Systemy operacyjne

Marek Grochowski

grochu@is.umk.pl
https://www.fizyka.umk.pl/~grochu/so/

Piotr Ablewski

piotra@is.umk.pl
https://piotra-kis.cloud.ablewski.pl/dydaktyka/SOPER

Wydział Fizyki, Astronomii i Informatyki Stosowanej UMK (2025/26)

Ostatnia aktualizacja: 14 grudnia 2025

Program wykładu

1. Wprowadzenie
   - Co to jest system komputerowy i system operacyjny?
   - Historia rozwoju systemów operacyjnych
2. Architektura współczesnych komputerów
   - Architektura i działanie procesora
   - Rodzaje i hierarchia pamięci
   - Magistrale i urządzenia peryferyjne
3. Jak system komputerowy wykonuje programy?
   - Monitor prosty, buforowanie
   - Wieloprogramowość i systemy z podziałem czasu
   - Systemy rozproszone, systemy czasu rzeczywistego

4. Przerwania
    - Systemy z obsługą przerwań
    - Dualny tryb pracy
    - Funkcje systemowe
5. Procesy
    - Model procesu i wątki
    - Stany procesów
6. Zarządzanie procesami
    - Algorytmy przydziału procesora
    - Komunikacja międzyprocesowa
7. Synchronizowanie procesów
    - Obszary krytyczne, wyścigi, zakleszczenia
    - Semafory i monitory

8. Zarządzanie pamięcią (P. Ablewski)
    - Wymiana pamięci i stronicowanie
    - Pamięć wirtualna
    - Algorytmy wymiany stron
9. Zarządzanie przestrzenią dyskową (P. Ablewski)
    - Fizyczna organizacja przestrzeni dyskowej
    - Systemy plików
    - Zarządzanie logicznymi wolumenami
10. Struktura systemów operacyjnych (P. Ablewski)
    - Rodzaje systemów i przykłady (Windows/Linux/Unix/MacOS)
    - Wirtualizacja, parawirtualizacja, kontenery
    - Współdzielenie systemów operacyjnych
    - Ładowanie systemu operacyjnego

Literatura

[1] A. S. Tanenbaum and H. Bos. *Systemy operacyjne*. Wydawnictwo Helion, 2015.

[2] William Stallings. *Systemy operacyjne. Architektura, funkcjonowanie i projektowanie*. Wydawnictwo Helion, 2018.

[3] A. Silberschatz, J. L. Peterson, and P. B. Galvin. *Podstawy systemów operacyjnych. Tom 1-2.* Wydawnictwo Naukowo-Techniczne, Warszawa, 2021.

[4] A. S. Tanenbaum and H. Bos. *Modern Operating Systems*. Pearson Education, 2022.

[5] K. Stencel. *Systemy operacyjne*. Wydawnictwo PJWSTK, Warszawa, 2004.

[6] A. M. Lister i R. D. Eager. *Wprowadzenie do systemów operacyjnych*. Wydawnictwo Naukowo-Techniczne, Warszawa, 1994.

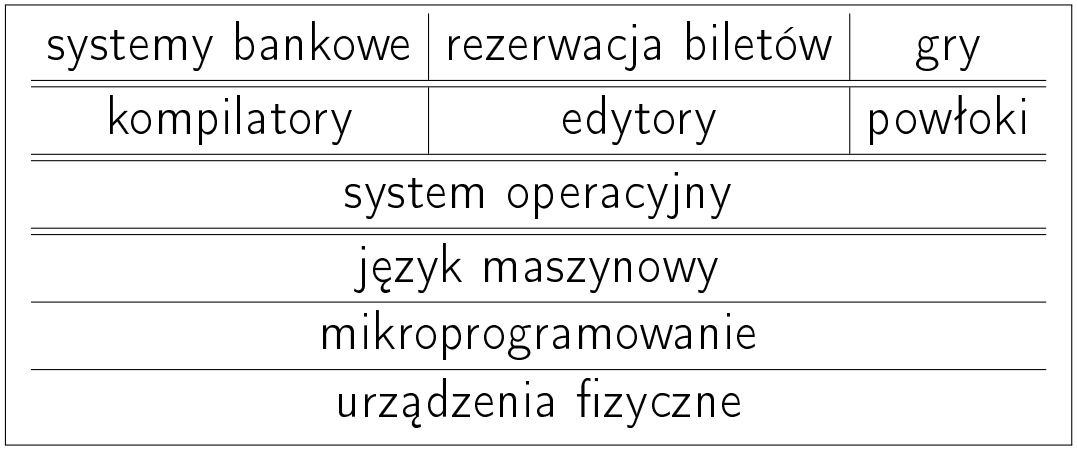
[7] L. Null i J. Lobur. *Struktura organizacyjna i architektura systemów komputerowych*. Wydawnictwo Helion, Gliwice, 2004.

[8] R. Love. *Linux kernel. Przewodnik programisty*. Wydawnictwo Helion, Gliwice, 2004.

Elementy systemu komputerowego

- **użytkownicy** (ludzie, maszyny, inne komputery), korzystają z systemu komputerowego
- **programy użytkowe** (kompilatory, edytory, systemy baz danych, gry) - wykonują zadania dla użytkowników
- **system operacyjny** - zarządza zasobami systemu komputerowego i udostępnia je programom użytkowym
- **sprzęt** - fizyczne komponenty komputera (procesor, pamięć, urządzenia I/O)

System komputerowy wg Tanenbauma



System komputerowy

Poziomy abstrakcji współczesnych systemów komputerowych
(wg Null i Lobur)

6	użytkownik	programy wykonywalne
5	języki wysokiego poziomy	C, C++, Java, FORTRAN, ...
4	asembler	kod asemblera
3	oprogramowanie systemowe	system operacyjny, biblioteki
2	maszyna	architektura zbioru rozkazów
1	sterowanie	mikrokod lub skonfigurowane sprzętowo
0	logika cyfrowa	obwody, bramki, itd.

Co to jest system operacyjny?

- System operacyjny jest **programem**; działa jako pośrednik między użytkownikiem komputera a sprzętem komputerowym; tworzy środowisko, w którym użytkownik może wykonywać programy.
- System operacyjny **nadzoruje i koordynuje** posługiwanie się sprzętem przez programy użytkowe, które pracują na zlecenie użytkowników.
- System operacyjny jest odpowiedzialny za
  - zarządzanie zasobami komputera
  - tworzenie wirtualnej maszyny (warstwy abstrakcji) dla programów użytkowych i programistów

Jądro systemu operacyjnego

- **Jądro (*kernel*)** – podstawowa część systemu operacyjnego, która działa nieustannie;
- Jądro działa w trybie uprzywilejowanym (zwanym także **trybem jądra**), który pozwala mu na wykonywanie wszystkich instrukcji procesora i pełny dostęp do sprzętu
- wszystkie pozostałe programy są programami użytkowymi, działają w **trybie użytkownika** i mają ograniczony dostęp do sprzętu

Cechy dobrego systemu operacyjnego

- **funkcjonalność** – system powinien zapewniać szeroki zakres usług dla użytkowników i programów
- **wydajność** – efektywne wykorzystanie zasobów, procesor powinien pracować dla użytkownika, minimalizacja opóźnień
- **skalowalność** – możliwość zwiększenia obciążenia i rozbudowy sprzętu bez utraty jakości działania
- **niezawodność** – wysoka dostępność, odporność na awarie, tzw. *five nines* (99.999% dostępności)
- **bezpieczeństwo** – ochrona danych i zasobów przed nieautoryzowanym dostępem, zapewnienie prywatności i integralności
- **łatwość korzystania i zarządzania** – intuicyjny interfejs, proste zarządzanie systemem i jego konfiguracją

Generacje komputerów

generacja	lata	technologia	szybkość (operacji/sek)
1	1946-1957	lampa próżniowa	40 000
2	1958-1964	tranzystor	200 000
3	1965-1971	mała i średnia skala scalenia	1 000 000
4	1972-1977	duża skala scalenia	10 000 000
5	1978-	bardzo duża skala scalenia	$>$ 100 000 000

SSI (*Small Scale Integration*) mała skala integracji, 10-100 elem./układ

MSI (*Medium Scale Integration*) średnia skala integracji, 100-1000 elem./układ

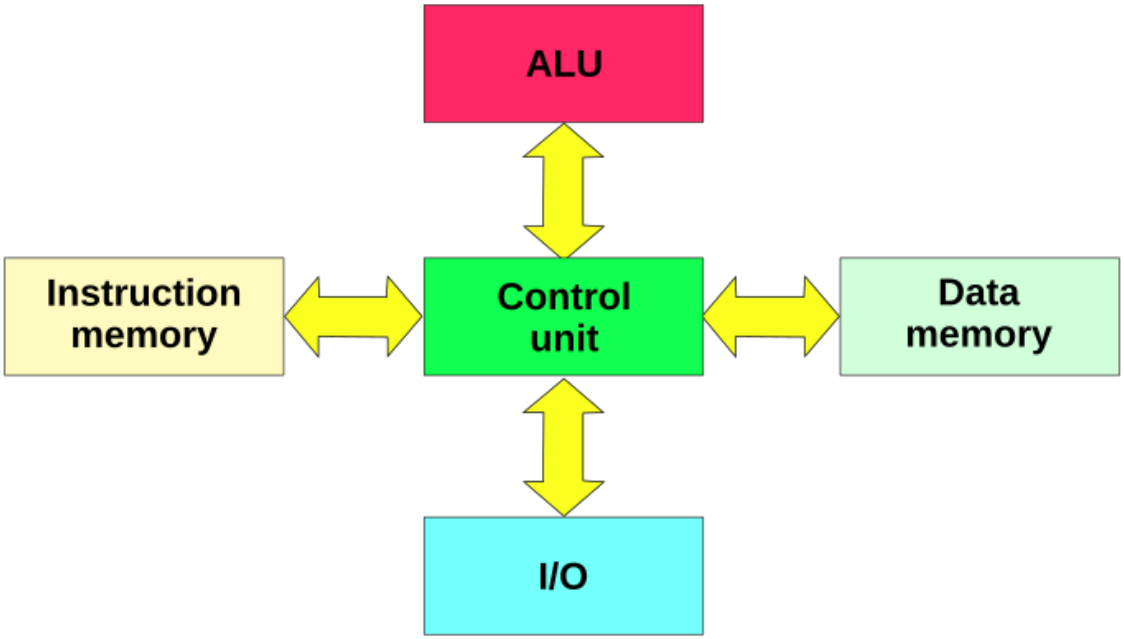
LSI (*Large Scale Integration*) duża skala integracji, 1000-10000 elem./układ

VLSI (*Vary Large Scale Integration*) bardzo duża skala integracji, powyżej 10000 elem./układ

Historia rozwoju komputerów i systemów operacyjnych

- 1642-1945 (0. generacja): mechaniczne maszyny liczące
  - maszyna analityczna Babbage'a (1835; Ada Byron)
  - ASCC (*Automatic Sequence Controlled Calculator*), Harvard Mark I, Howard Aiken, IBM (1944) - architektura harwardzka
- 1945-1953 (1. generacja): komputery na lampach próżniowych
  - brak systemów operacyjnych i języków programowania
  - bezpośrednie programowanie w języku maszynowym
  - przeprowadzanie wyłącznie obliczeń numerycznych
  - zastosowanie kart dziurkowanych do wprowadzania danych (początek lat 1950)

Architektura Harwardzka



Osobna pamięć i magistrale dla danych i instrukcji

https://en.wikipedia.org/wiki/Harvard_architecture

Architektura Harwardzka

Zalety:

- duża przepustowość: równoległy dostęp do danych i instrukcji
- bezpieczeństwo: pamięć instrukcji tylko do odczytu, brak możliwości modyfikacji kodu
- optymaliacja, np. szybsza pamięć dla instrukcji, większa dla danych

Wady:

- większa złożoność i koszt implementacji (dwa kontrolery pamięci, dwie magistrale)
- mniejsza elastyczność w programowaniu (trudności w modyfikacji pamięci instrukcji) i wykorzystaniu pamięci

Przykłady komputerów 1. generacji

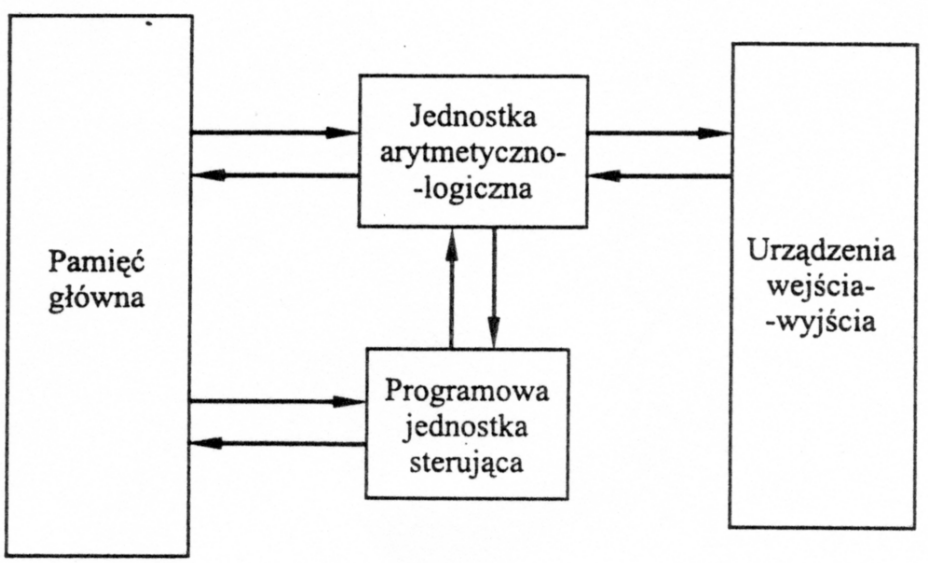
- **EDVAC** *Electronic Discrete Variable Computer* (1946, IAS[1])
  architektura von Neumanna
  pamięć główna przechowująca dane i rozkazy
  każdy element pamięci ma unikalny identyfikator - adres
  1000 słów 40 bitowych;
  2 instrukcje (20-to bitowe) w słowie

Pierwsze komputery komercyjne:

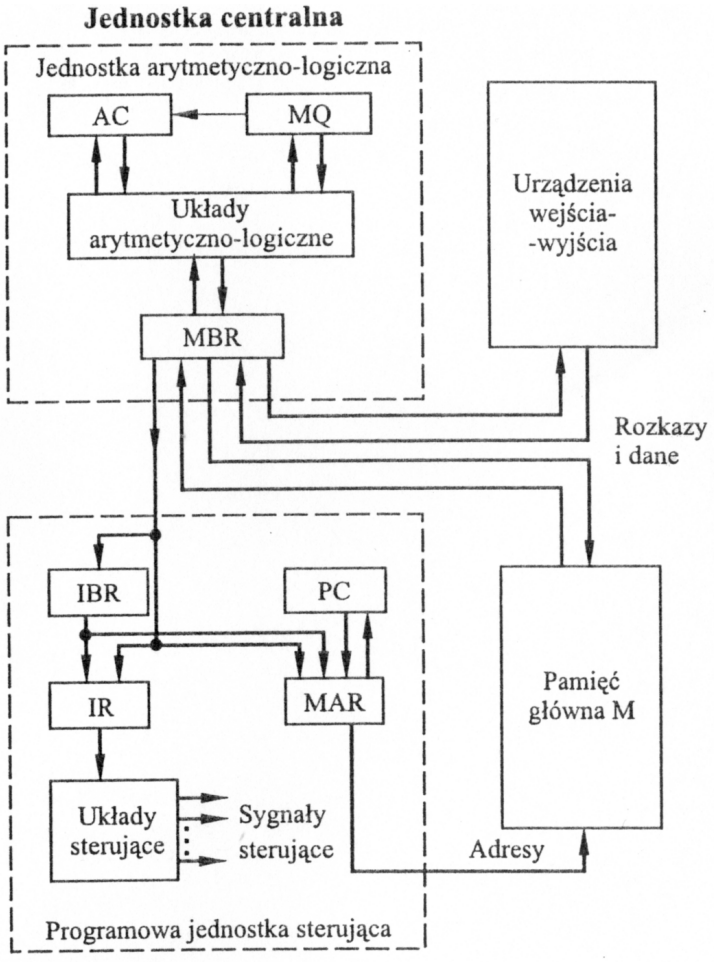
- **IBM 701** (1952) z przechowywanym programem do zastosowań naukowych, model 702 (1955) do zastosowań biznesowych

[1] Institute for Advanced Study, Princeton

Maszyna von Neumanna



- jednostka arytmetyczno-logiczna (ALU, *Arithmetic-Logic Unit*) wykonująca działania na danych binarnych
- jednostka sterująca interpretuje rozkazy z pamięci i powoduje ich wykonanie
- urządzenia wejścia-wyjścia, których pracą kieruje jednostka centralna



RYSUNEK 2.3. Struktura komputera IAS

Jednostka sterująca uruchamia IAS, pobierając rozkaz z pamięci i wykonując go (jeden rozkaz w określonym momencie)

- **rejestr buforowy pamięci** (MBR, *memory buffer register*) przechowywanie słowa, ktore ma być pobrane lub odesłane do pamięci
- **rejestr adresowy pamięci** (MAR,*memory address register*) – adres w pamięci słowa, które ma być zapisane/odczytane z MBR
- **rejestr rozkazów** (IR, *instruction register*) – 8-bitowy kod operacyjny rozkazu, który jest wykonywany
- **buforowy rejestr rozkazów** (IBR, *instruction buffer register*) czasowo przechowuje podręczny rozkaz ze słowa w pamięci
- **licznik programu** (PC, *program counter*) – adres kolejnej pary rozkazów
- **akumulator** (AC, *accumulator*) oraz **rejestr mnożenia i dzielenia** (MQ, *multiplier-quotient*) – czasowe przechowywanie argumentów i wyników operacji prowadzonych przez ALU.

Cykle komputera IAS:

- **cykl rozkazu:** kod operacji następnego rozkazu jest ładowany do IBR, a część adresowa – do MAR
- **cykl wykonania:** układy sterujące rozpoznają kod operacji i wykonują rozkaz, wysyłając odpowiednie sygnały sterujące, które powodują, że przenoszone są dane lub ALU wykonuje operację

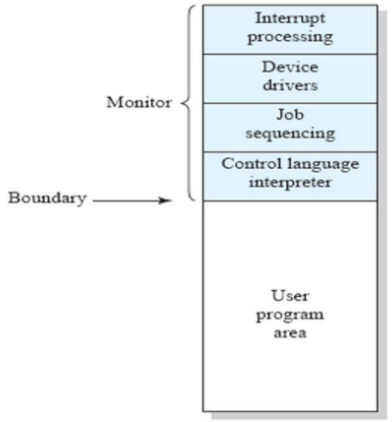
  Komputer IAS ma 21 rozkazów:

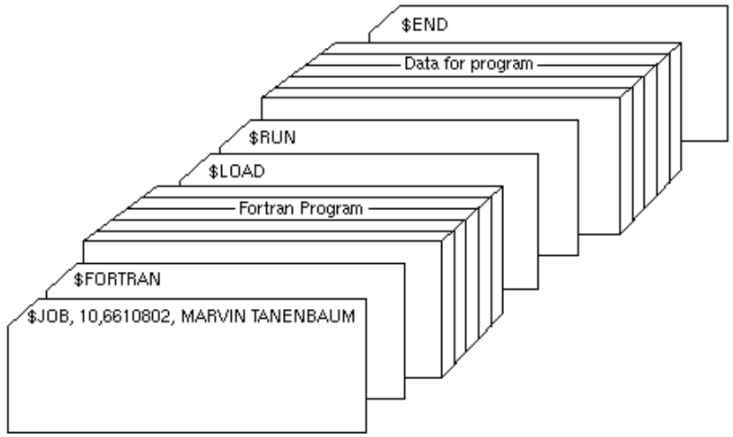
  – przenoszenie danych
  – rozgałęzienie bezwarunkowe
  – rozgałęzienie warunkowe
  – arytmetyka
  – modyfikowanie adresu

2. generacja: tranzystory i systemy wsadowe (1955-1965)

- zwiększona niezawodność
- bardziej złożone jednostki ALU oraz sterujące
- rozgraniczenie roli operatora i programisty
- rozwój języków programowania wysokiego poziomu (np. Fortran)
- system wsadowy i praca pośrednia (wczytywanie i drukowanie *off-line*)
- pojawia się oprogramowanie systemowe: proste **monitory** (np. FSM, *Fortran Monitor System*, IBSYS) i **karty sterowania zadaniami** (poprzednik powłoki i obecnych CLI)
- **monitor** wraz z językiem kontroli zadań (JCL, *Job Control Language*) umożliwiał **przetwarzanie wsadowe** (*batch processing*), czyli automatyczne wykonywanie zadań bez interwencji użytkownika

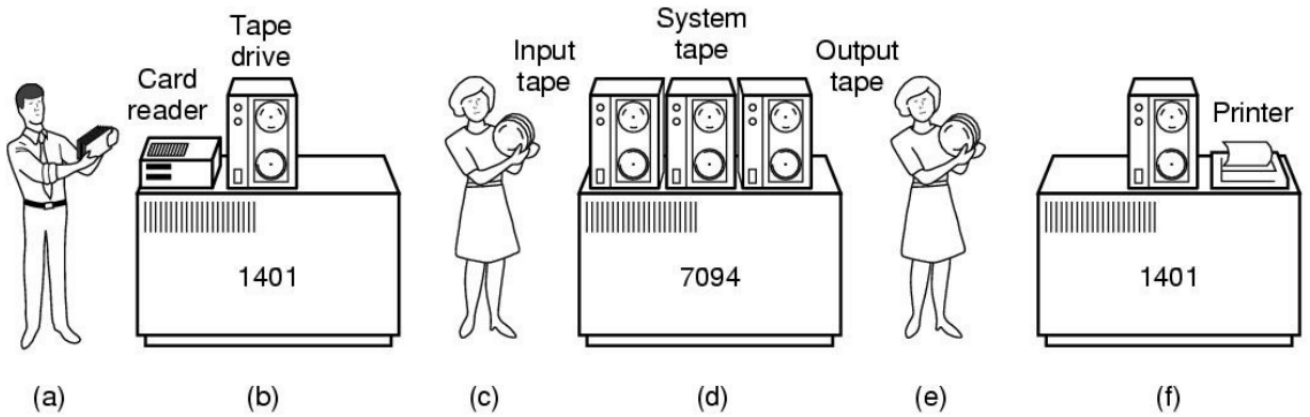
Monitor rezydujacy w pamięci





- specjalne karty sterujące, instrukcje monitora
- monitor i program rezydują jednocześnie w wydzielonych obszarach pamięci
- użytkownik nie ma bezpośredniego dostępu do procesora
- czytniki kart i drukarki powolne względem jednostki obliczeniowej

Wczesne systemy przetwarzania wsadowego



- praca w trybie pośrednim (off-line) - przyspieszenie obliczeń poprzez ładowanie zadań do pamięci z taśm oraz odczyt kart i drukowanie wierszy przy użyciu mniejszych maszyn
- nie trzeba wprowadzać żadnych zmian w programach aplikacji, aby przejść z trybu bezpośredniego na tryb off-line
- możliwość korzystania z wielu systemów wejściowych i wyjściowych

3. generacja: wieloprogramowość (1965-1980)

- rodzina komputerów o ujednoliconej architekturze do ogólnych zastosowań, IBM 360 (1964), ujednolicony system operacyjny (OS/360), przenośny (w teorii), przetwarzanie wsadowe
- **wieloprogramowość** - możliwość przechowywania w pamięci kilku programów jednocześnie i przełączania się między nimi
- **spooling** - kolejkowanie zadań wejścia/wyjścia, wyeliminowało małe komputery do obsługi kart
- pojawiaja się **systemy z podziałem czasu**:
  - interaktywny dostęp wielu użytkowników, współbieżne działanie wielu programów, ochrona pamięci, systemy plików,
  - *Compatible Time-Sharing System* (CTSS)
  - *Multiplexed Information and Computing Service* (MULTICS)
- minikomputery, DEC PDP 8 (1964)

4. generacja (1980-): komputery osobiste i sieci

- układy o dużym stopniu scalenia (VLSI)
- powstanie stacji roboczych i komputerów osobistych (architektura minikomputerów)
- sieciowe systemy operacyjne
- rozproszone systemy operacyjne (układy masywnie równoległe)
- wirtualizacja

5. generacja: systemy mobilne (1990-)

- lata 90-te: połączenie telefonu z PDA (*Personal Digital Assistant*) Nokia N9000 - potem nazwane smartfonem
- systemy operacyjne urządzeń mobilnych: Symbian OS, Windows Mobile, Palm OS, iOS, Android
- obecnie: tablety, smartfony, smartwatche, smartTV, IoT
- systemy wbudowane: RTOS, TinyOS, Contiki

Początki komputeryzacji w Polsce

- Pierwsze polskie komputery (lata 50. i 60. XX wieku)
  - Analogowy Analizator Równań Algebraicznych Liniowych (ARAL, 1954)
  - Analogowy Analizator Równań Różniczkowych (ARR), Elektroniczna Maszyna Automatycznie Licząca (EMAL)
  - XYZ (1958) wzorowany na IBM 701, pamięć naddźwiękową, 800 operacji/sek.
- System Automatycznego Kodowania (SAKO, 1960), „Polski FORTRAN"
- Odra 1002 (1962) – pierwszy komputer z zakładów Elwro, Odra 1003 (1964) – pierwszy seryjnie produkowany komputer (ostatni wyłączono 30/04/2010 po 34 latach pracy)
- 16-bitowy minikomputer K-202 J.Karpińskiego (1970-1973), pierwszy polski komputer zbudowany z użyciem układów scalonych, przewyższał pod względem szybkości pierwsze IBM PC oraz umożliwiał wielozadaniowość, wielodostępność i wieloprocesorowość.
- minikomputer Mera (1978)[2]

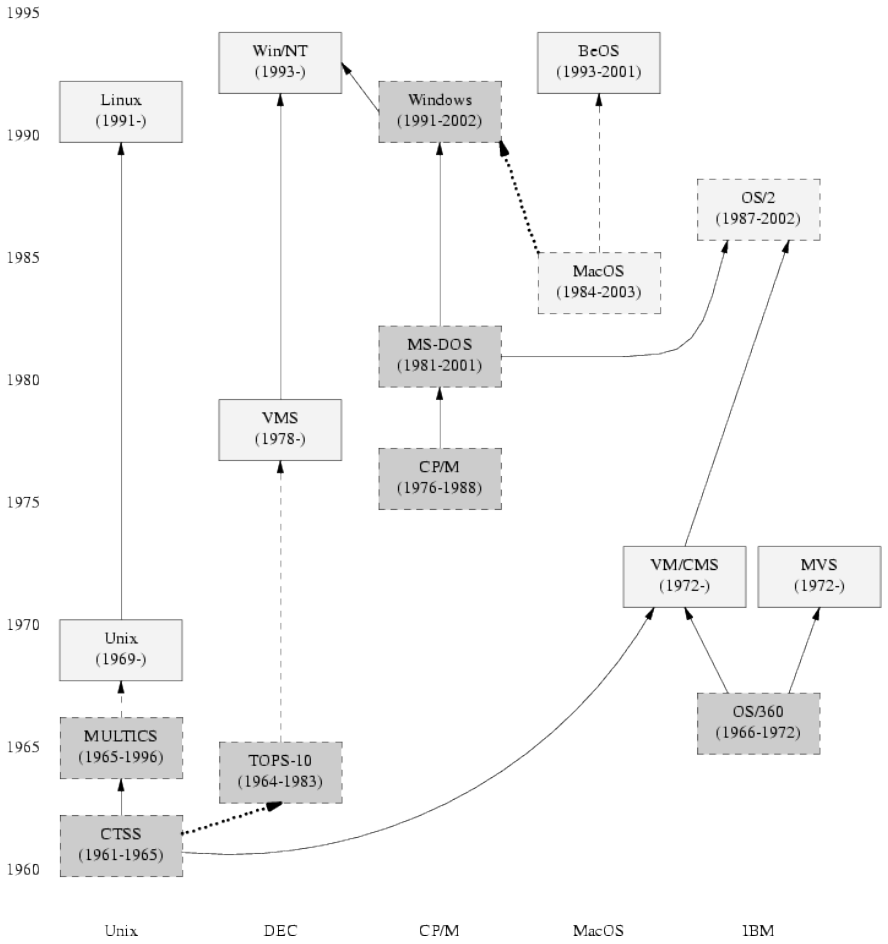
[2]Zob. System Mera 400

Systemy operacyjne z początków komputeryzacji w Polsce

- SOK-1 - tylko jeden program (komputer K-202)
- SODA System Operacyjny Dwu Aktywny (Odra) - dwa programy
- Egzekutor RTX (*Real Time Executive*), Mera 300
- Ładowacz (Mera 305)
- MSO 300 (Mera 306), system ogólnego przeznaczenia
- SOWA System Operacyjny Wielo Aktywny - potem nazwany CROOK
- CROOK 1-5, komputery K-202 i Mera 400, uniwersalny, wielozadaniowy, wielodostępowy system operacyjny z hierarchicznym systemem plików[3]

[3] Zob. https://mera400.pl/CROOK

Historia systemów operacyjnych wg E.S.Raymonda[4]

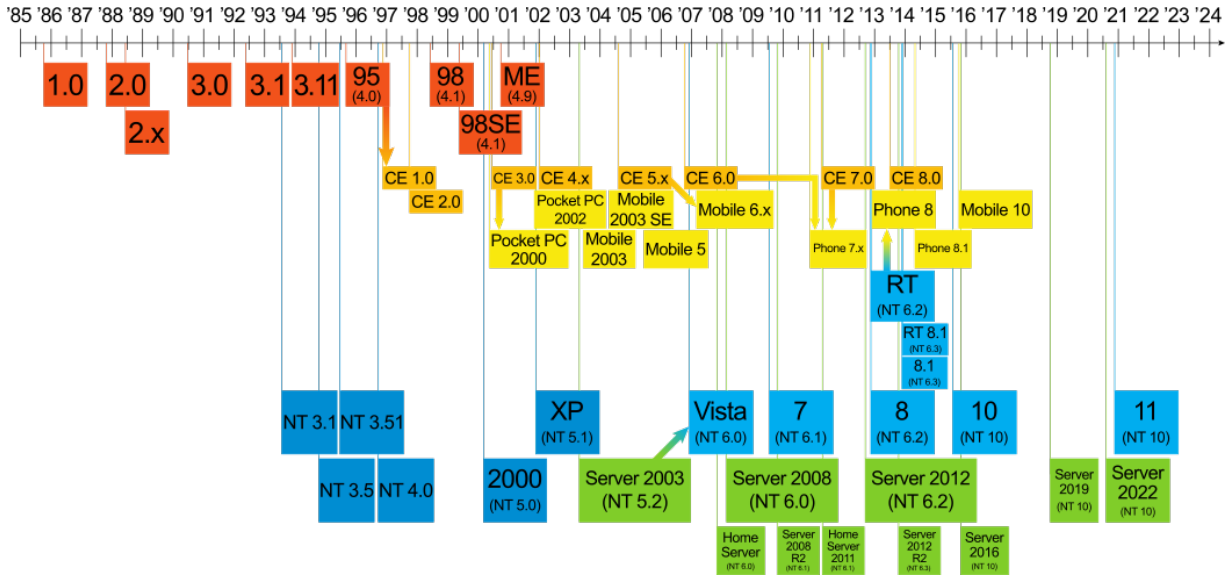


[4] *UNIX Sztuka programowania*, Wydawnictwo Helion, Gliwice, 2004. Także http://en.wikipedia.org/wiki/Timeline_of_operating_systems

MS-DOS, Windows, Mac OS

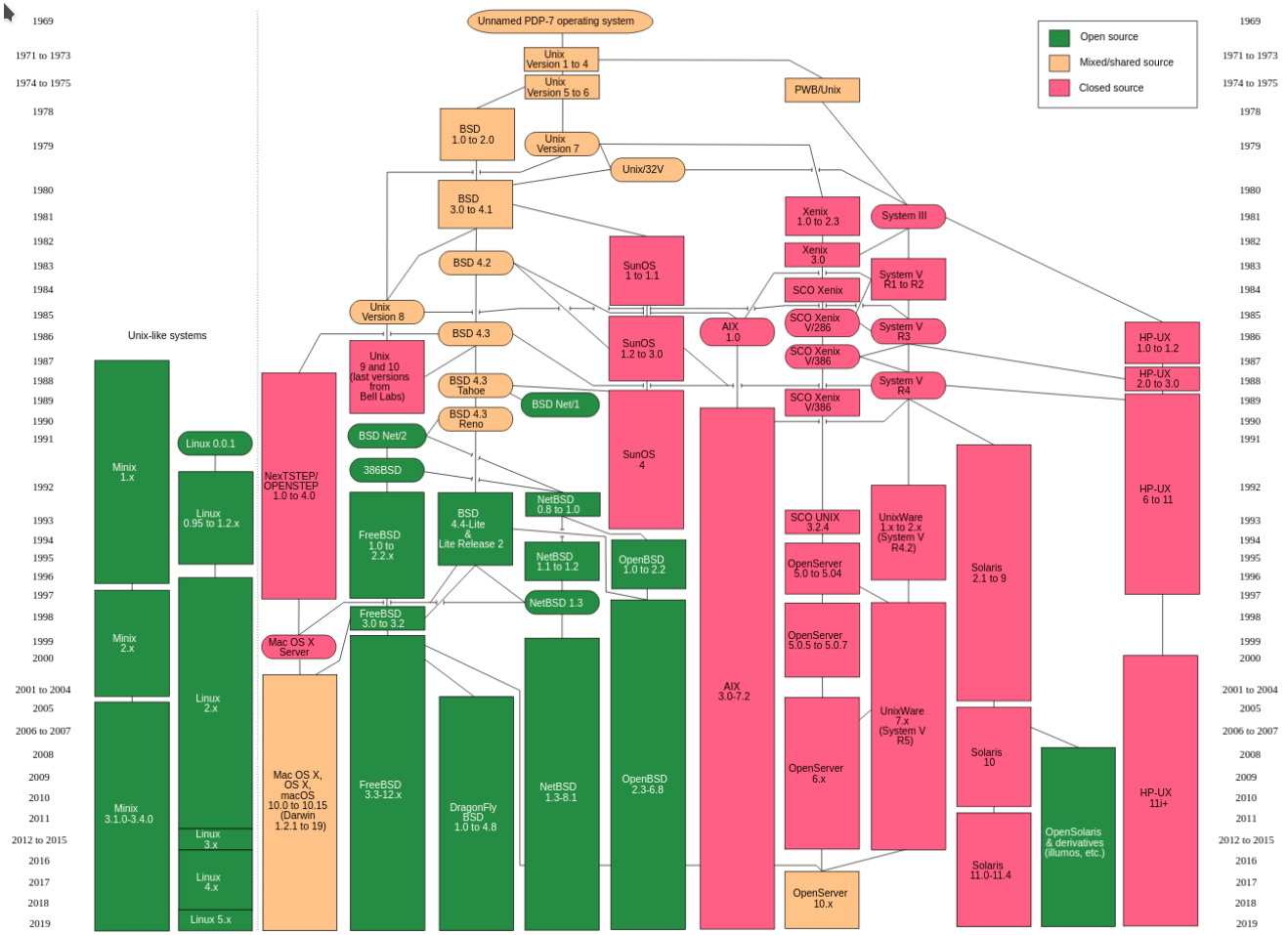
- 1974 pierwszy 8-mio bitowy procesor 8080 ogólnego przeznaczenia,
- dyskowy system CP/M (*Control Program for Microcomputer*), Gary Kildall, Digital Research, system zdominował rynek mikrokomputerów (m.in. Atari, ZX Spectrum) w latach 70-tych
- 1981 IBM PC, przez brak porozumienia z Digital Research, Bill Gates odkupuje DOS od lokalnej firmy i wprowadza jako MS-DOS - system dołączany do sprzętu
- 1983 IBM PC/AT - silna pozycja MS-DOS
- 1984 Apple Macintosh (Steve Jobs), pierwszy system z GUI i myszką, Mac OS X (obecnie macOS) adoptuje elementy jądra UNIX (BSD)
- 1985 Windows 1.0 - początkowo graficzna nakładka na MS-DOS
- 1995 Windows 95 pierwszy samodzielny system (MS-DOS do rozruchu i zgodności oprogramowania)

- Windows NT (*New Technology*) napisany na nowo, 32 bitowy, zgodny z W95/W98
- od 2000 dwie linie:[5]
  - serwerowa: Windows Server 2003, Windows Server 2008, ...
  - kliencka: oparte o Windows XP, Windows 7, 8, 10, ...



