika spójnej w odniesieniu do różnych urządzeń. Dostawcy mogą również dokonywać własnych ulepszeń i zamontowywać je w warstwie HAL bez oddziaływania na przestrzeń użytkownika. Przykładem takiego postępowania jest HwC (ang. *Hardware Composer* — kompozytor sprzętu) będący zależną od dostawcy implementacją HAL, która rozumie zdolności renderowania sprzętu znajdującego się poniżej. Zarządca powierzchni gładko współpracuje z różnymi implementacjami HwC, pochodzącymi od różnych dostawców.
- **Jądro Linuxa.** Jądro Linuxa jest przykrojone na miarę wymagań środowiska mobilnego.

Czynności

Czynność (aktywność, ang. *activity*) jest pojedynczym wizualnym komponentem interfejsu użytkownika, zawierającym obiekty takie jak wybory z menu, ikony i pola kontrolne. Każdy ekran aplikacji jest rozszerzeniem klasy Activity. Czynności korzystają z widoków (Views), aby tworzyć graficzne interfejsy użytkownika, wyświetlające informacje i reagujące na działania użytkownika. Czynności omówimy w rozdziale 4.

Zarządzanie zasilaniem

Android uzupełnia jądro Linuxa o dwie cechy mające usprawnić zdolność zarządzania zasilaniem. Są to alarmy i blokowanie czuwania (ang. *wakelocks*).

Możliwość alarmów jest zrealizowana w jądrze Linuxa i widoczna dla budowniczego aplikacji za pośrednictwem zarządcy alarmów (AlarmManager) w rodzimych bibliotekach środowiska wykonawczego (RunTime). Poprzez zarządcę alarmów aplikacja może zamówić usługę terminowego budzenia. Możliwość alarmów jest implementowana w jądrze, zatem alarm może zadziałać nawet wówczas, gdy system jest w trybie uśpienia. Umożliwia to systemowi przechodzenie w stan uśpienia i oszczędzanie energii także wtedy, gdy jest jakiś proces, który wymaga budzenia.

Blokowanie czuwania chroni system Android przed wejściem w stan uśpienia. Aplikacja może utrzymywać jedną z następujących blokad czuwania:

- `Full_Wake_Lock` (z ang. pełna blokada czuwania) — procesor jest włączony, pełna jasność ekranu, klawiatura czynna.
- `Partial_Wake_Lock` (z ang. częściowa blokada czuwania) — procesor włączony, ekran wyłączony, klawiatura wyłączona.
- `Screen_Dim_Wake_Lock` — procesor włączony, ekran przyciemniony, klawiatura wyłączona.
- `Screen_Bright_Wake_Lock` — procesor włączony, ekran jasny, klawiatura wyłączona.

Te blokady są zamawiane poprzez API, ilekroć aplikacja zażąda, aby któreś z urządzeń zewnętrznych będących pod kontrolą pozostało włączone. Jeśli nie istnieje żadna blokada czuwania blokująca urządzenie, zostaje ono wyłączone ze względu na dbałość o żywotność baterii.

Te obiekty jądra są widoczne dla aplikacji w przestrzeni użytkownika za pomocą plików */sys/power/wavelock*. Plików *wake_lock* i *wake_unlock* można użyć do zdefiniowania i przełączania blokady przez odpowiedni zapis w stosownym pliku.

2.12. PODSTAWOWE POJĘCIA, PYTANIA SPRAWDZAJĄCE I ZADANIA

Podstawowe pojęcia

Adres fizyczny	Planowanie	System z podziałem czasu
Adres rzeczywisty	Podział czasu	Średni czas do wystąpienia awarii (MTTF)
Adres wirtualny	Praca	Średni czas naprawy (MTTR)
Czas pracy	Proces	Tryb jądra
Jądro (ang. kernel, *nucleus*)	Programowanie seryjne	Tryb użytkownika
Jądro monolityczne	Projektowanie obiektowe	Wada
Jednoprogramowość	Przerwanie	Wątek
Język sterowania zadaniami (JCL)	Przestój	Wieloprogramowość
Kontekst wykonywania	Przetwarzanie wsadowe	Wieloprogramowy system operacyjny
Kwantowanie czasu	Rotacyjny (algorytm)	Wieloprzetwarzanie symetryczne (SMP)
Maszyna wirtualna	Rozkaz uprzywilejowany	Wielowątkowość
Mikrojądro	Rozproszony system operacyjny	Wielozadaniowość
Moduły ładowalne	Stan	Zadanie
Monitor	Stan procesu	Zarządzanie pamięcią
Monitor rezydentny	System operacyjny	
Niezawodność	System wsadowy	
Pamięć wirtualna		