[5] https://pl.wikipedia.org/wiki/Microsoft_Windows

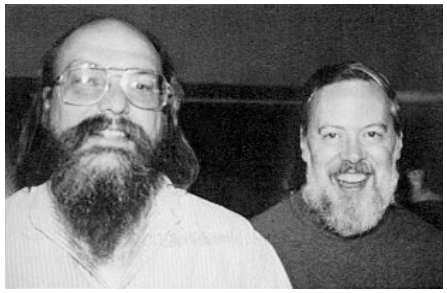
Historia systemu operacyjnego Unix



Wikipedia: History of Unix

Historia Unix

- 1969 port gry Space Travel na PDP-7 (Ken Thompson), powstaje system bazujący na MULTICS dla jednego użytkownika
- 1969 Unix, Ken Thompson, Denis Ritchie (AT&T Bell Labs)
- 1971 język C (D. Ritchie), kompilator
- 1974 udostępnienie źródeł Unix-a (przepisanych z użyciem C)
- 1977 Unix BSD rozwijany na Uniwersytecie Kalifornijskim w Berkeley (K.Thompson i B.Joy) - wiele ważnych zmian (pamięć wirt, TCP/IP)





Historia Unix

- 1983 firma AT&T komercjalizuje Unix System V
- brak standaryzacji różnych wersji Unix-a
  - IBM – AIX (SVR3)
  - DEC – Ultrix (4.2 BSD)
  - SUN – Solaris (SVR4)
  - HP – HP-UX (SVR4)
  - SGI – IRIX (BSD)
  - Microsoft i Santa Cruz Operations – Xenix (Unix Version 7)
- 1988 POSIX, Portable Operating System Interface,
  standard definiujący interfejsy systemowe i programowania aplikacji (API) na poziomie użytkownika, w celu zapewnienia zgodności oprogramowania (przenośności) z wariantami Uniksa i innych systemów operacyjnych

Historia GNU Linux

- 1984 GNU (*Gnu's Not Unix*), wolny (o swobodnym dostepie), uniksopodobny system operacyjny[6]
- 1985 GNU Manifesto, R. Stallman, powstaje Free Software Foundation
- 1989 oprogramowanie GNU udostępnione na licencji GPL (copyleft)
- 1991 jądro Linux udostepnione na licencji GNU, Linus Torvalds
  jądro wykorzystane w GNU/Linux
- 2000 powstaje The Linux Foundation





[6] www.gnu.org

comp.os.minix: wpis LBT z 25.08.1991

```
Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and
professional like gnu) for 386(486) AT clones. This has been
brewing since april, and is starting to get ready. I'd like any
feedback on things people like/dislike in minix, as my OS resembles
it somewhat (same physical layout of the file-system (due to
practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to
work. This implies that I'll get something practical within a few
months, and I'd like to know what features most people would want.
Any suggestions are welcome, but I won't promise I'll implement
them :-)

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded
fs. It is NOT portable (uses 386 task switching etc), and it
probably never will support anything other than AT hard disks, as
that's all I have :-(.
```

GNU/Linux

- GNU – Gnu's Not Unix, wolny, kompatybilny z Unix system operacyjny
- Linux jako jadro (GNU/Linux), ale jądro Hurd stale rozwijane
- asembler, kompilator GCC, glibc, program łaczacy, biblioteka GNU C, bash, GNU coreutils
- programy z projektów GNU https://www.gnu.org/software/
- GNU General Public License: http://www.gnu.org/licenses/gpl.html
  Powszechna Licencja Publiczna GNU:
  http://www.gnu.org/licenses/gpl.html
- Inne licencje: GNU Lesser GPL, GNU Library GPL, Modified BSD License, Perl Artistic License, Apache License, LaTeX Project Public License, Python Software Foundation License, PHP License, OpenSSL License, Sleepycat License, Common UNIX Printing System License Agreement, IBM Public License, . . .
- dystrybucje Linux: Slackware (1993), Debian (1993), Ubuntu (2004), RedHat, ...

Rozwój GNU/Linux

Data	Wersja	Linie kodu	
1991/09	0.01	10 K (1K assemblera)	88 plików
1991/11	0.11		
1992/02	0.12		GNU GPL
1992/03	0.95		X Window
1994/04	1.0.0	176 K	
1995/03	1.2.0	311 K	arch. Alpha, SPARC i MIPS, szyna PCI
1996/06	2.0.0	470 K	procesory SMP, SMB
1999/01	2.2.0	1.8 M	
1999/12	2.2.13		
2001/01	2.4.0	3.4 M	ISA Plug-And-Play, USB, PCMCIA, arch. 64 bit, LVM
2003/12	2.6.0	5.9 M	NuMA, hiperwątkowość
2011/07	3.0	14.6 M	
2012/09	3.2	15.9 M	
2014/12	3.18	19.0 M	
2015/07	4.0	22.2 M	
2017/08	4.13	24.8 M	
2020/01	4.19	27.8 M	
2020/08	5.8	28 442 673	70K plików
2024/10	6.11.3		aktualna stabilna wersja

https://kernel.org/

- 1998 powstaje Open Source Initiative (Eric. S. Raymond), „open source" w kontrze do „free software" – Oracle, Informix, Sysbase, IBM, Dell, HP i in. ogłaszają wsparcie dla ruchu open source
- 2000 SUSE Linux Enterprise Server (IBM S/390, x86), jedna z pierwszych komercyjnych dystrybucji Linuksa dla serwerów klasy enterprise (IBM S/390)
- 2001 IBM – 1 mld USD i 1500 programistów wspiera rozwój Linuksa
  Microsoft CEO S. Ballmer: *„Linux is a cancer [...] intellectual property destroyer"*
- 2002 Red Hat Linux Advanced Server – system klasy enterprise wspierany przez firmy Dell, IBM, HP, Oracle
- 2007 projekt Samba otrzymuje (po procesie) od Microsoftu dokumentację SMB
- 2009 Microsoft dokłada ponad 20 tys. linii kodu do jadra Linuksa (Hyper-V)
- 2014 Satya Nadella (CEO firmy Microsoft): *Microsoft loves Linux*
- 2016 Windows Subsystem for Linux,
  Microsoft przyłacza sie do Linux Foundation jako platynowy sponsor

Rozwój linux (raport 2020)

- jeden z największych projektów programistycznych rozwijanych przez społeczność
- regularne wydania co 8-10 tygodni, każde z istotnymi zmianami i usprawnieniami
- tempo zmian jądra jest wysokie i rośnie, około 12 K łatek wchodzi obecnie do wydania (w 2020 r. 10.7 commit/h))
- 1600 programistów z ponad 200 korporacji
- ok 12% zmian od miłośników Linuksa
- ok 52% od programistów różnych firm, m.in. Intel, Red Hat, IBM, SUSE, Linaro, Google, Samsung, AMD, Renesas, Texas Instruments, and Oracle
- łącznie od 2005 ponad 15,600 indywidualnych programistow z 1400 różnych firm wniosło wkład w rozwój

Współczesny rynek systemów operacyjnych

Komputery osobiste (PC, laptopy)

- Windows 73%
- OS X 15%
- Linux 4%
- Chrome OS 2%

Superkomputery

- Linux (TOP 500) 100%

Serwery (2021)

- Linux/Unix 77%
- Windows Server 22.7%

Tablety

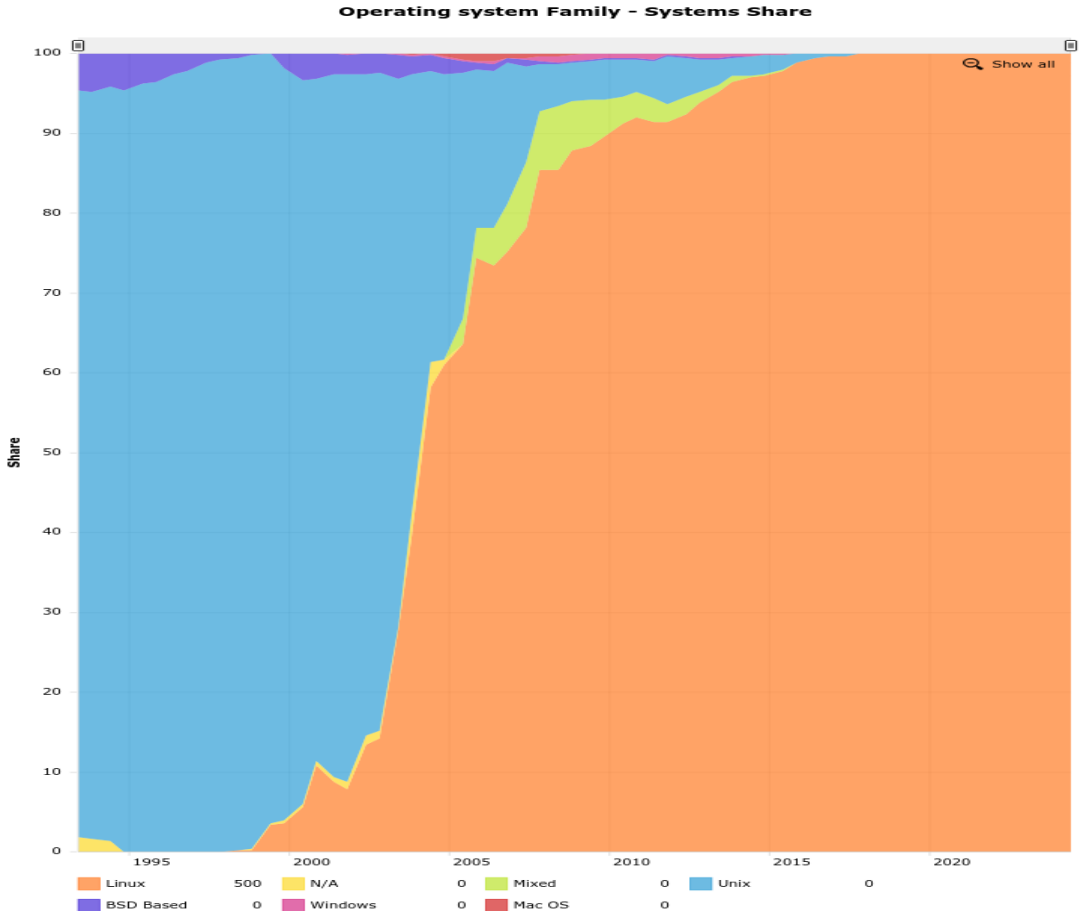
- iOS 55%
- Android 44%

Mobilne

- Android 72%
- iOS 27%

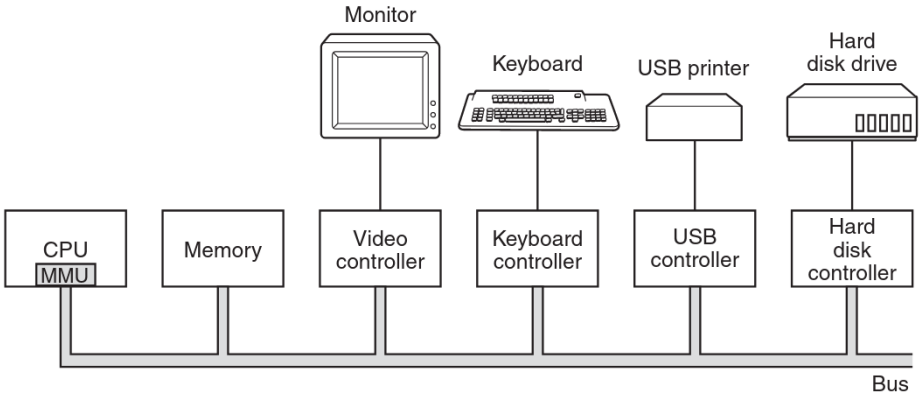
https://gs.statcounter.com
Wikipedia Usage share of operating systems

TOP 500: udział systemów operacyjnych (6-2024)



https://www.top500.org/statistics/overtime/

Architektura komputerów



- procesor
- pamięć
- sterowniki urządzań (kontrolery) wejścia-wyjścia, do których podłączone są urządzenia peryferyjne
- szyna (magistrala) systemowa łącząca te elementy

Procesor

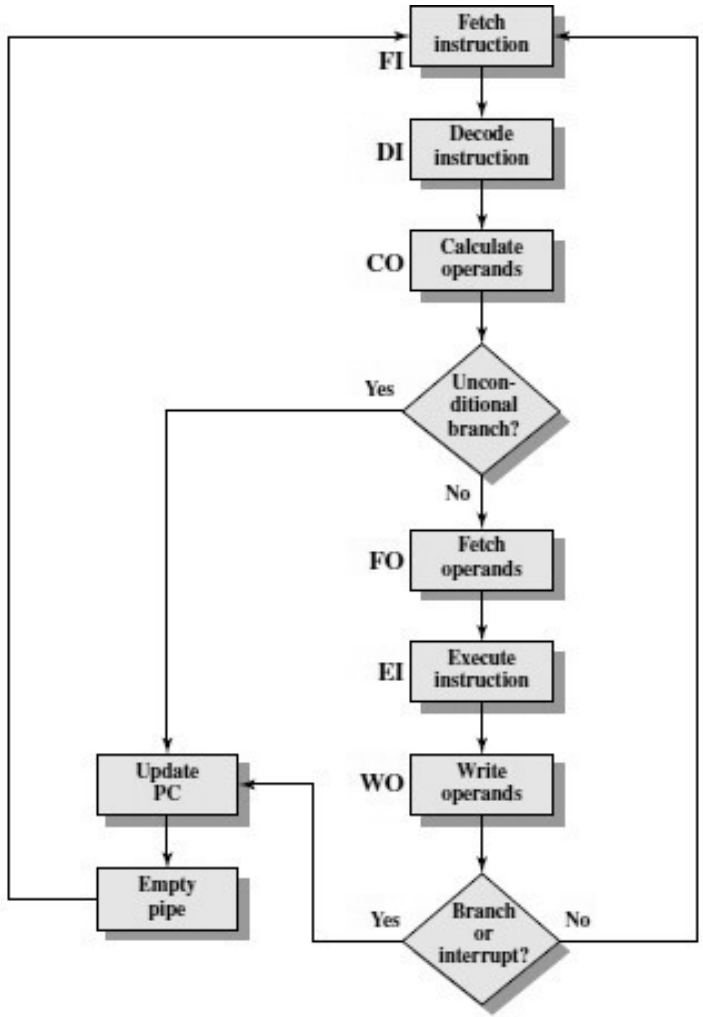
- rejestry
  - rejestry ogólnego zastosowania (x86: EAX, EBX, ECX, EDX)
  - licznik programu (*program counter*) - zawiera adres następnej instrukcji (x86: EIP)
  - wskaźnik stosu (*stack pointer*) wskazuje najwyższy element stosu wywołań w pamięci (x86: ESP)
- słowo statusu programu (PSW *program status word*)
  - wyniki instrukcji warunkowych, flagi, bity kontrolne, stan systemu
  - tryb ochrony (tryb jądra, użytkownika)
- potok instrukcji - pobranie, zdekodowanie, wykonanie instrukcji

System operacyjny musi znać stan wszystkich rejestrów. Podczas przełączania zadań zapisuje stan rejestrów, żeby móc je później przywrócić przy wznowieniu programu.

Architektura współczesnego procesora

- **przetwarzanie potokowe** - podział rozkazu na wyspecjalizowane etapy
- **superskalarność** - zrównoleglenie wykonania rozkazów w obrębie jednego procesora poprzez zwielokrotnienie skalarnych jednostek wykonawczych
- analiza przepływu danych, przewidywanie rozgałęzienia, spekulatywne wykonywanie - badanie zależności między rozkazami i wykonywanie ich poza kolejnością, aby zmniejszyć opóźnienia
- **hiperwątkowość** (*hyper-threading*) - wiele wątków w jednym procesorze
- **instrukcje SIMD** (*Single Instruction Multiple Data*) - jedna instrukcja operuje na wielu danych jednocześnie (np. multimedia, obliczenia naukowe), MMX (*MultiMedia Extensions*), SSE (*Streaming SIMD Extensions*), AMD 3DNow (*3D NO Waiting*)
- **wielordzeniowość** - wiele jednostek wykonawczych (rdzeni) w jednym procesorze
- wsparcie sprzętowe **wirtualizacji**: Intel VT (vmx), AMD V (svm)
- **CISC** i **RICS** - złożone i zredukowane zestawy rozkazów

Przetwarzanie rozkazów



FI: pobierz następną instrukcję do bufora

DI: dekodowanie instrukcji, określ kod operacji i specyfikację argumentów

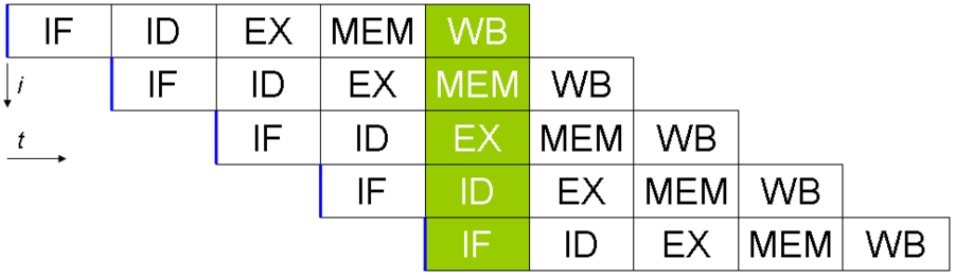
CO: oblicz efektywny adres danych źródłowych

FO: pobierz dane z pamięci jeżeli nie ma ich w rejestrach

EI: wykonaj instrukcję

WO: zapisz wynik w pamięci

Przetwarzanie potokowe



Klasyczny potok RISC:

IF=Instruction Fetch

ID=Instruction Decode

EX=Execute

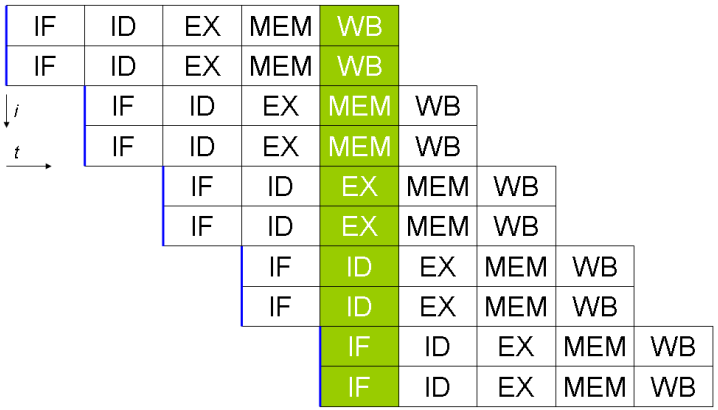
MEM=Memory access

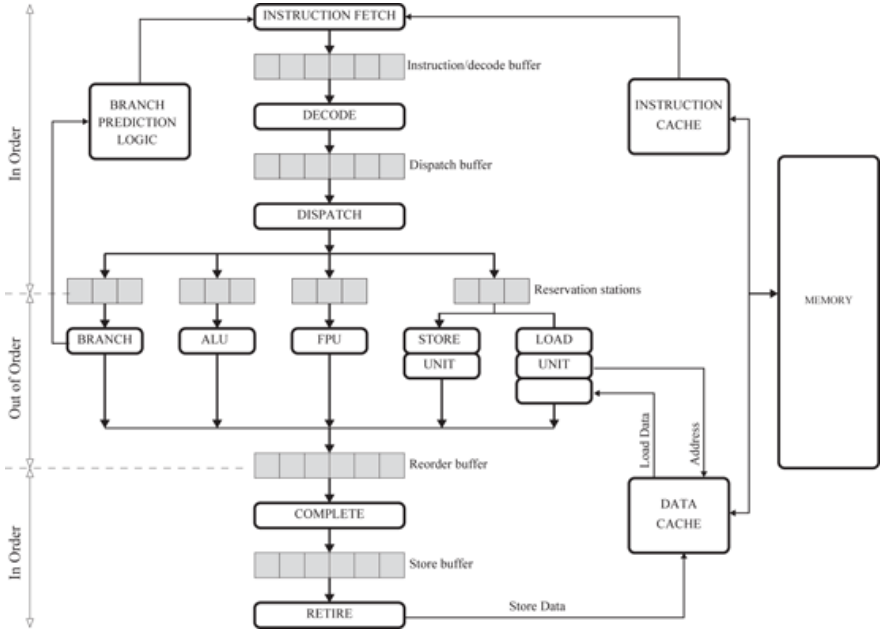
WB=Register write back

- podział rozkazu na wyspecjalizowane etapy
- dzięki podziałowi, różne instrukcje mogą być wykonywane równocześnie
- lepsza wydajność: zwiększenie liczby instrukcji wykonywanych w jednostce czasu
- problem z rozgałęzieniami (instrukcja skoku) i konflikty w dostepnie do zasobów

https://en.wikipedia.org/wiki/Instruction_pipelining

Superskalarność





- zrównoleglenie wykonania rozkazów w obrębie jednego procesora poprzez zwielokrotnienie skalarnych jednostek wykonawczych
- w jednym cyklu procesora wykonywane jest wiele instrukcji
- wykonanie poza kolejnością (*out-of-order*) instrukcji z bufora w celu zmaksymalizowania użycia CPU

http://en.wikipedia.org/wiki/Superscalar,
https://anubhavbhatla.github.io/projects/superscalar/

Ryzyka przetwarzania potokowego

Zatory w potoku poleceń pojawiają się, gdy instrukcje warunkowe nie pozwalają na sekwencyjne wykonanie instrukcji.

Możliwe ryzyka wykonania kodu poza kolejnością:

- **strukturalne** - gdy instrukcje w potoku nie moga być wykonane równocześnie
- **danych** - gdy instrukcje korzystają z tych samych danych i wynik zależy od kolejności operacji
- **sterowania** - wykonanie instrukcji po rozwidleniu ale przed wykonaniem instrukcji warunkowej. Instrukcje z błednej ścieżki muszą zostać odrzucone z potoku.

Ryzyka przetwarzania potokowego

Jak rodzić sobie z konfliktami:

- wykonywanie instrukcji obu ścieżek jednocześnie
- przewidywanie rozgałęzień (np. na podstawie statystyki poprzednich wykonań). Kod jest spekulatywnie wykonywany. W przypadku błednej predykcji efekt tego wykonania jest odrzucany
- *branch-free-code* technika programowania z minimalizowaniem rozgałęzień kodu
- bufor pętli - mała, bardzo szybka pamięć podręczna dla etapu pobierania (FI), jeśli wystąpi rozgałęzienie, sprzęt najpierw sprawdza, czy cel rozgałęzienia znajduje się w buforze pętli
- opóźnianie rozgałęzień - zmiana kolejności instrukcji tak aby instrukcje warunkowe można było wykonać jak najpóźniej

CISC (*Complex Instruction Set Computer*)

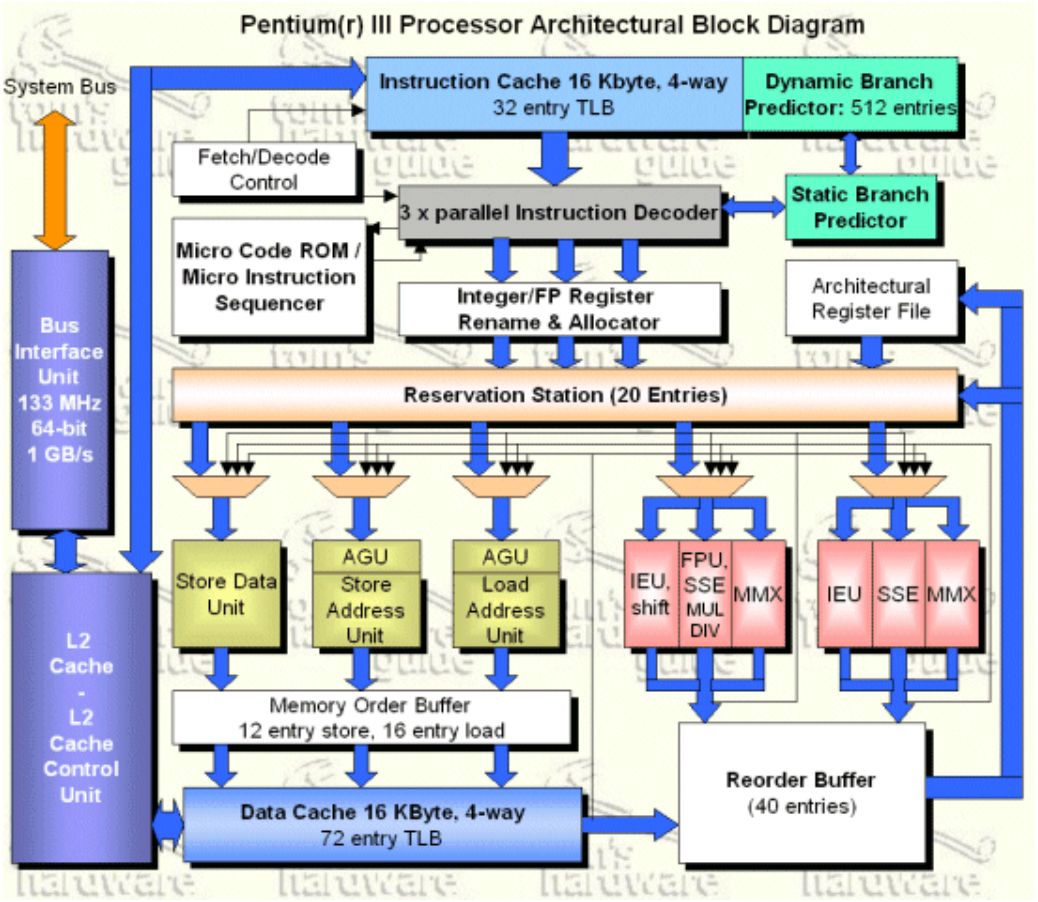
komputer o pełnej liście rozkazów

RISC (*Reduced Instruction Set Computer*)

komputer o zredukowanej liście rozkazów

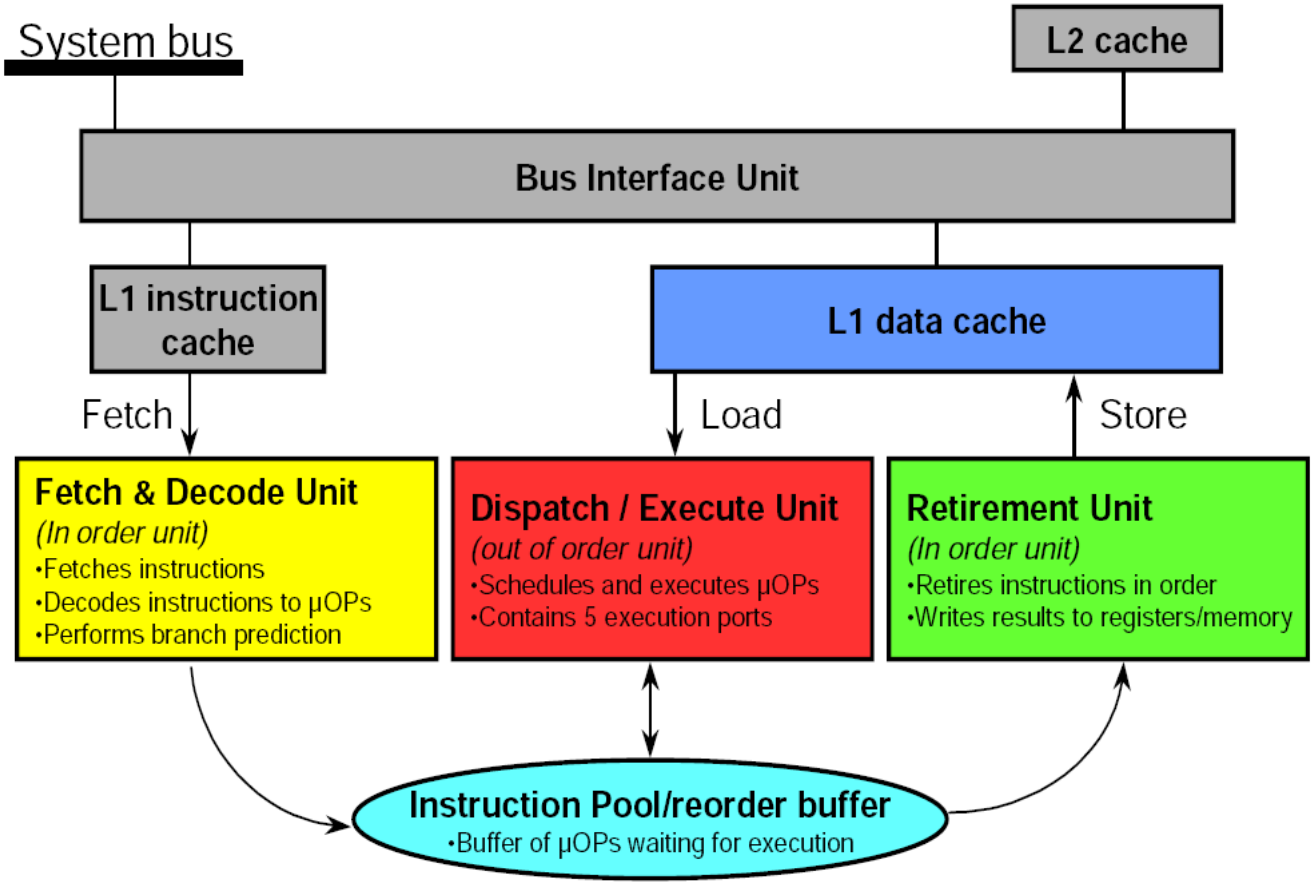
Cechy	RISC	CISC
Liczba instrukcji	Zredukowany zestaw (kilkadziesiąt)	Duży zestaw (100–200)
Złożoność instrukcji	Proste, jednolite	Złożone, wyspecjalizowane
Czas wykonania instrukcji	Zazwyczaj 1 cykl zegara	Wiele cykli zegara
Tryby adresowania	Ograniczone (2–4)	Duża liczba (5–20)
Długość instrukcji	Stała (np. 32 bity)	Zmienna
Odwołania do pamięci	Ograniczone (operacje rejestr-rejestr)	Częste (operacje pamięć-pamięć)
Liczba rejestrów	Duża liczba rejestrów ogólnego przeznaczenia	Mniejsza liczba rejestrów
Jednostka sterująca	Zaimplementowana sprzętowo	Mikroprogramowana (ROM)
Projektowanie potoku	Uproszczone, wydajne	Złożone, trudniejsze do optymalizacji
Przetwarzanie	Optymalizowane pod kątem potoku	Mikroprogramowane, mniej przewidywalne
Zastosowanie	Urządzenia mobilne, systemy oszczędne energetycznie, nowoczesne procesory	Starsze systemy, kompatybilność wsteczna
Przykłady	ARM, MIPS, RISC-V, Apple M1	x86 (Intel, AMD)

Występują też architektury mieszane - złożone instrukcje rozkładane są na prostsze (μOPS)



Zmodyfikowana architektura harwardzka architektura mieszana, łączy w cechy architektury harwardzkiej i architektury von Neumanna. Oddzielone obszary pamięci na dane i rozkazy.

Pentium II and Pentium III Processors Architecture



dynopt.dtc.umn.edu/documents/IA-OPTIMIZATION-Ref-Manual.pdf

Pamięć: rodzaje, własności

- szybki rozwój pojemności pamięci i szybkości procesorów
- wolniejszy przyrost szybkości przesyłania danych pomiędzy procesorem i pamięcią

Interfejs pomiędzy pamięcią główną a procesorem jest najbardziej krytycznym elementem całego komputera, ponieważ jest on odpowiedzialny za przepływ rozkazów i danych pomiędzy tymi układami.

Jeśli dostęp do pamięci jest niewystarczający, to cykle procesora są marnowane (**głodzenie procesora**).

Wymagania dotyczące pamięci: niska cena, duża pojemność, szybkość działania (porównywalna z szybkością CPU)

Hierarchia pamięci:

- rejestry (SRAM)
- pamięć podręczna, cache (SRAM)
- pamięć główna (MCDRAM, DRAM, NVRAM, NVDIMM)
- pamięć dyskowa (HDD, SDD)
- pamięć taśmowa, dyski optyczne

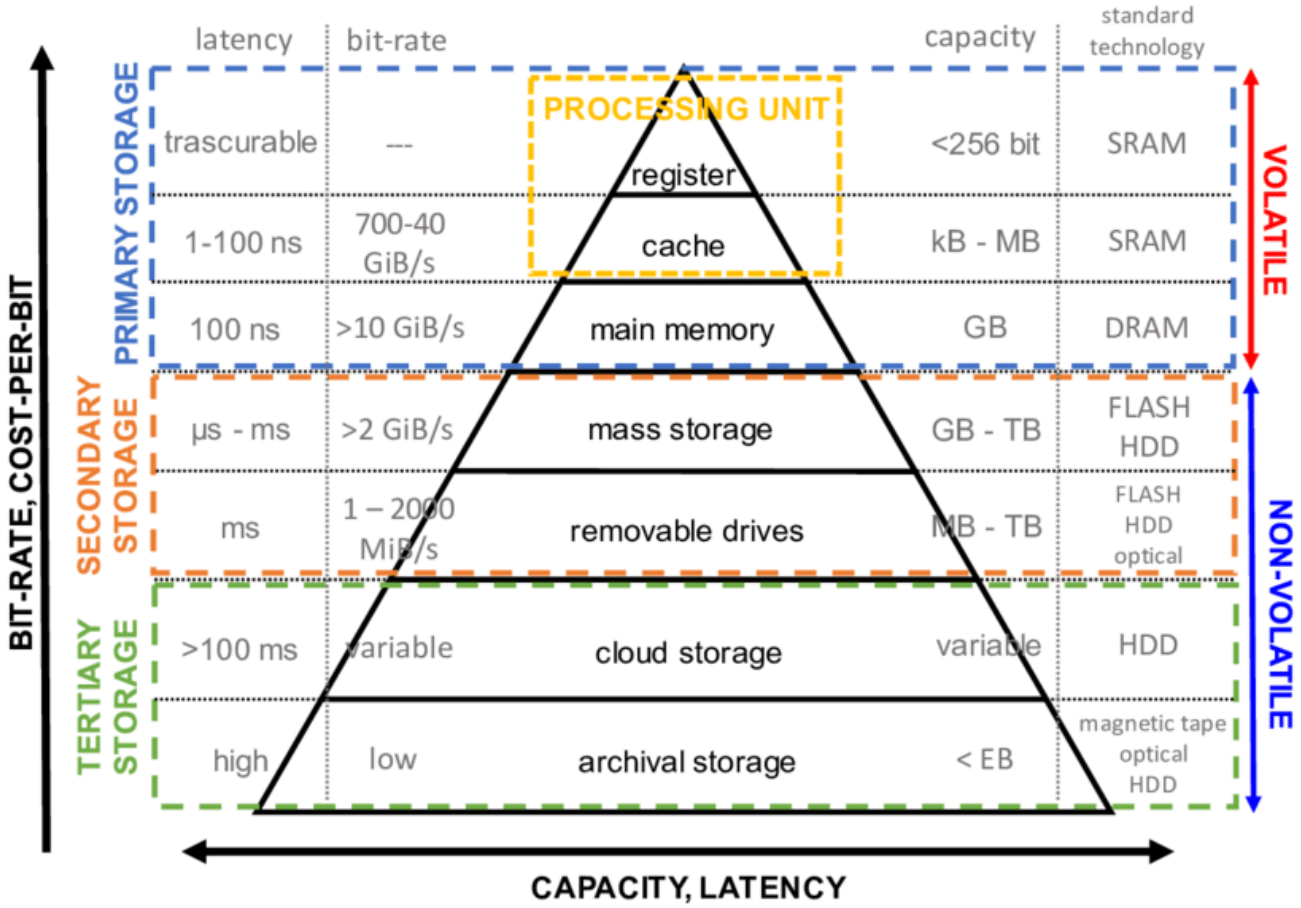
Koszt trafienia/chybienia (w cyklach) dla procesora Pentium M:

rejestr	$\leqslant 1$
L1d	≈ 3
L2	≈ 14
pamięć główna	≈ 240

Zob.: Interactive latency

```
$ curl cheat.sh/latencies
```

Hierarchia pamięci[7]



[7] E. Gemo, *The design and analysis of novel integrated phase-change photonic memory and computing devices*, 2021

Hierarchia pamięci

Im niżej w hierarchi od CPU

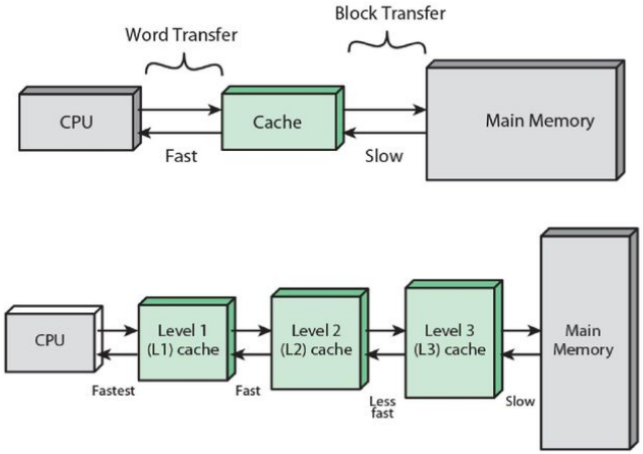
- malejąca cena/bit
- rosnąca pojemność
- rosnący czas dostępu
- malejąca częstość dostępu przez preceosor

Rodzaj dostępu do pamięci:

- dostęp sekwencyjny
  - odczyt/zapis wyłącznie w określonej sekwencji liniowej (np. pamięć taśmowa)
  - czas dostepu zależy od położenia danych
- dostęp swobodny
  - pozycja w pamięci dostępna za pomocą unikatowego adresu (pamięć RAM)
  - czas dostepu stały, nie zalezy od lokalizacji danych
- dostęp bezpośredni
  - pamięć podzielona na bloki (lub rekordy) o unikatowych adresach (np. pamięć dyskowa)
  - najpierw dostęp (swobodny) do najbliższego otoczenia (wybór bloku) a potem poszukiwanie loakcji koncowej (sekwencyjne)
- dostęp skojarzeniowy
  - rodzaj dostępu swobodnego, który umożliwia porównywanie i specyficzne badanie zgodności wybranych bitów wewnątrz słowa (zachodzi to dla wszystkich słów jednocześnie)
  - słowo jest wyprowadzane na podstawie części swojej zawartości, a nie na podstawie adresu (zastosowanie: pamięć podręczna procesora)
  - stały czas dostępu

Pamięć podręczna (cache)

działa jako pośrednik między ultra-szybkim procesorem a wolną pamięcią główną



- przechowuje kopie fragmentu pamięci głównej
- proceseor najpierw sprawdzana obecność danych w pamięci podręcznej
- jeśli danych brakuje (chybienie), to do pamięci podręcznej wczytywany jest właściwy blok pamięci głownej

Lokalność przestrzenna

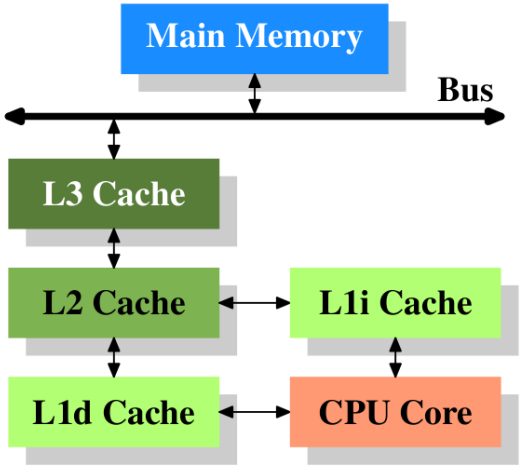
- tendencja do częstych odwołań do danych położonych blisko siebie w pamięci
- takie dane mają szansę trafić do tego samego bloku w pamięci podręcznej

Lokalność czasowa

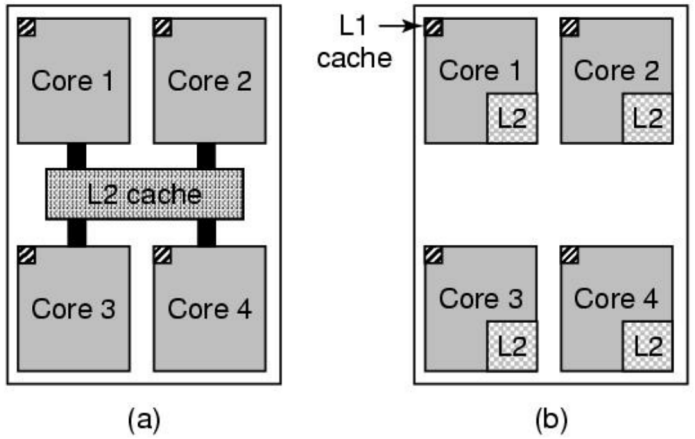
- tendencja do częstych odwołań do tych samych danych w krótkich odstępach czasu
- dane często wykorzystywane mają szansę pozostać w pamięci podręcznej pomiędzy rządaniami

Procesor i pamięć

Architektura jednoprocesorowa



Architektura wieloprocesorowa

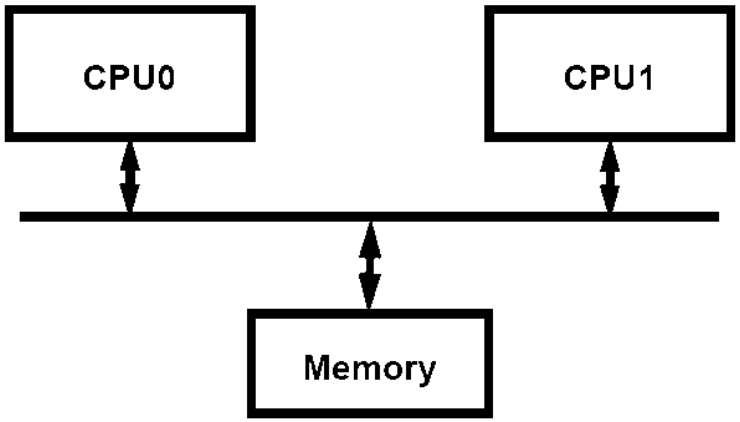


a) współdzielony L2 przez 4 procesory
b) niezależne L1 i L2 w układzie 4 procesorów

U.Drepper, Red Hat Inc. *What every programmer should know about memory*

Wieloprocesorowość symetryczna (SMP)

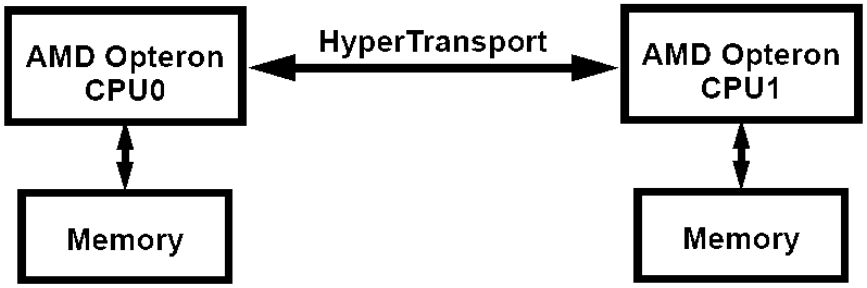
Symmetric Multiprocessing



- procesory współdzielą zasoby pamięci oraz urządzenia wejścia/wyjścia przy pomocy magistrali systemowej
- każdy z procesorów może zostać przypisany do wykonywania konkretnego zadania, tak aby w systemie następowało równoważenie obciążenia (*load balancing*)
- przepustowość szyny i pamięci ogranicza szybkość działania procesorów

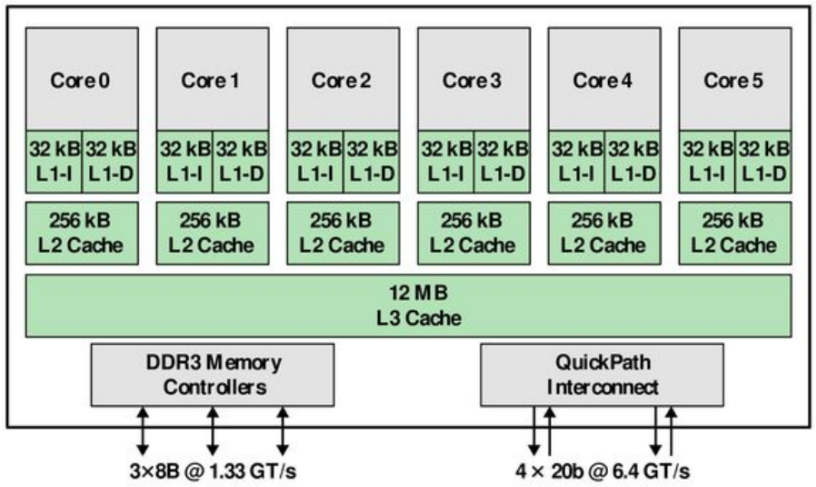
Niejednolity dostęp do pamięci (NUMA)

Non-Uniform Memory Access



- minimalizacja opóźnień dostępu do pamięci poprzez przydzielenie pamięci lokalnej dla każdego procesora
- każdy procesor posiada własny kontroler pamięci lokalnej
- procesory łączone między sobą za pomocą szyny o dużej przepustowości i niskich opóźnieniach (HyperTransport), która nie ma bezpośredniego połącznia z układem pamięci
- wydłużony czas dostępu do pamięci zdalnej (połączonej z innym procesorem)
- wymaga właściwej organizacji danych w pamięci aby zapewnić efektywność

Architekruty wieloprocesorowe



Rysunek 1: Intel Core-i7 990X

- wiele procesorów (rdzeni) na jednej kości
- każdy rdzeń posiada jednostkę wykonawczą, rejestry, lokalną pamięć podręczną, rdzeńie mogą być wielwątkowe
- rdzenie współdzielą pamięć L3
- pamięć połączona z procesorem za pomocą szybkiej szyny (DDR3), urządzenia I/O za pomocą osobnej szyny (PCIe)

Rodzaje pamięci półprzewodnikowych:

RAM (*Random Access Memory*) pamięć o dostępie swobodnym – odczyt-zapis, wymazywanie/zapisywanie elektryczne na poziomie bajta

Static RAM (SRAM) - statyczna pamięć o dostępie swobodnym

- rejestry i pamięć podręczna
- ulotna, przechowuje dane tak długo, jak długo włączone jest zasilanie
- każdy bit przechowywany w układzie zbudowanym z 6 transystorów (mniejsza gęstość danych niż DRAM)

Dynamic RAM (DRAM) - pamięć dynamiczna

- pamięć głowna komputera
- ulotna, wymaga okresowego odświeżania zawartości

Pamięci dynamiczne

Synchronous DRAM (SDRAM) - pamięc dynamiczna pracująca synchronicznie z magistralą systemową

	napięcie	pojemność	przepustowość	rok
single data rate				
SDR SDRAM	3.3 V	512 MB	533-1066 MB/s	1996
double data rate				
DDR SDRAM:	2.5 V	1 GB	1.6-3.2 GB/s	2000
DDR2 SDRAM	1.8 V	4 GB	3.2-6.4 GB/s	2003
DDR3 SDRAM	1.5 V	8 GB	6.4-19.2 GB/s	2007
DDR4 SDRAM	1.2 V	16 GB	12.8-25.6 GB/s	2012
DDR5 SDRAM	1.1 V	64 GB	25.6-64 GB/s	2020

$$\text{moc} = \text{pojemność} \times \text{napięcie}^2 \times \text{częstotliwość}$$

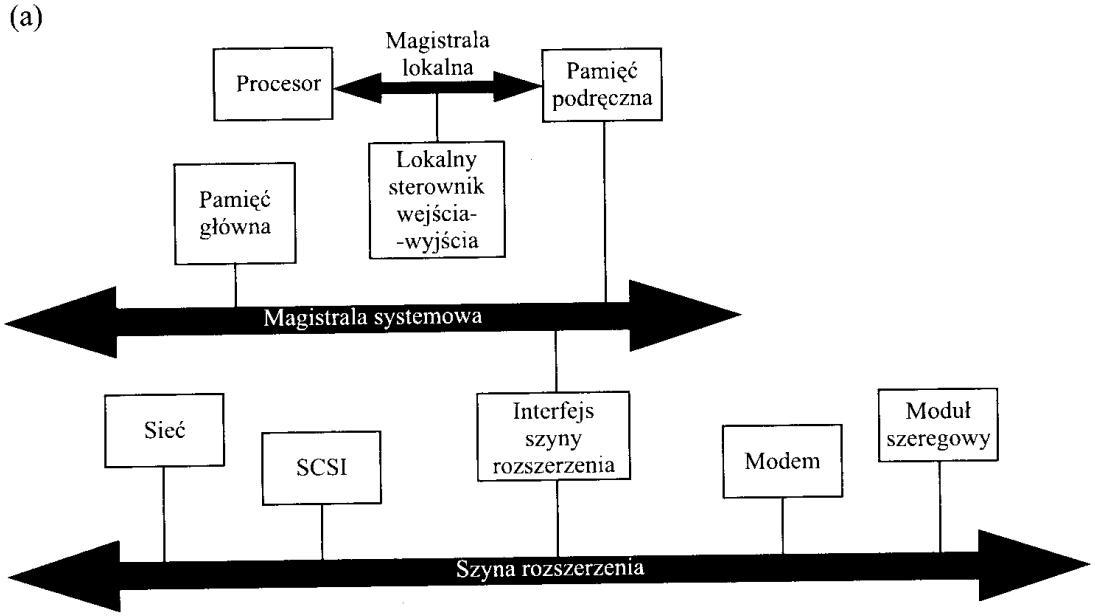
DDR Generations: Memory Density and Speed

Rodzaje pamięci półprzewodnikowych nieulotnych

- **ROM** (*Read-Only Memory*) pamięć stała – tylko odczyt, zapisywanie w trakcie produkcji, tania i szybka
- **PROM** (*Programmable ROM*) programowalna pamięć jednokrotnego zapisu
- **EPROM** (*Erasable and Programmable ROM*) wymazywalna i programowalna pamięć stała – głównie odczyt, wymazywanie światłem UV, zapisywanie elektryczne
- **pamięć błyskawiczna** (*flash memory*) – wymazywanie elektryczne na poziomie bloku (256 B-16 KB), zapisywanie elektryczne; szybki odczyt, wolniejszy zapis (SSD, pamięci USB, karty pamięci MC, MMC, SM)
- **NVRAM** (*Non-Volatile RAM*) – pamięć typu SRAM/DRAM z baterią podtrzymującą zawartość po odłączeniu zasilania (zastosoanie BIOS)[8]
- **NVDIMM** (*Non-Volatile Dual In-line Memory Module*) – pamięć typu DRAM powiązana z układami pamięci błyskawicznej, co pozwala zachować zawartość pamięci ulotnej po awarii zasilania

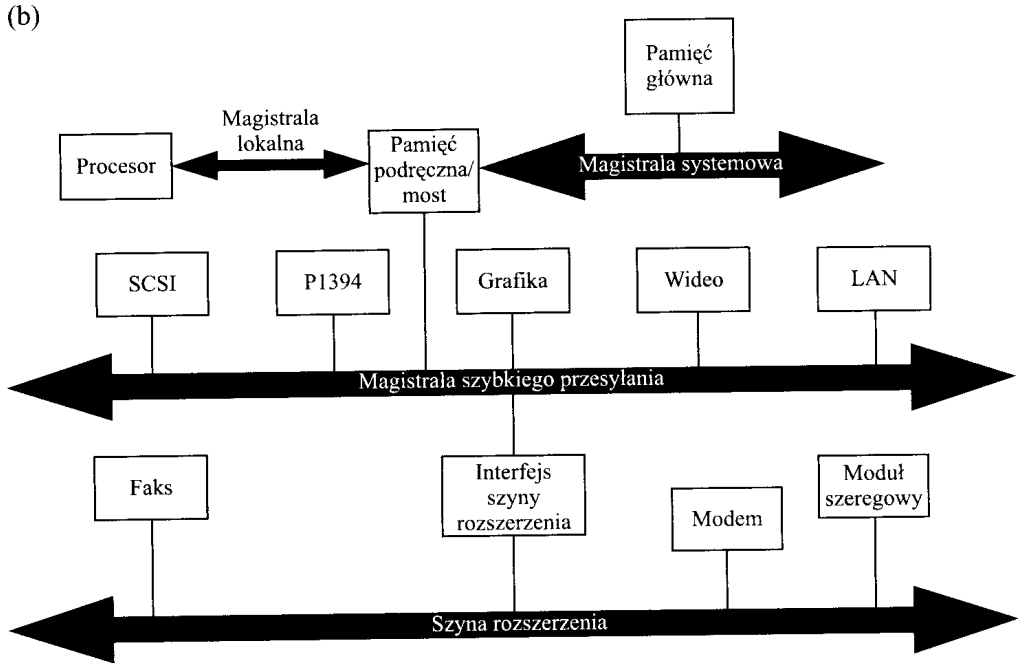
[8] Także EEPROM (*Electrically Erasable and Programmable ROM*) wymazywalna i programowalna pamięć stała (na poziomie bajtu) – głównie odczyt, wymazywanie/zapisywanie elektryczne; szybki odczyt, wolny zapis.

Magistrale (szyny) systemowe



- komunikacja między szyną systemową i szyną rozszerzenia może nastepować bez angażowania CPU

Magistrale (szyny) systemowe



- hierarchiczna magistrala o wysokiej wydajności
- urządzenia o większych wymaganiach są blizej CPU

Magistrale urządzeń

Nazwa	Rok powstania	Aktualna wersja	Przepustowość	Typ / Szerokość linii	Częstotliwość	Zastosowania
ISA	1981	brak	8–16 MB/s	równoległa 8–16 bitów	8 MHz	karty rozszerzeń, modemy, karty dźwiękowe
EISA	1988	brak	33 MB/s	równoległa 32 bity	8.33 MHz	karty rozszerzeń, karty graficzne
PCI	1992	PCI 2.3	533 MB/s	równoległa 32–64 bity	33–66 MHz	karty rozszerzeń, karty graficzne, karty sieciowe
PCIe	2004	PCIe 6.0	256 GB/s (x16)	szeregowa 1, 2, 4, 8, 16 linii	64 GHz	karty graficzne, dyski SSD, karty rozszerzeń
USB	1996	USB4 2.0	10 GB/s	szeregowa 1 para różnicowa	20 GHz	urządzenia peryferyjne, pamięci masowe, urządzenia mobilne
IDE/ATA	1986	ATA/133	133 MB/s	równoległa 16 bitów	133 MHz	dyski twarde, napędy optyczne
AGP	1997	AGP 3.0	2.1 GB/s	równoległa 32 bity	66 MHz	karty graficzne
SATA	2003	SATA III	600 MB/s	szeregowa 1 para różnicowa	6 GHz	dyski twarde, dyski SSD, napędy optyczne
SCSI	1986	SCSI Ultra-640	640 MB/s	równoległa 8-16 bitów	160 MHz	dyski twarde, skanery, serwery
SAS	2004	SAS 4	2.3 GB/s	szeregowa 1 para różnicowa	22.5 GHz	dyski twarde, serwery, pamięci masowe

ISA (*Industry Standard Architecture*, IBM 1980)

- wymaga ręcznej konfiguracja IRQ, adresów I/O oraz kanału DMA

EISA (*Extended ISA*, Gang of Nine 1989)

- rozszerzenie ISA umożliwiające 32-bitowy transfer danych
- adresowanie do 4 GB RAM
- *Plug and Play*, prez problemy z konfliktami IRQ nazywany *plug and pray*

Ograniczenia:

- brak wsparcia w nowoczesnych systemach operacyjnych (np. Windows 10, Linux nie obsługuje ISA natywnie)
- przestarzała, zastąpiona przez PCI.

PCI (*Peripheral Component Interconnect*, Intel 1992)

- obsługuje standard podłącz i używaj (PnP, *Plug and Play*), autokonfiguracja urządzań
- system odpytuje PCI przy starcie o dostępne urządzenia
- każde urządzenie na osobnej szynie z osobnym adresem
- szerokie wsparcie w systemach operacyjnych (Windows, Linux, macOS)
- możliwość użycia przerwań współdzielonych (*IRQ sharing*), co poprawia skalowalność

Ograniczenia:

- szyna współdzielona, ograniczona przepustowość
- ograniczona liczba urządzeń (maksymalnie 5 slotów na mostku PCI, 256 urządzeń w systemie)
- konflikty przerwań (IRQ) w starszych systemach (np. Windows 98)
- popularna w latach 90., zastąpiona przez PCIe.

Magistrala PCI Express

- wysoka przepustowość (do 64 GB/s w wersji 6.0), idealna dla kart graficznych, dysków NVMe i sieciowych kart wysokiej wydajności.
- niski opóźnienia i skalowalność dzięki architekturze *point-to-point*
- każde urządzenie jest połączone bezpośrednio z kontrolerem
- wsparcie dla *hot-plugging* (podłączania urządzeń „na gorąco")
- pełna kompatybilność z nowoczesnymi systemami operacyjnymi
- obsługa wielu linii (x1, x4, x8, x16), zależnie do potrzeb urządzenia
- zastępuje PCI i AGP (Advanced Graphics Port, PCI dla kart graficznych)
- brak linii przerwań IRQ, używa sygnałów MSI (Message Signaled Interrupts)

Wady i problemy:

- wyższy koszt implementacji w porównaniu do PCI
- wymaga nowoczesnych sterowników, co może powodować problemy z kompatybilnością w starszych systemach

Uniwersalna szyna szeregowa (*Universal Serial Bus*, USB, 1995)

- standard szyny zewnętrznej do podłączania do komputera do 127 urządzeń peryferyjnych (jeden IRQ)
- *Plug and Play* — automatyczna detekcja i konfiguracja urządzeń
- *hot plugging* - podłączenie nie wymaga wyłaczenia zasilania
- łatwość rozmnażania portów i wydłużania połączenia poprzez zastosowanie maksymalnie pięciu koncentratorów (*USB hubs*),
- załącze uniwersalne, wsparcei szerokiej gamy urządzań (klawiatura, mysz, drukarki, skanery, kamery wideo, dyski)
- szeroka kompatybilność z systemami operacyjnymi (Windows, Linux, macOS, Android)

Ograniczenia:

- wymaga obecności dokładnie jednego kontolera po stronie hosta (niemożliwe podłączenie dwóch komputerów)

IDE (*Integrated Drive Electronics*, 1986) właściwie **ATA** (*Advanced Technology Attachement*), od 2003 – **PATA** (*Parallel ATA*)

- szyna do łączenia dysków HDD i napędow CDROM (40 pinów)
- MTBF $\approx 1.2 \times 10^6$ h [9] (prawd. awarii dysku w czasie roku $\frac{24h \cdot 356}{MRFB} \approx 7\%$)
- zastąpiony przez SATA

Ograniczenia:

- niska przepustowość (do 133 MB/s w ATA/133), co ogranicza wydajność nowoczesnych dysków.
- bak wsparcia dla *hot-plugging*
- ograniczona liczba urządzeń, każdy sterownik IDE potrzebuje IRQ i może obsłużyć 2 urządzenia wewnętrzne (*master* i *slave*)
- brak wsparcia w nowoczesnych systemach operacyjnych (wymagany tryb kompatybilności lub emulacja)

[9] MTBF (*Mean Time Between Failures*) – czas po jakim z pewnej puli urządzeń zostaje połowa sprawnych. Z 1000 dysków o MTBF 1.2×10^6 h będzie się psuł jeden co (średnio) $\frac{MTFB}{24h \cdot 1000} = 50$ dni.

Szeregowy ATA (*Serial ATA*, SATA)

- kolejkowanie zadań w celu minimalizacji liczby skosków głowicy
- powielacze portów - możliwość podłoczęnia SATA do wielu urządzeń
- wyznacznik portu - mozliwość podłączenia dwóch portów do tego samego SATA (nadmiarowa ścieżka)
- połączenia punkt-punkt
- możliwość podłączania wewnętrzych i zewnętrznych urządzeń
- CRC (*Cyclic Redundancy Check*) dla pakietów danych, rozkazów i statusu
- łatwość przyłączenia (wsparcie dla *hot swapping* - wymiana urządzenia bez rerestaru systemu)
- Szerokie wsparcie w systemach operacyjnych (Windows, Linux, macOS).
- MTBF $\approx 2.5 \times 10^6$ h

SCSI (*Small Computer Systems Interface*)

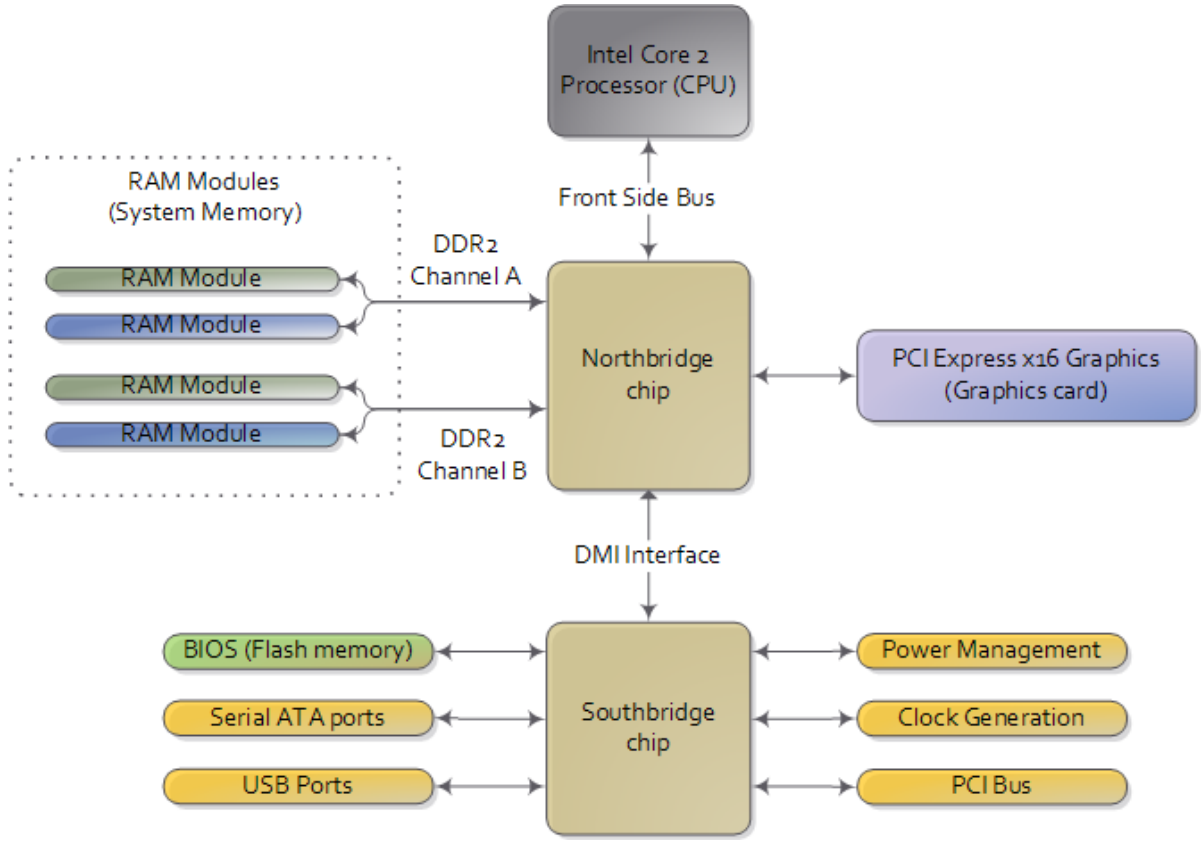
- wysoka wydajność i niezawodność, szczególnie dla stacji roboczych i serwerów (odpowiednik ATA w PC ale mniej obciążający CPU)
- jeden IRQ, obsługa do 16 urządzeń (w tym sterownik), łącze wielopunktowe (*multidrop*) dyski, napędy CD-ROM, napędy taśmowe (*streamer*), skanery
- maksymalna długość do 25 m
- możliwość podłączania urządzeń o różnej prędkości na tym samym łańcuchu.
- MTBF $\approx 1.6 \times 10^6$ h

https://en.wikipedia.org/wiki/Parallel_SCSI

SAS (*Serial Attached SCSI*)

- szeregowa szyna, zastąpiła SCSI
- jeden IRQ, połączenie punkt-punkt (*initiator-target*), tryb pełnego dupleksu
- zastosowanie ekspanderów (*expander*) daje możliwość podłączenia do 65535 urządzeń
- MTBF $\approx 1.6 \times 10^6$ h

Architektura płyty głównej



G. Duarte http://duartes.org/gustavo/blog/post/motherboard-chipsets-memory-map

Chipset

- **Front Side Bus** - magistrala wyjściowa procesora
  Komunikacja procesora z resztą świata:
  - adresy pamięci RAM
  - adresy I/O mapowane na adresy pamięci (Memory Mapped I/O)
  - przerwania i sygnały
- **mostek północny** - łączy CPU z urządzeniami dużej szybkości: RAM, GPU, PCIe, Gigabit Ethenet
- **mostek południowy** - współpraca z pozostałymi urządzeniami ICH (I/O Controller Hub) Intel, FCH (Fusion CH) AMD
- elementy chipsetu: sterownik CPU, sterownik DRAM, sterownik cache, sterownik przerwań, sterownik DMA, sterownik magistrali,
- mostek północny obecnie często zintegrowany z południowym lub nawet z CPU

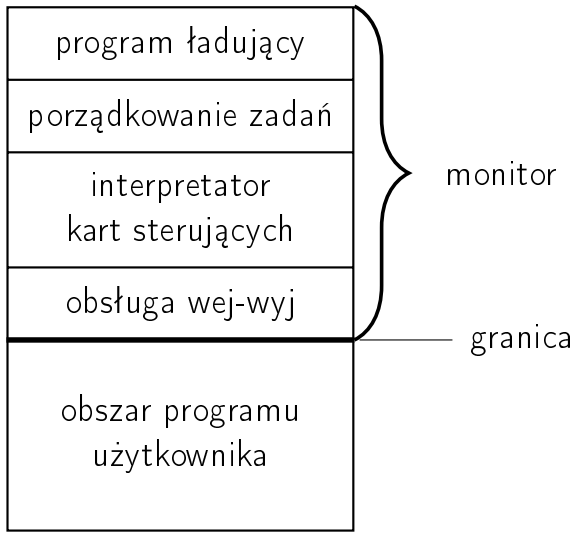
Jak system wykonuje zadania?