Pytania sprawdzające

2.1. Jakie są trzy cele projektowania SO?

2.2. Co to jest jądro SO?

2.3. Co to jest wieloprogramowość?

2.4. Co to jest proces?

2.5. Jak wygląda kontekst wykonywania procesu używany przez SO?

2.6. Wymień i krótko omów pięć powinności typowego SO dotyczących zarządzania pamięcią.

2.7. Wyjaśnij różnicę między adresem rzeczywistym a adresem wirtualnym.

2.8. Opisz technikę planowania rotacyjnego.

2.9. Wyjaśnij różnicę między jądrem monolitycznym a mikrojądrem.

2.10. Co to jest wielowątkowość?

2.11. Wymień podstawowe zagadnienia projektowania systemu operacyjnego typu SMP.

Zadania

2.1. Przypuśćmy, że mamy wieloprogramowany komputer, w którym każde zadanie ma identyczną charakterystykę. W jednym przydzielonym zadaniu okresie obliczeń T połowa czasu jest spędzana na operacjach wejścia-wyjścia, a połowa na zatrudnianiu procesora. Każde zadanie działa przez N takich okresów. Zakładając proste planowanie rotacyjne oraz możliwość zachodzenia w tym samym czasie działań na wejściu-wyjściu i w procesorze, określ następujące wielkości:

- czas cyklu przetwarzania = faktyczny czas do zakończenia zadania;
- przepustowość = średnia liczba zadań kończonych w okresie T;
- wykorzystanie procesora = procent czasu, w którym procesor jest aktywny (nie czeka).

Oblicz te wielkości dla jednego, dwóch i czterech zadań wykonywanych jednocześnie, zakładając, że okres T podlega następującej dystrybucji:

a. Pierwsza połowa idzie na wejście-wyjście, druga na procesor.

b. Wejście-wyjście w pierwszej i czwartej ćwierci, procesor w drugiej i trzeciej ćwierci.

2.2. **Program ograniczony przez wejście-wyjście** (ang. *I/O-bound program*) to taki, który działając samotnie, spędzałby więcej czasu na oczekiwaniu na wejście-wyjście niż na używaniu procesora. **Program ograniczony przez procesor** (ang. *CPU-bound program*) zachowuje się odwrotnie. Załóżmy, że algorytm planowania krótkoterminowego faworyzuje programy, które ostatnio zużywały mniej czasu procesora. Wyjaśnij, dlaczego ten algorytm faworyzuje programy ograniczone przez wejście-wyjście, a jednak nie odmawia stale czasu procesora programom ograniczonym przez procesor.

2.3. Porównaj zasady planowania nadające się do użycia, gdy chodzi o optymalizację systemu z podziałem czasu, z tymi, z których należałoby skorzystać do optymalizowania **wieloprogramowego systemu wsadowego**.

2.4. Co jest celem wywołań systemowych? Jak wywołania systemowe mają się do SO i do koncepcji dualnego trybu działania (trybu jądra i trybu użytkownika)?

2.5. W systemie operacyjnym OS/360 komputerów głównych IBM jednym z ważniejszych modułów w jądrze jest zarządca zasobów systemu (ang. *System Resource Manager* — SRM). Moduł ten odpowiada za rozdzielnictwo zasobów między przestrzenie adresowe (procesy). SRM sprawia, że stopień wyrafinowania OS/360 jest niespotykany w systemach operacyjnych. Żaden inny SO komputera głównego (ang. *mainframe*) i z pewnością żaden SO innego typu nie może się równać z funkcjami wykonywanymi przez SRM. Pojęcie zasobu obejmuje procesor, pamięć rzeczywistą i kanały wejścia-wyjścia. SRM gromadzi statystyki dotyczące wykorzystania procesora, kanału i rozmaitych kluczowych struktur danych. Jego celem jest osiąganie optymalnego działania na podstawie monitorowania i analizy tych danych. W danej instalacji z góry ustala się różne cele efektywności służące jako wskazówki dla SRM, który dynamicznie modyfikuje instalację i charakterystykę wydajnościową zadań na podstawie wykorzystania systemu. Z drugiej strony, SRM dostarcza raporty, które doświadczonemu operatorowi umożliwiają doskonalenie konfiguracji i ustawień parametrów w celu poprawiania obsługi użytkowników.