- Przetwarzanie wsadowe, monitor prosty, praca pośrednia
- Buforowanie, Spooling
- Wieloprogramowość
- Wielozadaniowość, wielodostępowość, systemy z podziałem czasu
- Wieloprocesorowość, systemy HPC
- Systemy rozproszone, klastrowe i chmurowe
- Systemy czasu rzeczywistego

Monitor prosty

- wczesne rozwiązania zarządzania zadaniami w systemach operacyjnych
- **przetwarzanie wsadowe** (*batch processing*) - zadania gromadzone w postaci wsadów są kolejno wykonywane, bez interakcji użytkownika
- **monitor rezydujący** – stale obecny w pamięci, automatyczne przekazuje sterowania od zadania do zadania, monitoruje i zarządza określonymi zasobami lub zdarzeniami
- automatyczne porządkowanie zadań - przydzielanie czasu procesora poszczególnym zadaniom w sposób sekwencyjny lub według prostych reguł
- każde zadanie musiało zakończyć się lub zrezygnować z procesora, zanim kolejne mogło zostać uruchomione
- w najprostszych monitorach brak mechanizmu przerwań sprzętowych - programista musiał jawnie wywoływać funkcje monitora, aby oddać sterowanie

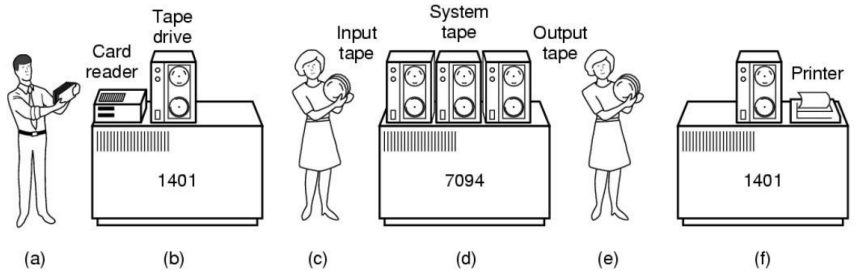
Monitor rezydujący w pamięci



- **ochrona pamięci** - program nie może ingerować w obszar pamięci monitora
- **uprzywilejowane instrukcje** - pewne instrukcje moga być wykonywane wyłącznie przez monitor (tryb jądra)

Praca pośrednia (off-line)

- **Praca pośrednia** - przetwarzanie danych, gdzie urządzenia wejścia/wyjścia (np. czytniki kart, drukarki, taśmy magnetyczne) nie są bezpośrednio sterowane przez procesor komputera, lecz przez odrębne urządzenia lub programy pośredniczące
- dane są najpierw zbierane lub przygotowywane przez mniejsze, specjalizowane urządzenia, a następnie przekazywane do komputera w blokach (np. na taśmie magnetycznej)
- procesor nie traci czasu na oczekiwanie na wolne urządzenia we/wy



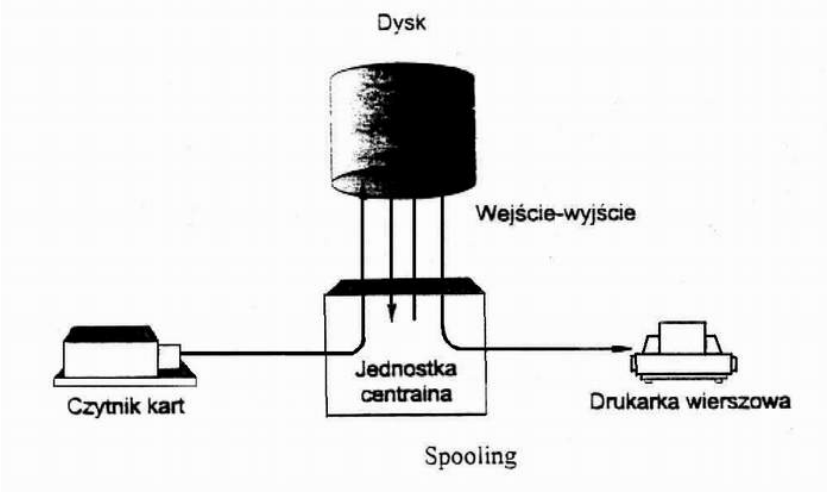
Buforowanie

Buforowanie jest metodą usprawniającą wykonywanie obliczeń i operacji wej-wyj dla danego zadania.

- polega na tymczasowym przechowywaniu danych w pamięci (buforze) przed ich dalszym przetwarzaniem lub przesyłaniem
- buforowanie zmniejsza liczbę operacji I/O, przyspiesza dostęp do danych
- procesor może pracować nad innymi zadaniami, podczas gdy dane są ładowane do bufora. To pozwala na lepsze wykorzystanie zasobów procesora i zmniejsza czas oczekiwania na dane
- przykłady: pamięć podręczna procesora, bufor dyskowy, buforowanie w systemach plików, buforowanie w sieciach komputerowych (DNS cache)

Spooling

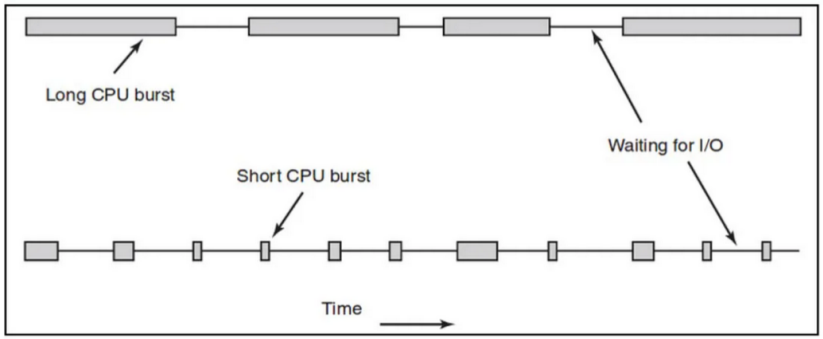
- **spooling** (*Simultaneous Peripheral Operation On-Line*) jednoczesna bezpośrednia praca urządzeń
- dane są przechowywane w plikach na dysku w celu ich późniejszego przetwarzania przez urządzenia peryferyjne (np. drukarki)
- spooling umożliwia równoczesne wykonywanie operacji wej-wyj jednego zadania i obliczeń dla innych zadań znajdujących się we wsadzie
- spooling wytwarza pulę zadań do wykonania i umożliwia planowanie zadań i wieloprogramowość



Spooling

Zadania ograniczone CPU i ograniczone I/O

- Zadania ograniczone operacjami wej-wyj (*I/O bound*)
  - prędkość przetwarzania zależna od szybkości urządzeń wej-wyj.
  - buforowanie może zwiększyć przepustowość poprzez grupowanie wielu operacji I/O
- Zadania ograniczone procesorem (*CPU bound*)
  - prędkość przetwarzania ograniczona przez szybkość CPU (bufor wejściowy zawsze pełny, bufor wyjściowy zawsze pusty)
  - buforowanie zwiększa wydajność gromadząc często używane dane i zmniejszając czas dostępu

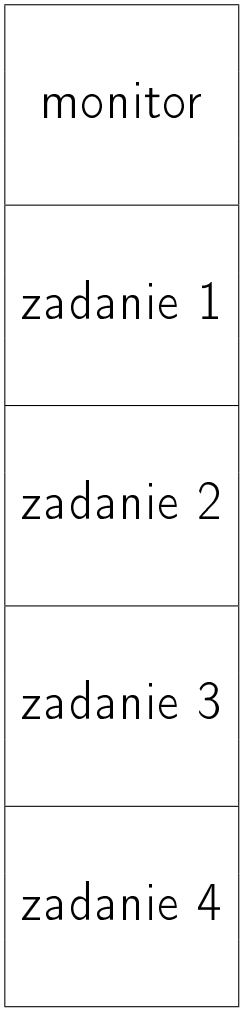


Wieloprogramowość

- **wieloprogramowość** - w pamięci rezyduje więcej niż jeden program użytkownika, ale tylko jeden jest wykonywany w danym momencie przez procesor
- w momencie zablokowania (np. operacja I/O) procesor jest przełaczany do innego zadania
- celem jest maksymalizacja wykorzystania CPU

Realizacja wieloprogramowości wymaga skomplikowanego systemu operacyjnego:

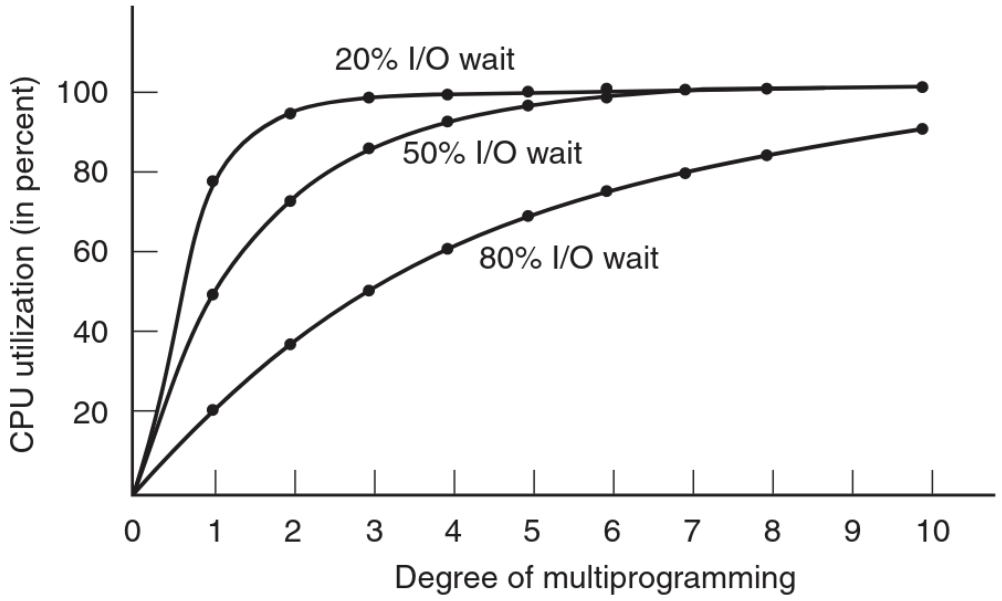
- ochrona pamięci
- planowanie przydziału procesora, kolejka zadań
- zarzadzanie przydziałem pamięci
- ochrona zasobów



Wieloprogramowość: wykorzystanie CPU

Prosty model wykorzystania CPU:

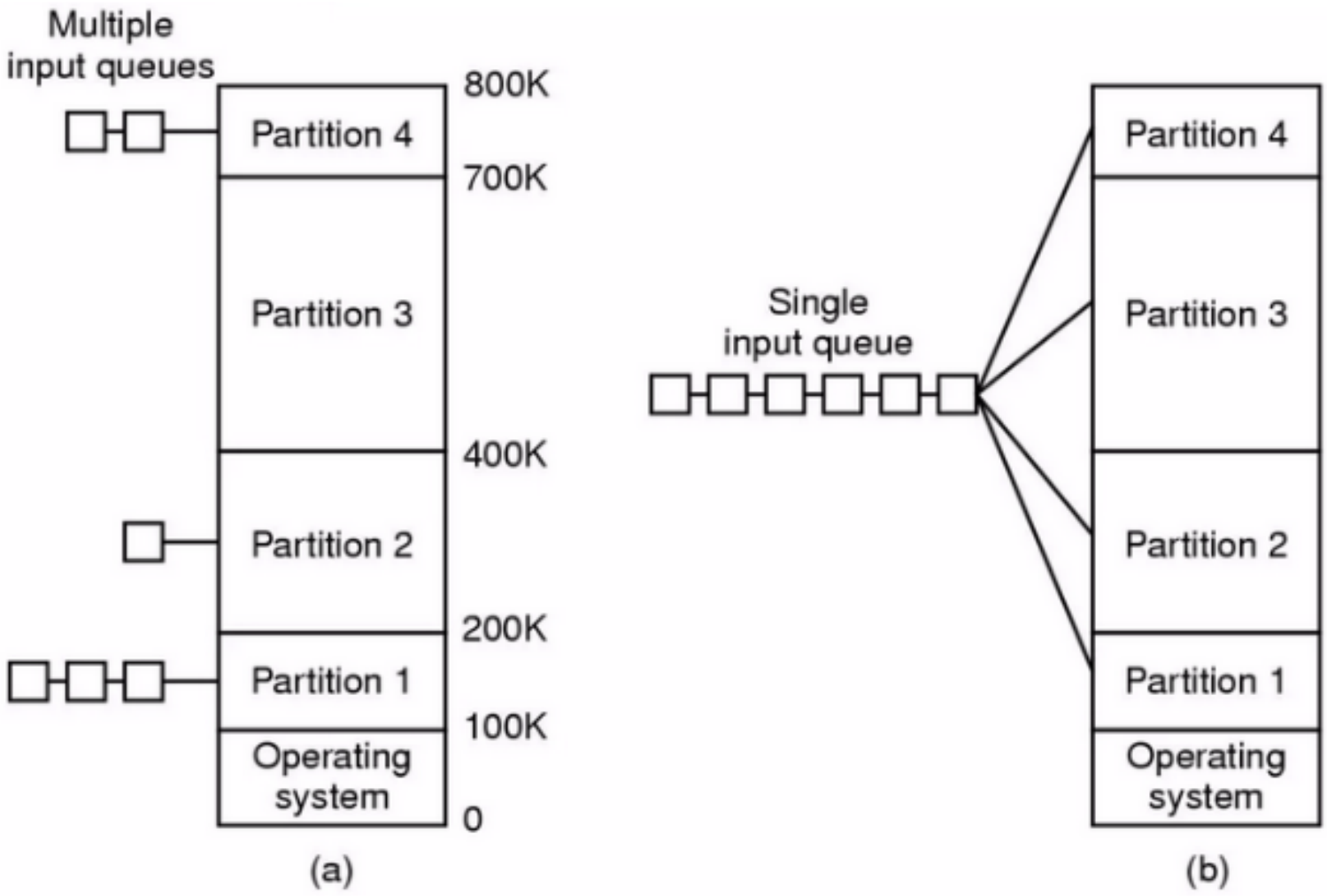
- n-procesów, każdy spędza część swego czasu w stanie I/O
- p - prawdopodobieństwo, że pojedynczy proces w danym przedziale czasowym nie korzysta z procesora (oczekuje na I/O)
- p^n – prawdopodobieństwo, że n-procesów będzie w stanie I/O
- wykorzystanie CPU $= 1 - p^n$



Wieloprogramowość: wykorzystanie pamięci

- podział pamięci:
  - partycje o **stałym** rozmiarze (jednakowej lub różnej wielkości)
  - **dynamiczny** - obszary pamięci dopasowane do rozmiaru zadania
- kolejka zadań oczekujących na przydział pamięci:
  - jedna wspólna kolejka dla wszystkich partycji
  - osobne kolejki dla każdej partycji
- algorytm wyboru zadania z kolejki, np. FIFO
- problem fragmentacji pamięci:
  - **wewnętrzna** - niewykorzystana pamięć wewnątrz przydzielonego bloku, gdy blok pamięci większy od rozmiaru zadania
  - **zewnętrzna** - niewykorzystane obszary pomiędzy przydzielonymi blokami, wolna pamięć jest rozproszona w wielu małych, nieciągłych obszarach

Wieloprogramowość: wykorzystanie pamięci



Wieloprogramowość ze stałym podziałem pamięci

a.) Osobne kolejki zadań do każdej partycji

- zadanie trafia do kolejki do najmniejszej pasującej partycji
- duże partycje moga pozostawać puste, nawet gdy są zadania w innych kolejkach, które mogłyby się zmieścić
- małe zadania muszą czekać

b.) Wspólna kolejka do wszystkich segmentów:

- zadanie z początku kolejki trafia do pierwszej wolnej partycji
- może powodować marnowanie pamięci na małe zadania
- potrzebny alg. wyszukiwania zadan w kolejce pasujących do wolnego obszaru - ale to dyskryminuje małe zadania

Wieloprogramowość z dynamicznym podziałem pamięci

- obszary tworzone dynamicznie dopasowane do rozmiaru zadania
- brak fragmentacji wewnętrznej - efektywne wykorzystanie pamięci RAM
- konieczność wykonywania upakowań (defragmentacji) w celu uniknięcia fragmentacji zewnętrznej (niewykorzystane obszary pomiędzy zaalokowanymi blokami), co zwiększa użycie CPU

Obecnie stosowane rozwiązanie: pamięć wirtualna, segmentacja o stałym rozmiarze i mechanizm stronicowania

Wielozadaniowość

- praca wsadowa a praca interakcyjna
- **wielozadaniowość** (*multitasking*) - wykonywanie wielu procesów w określonym przedziale czasu. Użytkownik ma wrażenie, że programy działają jednocześnie:
  - **współbieżna** na jednym procesorze przez szybkie przełączanie zadań (podział czasu)
  - **równoległa** na wielu procesorach
- **wielodostępowość** - wielu użytkowników korzysta z systemu jednocześnie (np. terminale podłączone do jednego komputera)

Wielozadaniowość w systemach z podziałem czasu

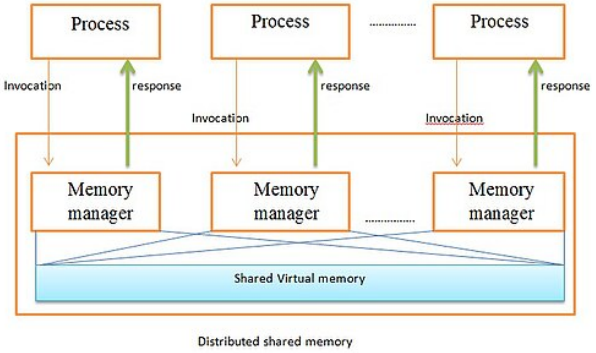
- **system z podziałem czasu** – system wielozadaniowy, w którym procesor jest przydzielany procesom w krotkich, równych porcjach czasu (kwantach), gdy kwant się kończy to CPU przedzielane innemu procesowi
- celem jest **interaktywność** - każdy użytkownik otrzymuje „wrażenie" ciągłej pracy systemu

Systemy z podziałem czasu są skomplikowane, gdyż wymagają:

- realizacji mechanizmów działań współbieżnych
- zarządzania pamięcią
- ochrony pamięci
- planowania przydziału CPU
- administrowania pamięcią dyskową
- systemu plików dostępnych bezpośrednio

Systemy HPC (*High Performance Computing*)

- **wieloprocesowość** (*multiprocessing*) - systemy posiadające więcej niż jeden procesor pozwalające na równoczesne przetwarzanie zadań
- **SMP** (*Symmetric MultiProcessing*)
  - współdzielona pamięć, I/O
  - wspólny system operacyjny
  - jednostki CPU są równoważne
- **DSM** (*Distributed Shared Memory*)
  - rozdzielone fizycznie pamięci są adresowane wspólną przestrzenią adresową
  - tańszy niż SMP i dobrze się skaluje
  - pozwala na obsługe dużych baz danych
  - wolniejsza komunikacja między węzłami niż SMP



Systemy HPC (*High Performance Computing*)

- **MPP** (*Massively Parallel Processor*)
  - setki/tysiące procesorów, rozproszona pamięć
  - kazdy CPU posiada własny system operacyjny, własną pamięć
  - podsystemy komunikują się łączami o dużej szybkości
- **klastry obliczeniowe**
  - zbiór węzłów pracujących wspólnie widocznych jako jedna jednostka obliczeniowa
  - połączenie przez sieć lokalną lub szybkie łącza
- **przetwarzanie sieciowe (grid)**
  - zbiór geograficznie rozproszonych zasobów komputerowych

Rygorystyczne systemy czasu rzeczywistego

- sterowniki urządzeń o ścisłe określonym zastosowaniu: nadzorowanie procesów produkcyjnych, eksperymentów naukowych, kierowanie sygnalizacją świetlną, autopilot, itp.
- działanie podlega ostrym rygorom czasowym, nawet kosztem efektywnego wykorzystania systemu

Łagodne systemy czasu rzeczywistego

- działanie podlega złagodzonym rygorom czasowym
- niespełnienie wymagań czasowych nie jest niebezpieczne ale powoduje pogorszenie jakości usług świadczonych przez system
- zastosowania: techniki multimedialne, tworzenie wirtualnej rzeczywistości, urządzenia zdolne do samodzielnej eksploracji

Obsługa urządzeń wejścia-wyjścia

Jak ma współdziałać jednostka centralna z urządzeniami wej-wyj?

- sterownik urządzenia, mapowanie rejestrów urządzenia, MMIO
- aktywne czekanie
- odpytywanie
- przerwania
- bezpośredni dostęp do pamięci

Sterowniki i moduły sterujące urządzeń

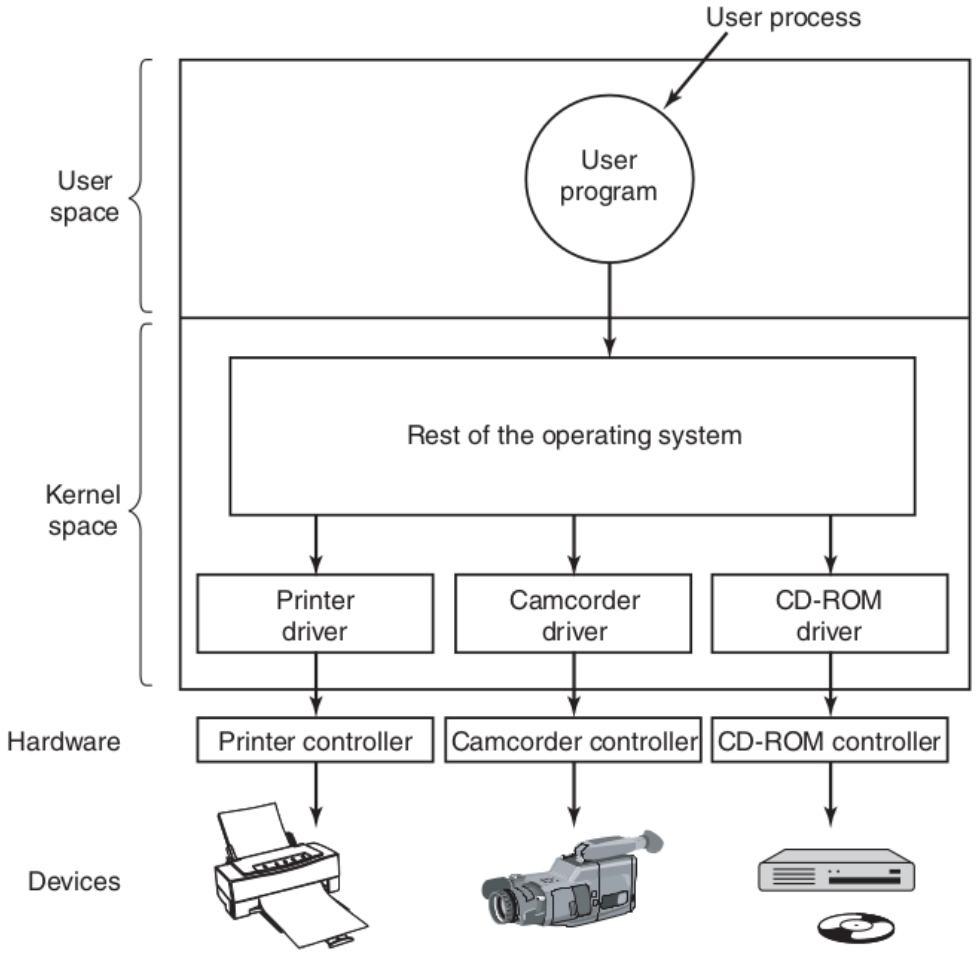


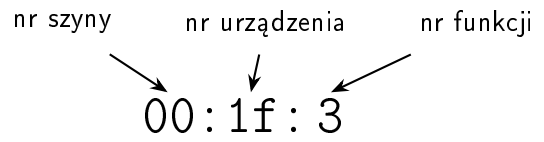
Figure 5-12. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

Sterowniki i moduły sterujące urządzeń

- **Sterownik urządzenia** (*device controller*) związany jest z konkretnym urządzeniem i rozporządza lokalnym buforem i zbiorem rejestrów o specjalnym przeznaczeniu. Odpowiada za przesyłanie danych między urządzeniem zewnętrznym, a własnym buforem.
- **Moduł sterujący (obsługi) urządzenia** (*driver*) jest odpowiedzialny od strony systemu operacyjnego za komunikację ze sterownikiem urządzenia.

```
00:1f.3 SMBus: Intel Corporation 8 Series SMBus Controller (rev 04)
        Subsystem: Lenovo Device 220c
        Flags: medium devsel, IRQ 18
        Memory at e0638000 (64-bit, non-prefetchable) [size=256]
        I/O ports at efa0 [size=32]
        Kernel driver in use: i801_smbus
        Kernel modules: i2c_i801
```

Pliki `/proc/iomem`, `/proc/ioports`, polecenie: `lspci -v`

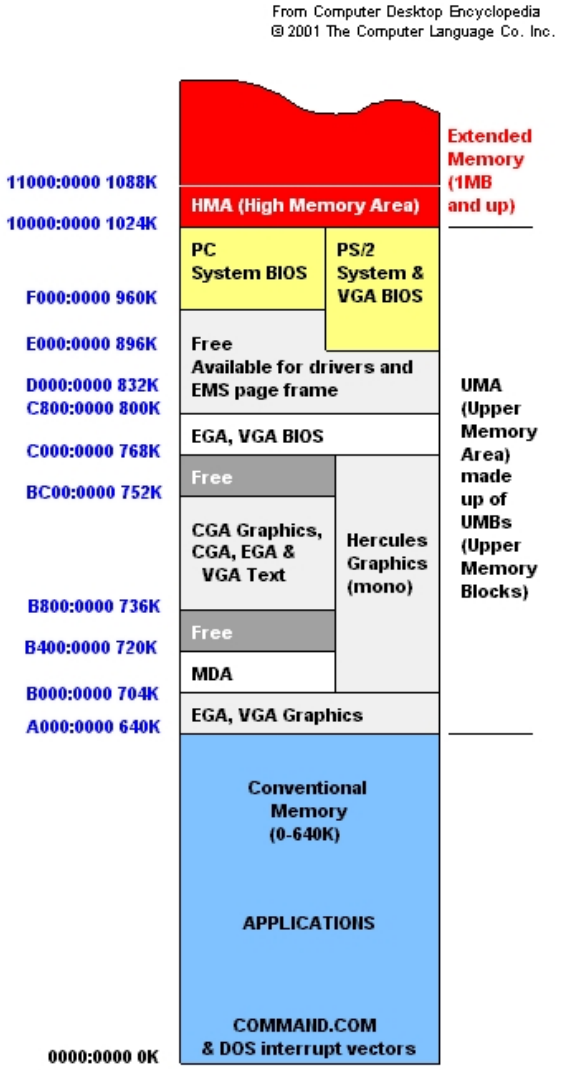


MMIO (memory-mapped input/output)

- komunikacja z urządzeniem - zapis/odczyt rejestrów urządzenia
- zbiór rejestrów urządzenia tworzy **przestrzeń portów wej-wyj**
  - potrzebne specjalne operacje dostępu `IN-OUT`
- odwzorowanie rejestrów w przestrzeni adresowej pamięci MMIO (*memory-mapped input/output*)
  - nie potrzeba dodatkowych instrukcji obsługi
  - operacja zapisu (odczytu) adresu pamięci, skutkuje zapisem (odczytem) fizycznego rejestru urządzenia
  - ochrona dostępu realizowana za pomocą mechanizmów ochrony pamięci
  - część przestrzeni adresowej jest zarezerwowana dla urządzeń, nie może być użyta do komunikacji z pamięcią - zjawisko dziury w pamięci[10]

[10] https://pl.wikipedia.org/wiki/MMIO

MMIO (memory-mapped input/output)




```
      0.0000 M        0.0039 M  Reserved
      0.0039 M        0.6133 M  System RAM
      0.6133 M        0.6250 M  Reserved
      0.6250 M        0.7500 M  PCI Bus 0000:00
      0.7500 M        0.8076 M  Video ROM
      0.8125 M        0.8281 M  pnp 00:00
      0.8281 M        0.8437 M  pnp 00:00
      0.8438 M        0.8594 M  pnp 00:00
      0.8594 M        0.8750 M  pnp 00:00
      0.8750 M        1.0000 M  Reserved
      0.9375 M        1.0000 M  System ROM
      1.0000 M     2785.2656 M  System RAM
    352.0000 M      360.6260 M  Kernel code
    360.6260 M      368.3481 M  Kernel data
    370.6484 M      372.5781 M  Kernel bss
   2785.2656 M     3000.4531 M  Reserved
   3000.4531 M     3000.7656 M  ACPI Non-volatile Storage
   3000.7656 M     3020.4648 M  Reserved
   3020.4648 M     3022.4961 M  ACPI Non-volatile Storage
   3022.4961 M     3022.9961 M  ACPI Tables
   ...
   4077.2500 M     4077.2969 M  PCI Bus 0000:00
   4077.2500 M     4077.2695 M  TPM
   4077.2695 M     4077.2969 M  pnp 00:01
   4078.0000 M     4078.0039 M  Local APIC
   4078.0000 M     4078.0039 M  Reserved
   4092.0000 M     4096.0000 M  Reserved
```

Aktywne czekanie

Aktywne czekanie (*busy waiting*) polega na ciągłym sprawdzaniu stanu zasobów (np. rejestrów, pamięci) w celu ustalenia, czy są one dostępne do dalszej pracy.

1. sprawdź czy drukarka jest gotowa na przyjęcie następnego znaku
2. jeśli nie jest gotowa, to idź do punktu 1
3. jeśli drukarka jest gotowa (po wydrukowaniu znaku), to sprawdź czy jest do wydrukowania nowy znak
4. jeśli jest nowy znak, to idź do punktu 1
5. jeśli nie ma więcej znaków, to drukowanie zostało zakończone

Odpytywanie (*polling*)

Procesor regularnie sprawdza stan kolejnych urządzeń (np. dysków, portów, itp.) w celu ustalenia, czy są one gotowe do wykonania operacji

1. wybierz kolejne urządzenie wymagające obsługi
2. sprawdź, czy to urządzenie wymaga obsługi
3. jeśli tak, to uruchom procedurę obsługi urządzenia
4. jeśli nie, to przejdź do punktu 1.

Obsługa przerwań

Przerwanie to żądanie wysyłane do procesora, aby przerwał aktualnie wykonywane zadanie i zajął się obsługą zdarzenia

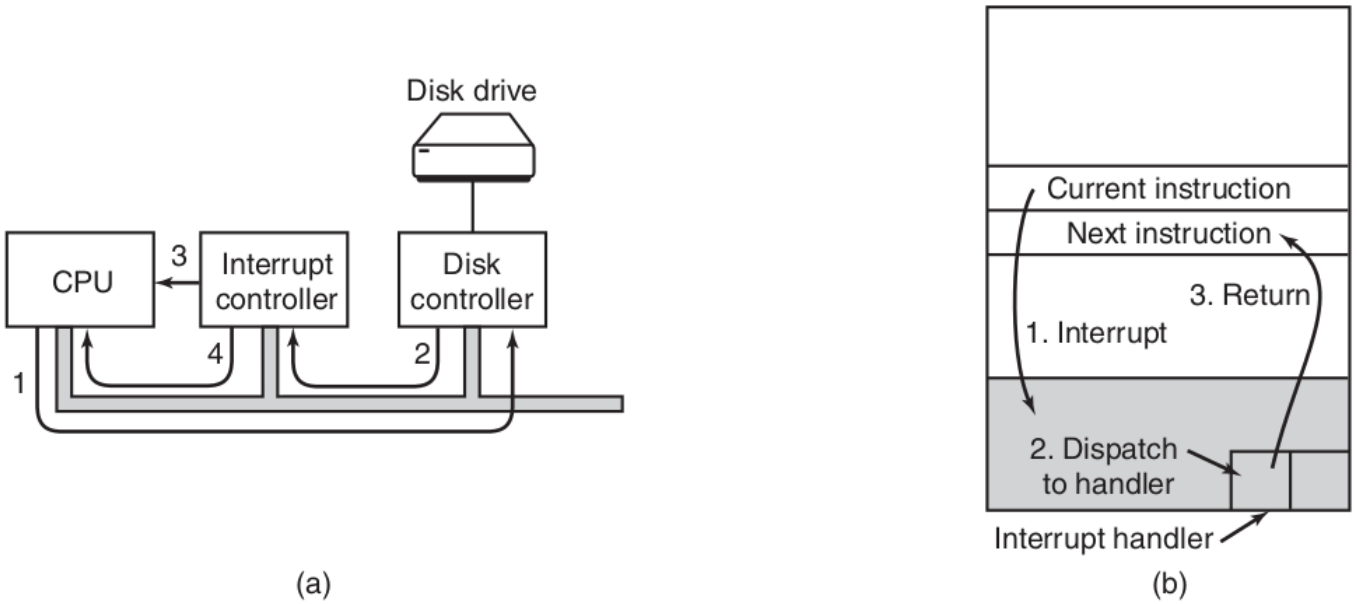


Figure 1-11. (a) The steps in starting an I/O device and getting an interrupt. (b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program.

Obsługa przerwań

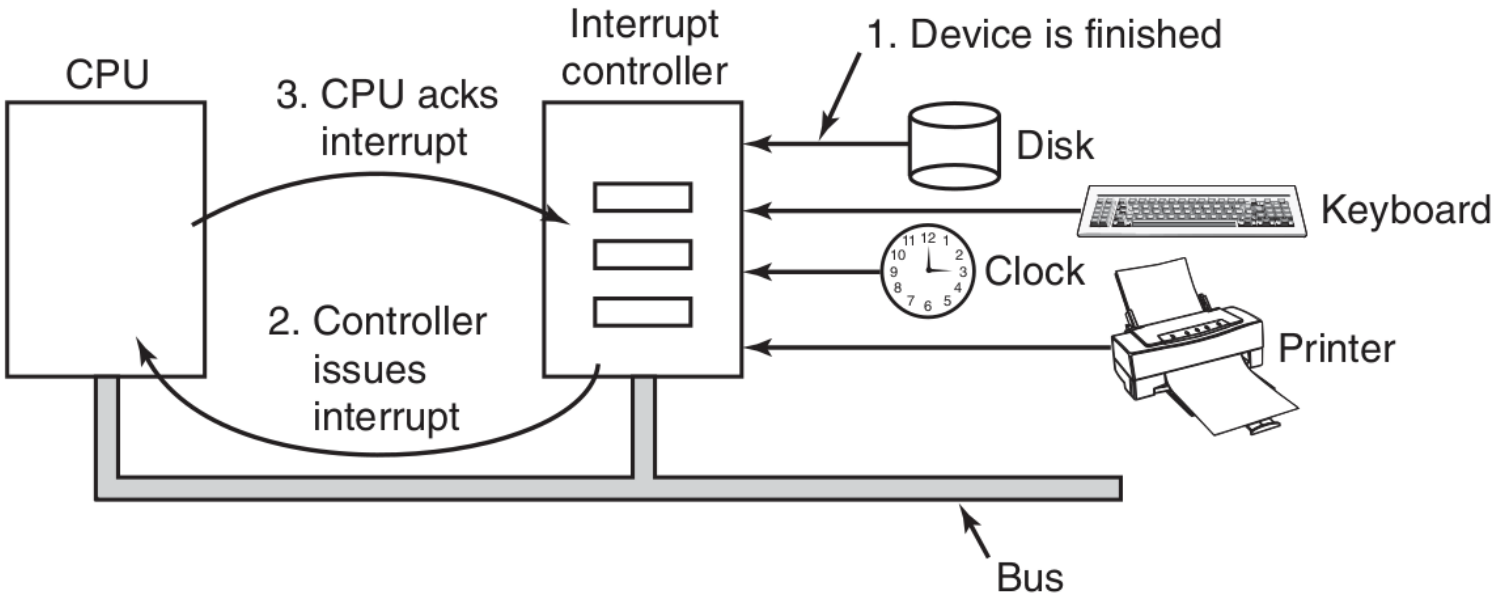


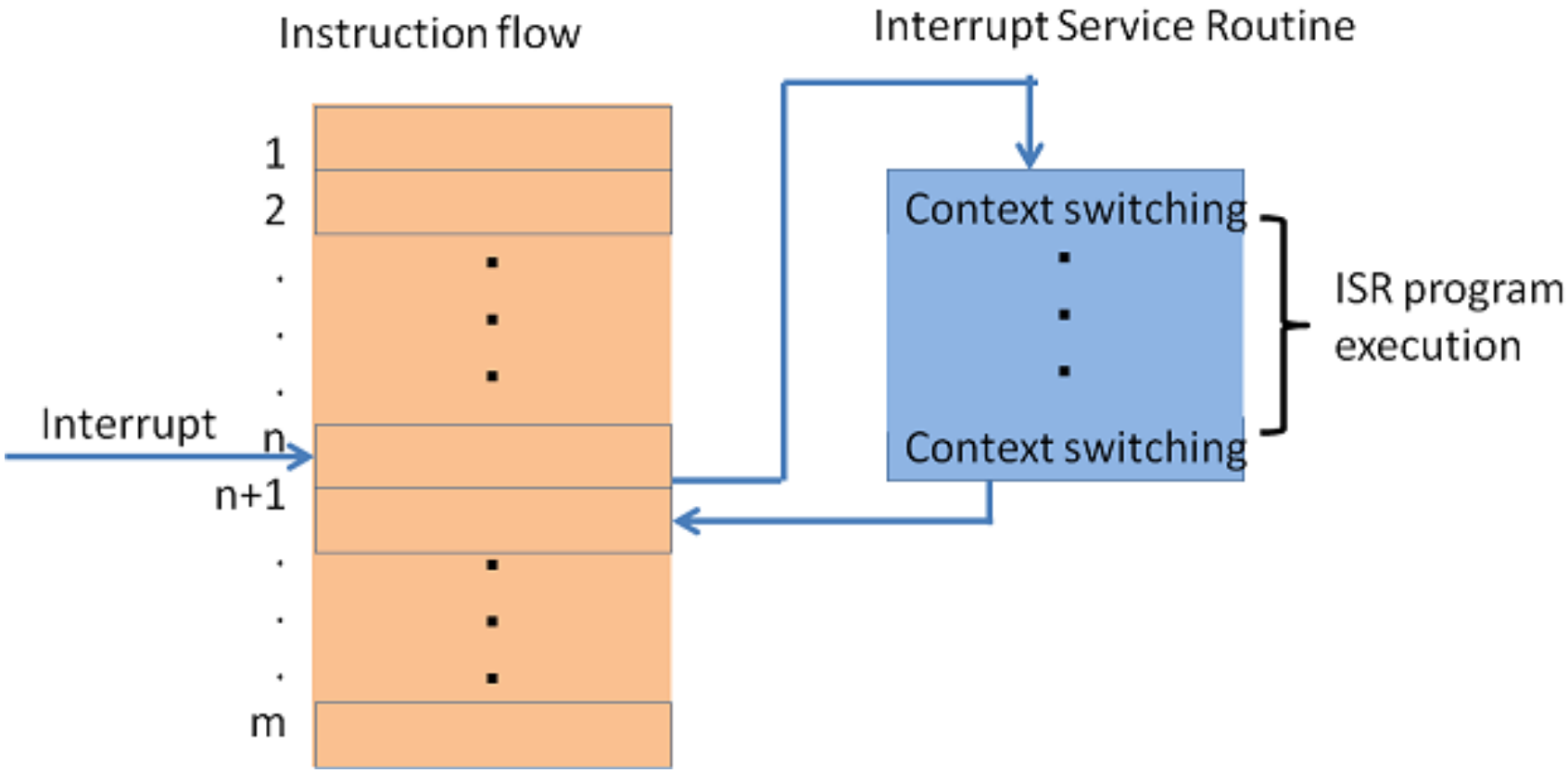
Figure 5-5. How an interrupt happens. The connections between the devices and the controller actually use interrupt lines on the bus rather than dedicated wires.

Przerwania: działanie

- jednostka centralna inicjuje przesyłanie danych przez wprowadzenie pewnych wartości do odpowiednich rejestrów sterownika urządzenia
- sterownik urządzenia rozpoczyna działanie (gromadzenie danych w buforze)
- sterownik urządzenia powiadamia procesor o zakończonej pracy generując określone przerwanie
- procesor wstrzymuje bieżącą pracę, odkłada na stos adres przerwanego rozkazu, określa źródło przerwania i przekazuje sterowanie do procedury obsługi przerwania *(interrupt handler, interrupt service routine (ISR))* wykorzystując wektor przerwań lub odpytywanie
- przesłanie danych z bufora do programu użytkownika
- wznowienie przerwanej pracy

Procedura obsługi przerwania jest częścią modułu obsługi urządzenia, czyli kodu jądra zarządzającego urządzeniem.

Procedura obsługi przerwania



Przerwania: obsługa

- przerwanie sprzętowe może zostać zainicjowane w dowolnym momencie
- procedura obsługi przerwania powinna zostać uruchomiona jak najszybciej i nie może byc przerwana
  - wymagania urządzenia: jak najszybsza obsługa przerwania
  - wymagania systemu operacyjnego: jak najkrótsza obsługa
  - co gdy w podczas obsługi przerwania wystąpi nastepne żądanie?

- możliwe zagnieżdżanie obsługi wywołań lub odraczanie wykonania
- w Linux obsługa przerwania podzielona na etapy:
  - górna połówka (*top half*) - niezbędne operacje
  - dolna połówka (*bottom half*) - odłożone do późniejszej obsługi

http://www.jonmasters.org/blog/2007/12/12/everything-you-know-about-interrupts-is-wrong/

Przerwania: podział

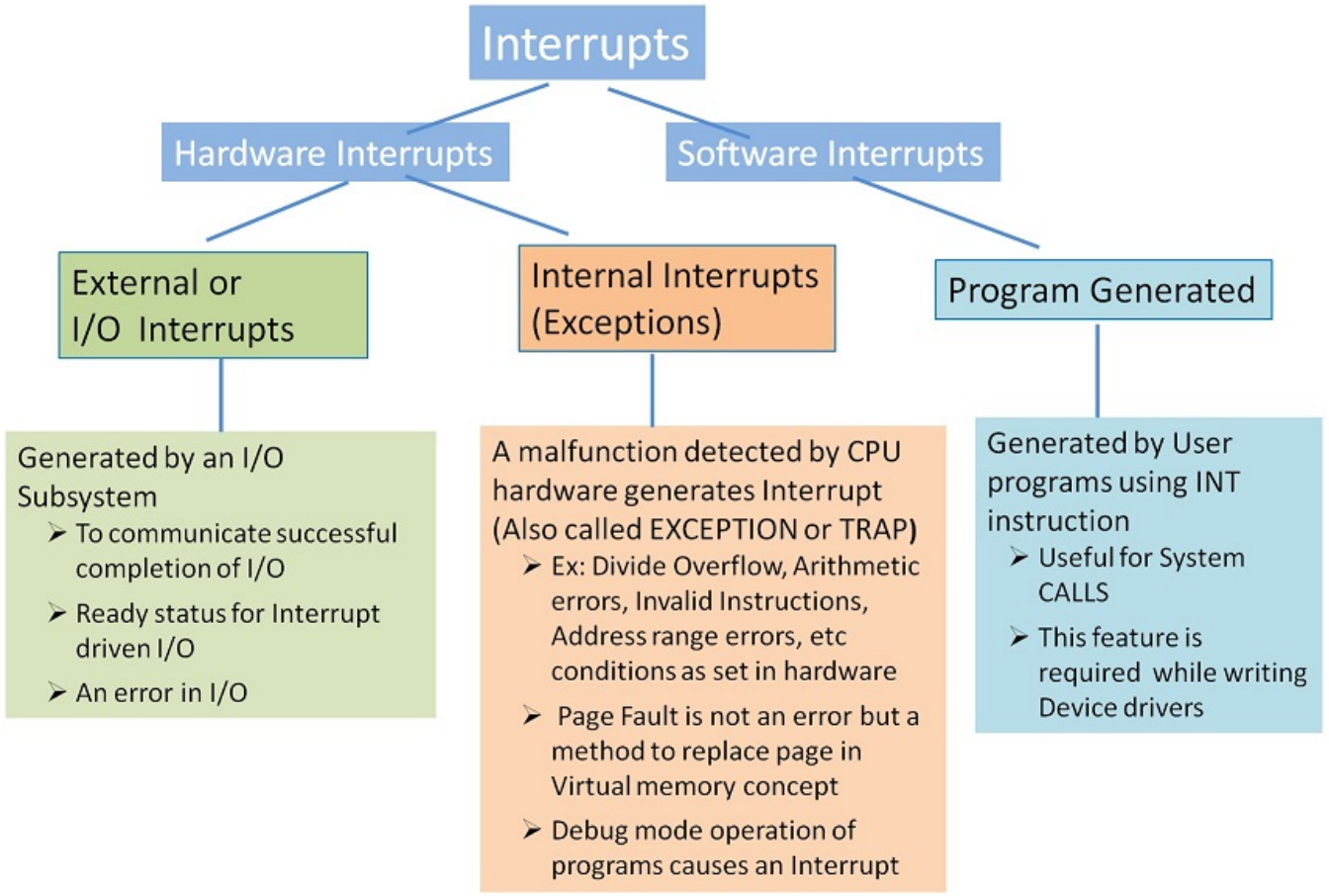
- **asynchroniczne** (przerwania) – generowane przez urządzenia sprzętowe w dowolnym czasie, niezależnie od sygnałów zegarowych procesora, np. przerwanie z karty sieciowej po odebraniu pakietu danych
- **synchroniczne** (wyjątki) – generowane w wyniku wykonania instrukcji, tworzone przez jednostkę sterowania procesora w celu obsłużenia sytuacji wynikającej z wykonania instrukcji, np. dzielenie przez zero, wywołania systemowe

Przerwania: źródła

Systemy z obsługą przerwań są kierowane zdarzeniami generowanymi przez

- przerwania sprzętowe (*interrupts*)
  - zegar - operacje wykonywane w równych odstępach czasu
  - urządzenia I/O - generowane przez sterownik urządzenia
  - awaria sprzętu
- przerwania wywołane instrukcjami programów (*exceptions*)
  - błędy programów (*faults*), nieprawidłowe operacje,
  - załamania (*aborts*)
  - pułapki (*traps*), przerwania programowe

Przerwania: źródła



Przerwania sprzętowe: podział wg dokumentacji Intela

- **maskowalne** – mogą być ignorowane
  - dwa stany przerwań: zamaskowany (ignorowany), nie zamaskowany
  - możemy tymczasowo odłożyć uruchomienie procedury obsługi przerwań, do czasu, gdy przerwanie zostanie ponownie włączone
  - wszystkie IRQ (*Interrupt ReQuests*) generowane przez urządzenia I/O powodują powstanie przerwań maskowalnych
  - sygnalizowane przez pin INT procesora.
- **niemaskowalne** – nie mogą być ignorowane
  - krytyczne zdarzenia (np. awaria sprzętu) zawsze rozpoznawane przez CPU
  - sygnalizowane przez pin NMI (*NonMaskable Interrupt*)

Wyjątki wykrywane przez procesor

Wyjątki generowane w sytuacji wystąpienia nieprawidłowości przy wykonywaniu instrukcji przez procesor

- **błąd** (*fault*) – po obsłużeniu wyjątku, procesor może wznowić wykonanie od instrukcji, która spowodowała błąd.
  np. błąd strony, dzielenie przez zero, naruszenie ochrony pamięci
- **pułapka** (*trap*) – zgłaszana po wykonaniu instrukcji pułapki. Po obsłużeniu, procesor wznawia wykonanie od nastepnej instrukcji.
  np. śledzenie programu podczas debugowania
- **załamanie** (*abort*) – wystąpił poważny błąd i jednostka sterowania nie może zachować adresu instrukcji; zwykle oznacza zakończenie procesu, który spowodował załamanie.
  np. krytyczny bład sprzętu.

Wyjątki programowe – występują na żądanie programisty (np. instrukcja `INT N`), są obsługiwane za pomocą pułapek
np. wywołanie funkcji systemowej, debugowanie programu

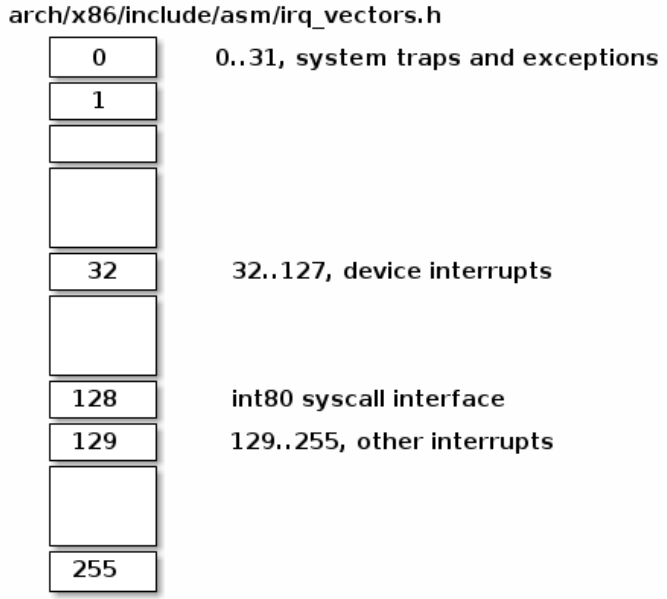
Przerwania: identyfikacja

Każde przerwanie lub wyjątek jest identyfikowane przez liczbę z zakresu od 0 do 255 (Intel nazywa tę 8-bitową liczbę bez znaku **wektorem**).

Tablica deskryptorów przerwań IDT

Interrupt descriptor table

- struktura używana w architekturach x86 do implementacji wektora przerwań
- odwzorowuje numer przerwania na adres funkcji obsługi
- tablica 256 wektorów po 8B, max. rozmiar 2048B
- może rezydować w dowolnym miejscu pamięci, pozyzję tablicy wskazuje rejestr procesora `IDTR`

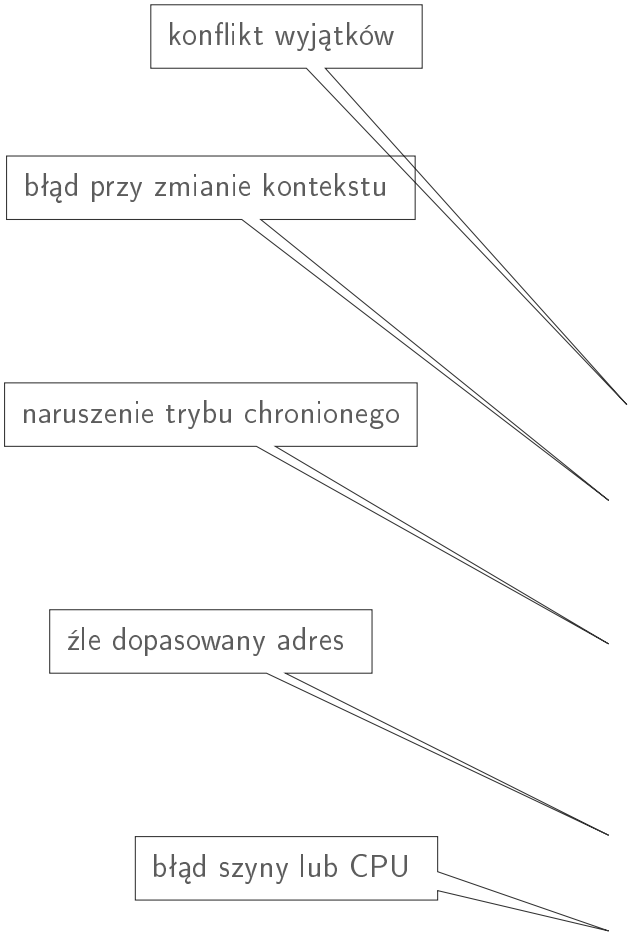


Tablica deskryptorów przerwań IDT

Rodzaje zdarzeń:

- wyjątki procesora 0-31, stałe mapowanie
- przerwania sprzętowe odwzorowują IRQ sprzętu zależnie od kontrolera przerwań 32-127
- przerwania programowe 128-255 definiowane przez BIOS i system operacyjny. Wzbudzane instrukcją asemblera `INT X` przez programy, moduły obsługi urządzeń lub inne funkcje obsługi przerwań.
- w systemie Linux wektor 128 (`0x80`) wykorzystywany do wywołań funkcji systemowych

Wyjątki procesora x86



#	Wyjątek	Rodzaj	Sygnał
0	*Divide error*	fault	SIGFPE
1	*Debug*	fault or trap	SIGTRAP
2	NMI	interrupt	
3	*Breakpoint*	trap	SIGTRAP
4	*Overflow*	trap	SIGSEGV
5	*Bounds check*	fault	SIGSEGV
6	*Invalid opcode*	fault	SIGILL
7	*Device not available*	fault	
8	*Double fault*	abort	
9	*Coprocessor segment overrun*	abort	SIGFPE
10	*Invalid TSS*	fault	SIGSEGV
11	*Segment not present*	fault	SIGBUS
12	*Stack exception*	fault	SIGBUS
13	*General protection*	fault	SIGSEGV
14	*Page fault*	fault	SIGSEGV
15	zarezerowana przez Intela		
16	*Floating point error*	fault	SIGBUS
17	*Alignment check*	fault	SIGSEGV
18	*Machine check*	abort	
19	*SIMD floating point exceptoin*	fault	SIGFPE

https://wiki.osdev.org/Exceptions

Wyjątki procesora x86

- jądro systemu dostarcza funkcje obsługi wyjątków (*exception handler*)
- wyjątki 20-31 - zarezerwowane do przyszłych zastosowań
- w UNIX/Linux funkcje obsługi wyjątku procesora (zazwyczaj) wysyłają sygnały do procesu powodującego wystapienie wyjątku

Przyporządkowanie IRQ do urządzeń I/O

IRQ	INT	urządzenie
0	32	zegar
1	33	klawiatura
2	34	kaskada PIC
3	35	drugi port szeregowy
4	36	pierwszy port szeregowy
5	37	karta dźwiękowa
6	38	stacja dysków
7	39	port równoległy
8	40	zegar systemowy
11	43	interfejs sieciowy
12	44	mysz PS/2
13	45	koprocesor matematyczny
14	46	pierwszy łańcuch sterownik dysków EIDE
15	47	drugi łańcuch sterownika dysków EIDE

- PIC *Programmable interrupt controller* dla jednostek jednoprocesorowych
- chip 8259A obsługuje 8 linii IRQ, w połaczeniu kaskadowym (pin 2) do 15 linii IRQ
- PIC połaczony do pinu INTR procesora

Sterownik przerwań PIC w systemach Intel PC

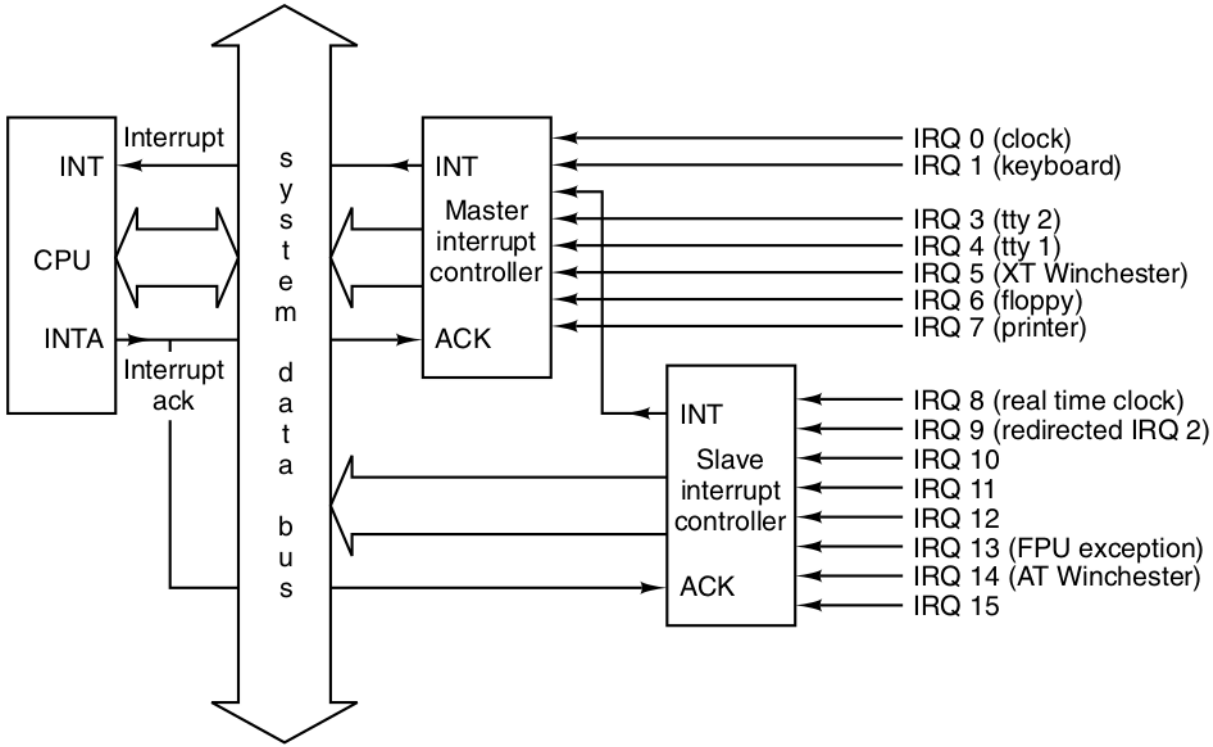


Figure 2-39. Interrupt processing hardware on a 32-bit Intel PC.

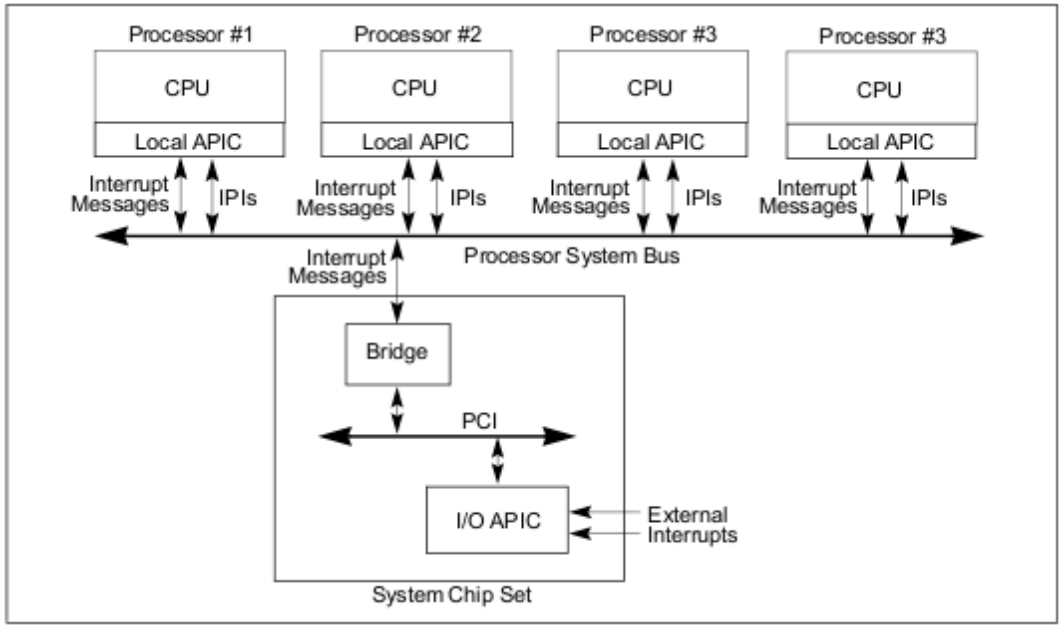
Sterownik przerwań

- linie IRQ łączą urządzenia z pinami wejściowymi programowalnego sterownika przerwań (*Programmable Interrupt Controller*, PIC)
- PIC monitoruje linie IRQ sprawdzając podnoszone sygnały na liniach
- PIC łaczy się z pinami INT i MNI procesora i udostępnia mu porty (rejestry) do komunikacji

Gdy urządzenie wymaga obsługi:

1. sterownik urządzenia generuje przerwanie podnosząc sygnał na linii IRQ (*Interrupt ReQuest*, żądanie przerwania)
2. PIC tłumaczy otrzymany IRQ na odpowiedni wektor
3. zachowuje wektor w porcie I/O sterownika przerwań, co pozwala na odczytanie go przez procesor za pomocą szyny danych
4. wysyła sygnał do pinu INT procesora (generuje przerwanie)
5. czeka, aż procesor potwierdzi otrzymanie sygnału przerwania (w tym czasie PIC nie generuje nowych przerwań)
6. CPU potwierdza przerwanie i uruchamia obsługę przerwania

APIC dla procesorów Intel Xeon w systemach SMP



- IPI – *Inter-Processor Interrupt* w systemie wieloprocesorowym (SMP) procesor może zażądać działania ze strony innego procesora (tzw. klepnięcie w ramię): synchronizacja pamięci cache, zatrzymanie systemu, itp.

APIC (*Advanced Programmable Interrupt Controller*)

- Local APIC osadzony wewnątrz każdego procesora
  - obsługuje przerwania lokalne, wyjątki procesora x86, wektor 0-31
  - generuje i akceptuje przerwania między-procesorowe (IPIs)
  - udostępnia wysokiej rozdzielczości zegar (mikrosekundy)
  - przekazanie do procesora przerwań zewnętrznych z I/O APIC
- I/O APIC w chipsecie płyty głownej, przy szynie systemowej
  - zbiera przerwania od urządzeń i przekierowuje do Local APIC
  - przerwania zewnętrzne IRQ są tłumaczone na komunikaty i wysyłane do LAPIC
  - standardowo 24 linie IRQ (0-23), programowalne rejestry
  - programowalna tabela przekierowań (*Interrupt Redirection Table*): mapowanie IRQ na wektor przerwań, priorytety, przypisanie docelowych procesorów

Przerwania sygnalizowane komunikatami (MSI)

Message Signaled Interrupts

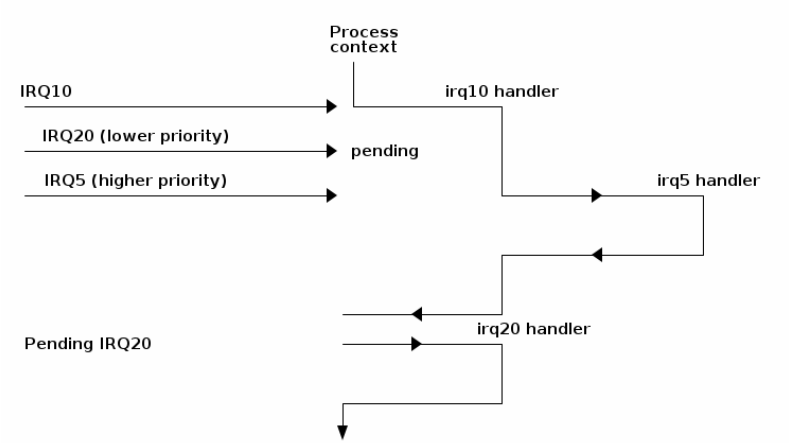
- mechanizm przerwań zdefiniowany w specyfikacji PCI 2.2 (MSI) i PCIe (MSI-X), obsługiwany przez APIC
- wykorzystuje szynę danych zamiast dedykowanych linii IRQ (PCIe nie posiada linii IRQ), obsługa do 2048 przerwań na urządzanie, każda funkcja urządzenia może mieć własne przerwanie
- przerwanie zgłaszane komunikatem poprzez zapis do adresów pamięci (MMIO) kontrolera przerwań Local APIC, komunikat nie niesie żadnych danych, jest małym pakietem identyfikującym źródło przerwania
- szybszy w działaniu (rzędy wielkości mniejsze opóźnienia niż w przypadku linii IRQ)
- lepsze wsparcie dla systemów wieloprocesorowych – możliwe kierowanie przerwań do określonych rdzeni CPU, co poprawia równoważenie obciążenia
- wymaga wsparcia systemu operacyjnego i sterowników urządzeń

Sterowanie przerwaniami

Synchronizacja dostępu do współdzielonych danych pomiędzy procedurą obsługi przerwań i innymi potencjalnymi współbieżnymi działaniami (np. inicjalizacja sterownika, przetwarzanie danych sterownika), często wymaga włączenia i wyłączenia przerwań w kontrolowany sposób:

- z poziomu urządzenia - przez zaprogramowanie rejestrów sterujących urządzenia
- z poziomu PIC - oprogramowanie PIC włącza/wyłącza daną linię IRQ
  - żądania na wyłączonej linii nie są gubione, PIC prześle je do CPU jak tylko linia zostanie włączona
  - umożliwia szeregowe przetwarzanie przerwań danego typu
- z poziomu procesora - np. instrukcje *cli* (*CLear Interrupt flag*), *sti* (*SeT Interrupt flag*) w architekturze x86 ustawiają maskowanie przerwania, wówczas przerwanie jest ignorowane (czasowo) przez procesor

Priorytety przerwań



- priorytety przerwań są wspierane przez większość architektur, choc ich obsługa jest problematyczna i rzadko wystepuje w systemach ogólnego przeznaczenia,
- priorytety są przydatne w systemach czasu rzeczywistego, gdzie wymagany jest determinizm czasowy - reagowanie systemu na zdarzenia w określonym czasie
- obsługa przerwania o wyższym priorytecie przerywa obsługę przerwania o niższym priorytecie (wywołanie wielokrotne)
- przerwania o niższym priorytecie są odłożone do późniejszego wykonania
- wyjątki procesora x86 mają wyższy priorytet niż przerwania I/O

Problemy operacji wej-wyj

Podczas programowej obsługi wej-wyj (aktywnie czekanie, odpytywanie) oraz w obsłudze sterowanej przerwaniami procesor jest czynnie zaangażowany w przesyłanie danych pomiędzy pamięcią a urządzeniami wej-wyj:

- szybkość transferu wej-wyj jest ograniczona szybkością, z jaką procesor może testować i obsługiwać urządzenie
- procesor jest zajęty zarządzaniem przesyłem danych z wejścia i na wyjście

Szybkie urządzenia wej-wyj oraz urządzenia przesyłające duże ilości danych wymagają bezpośredniego dostępu do pamięci (DMA, *Direct Memory Access*).