Zadanie dotyczy jednego przykładu aktywności SRM. Rzeczywista pamięć jest podzielona na bloki równej długości, zwane ramkami, przy czym mogą ich być tysiące. W każdej ramce może znajdować się blok pamięci wirtualnej określany jako strona. SRM otrzymuje sterowanie w przybliżeniu 20 razy na sekundę i sprawdza każdą ramkę bez wyjątku. Jeśli do strony nie było odniesień ani nie została ona zmieniona, licznik jest zwiększany o 1. W miarę upływu czasu SRM uśrednia te wartości, aby określić średnią liczbę sekund, przez które ramka strony w systemie pozostawała nietknięta. Jaki może być tego cel i jakie działanie mógłby podjąć SRM?

2.6. Wieloprocesor o ośmiu procesorach ma przydzielonych 20 przewijaków taśmy. W systemie zlecono do wykonania dużą liczbę zadań, przy czym każde wymaga do zakończenia działania najwyżej czterech przewijaków. Załóżmy, że każde zadanie rozpoczyna działanie tylko z trzema przewijakami na długo przed zapotrzebowaniem czwartego krótko przed końcem. Załóżmy również nieustanne dostarczanie takich zadań.

a. Zakładamy, że planista w SO nie rozpocznie zadania dopóty, dopóki nie będą dostępne cztery przewijaki taśmy. Po rozpoczęciu zadania cztery przewijaki są przydzielane natychmiast i nie są zwalniane aż do skończenia zadania. Ile zadań maksymalnie będzie w trakcie jednoczesnego wykonywania? Ile wynosi maksymalna i minimalna liczba przewijaków taśmy, które mogą pozostawać bezczynne wskutek zastosowania takiej polityki przydziału?

b. Zaproponuj alternatywną politykę, aby poprawić wykorzystanie przewijaków taśmy, a jednocześnie uniknąć zakleszczenia w systemie. Ile wyniesie maksymalna liczba zadań, które mogą być wykonywane jednocześnie? Jakie są ograniczenia liczby próżnujących przewijaków taśmy?

SKOROWIDZ

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

T

U

V

W

Z

Ż

μ

Systemy operacyjne cały czas są wzbogacane o innowacje i ulepszenia. Stają się też coraz bardziej wyspecjalizowane, co wynika z rosnącej różnorodności maszyn, które mają obsłużyć. Wystarczy tu wskazać systemy wbudowane, smartfony, komputery osobiste i komputery główne (ang. *mainframe*) oraz superkomputery, a także specjalne systemy czasu rzeczywistego. Znajomość wewnętrznych mechanizmów systemu i architektury jądra okazuje się zatem niezwykle istotna dla programistów i inżynierów. Bez tej wiedzy trudno mówić o niezawodności tworzonego oprogramowania.

Ta książka jest kolejnym, gruntownie przejrzanym i zaktualizowanym wydaniem klasycznego podręcznika, w którym jasno i wyczerpująco wyjaśniono koncepcje, strukturę i mechanizmy rządzące funkcjonowaniem nowoczesnych systemów operacyjnych. Wnikliwie omówiono również podstawowe zasady projektowania systemów operacyjnych i powiązano je ze współczesnymi zagadnieniami projektowymi oraz kierunkami rozwoju systemów operacyjnych. Aby zilustrować prezentowane treści, odniesiono się do czterech systemów: Windows, Android, Unix i Linux. W ten sposób koncepcje projektowe omawiane w danym rozdziale są natychmiast popierane rzeczywistymi przykładami.

Najważniejsze zagadnienia:

- Przegląd systemów operacyjnych
- Współbieżność i rozproszone zarządzanie procesami
- Zarządzanie pamięcią i pamięć wirtualna
- Bezpieczeństwo systemów operacyjnych
- Operacje wejścia-wyjścia i zarządzanie przestrzenią dyskową
- Odporność na awarie

DR WILLIAM STALLINGS jest autorem wielokrotnie nagradzanych książek poświęconych bezpieczeństwu systemów informatycznych, sieciom i architekturze komputerów. Projektował i implementował zestawy protokołów sieciowych na różne komputery i systemy operacyjne. Doradzał agencjom rządowym oraz dostawcom komputerów i oprogramowania w kwestiach projektowania i użytkowania oprogramowania i produktów sieciowych. Jest członkiem kolegium redakcyjnego czasopisma „Cryptologia", naukowego periodyku poświęconego zagadnieniom szyfrowania.

Systemy operacyjne: poznaj i zaprojektuj!

ISBN 978-83-283-3759-6
9 788328 337596
Cena: 129,00 zł