Bezpośredni dostęp do pamięci (DMA)

Bezpośredni dostęp do pamięci jest realizowany przez osobny kontroler na magistrali systemowej (DMAC, *DMA Controller*, część mostka południowego) lub bezpośrednio w kontrolerach urządzeń wej-wyj (np. karty sieciowe, dyski SSD).

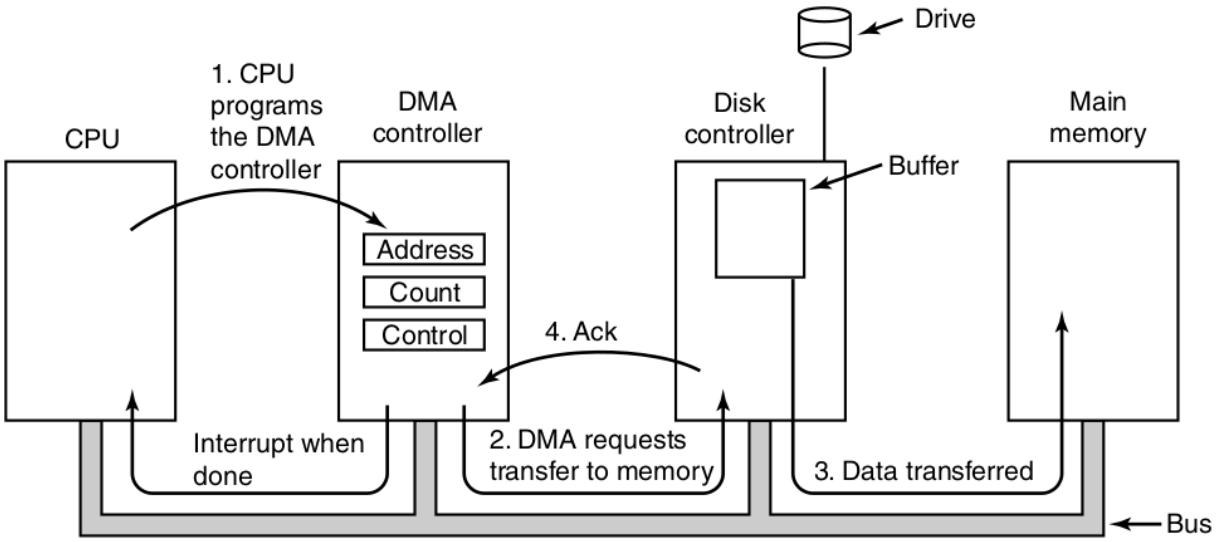


Figure 3-4. Operation of a DMA transfer.

Bezpośredni dostęp do pamięci (DMA)

Transfer danych wymaga przekazania przez procesor do modułu DMA rozkazu zawierającego informacje:

- czy wymagany jest odczyt/zapis
- adres urządzenia wej-wyj
- adres początkowej komórki pamięci z danymi (na dane)
- liczbę słów do przesłania

Procesor inicjuje przesłanie danych i kontynuuje przetwarzanie do momentu nadejścia przerwania od modułu DMAC.

DMA pozwala urządzeniom peryferyjnym omijać procesor i przesyłać dane bezpośrednio między pamięcią RAM a urządzeniem, co znacząco odciąża CPU i przyspiesza operacje wejścia/wyjścia.

Przyporządkowywanie przerwań

```
$ less /var/log/syslog

Sep 22 09: ... kernel: ttyS00 at 0x03f8 (irq = 4) is a 16550A
Sep 22 09: ... kernel: PCI: Found IRQ 5 for device 00:08.0
Sep 22 09: ... kernel: maestro: Configuring ESS Maestro 2E found at IO 0xD800 IRQ 5
Sep 22 09: ... kernel: parport0: irq 7 detected
Sep 22 09: ... kernel: lp0: using parport0 (polling).
Sep 22 09: ... kernel: ttyS04 at port 0x4880 (irq = 11) is a 16550A
Sep 22 22: ... kernel: eth0: Xircom Cardbus Adapter rev 3 at 0x4800, 00:10:A4:D2:52:55, IRQ 11.
Sep 22 09: ... kernel: ide0 at 0x1f0-0x1f7,0x3f6 on irq 14
Sep 22 09: ... kernel: ide1 at 0x170-0x177,0x376 on irq 15
```

Monitorowanie przerwań sprzętowych

```
$ uname -sr
Linux 2.6.18-92.1.10.el5 x86_64+

$ cat /proc/interrupts
           CPU0       CPU1
  0:  799260896          0    IO-APIC-edge  timer
  1:        140        100    IO-APIC-edge  i8042
  6:          5          0    IO-APIC-edge  floppy
  8:          0          0    IO-APIC-edge  rtc
  9:          0          0    IO-APIC-level  acpi
 12:        115          0    IO-APIC-edge  i8042
 15:    7173047       3915    IO-APIC-edge  ide1
169:          0          0    IO-APIC-level  ohci_hcd:usb1, ohci_hcd:usb2
177:     567795     125000    IO-APIC-level  ioc0
185:    6118456      40533    IO-APIC-level  eth0
NMI:       2770       1163
LOC:  799176280  799176233
ERR:          0
```

- edge - zgłoszenie na linii IRQ przez zmianę poziomu sygnału, krótkotrwałe zdarzenia (np. naciśnięcie klawisza, sygnał z timera)
- `level` - zgłoszenie na linii IRQ przez podniesienie poziomu sygnału, dłuższe zdarzenia wymagające potwierdzenia (np. transfer danych z dysku)

Monitorowanie przerwań sprzętowych

```
# uname -sr
Linux 6.9.3-76060903-generic

# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  1:      17024          0          0         12  IR-IO-APIC    1-edge      i8042
  8:          0          0          0          0  IR-IO-APIC    8-edge      rtc0
  9:      47343       2532          0          0  IR-IO-APIC    9-fasteoi   acpi
 12:     100109          0       2004          0  IR-IO-APIC   12-edge      i8042
 14:          0          0          0          0  IR-IO-APIC   14-fasteoi   INT34BB:00
 16:          0          0          0          0  IR-IO-APIC   16-fasteoi   idma64.0, i2c_designware.0, i801_smbus
120:          0          0          0          0  DMAR-MSI      0-edge      dmar0
121:          0          0          0          0  DMAR-MSI      1-edge      dmar1
122:          0          0          0          0  IR-PCI-MSI-0000:00:1c.0    0-edge      PCIe PME
...
164:      11391          0          0          0  IR-PCI-MSIX-0000:3d:00.0    1-edge      nvme0q1
...
NMI:          0          0          0          0   Non-maskable interrupts
LOC:   49586732   34698221   31900639   31100941   Local timer interrupts
SPU:          0          0          0          0   Spurious interrupts
PMI:          0          0          0          0   Performance monitoring interrupts
IWI:    7828372     588193     485311     404572   IRQ work interrupts
RTR:          0          0          0          0   APIC ICR read retries
RES:     451370     469078     472718     500172   Rescheduling interrupts
CAL:    9657155    6803847    6046705    5690671   Function call interrupts
TLB:    3807589    4242390    4227898    4149771   TLB shootdowns
TRM:         16         16         16         16   Thermal event interrupts
THR:          0          0          0          0   Threshold APIC interrupts
DFR:          0          0          0          0   Deferred Error APIC interrupts
MCE:          0          0          0          0   Machine check exceptions
MCP:        356        349        349        349   Machine check polls
ERR:          0
MIS:          0
```

Przerwania w Linux: górna i dolna połówka

- górna połówka (*top half*)
  - uruchamiany bezpośrednio po odbiorze sygnału przerwania
  - realizuje tylko te czynności, które mają kluczowe znaczenie dla funkcjonowania urządzenia (przyjęcie przerwania, restart urządzenia)
  - uruchomienie funkcji obsługi przerwań urządzeń
  - zlecenie wykoanania odroczonej akcji (dolnej połówki)
  - w tym czasie blokowana jest obsługa pozostałych przerwań
  - wykonywanie zbyt długich operacji podczas blokowania przerwań powoduje opóźnienia, problemy z wydajnością i gubienie przerwań
- dolna połówka (*bottom half*)
  - obejmuje czynności obsługi, które mogą być odłożone w czasie i wykonane w momencie mniejszego obciazenia systemu
  - przerwania nie są blokowane w czasie obsługi

Odroczone akcje w Linux

softIRQ

- działają w kotekście jądra, zlecane przez funkcję obsługi przerwania (dolna połówka)
- przydzielane statycznie, użycie ograniczone do wybranych podsystemów z wymogami małych opóźnień i wysokiej częstotliwości
- obsługa może być zlecana równolegle na wielu CPU

tasklet - szczególny rodzaj softIRQ

- mogą być dynamicznie alokowane (np. podczasa ładowania modułu)
- ogsługa szeregowa, pojedyncza akcja może być wykonana na jednym procesorze

workqueues

- zadania odroczone wykonywane w kontekście procesu
- w odróżnieniu od softIRQ i taskletów mogą być blokowane

softirq - statystyki przerwań

```
$ uname -sr
Linux 6.9.3-76060903-generic

$ cat /proc/softirq
                    CPU0       CPU1       CPU2       CPU3
          HI:   23555600    3297692    3334936    3477097
       TIMER:    8655197    3098823    2571420    2336278
      NET_TX:         51         10          8         14
      NET_RX:    3089136      33591      32130      78008
       BLOCK:        367        472        921        324
    IRQ_POLL:         45          0          2          4
     TASKLET:     268745       2291       2068       2142
       SCHED:   21443427   12885249   11074332   10525505
     HRTIMER:       2452         13         10          6
         RCU:    7329331    6460572    6142318    6003988
```

`HI` - najwyższy priorytet
`TIMER` - przerwania zegarowe
`NET_TX` - wysyłanie pakietów sieciowych
`NET_RX` - odbiór pakietów sieciowych
`BLOCK` - operacje blokowe (dyskowe)
`IRQ_POLL` - obsługa przerwań przez odpytywanie
`TASKLET` - tasklety
`SCHED` - wiązane z planistą procesów (np. bilansowanie obciążenia)
`HRTIMER` - wysokorozdzielcze timery
`RCU` - mechanizm synchronizacji jądra Read-Copy Update

Monitorowanie przerwań sprzętowych: klawiatura

Działanie: naciśnięcie i przytrzymanie pojedynczego klawisza

```
root@nscobie ~$ sar -I 1 1
Linux 5.14.16-101.fc33.x86_64 (nscobie)         15.11.2021      _x86_64_        (8 CPU)

09:38:15          INTR    intr/s
09:38:16             1      1,00
09:38:17             1      0,00
09:38:18             1      1,00
09:38:19             1     24,00
09:38:20             1     30,00
09:38:21             1     29,00
09:38:22             1     29,00
09:38:23             1     29,00
09:38:24             1     29,00
09:38:25             1     30,00
09:38:26             1     29,00
09:38:27             1     29,00
09:38:28             1      3,00
^C
Średnia:             1     20,23
```

Monitorowanie przerwań sprzętowych: myszka/gładzik

Działanie: szybkie ruchy myszką

```
root@nscobie ~$ sar -I 12 1
Linux 5.14.16-101.fc33.x86_64 (nscobie)         15.11.2021      _x86_64_        (8 CPU)

09:45:56         INTR    intr/s
09:45:57           12      0,00
09:45:58           12     71,00
09:45:59           12    469,00
09:46:00           12    471,00
09:46:01           12    471,00
09:46:02           12    469,00
09:46:03           12    472,00
09:46:04           12    469,00
09:46:05           12    473,00
09:46:06           12    349,00
09:46:07           12    356,00
09:46:08           12    472,00
09:46:09           12    102,00
09:46:10           12      0,00
^C
Średnia:           12    273,02
```

Monitorowanie przerwań sprzętowych: wlan0

Działanie: `ping -i 0.002 -c 1000 192.168.1.1`

```
$ sar -I 174 1
Linux 6.9.3-76060903-generic (kis)  01.11.2024  _x86_64_ (8 CPU)

22:44:25         INTR    intr/s
22:44:26          174      8,00
22:44:27          174      6,00
22:44:28          174     67,00
22:44:29          174    980,00
22:44:30          174   1845,00
22:44:31          174    674,00
22:44:32          174     16,00
22:44:33          174      6,00
22:44:34          174     24,00
22:44:35          174     11,00
^C

Średnia:          174    363,70
```

Wieloprgramowość i ochrona sprzętowa

- w systemach wieloprogramowych i z podziałem czasu operacje wej-wyj nakładają się na działanie jednostki centralnej (asynchroniczne wej-wyj).
- zarządzanie wieloma programami wymaga odpowiednich mechanizmów ochrony, aby zapewnić bezpieczeństwo i stabilność systemu operacyjnego
- rolą jądra systemu operacyjnego jest nadzór nad zadobami i udostępnienie interfejsu dostępu do sprzętu
- system operacyjny i jądro systemu są programami, więc potrzebny mechanizm odróżniający kod wykonywany przez użytkonika od kodu jądra systemu

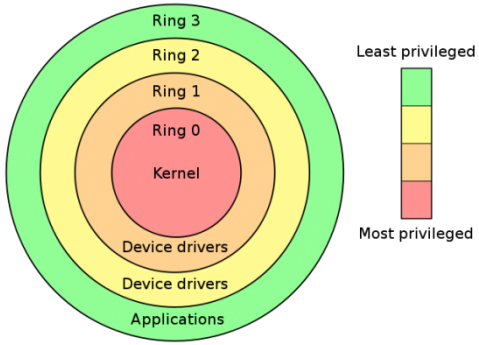
Ochrona sprzętowa

- generowanie przerwania od czasomierza (co kwant czasu) [11], poprawne działenie jest kluczowe dla stabilności działania systemu
- ochrona pamięci realizowana przez sprzęt (obecnie stronicowanie, dawniej adres bazowy i graniczny), zapewnia izolację pamięci między procesami
- ochrona wektora przerwań, procedur wej-wyj, które umożliwiają bezpieczne i kontrolowane przerywanie działania procesora w celu obsługi zdarzeń zewnętrznych
- wszelkie rozkazy wej-wyj są uprzywilejowane, tylko system operacyjny może komunikować się z urządzeniami,
- tryby pracy procesora: tryb użytkonika, tryb jądra (pierścienie uprzywilejowania)

[11] W jądrach >= 2.6 pojawia się możliwość zastosowania tzw. *dynamic ticks*, tzn. że przerwania zegarowe są generowane w razie potrzeby zarówno wtedy, kiedy system jest zajęty, jak i wtedy kiedy pracuje w trybie "idle"(CPU idle state).

Pierścienie uprzywilejowania

Mechanizm sprzętowy (zaimplementowany w procesorze) służący do izolacji i kontroli dostępu do zasobów systemu.



Cztery pierścienie (*Ring* 0-3) w architekturze x86:

- pierścień 3 - najniższe uprawnienia, dostępny dla użytkownika
- pierścień 1, 2 - rzadko używany, czasem dla modułów urządzeń wej-wyj
- pierścień 0 - najwyższe uprawnienia, uprzywilejowane opeacje, tylko jądro systemu, niedostępny bezpośrednio dla programistów i użytkowników

http://en.wikipedia.org/wiki/Ring_(computer_security)

Dualny tryb pracy

- system operacyjny musi gwarantować, że niepoprawny program nie będzie mógł zakłócić działania innych programów
- niepoprawnie działający program musi generować przerwanie
- ochronie muszą podlegać wszelkie zasoby dzielone
- sprzęt pozwalający odróżnić dwa tryby pracy: **tryb użytkownika** oraz **tryb jądra** (monitora, nadzorcy, systemu, uprzywilejowany)
- każde przerwanie powoduje przejście systemu z trybu użytkownika do trybu jądra
- dualny tryb pracy jest uzupełniony listą uprzywilejowanych rozkazów maszynowych, które mogą być wykonywane tylko w trybie jądra

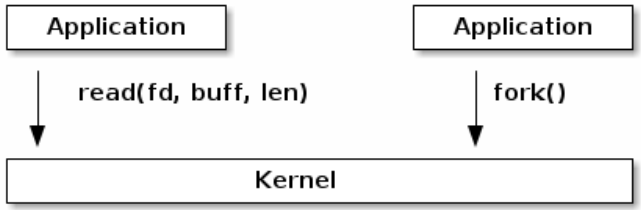
Dualny tryb pracy (cd)

- **tryb jądra (nadzorcy, monitora)** – w tym trybie wykonywane są rozkazy uprzywilejowane, np. rozkazy wejścia-wyjścia, zmieniające stan rejestrów zarządzających pamięcią i czasomierzem, rozkaz *halt*, rozkazy włączania/wyłączania przerwań
- **tryb użytkownika** – aplikacje użytkowe działają z ograniczonymi uprawnieniami, mogą zlecić systemowi wykonanie operacji zastrzerzonych (np. dostęp do zasobów) poprzez wywołania systemowe (*system calls*).

  Odwołanie do systemu jest traktowane przez sprzęt jak przerwanie programowe. Poprzez wektor przerwań sterowanie przekazywane jest do odpowiedniej procedury obsługi w systemie operacyjnym. .

Przerwania programowe i wywołania systemowe

Z punktu widzenia użytkownika/programisty wywołania systemowe są „usługami" oferowanymi przez jądro systemu aplikacjom użytkownika i przypominają API bibliotek, tj. są opisane jako wywołanie funkcji z nazwą, parametrami i wartością zwracaną[12]



Wywołania systemowe nie są w rzeczywistości wywołaniami funkcji, ale konkretnymi instrukcjami asemblera (specyficznymi dla architektury i jądra), które wykonują:

- identyfikują wywołanie systemowe i jego parametry
- przełaczają tryb wykonania z trybu użytkownika do trybu jądra
- odbierają rezultat wywołania systemowego

[12] API Linuksa jest zgodny ze standardem POSIX (*Portable Operating System Interface*).

- biblioteki systemowe udostępniają funkcje, które implementują wlaściwe wywołania systemowe, w ten sposób łatwo z nich korzystać w aplikacjach
- interfejs wywołań systemowych w Linuksie (większości Uniksów) stanowi częściowo biblioteka C (`glibc`)

Program:	wywołanie funkcji printf()
Biblioteka C:	printf() $\longrightarrow$ write()
Jądro:	wywołanie systemowe write()

- wywołania systemowe są niezbędne, gdyż programy działające w przestrzeni użytkownika nie mogą wykonać kodu jądra
- programy nie mogą bezpośrednio zainicjować wywołania funkcji operacji jądra, ponieważ jądro jest umieszczone i działa w chronionym obszarze pamięci operacyjnej.

Przerwania programowe i wywołania systemowe

- przerwanie programowe realizuje w procesorach rodziny x86 instrukcja `INT n`, gdzie `n` jest numerem wektora przerwania
- `INT $0x80` powoduje przełączenie systemu do trybu jądra i rozpoczęcie wykonywania wektora przerwania 128, który wskazuje na adres procedury obsługi wywołań systemowych `system_call()`
- konkretne wydołanie systemowe jest identyfikowane przez numer wywołania, który jest umieszczany w rejestrze `eax`
- rejestry (`ebx`, `ecx`, `edx`, `esi`, `edi`) są (zwykle) używane do przekazywania argumentów wywołania systemowego. Przed użyciem dokonywana jest dokładna weryfikacja przekazanych argumentów.

Przerwania programowe i wywołania systemowe

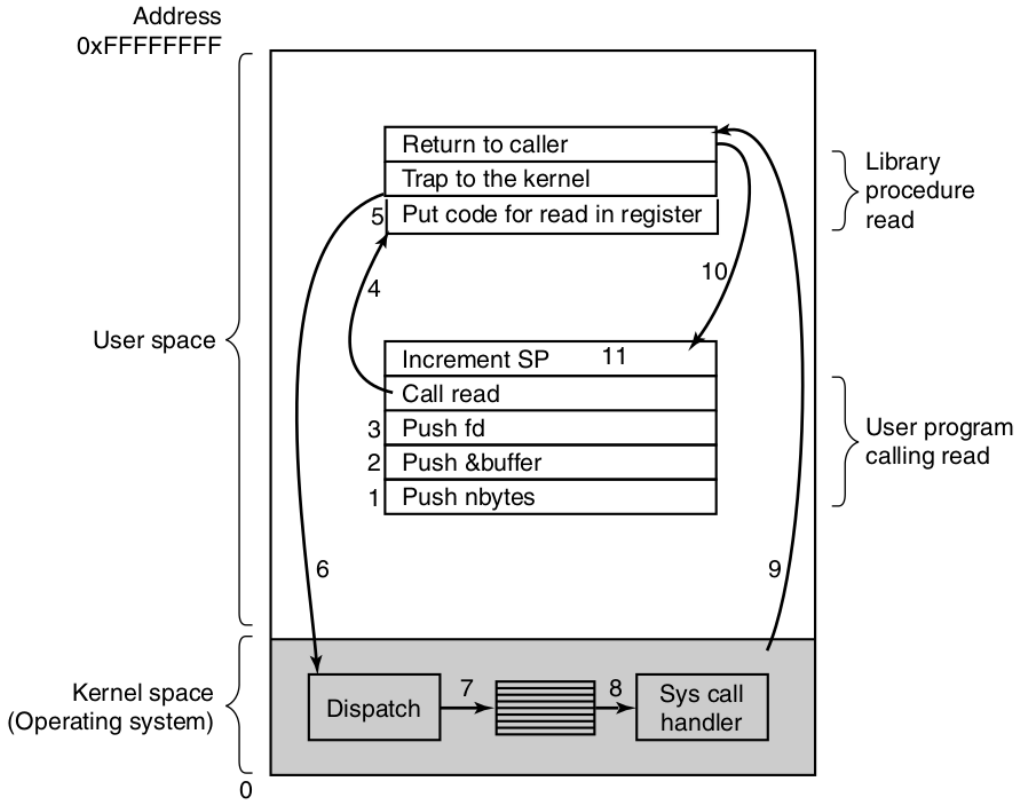


Figure 1-16. The 11 steps in making the system call read(fd, buffer, nbytes).

Przykład wywołania funkcji `read(fd, buffer, nbytes)`

Przerwania programowe i wywołania systemowe

Fragmenty pliku /usr/include/asm/unistd_64.h

Fedora 24

```
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
#define __NR_fstat 5
#define __NR_lstat 6
...
#define __NR_mmap 9
#define __NR_mprotect 10
#define __NR_munmap 11
#define __NR_brk 12
...
#define __NR_access 21
#define __NR_pipe 22
#define __NR_select 23
#define __NR_sched_yield 24
...
#define __NR_clone 56
#define __NR_fork 57
#define __NR_vfork 58
#define __NR_execve 59
#define __NR_exit 60
#define __NR_kill 62
#define __NR_uname 63
...
#define __NR_preadv2 327
#define __NR_pwritev2 328
```

Fedora 33

```
...
#define __NR_pkey_free 331
#define __NR_statx 332
#define __NR_io_pgetevents 333
#define __NR_rseq 334
...
#define __NR_pidfd_send_signal 424
#define __NR_pidfd_send_signal 424
#define __NR_io_uring_setup 425
#define __NR_io_uring_enter 426
...
#define __NR_mount_setattr 442
#define __NR_quotactl_fd 443
#define __NR_landlock_create_ruleset 444
#define __NR_landlock_add_rule 445
#define __NR_landlock_restrict_self 446
#define __NR_memfd_secret 447
```

Przerwania programowe: śledzenie

```
$ strace -n echo "Witaj Świecie"
[  59] execve("/usr/bin/echo", ["echo", "Witaj \305\232wiecie"], 0x7ffc6d7a05e0 /* 82 vars */) = 0
[  12] brk(NULL)                        = 0x636ecc694000
[   9] mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7d38fe8d6000
[  21] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (Nie ma takiego pliku ani katalogu)
[ 257] openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
[ 262] newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=147643, ...}, AT_EMPTY_PATH) = 0
[   9] mmap(NULL, 147643, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7d38fe8b1000
[   3] close(3)                         = 0
[ 257] openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
[   0] read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) = 832
...
[ 257] openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
[ 262] newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=18671040, ...}, AT_EMPTY_PATH) = 0
[   9] mmap(NULL, 18671040, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7d38fd400000
[   3] close(3)                         = 0
[ 262] newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
[   1] write(1, "Witaj \305\232wiecie\n", 15) = 15
[   3] close(1)                         = 0
[   3] close(2)                         = 0
[ 231] exit_group(0)                    = ?
[ 231] +++ exited with 0 +++
```

Przerwania programowe: śledzenie

```
$ strace -c cat /etc/hosts > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 35.84    0.000944         944         1           execve
 11.92    0.000314          28        11         6 openat
 11.20    0.000295          29        10           mmap
  6.42    0.000169          24         7           close
  5.58    0.000147          36         4           brk
  5.28    0.000139          23         6           fstat
  4.78    0.000126          31         4           mprotect
  4.71    0.000124          24         5           read
  3.83    0.000101          50         2         1 arch_prctl
  3.49    0.000092          46         2           munmap
  3.26    0.000086          21         4           pread64
  2.01    0.000053          53         1         1 access
  0.91    0.000024          24         1           write
  0.76    0.000020          20         1           fadvise64
------ ----------- ----------- --------- --------- ----------------
100.00    0.002634          44        59         8 total
```

Przerwania programowe: śledzenie

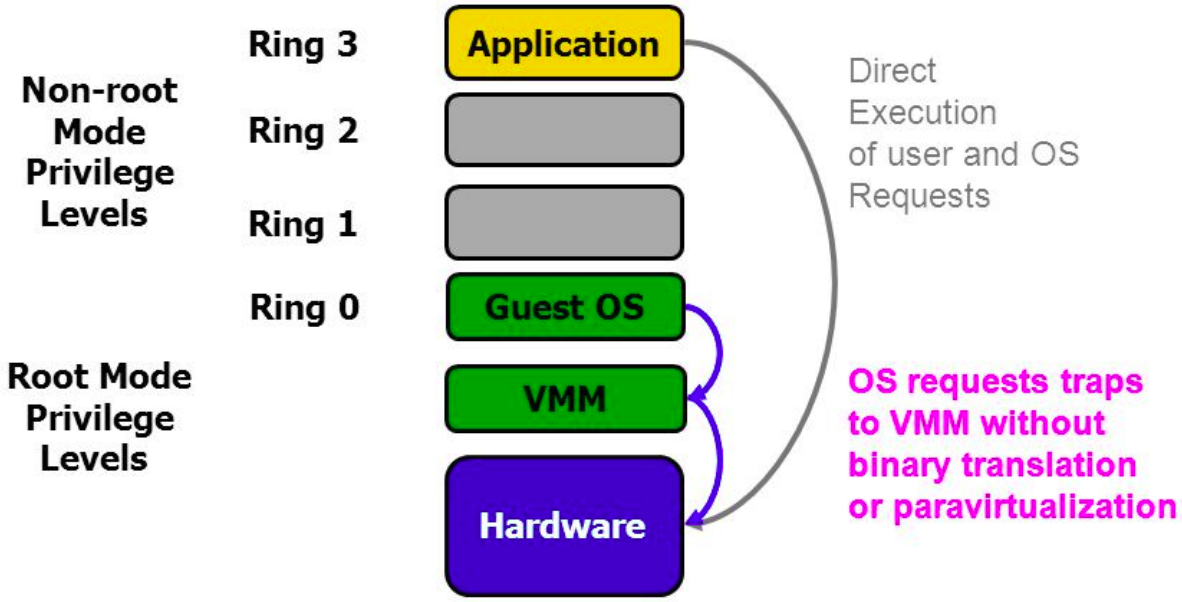
```
$ strace -c firefox
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ------------------
 26.32    0.364886          45      7963       505 futex
 13.59    0.188327          13     14104     12926 recvmsg
 11.21    0.155370          14     11038           poll
  7.90    0.109478          18      5802           write
  7.60    0.105350          18      5705         1 madvise
  5.28    0.073184        2152        34        16 wait4
  5.19    0.071983          12      5705         3 read
  4.38    0.060772          23      2565           mprotect
  3.09    0.042794          20      2041           mmap
  3.08    0.042744          41      1023           munmap
   ...
  0.00    0.000000           0         1           set_robust_list
  0.00    0.000000           0         1           inotify_init1
  0.00    0.000000           0         1           sched_setattr
  0.00    0.000000           0         1           sched_getattr
------ ----------- ----------- --------- --------- ------------------
100.00    1.386228          20     68379     14862 total
```

Czas spędzony w trybie jądra

```
$ /bin/time -p ping -i 0.002 -c 500  127.0.0.1 > /dev/null
real 0.99
user 0.19
sys 0.80
```

- `real` czas rzeczywisty
- `user` czas CPU spędzony w trybie urzytkownika
- `sys` czas CPU spędzony w trybie jądra

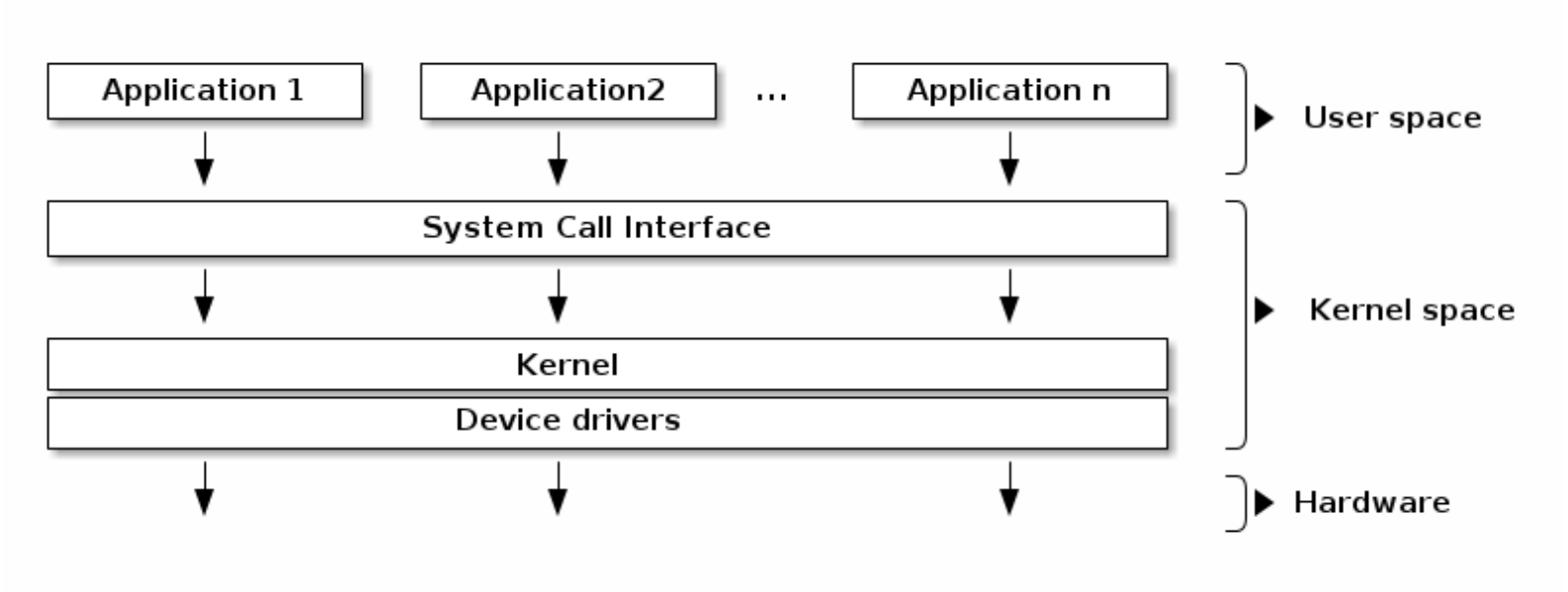
Pierścienie uprzywilejowania i wirtualizacja



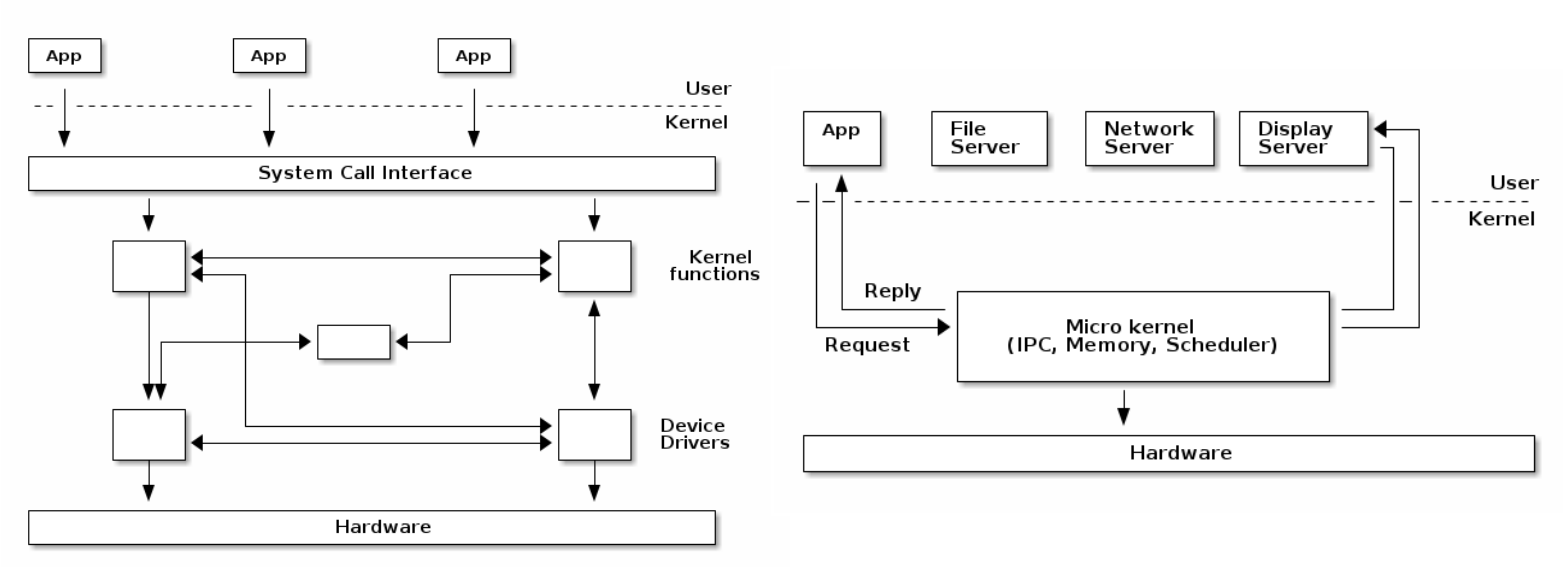
Wirtualizacja: Hypervisory (np. KVM) działają w Ring -1, aby kontrolować systemy gościnne w Ring 0.

http://slideplayer.com/slide/5084764/

Typowa architektura systemu

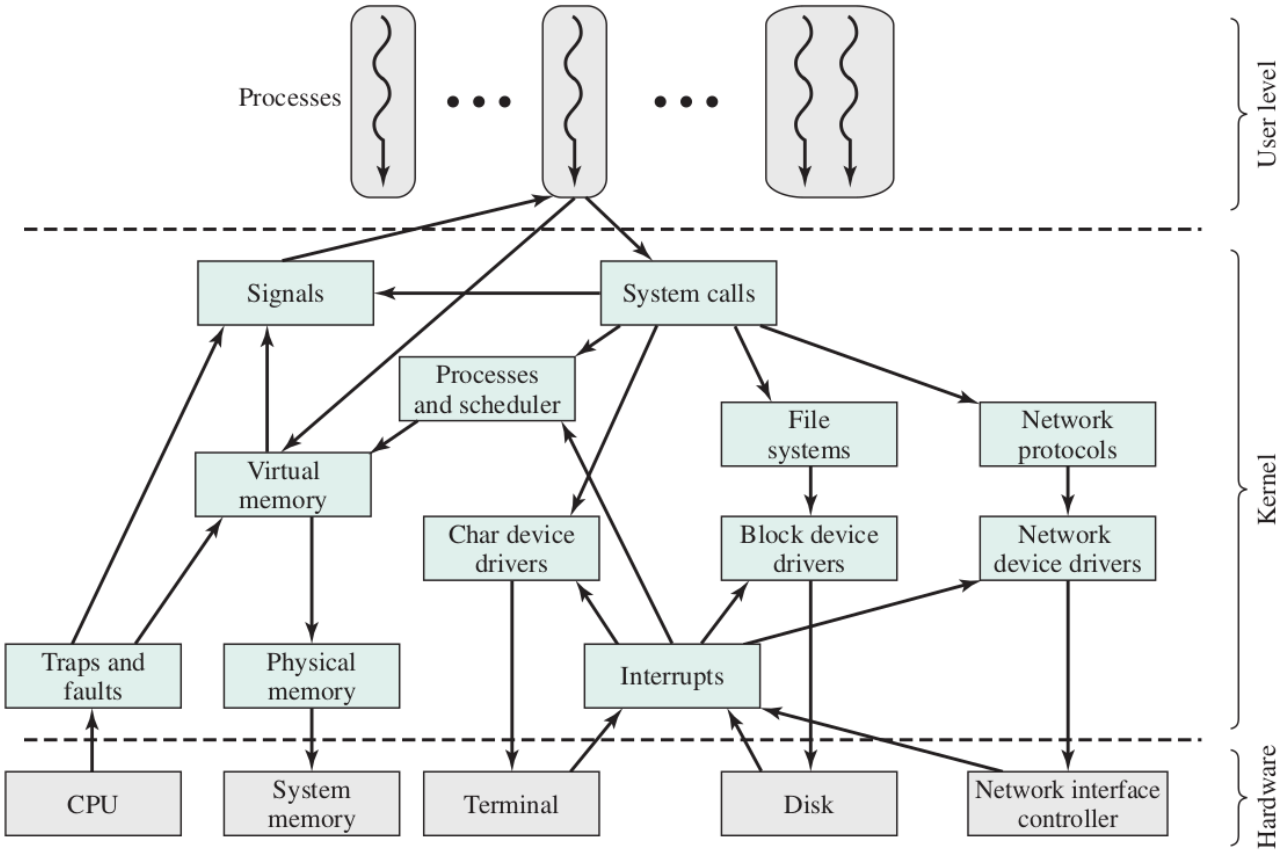


Monolityczne jądro vs. mikrojądro



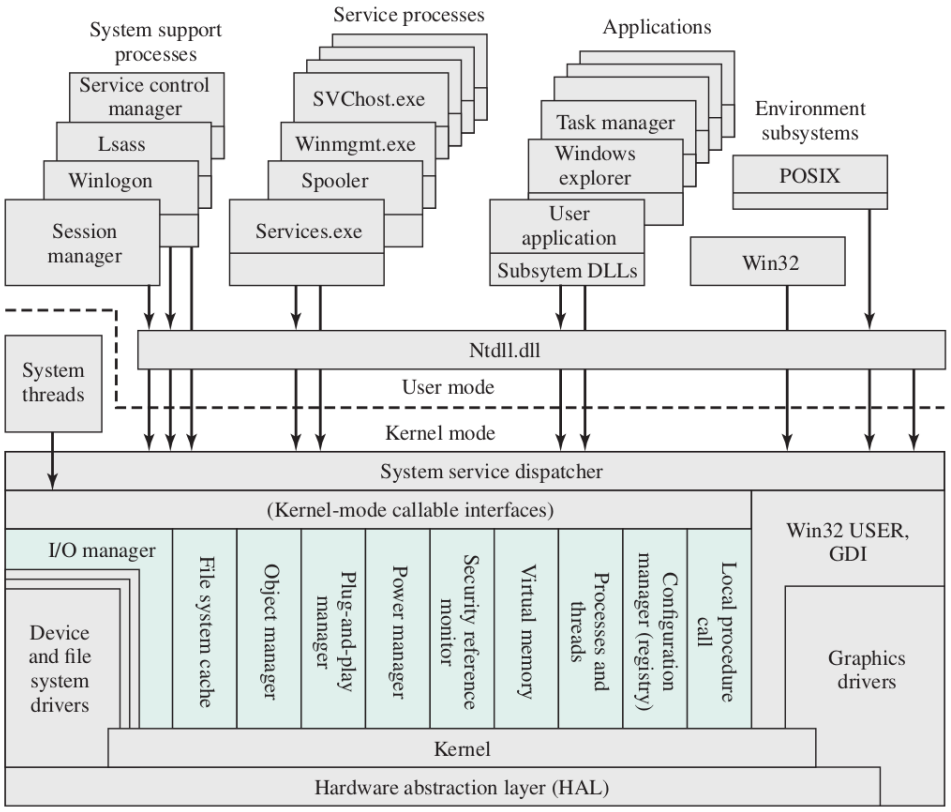
- **Jądro monolityczne** nie ma ochrony dostępu pomiędzy różnymi podsystemami jądra i w którym funkcje publiczne mogą być bezpośrednio wywoływane pomiędzy różnymi podsystemami (Linux)
- **Mikrojądro** - duże części jądra są przed sobą chronione i zwykle działają jako usługi w przestrzeni użytkownika. Ponieważ znaczna część jądra działa w trybie użytkownika, pozostały kod działający w trybie jądra jest znacznie mniejszy, stąd termin mikrojądro.
- jądra hybrydowe (Mac OS X, Windows) - jednak większość monolitycznych usług działa tu w trybie jądra, więc można kwalifikować je jako monolityczne
- Debata o wyższości mikrojądra nad monolitycznym jądrem Tanenbaum–Torvalds debate

Architektura jądra Linux



W. Stallings, *Operating Systems: Internals and Design Principles*, 9th Edition, Pearson, 2018.

Architektura jądra Windows



Egzekutor (*Executive*) - zbiór modułów jądra Windows (na rys. zaznaczony kolorem) dostsarczający API oprogramowaniu działającemu w trybie uzytkonika.

W. Stallings, *Operating Systems: Internals and Design Principles*, 9th Edition, Pearson, 2018.

Proces vs. program

Proces to program, który jest wykonywany

- jest dynamiczny, jest uruchomioną instancją programu
- jest aktywny, wykonuje się w pamięci zużywając zasoby systemowe
- posiada bieżący stan wykonywania, w tym licznik programu, stos, rejestry i inne zasoby systemowe; licznik programu wskazuje następną instrukcję do wykonania
- wykonanie procesu musi przebiegać w sposób sekwencyjny

Program to zestaw instrukcji, które mają na celu wykonanie określonego zadania

- jest statyczny, to plik (zbiór bitów) przechowywany na dysku
- jest bierny, nie wykonuje się
- nie jest procesem

Procesy: zadania systemu operacyjnego

System operacyjny odpowiada za wykonywanie następujących czynności:

- tworzenie i usuwanie procesów
- zarządzanie stanem procesów (wstrzymywanie i wznawianie procesów)
- dostarczanie mechanizmów komunikacji między procesami
- synchronizacja procesów, zapobieganie problemom takim jak zakleszczenia i wyścigi
- obsługa przerwań i sygnałów
- zapewnienie ochrony i bezpieczeństwa, izolacja procesów i zarządzanie uprawnieniami dostępu do zasobów
- monitorowanie działania procesów i dostarczanie narzędzi do diagnostyki
- zarządzanie przydziałem pamięci dla procesów, wirtualizacja pamięci i ochrona dostępu

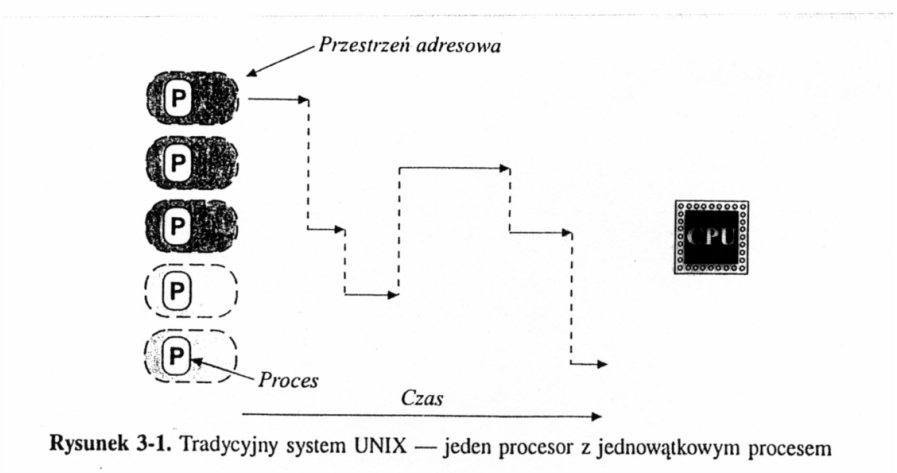
Procesy: rodzaje

Proces stanowi jednostkę pracy w systemie.

System składa się ze zbioru procesów:

- procesy systemu operacyjnego (wykonują kod systemu)
- procesy użytkowników (wykonują kod programów użytkowników)
- procesy działające w tle (demony, usługi) - wykonują zadania bez bezpośredniej interakcji z użytkownikiem
- procesy interaktywne (np. terminale, aplikacje GUI)
- procesy wsadowe (*batch*) - uruchamiane bez interakcji użytkownika, często jako zadania wsadowe wykonywane sekwencyjnie

Współbieżność (pseudo-paralelizm) - w jedno-procesorowym systemie z podziałem czasu w każdej chwili wykonuje się tylko jeden proces, ale z uwagi na przełączanie kontekstu powstaje wrażenie równoczesnej pracy wielu procesów.



Rysunek 3-1. Tradycyjny system UNIX — jeden procesor z jednowątkowym procesem

Przetwarzanie równoległe - jednoczesne wykonywanie wielu procesów na różnych procesorach, procesy są rzeczywiście wykonywane jednocześnie.

Tworzenie procesów

Zdarzenia powodujące powstawanie procesów

- inicjalizacja systemu
- wykonanie przez działający proces funkcji systemowej tworzącej procesy
  - Unix/Linux: `fork()`
  - Windows WinAPI: `CreateProcess()`
- żądanie użytkownika w systemie interaktywnym
- inicjacja zadania wsadowego w systemach wsadowych

Zakończenie procesu

Zdarzenia powodujące zakończenie działania procesów:

- normalne wyjście po zakończeniu pracy (dobrowolne)
  - Unix/Linux: `exit()`
  - Windows WinAPI: `ExitProcess()`
- wyjście zakończone błędem (dobrowolne), gdy proces wykryje sytuację uniemożliwiającą dalsze wykonie
- krytyczny błąd (niedobrowolne), w wyniku wykonania niepoprawnej instrukcji
- zabicie przez inny (uprawniony) proces (niedobrowolne), poprzez wywołanie funkcji systemowej
  - Unix/Linux: `kill()`
  - Windows WinAPI: `TerminateProcess()`

Hierarcha procesów

- proces, który tworzy nowy proces, nazywany jest procesem **macierzystym** (*parent*) lub **nadrzędnym**, a nowo utworzony proces to proces **potomny** (*child*)
- w UNIX/Linux procesy tworzą wyraźną hierarchię (drzewo procesów), pierwszy proces (korzeń) startuje podczas inicjacji systemu (np. `init`, `systemd`), a wszystkie pozostałe są jego potomkami bezpośrednimi lub pośrednimi
- proces wraz ze wszystkimi potomkami i przodkami tworzy **grupę procesów**, które wspólnie moga odbierać sygnały, choć każdy z nich może dostarczać indywidualną funkcję obsługi sygnału lub go zignorować
- nie wszystkie systemy stosują hierarchię s takim samym stopniu: w Windows procesy nie tworzą wyraźnej hierarchii. Proces nadrzędny uzyskuje **uchwyt** do potomka, za pomocą którego może go kontrolować ale uchwyt może być przekazany innemu procesowi

Hierarchia procesów

```
$ pstree
systemd-+-ModemManager---2*[{ModemManager}]
        |-NetworkManager---2*[{NetworkManager}]
        |-accounts-daemon---2*[{accounts-daemon}]
        |-acpid
        |-atop
        |-atopacctd
        |-avahi-daemon---avahi-daemon
        |-blueman-mechani---2*[{blueman-mechani}]
        |-bluetoothd
        |-cron
        |-gnome-keyring-d-+-ssh-agent
        |-python3
        |-sshd
        |-systemd-+-(sd-pam)
        |         `-gnome-terminal--+-2*[zsh]
        |                           |-zsh---pstree
        |                           `-3*[{gnome-terminal-}]
        `-systemd-journal

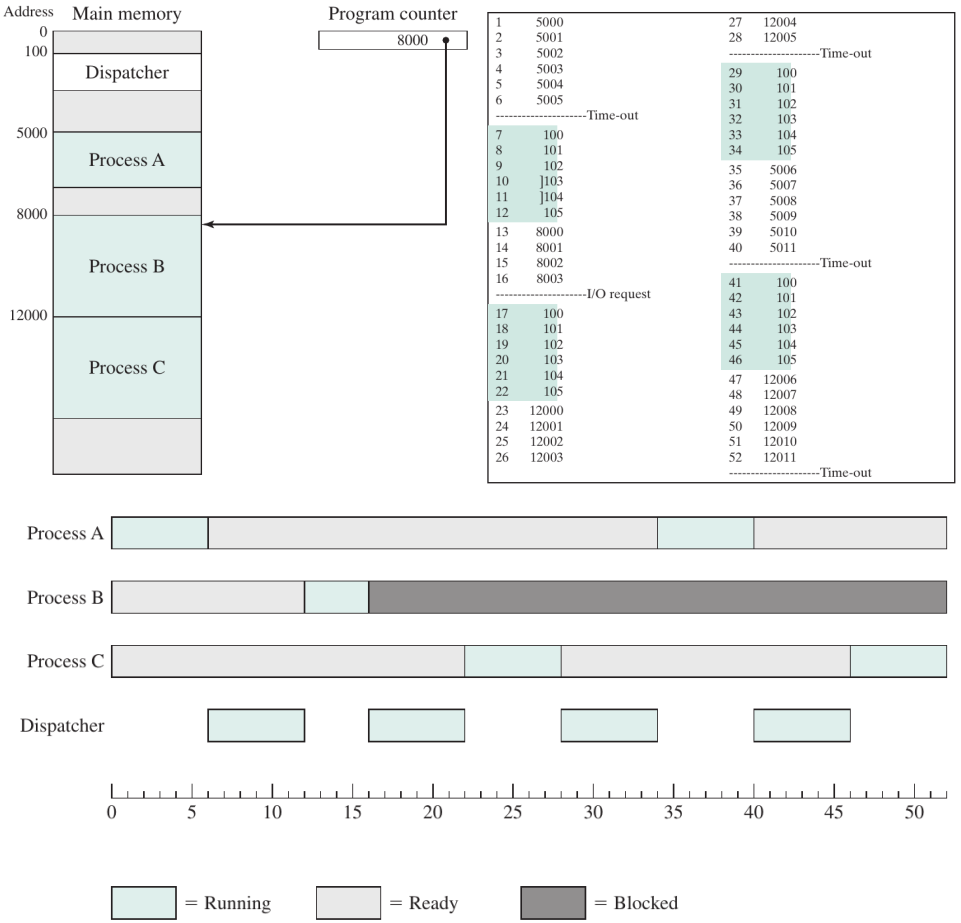
$ pstree -s $$
systemd---systemd---gnome-terminal----zsh---pstree
```

Procesy: stany

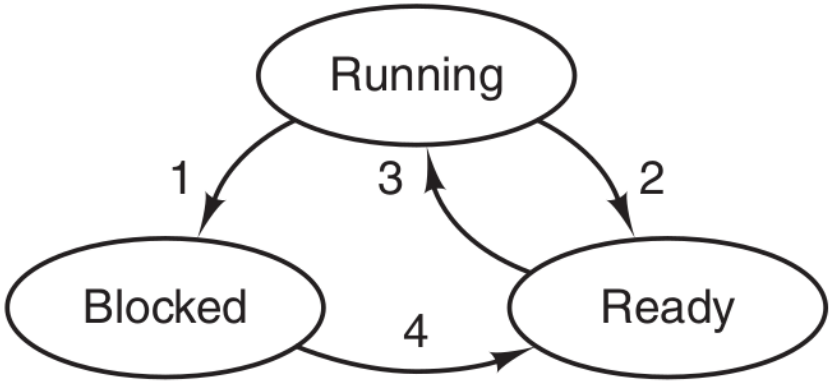
- **gotowy** (*ready*) – proces czeka na przydział procesora
- **bieżący** (*running*) – aktualnie używający CPU, są wykonywane instrukcje
- **zablokowany** (oczekujący, *blocked*) – czeka na wystąpienie jakiegoś zdarzenia (np. zakończenia operacji wejścia-wyjścia)



Przykładowy przebieg procesów w systemie

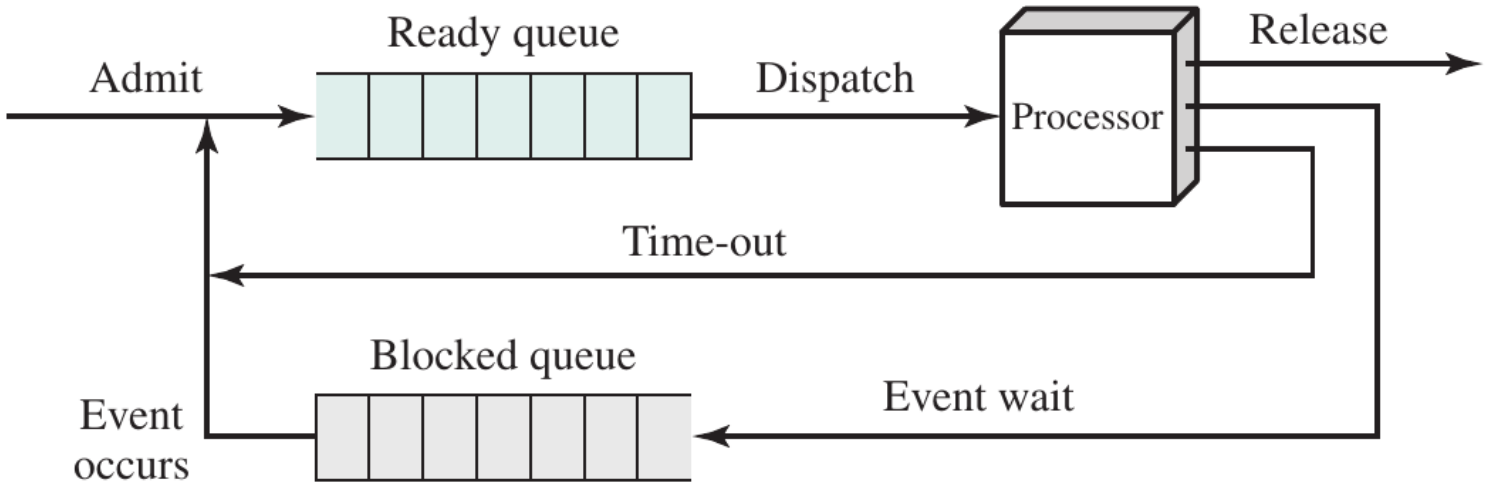


Procesy: zmiana stanu



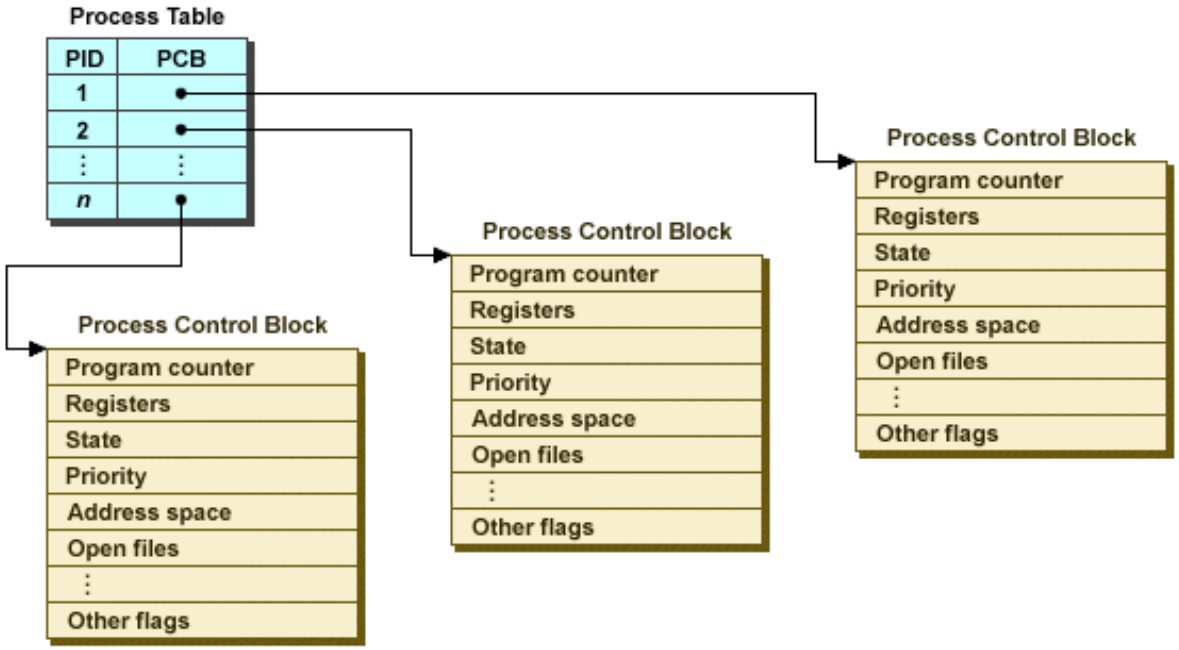
1. proces zablokowany w oczekiwaniu na zdarzenie (oczekiwanie wejścia)
2. planista wybiera inny proces do wykonania
3. planista wybiera ten proces do wykonania
4. zachodzi zdarzenie odblokowujące proces (wejście staje się dostępne)

Kolejka procesów w stanie gotowości i zablokowania



Procesy: zarządzanie

Tablica procesów - zarządzana przez system operacyjny tablica zawierająca **bloki kontrolne procesów**, po jednej strukturze na proces



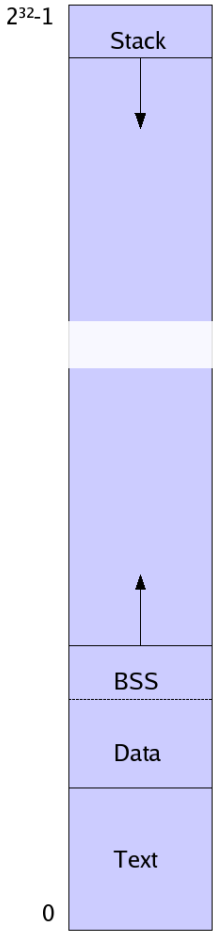
Blok kontrolny procesu zawiera wszystkie niezbędne informacje wymagane do zarządzania procesem, wszystko co musi być zapisane, gdy proces opuszcza stan bieżący, tak aby możliwe było wznowienie jego wykonania

- rejestry procesora, licznik programu, słowo stanu programu, wskaźnik stosu
- informacje o wykorzystanych zasobach (wykaz otwartych plików, ograniczenia pamięci, itp.)
- pozostałe informacje wymagane do zarzadzania procesem, planowania przydziału procesora, identyfikacji procesu, itp.

Typowe pola bloku kontrolnego procesu

zarządzanie procesem	zarządzanie pamięcią	zarządzanie plikami
rejestry	wskaznik do informacji segmentu tekstu	katalog głowny
licznik programu	wskaznik do informacji segmentu danych	katalog roboczy
słowo stanu programu	wskaznik do informacji segmentu stosu	deskryptowy plików
wskaznik stosu		identyfikator użytkownika
stan procesu		identyfikator grupy
priorytet		
parametry szeregowania		
identyfikator procesu		
identyfikator rodzica		
grupa procesów		
sygnały		
czas rozpoczęcia procesu		
wykorzystany czas CPU		
czas CPU procesów potomnych		

Procesy: przestrzeń adresowa



Z procesem związana jest określona wirtualna przestrzeń adresowa zawierająca segmenty:

- programu (*text*),
- danych zainicjowanych (*data*),
- danych niezainicjowanych (*bss*),
- sterty (*heap*),
- stosu (*stack*)

Przykład rozmiarów segmentów programu:

```
$ size /bin/ps
   text    data     bss     dec     hex filename
  71928     769  131967  204664   31f78 /bin/ps
```

Przykład: polecenie pmap

Komenda pmap (*process memory map*) wyświetla odwzorowanie pamięci procesu (plik /proc/PID/maps).

```
$ pmap -x $$

9719:   zsh
Adres                 KB     RSS  Brudne Tryb  Odwzorowanie
00005fff8366e000      92      92       0 r---- zsh
00005fff83685000     760     760       0 r-x-- zsh
00005fff83743000     108      64       0 r---- zsh
...
00007f398a600000     160     160       0 r---- libc.so.6
00007f398a628000    1620    1440       0 r-x-- libc.so.6
00007f398a7bd000     352     192       0 r---- libc.so.6
...
00007f398aaf6000       8       8       0 r-x--   [ anon ]
00007f398aaf8000       8       8       0 r---- ld-linux-x86-64.so.2
00007f398aafa000     168     168       0 r-x-- ld-linux-x86-64.so.2
00007ffd0a6b5000     356     356     356 rw---   [ stos ]
ffffffffff600000       4       0       0 --x--   [ anon ]
---------------- ------- ------- -------
razem kB           33256   12484    3884
```

Przykład: polecenie ps

Komenda ps (*process status*) pozwala obserwować i określać stan procesów w systemie.

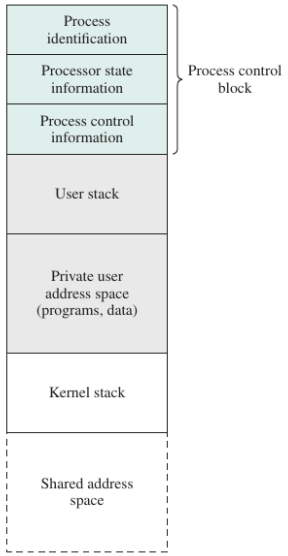
```
$ ps -elf
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root         1     0  0  80   0 - 55324 ep_pol Nov10 ?        00:00:15 /usr/lib/systemd/systemd
1 S root         2     0  0  80   0 -     0 kthrea Nov10 ?        00:00:00 [kthreadd]
1 S root         4     2  0  60 -20 -     0 worker Nov10 ?        00:00:00 [kworker/0:0H]
1 S root         6     2  0  60 -20 -     0 rescue Nov10 ?        00:00:00 [mm_percpu_wq]
...
4 S avahi      820     1  0  80   0 - 12035 poll_s Nov10 ?        00:00:12 avahi-daemon: running [sc
4 S root       822     1  0  80   0 - 98305 poll_s Nov10 ?        00:00:03 /usr/libexec/accounts-dae
4 S rtkit      823     1  0  81   1 - 45995 poll_s Nov10 ?        00:00:00 /usr/libexec/rtkit-daemon
4 S root       824     1  0  80   0 -  6329 hrtime Nov10 ?        00:00:00 /usr/sbin/smartd -n -q ne
...
0 S jkob      1987  1842  0  80   0 - 30643 poll_s Nov10 pts/2    00:00:00 /bin/bash
...
4 R root     32476 25560  0  80   0 - 36787 -      15:42 pts/22   00:00:00 ps -elf
```

S - stan procesu, UID - identyfikator użytkownika, PID - identyfikator procesu, PPID - identyfikator procesu macierzystego, PRI - priorytet, NI - wartość nice, TIME - całkowity czas CPU użyty przez proces, CMD - polecenie uruchamiające proces

Stos jądra procesu

W PC i stacjach roboczych typowo każdy proces posiada dwa stosy:

- stos użytkownika (użytkownika procesu)
- stos jądra (używany podczas wykonywania kodu jądra)



Kod jądra systemu operacyjnego oraz dane znajdują się we współdzielonej przestrzeni adresowej wszystkich procesów. Wywołanie systemowe i obsługa przerwań powodują przełączenie ze stosu użytkownika na stos jądra.

Wątki

Wątek – (lekki) proces działający w tej samej wirtualnej przestrzeni adresowej, co tworzący go (ciężki) proces. Stan wątku jest zdefiniowany przez małą, odrębną ilość danych (własny stan rejestrów i stos).

Grupa równoprawnych wątków współdzieli stan i zasoby procesu

- kod
- przestrzeń adresową
- otwarte pliki
- zasoby systemu
- uprawnienia (należą do tego samego użytkownika)

Wątki ze sobą współpracują, a nie współzawodniczą (tak jak procesy).

Wątek, to podstawowa jednostka wykorzystania procesora.

Procesy i wątki

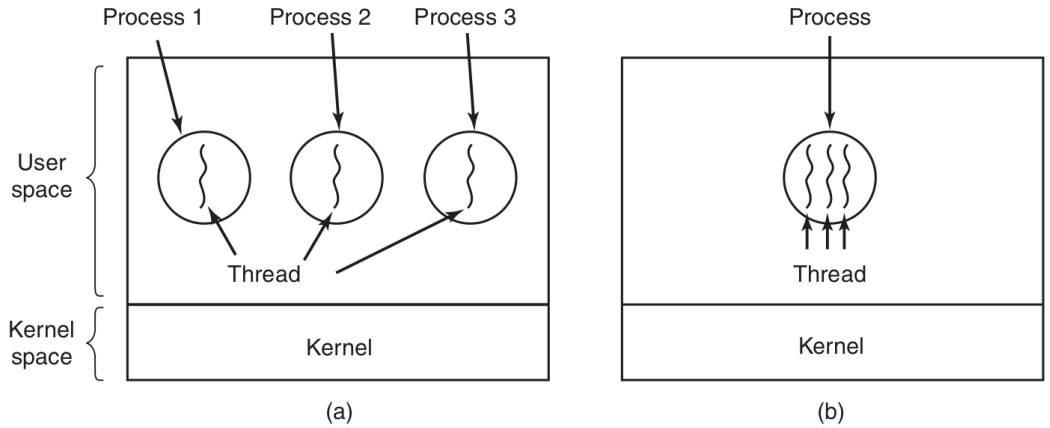
Procesy grupują zasoby, zaś wątki są jednostkami wykonawczymi pracującymi na CPU.

Wątki posiadają własny:

- licznik programu, wskazujący następną instrukcję do wykonania
- stos z historią wywołań procedur
- rejestry, przechowujące wartości zmiennych lokalnych
- stan, identycznie do stanu procesu: gotowy, bieżący, zablokowany, zakończony

Wielowątkowość - gdy możliwe jest występowanie wielu wątków w procesie

Procesy i wątki

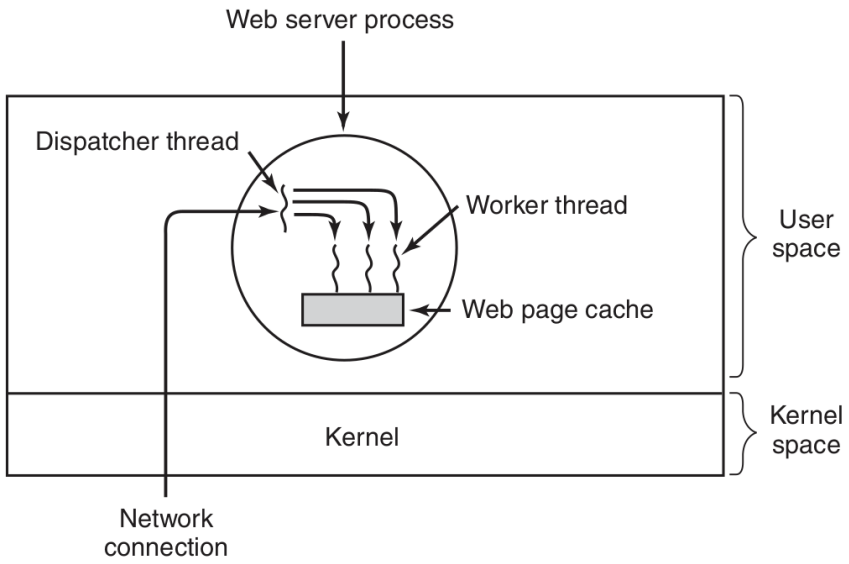


Proces	Wątek
przestrzeń adresowa	licznik programu
zmienne globalne	rejestry
otwarte pliki	stos
procesy potomne	stan
zaległe alarmy	
sygnały i procedury obsługi sygnałów	
informacje dotyczące statystyk	

Zalety wątków

- przełączanie procesora między wątkami w procesie trwa krócej niż przełączanie między procesami
- tworzenie i niszczenie wątku w istniejącym procesie zajmuje o wiele mniej czasu niż tworzenie i niszczenie procesu (nawet 10-100 razy szybsze)
- lepsza realizacja przetwarzania współbieżnego na maszynach wieloprocesorowych
- sprawniejsza komunikacja między wątkami niż między procesami (wątki współdzielą pamięć procesu), komunikacja nie wymaga angażowania jądra

Przykład zastosowania wątków: serwer WWW




```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

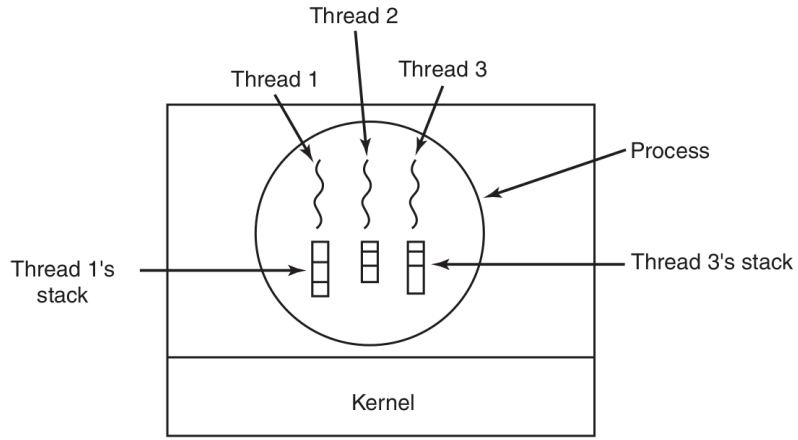
(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

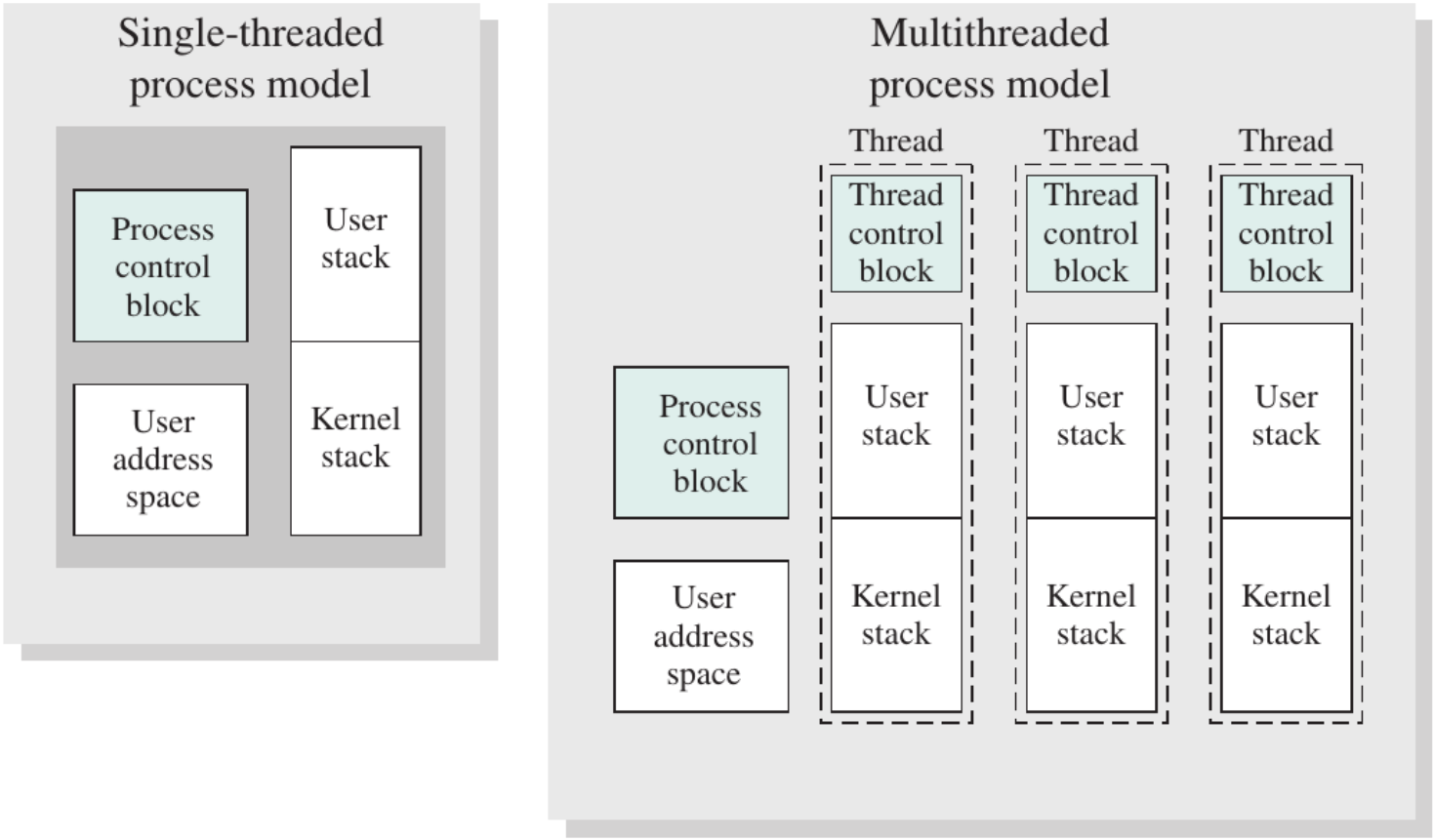
Przykład serwera realizowanego za pomocą wątku dyspozytora i wątków roboczych

Stos procesu

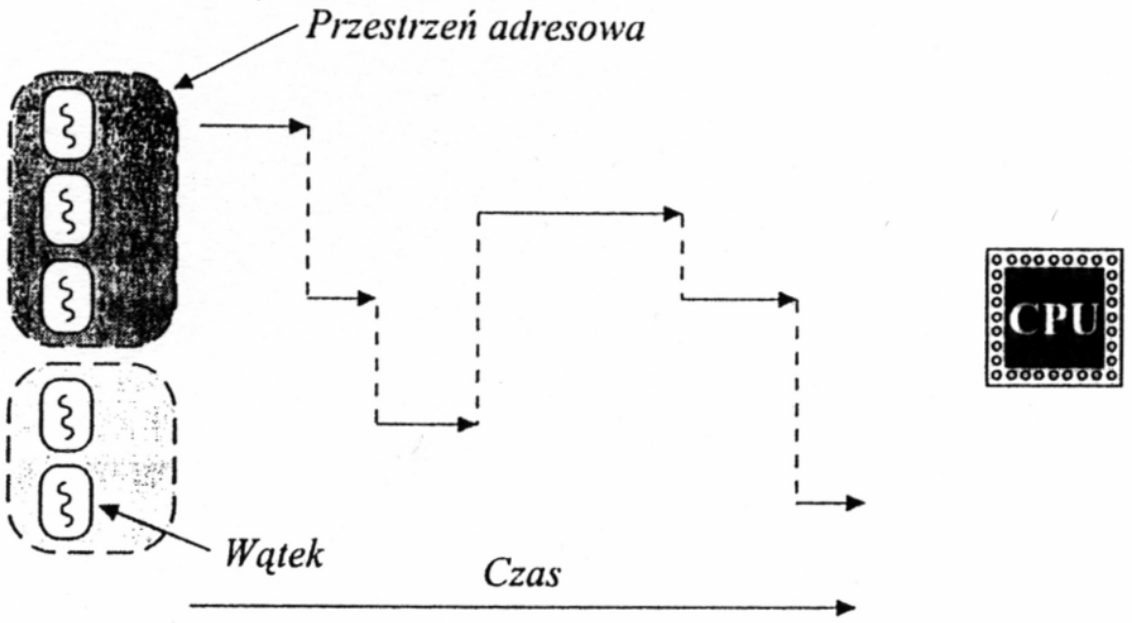


- każdy wątek posiada własny stos
- stos zawiera po jednej ramce dla każdej procedury, która została uruchomian a z której jeszcze nie nastapił powrót
- ramka zawiera zmienne lokalne procedury i adres powrotu
- każdy watek może uruchamiać różne procedury i posiadać odmienną historię wywołań

Model procesu jednowątkowego i wielowątkowego



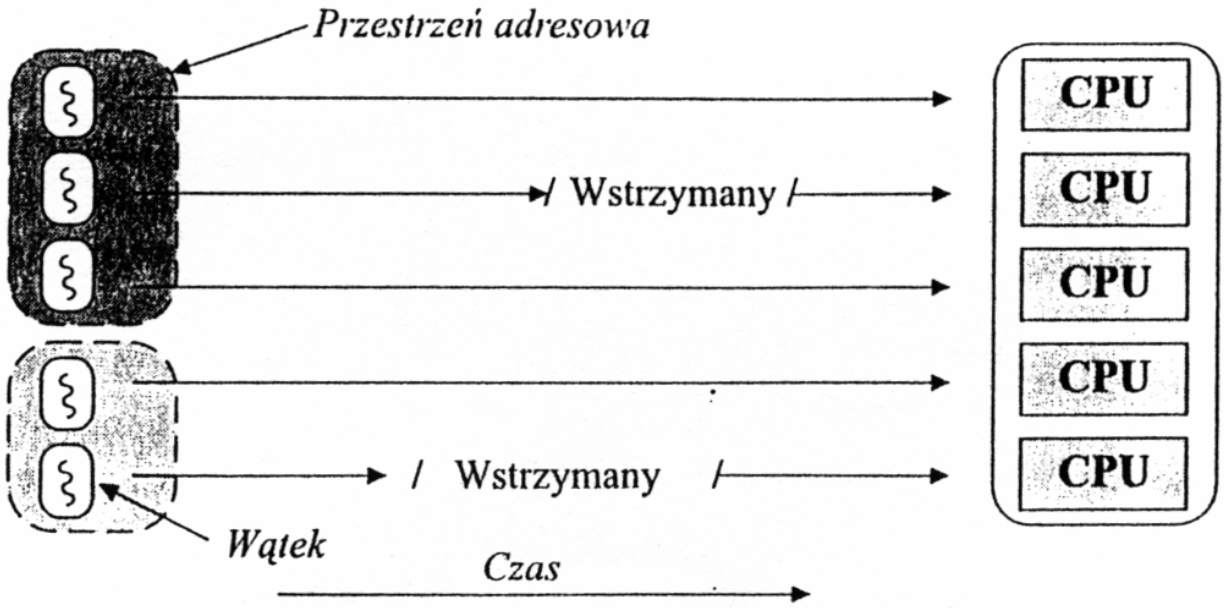
Procesy wielowątkowe w systemie z jednym CPU



Rysunek 3-2. Procesy wielowątkowe w systemie z jednym procesorem

Niektóre architektury CPU posiadają wsparcie sprzętowe wielowątkowości pozwalając na przełaczenie miedzy wątkami w skali nanosekund (*Hyper-Threading*)

Procesy wielowątkowe w wieloprocesorze

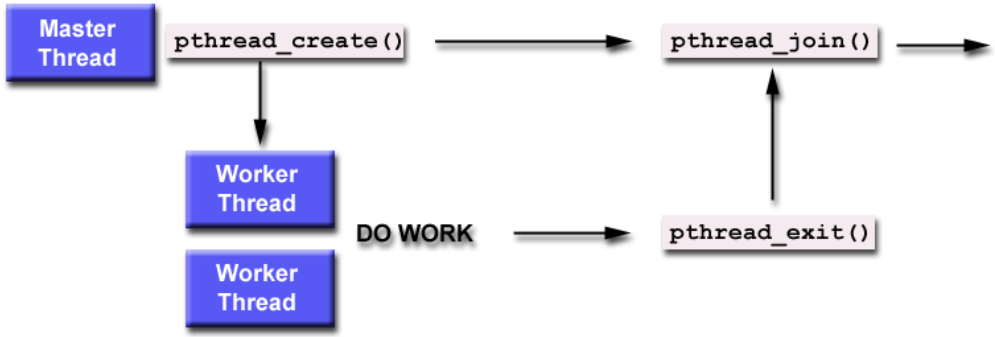


Rysunek 3-3. Procesy wielowątkowe w wieloprocesorze

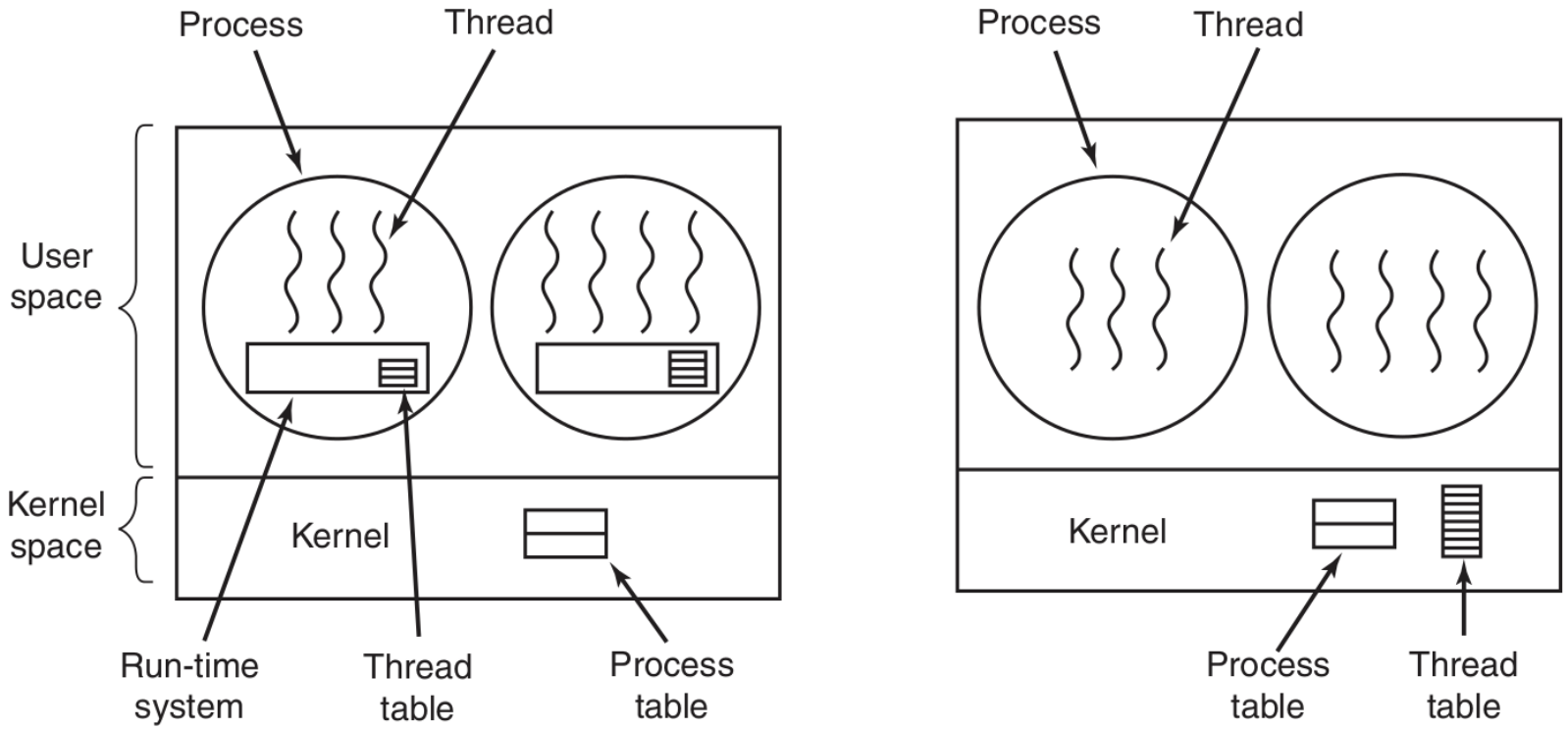
Nowoczesne procesory mają wiele rdzeni, co pozwala na jednoczesne wykonywanie wielu wątków w sposób rzeczywiście równoległy

Zarządzanie wątkami

- nowo utworzony proces zazwyczaj startuje z pojedynczym wątkiem
- wątek może utworzyć nowy wątek (funkcja `thread_create`) przekazując mu procedurę do wykonania
- wątek po wykonaniu pracy zgłasza zamknięcie (funkcja `thread_exit`)
- wątek może oczekiwać na zakończenie innego wątku (funkcja `thread_join`), jest wówczas w stanie zablokowanym
- wątek może dobrowolnie zrzec się CPU (funkcja `thread_yeld`) tak aby dać innym wątkom szansę na wykonanie

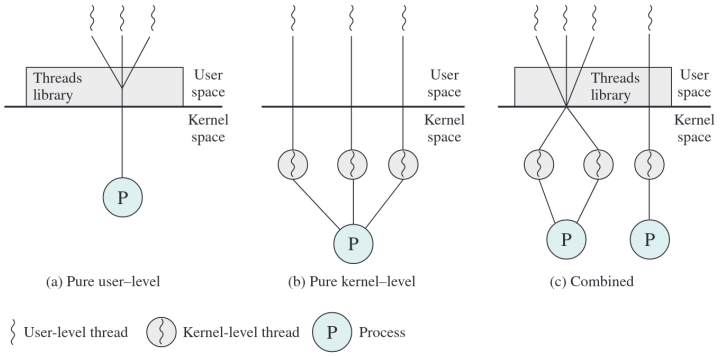


Wątki poziomu użytkownika i poziomu jądra



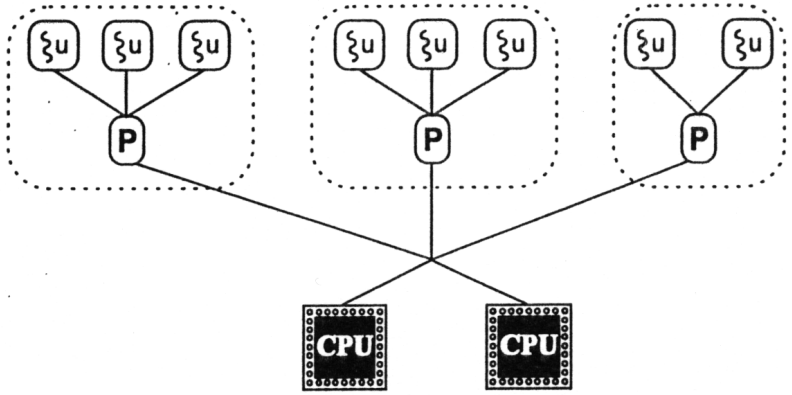
Rodzaje wątków

- **wątki poziomu użytkownika** (*user-level threads*) - zarządzane przez bibliotekę w przestrzeni użytkownika, np. GNU Portable Threads
- **wątki poziomu jądra** (*kernel-level threads*) zarządzane bezpośrednio przez jądro systemu operacyjnego, np. PThreads (p-wątki, wątki zgodne z POSIX), Windows Thread API, Native POSIX Thread Library (NPTL)
- **wątki hybrydowe (mieszane)** (*hybrid threads*) - powiązanie wątków użytkownika z wątkami jądra, np. procesy lekkie (*lightweight processes*, LWP), włókna (*fiber*) w Windows, NPTL w Linux

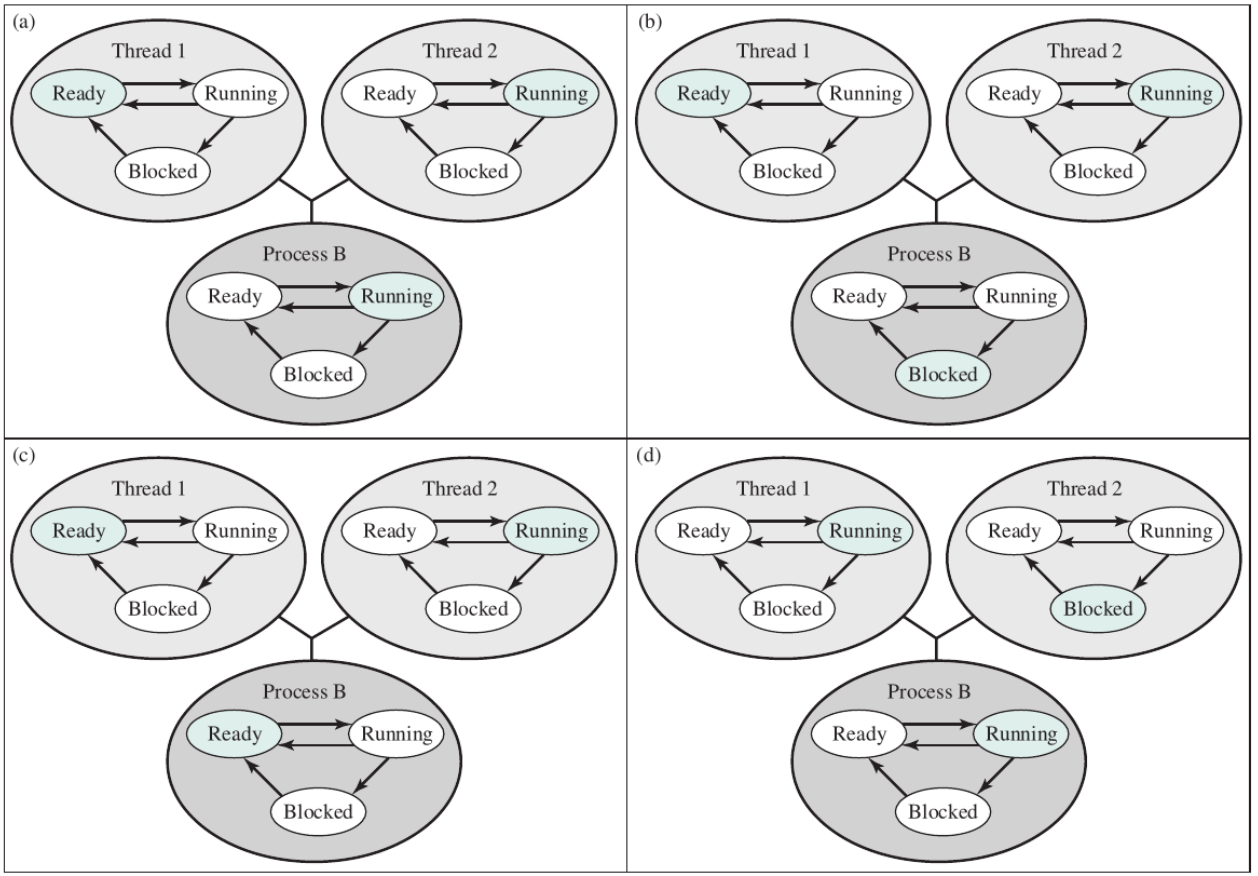


Wątki poziomu użytkownika

- wątki działające w przestrzeni użytkownika są implementowane i zarządzane przez bibliotekę
- każdy proces przechowuje własną **tablicę wątków**, wskazującą na zasoby wątków związanych z procesem (licznik programu, rejestry, stos)
- kontekst wątku jest zapamiętywany i odtwarzany z poziomu użytkownika, bez angażowania jądra
- jądro zarządza procesem (nie wie o istnieniu wątków użytkownika), wywłaszczając proces użytkownika wywłaszcza związane z nim wątki



Stany wątków użytkownika a stany procesu



operacja blokująca wątku użytkownika blokuje proces

proces wyczerpał kwant czasu lub został wywłaszczony

wątek 2 blokuje się na rzecz wątku 1

Planowanie wątku odbywa się na poziomie użytkownika (jądro nie jest świadome stanu wątków) a planowania procesu realizowane jest przez jądro systemu operacyjnego

Zalety wątków użytkownika

- przełaczne wątków jest bardzo szybkie, nie jest wymagane przejście do trybu jądra
- wątki użytkownika są wydajne, nie zużywają zasobów jądra (jeśli nie są związane z wątkiem poziomu jądra)
- każdy proces może mieć indywidualny schemat zarządzania wątkami, dostosowany do potrzeb aplikacji
- mogą być implementowane w systemach operacyjnych, które nie wspierają wielowątkowości na poziomie jądra

Problemy wątków użytkownika

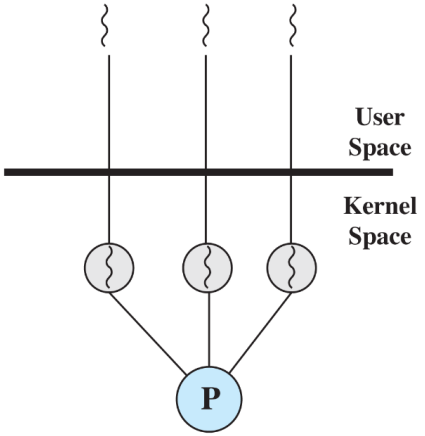
- biblioteka szereguje wątki, jądro procesy;
  - tylko jeden wątek procesu może być wykonywany w danym momencie nawet na maszynie wieloprocesorowej,
  - wątki zwiększają poziom współbieżności, ale nie równoległości
- operacje blokujące (wywołania systemowe, błędy strony) powodują zablokowanie procesu, więc i wszystkich wątków w procesie
- wywołania systemowe generują przerwanie i przejście do trybu jądra, jest to kosztowne
- uruchomiony wątek zajmuje CPU nie dając możliwości działania innym wątkom, dopóki nie zrzeknie się procesora (*yeld*)

Oałanianie (ang *jacketing*) - technika polegająca na owinięciu wywołań systemowych wątku użytkownika w funkcje, które zarządzają blokowaniem i odblokowywaniem innych wątków w procesie.

Wątki poziomu jądra

- wątki poziomu jądra (ang. *kernel-level threads*) są zarządzane bezpośrednio przez jądro systemu operacyjnego
- jądro utrzymuje tablicę wątków, podobnie jak tablicę procesów
- jądro jest świadome istnienia wątków i może nimi zarządzać, planuje wątki na procesorach, przełącza je, tworzy i niszczy
- aplikacje i programiści korzystają z interfejsów API udostępnianych przez system operacyjny, które abstrahują zarządzanie wątkami

Windows API:	`CreateThread()`
POSIX:	`pthread_create()`
C++ 11:	`std::thread`



Wątki poziomu jądra: wady i zalety

Zalety:

- przetwarzanie równoległe: system operacyjny może jednocześnie planować wiele wątków z jednego procesu na wielu procesorach
- w momencie zablokowania wątku system może wznowić wykonywanie innego, gotowego wątku (także z innego procesu)
- jądro może precyzyjnie kontrolować dostęp do zasobów i zarządzać priorytetami wątków
- wątki poziomu jądra nie wymagają implementacji nieblokujących wywołań systemowych, w momencie wywołania blokującego inny wątek może być wykonywany

Wady:

- przełączanie wątków przez jądro jest wolniejsze niż w przypadku wątków użytkownika, ponieważ wymaga przejścia do trybu jądra
- tworzenie oraz niszczenie wątków odbywa się przez odwołania do jądra, jest to kosztowniejsze niż w przypadku wątków użytkownika

Procesy i wątki: porównanie

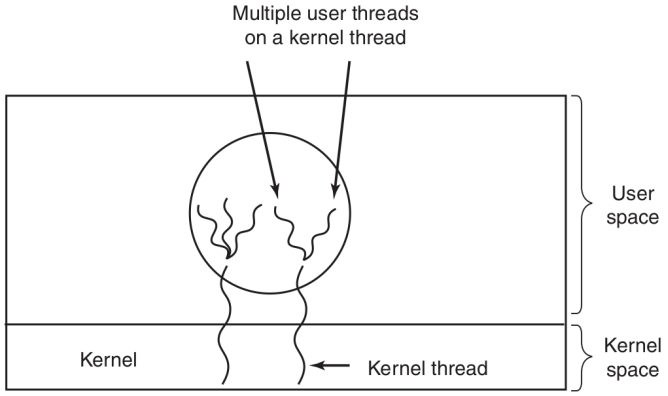
- **Null fork:** czas mierzony od utworzenia poprzez jego szeregowanie, wykonywanie, aż do zakończenia procesu (wątku), który wykonuje pustą procedurę (miara obciążenia spowodowanego przez tworzenie procesu (wątku))
- **Signal-Wait:** czas potrzebny, aby proces (wątek) przesłał sygnał oczekiwania innemu procesowi (wątkowi) i otrzymał odpowiedź zawierającą warunek (miara obciążenia spowodowanego wzajemną synchronizacją)

Czas oczekiwania (μs) na wykonanie operacji przez wątki użytkownika (UT), wątki jądra (KT) i procesy (P).[13]

	UT	KT	P
Null fork	34	948	11300
Signal-wait	37	441	1840

[13] VAX, system uniksopodobny

Hybrydowe implementacje wątków



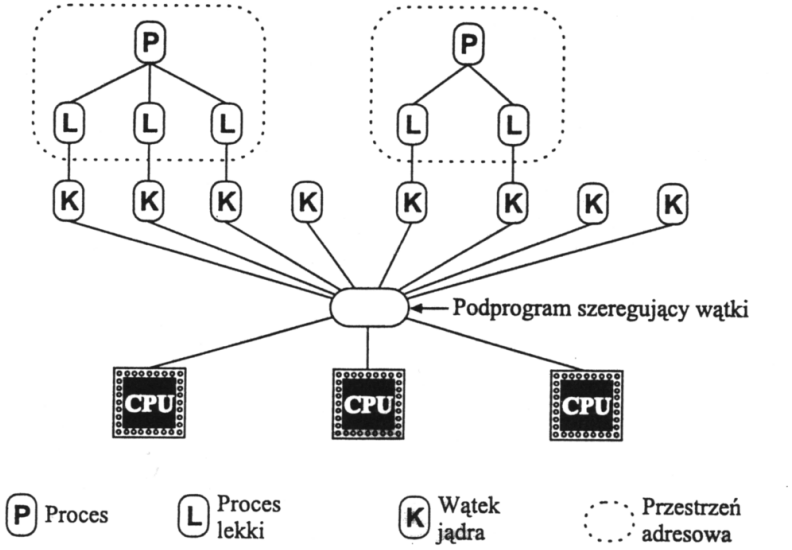
- podejście mieszane, wykorzystuje wątki jądra i użytkownika
- wątki poziomu jądra są zwielokrotniane na poziomie użytkownika
- jądro systemu zarządza wyłącznie wątkami jądra
- zarządzanie wątkami użytkownika odbywa w ramach procesu, analogicznie do tradycyjnego modelu wątku użytkownika

Zalety hybrydowych wątków

- większość prac związanych z zarządzaniem wątkami (tworzenie, synchronizacja) odbywa się na poziomie użytkownika
- programista może dobrać liczbę wątków jądra do konkretnej aplikacji i procesora
- wiele wątków tej samej aplikacji może być wykonywanych równolegle na wielu procesorach
- wątki mogą być blokowane niezależnie, bez blokowania całego procesu

Procesy lekkie, LWP

- Proces lekki (*light-weight process*, LWP) jest wspieranym przez jądro wątkiem z przestrzeni użytkownika
- z procesem może być związanych wiele procesów lekkich, z których każdy jest wspierany przez osobny wątek jądra (relacja 1:1)



https://www.thegeekstuff.com/2013/11/linux-process-and-threads/

Pojęcie procesów lekkich bywa używane w kilku znaczeniach:

- oryginalnie w Solaris (Unix) oznacza mapowanie wątku użytkownika na wątek jądra
- w Linux proces lekki to jednostka wykonawcza (zadanie) planowana przez jądro, może być procesem lub wątkiem (oba są reprezentowane przez strukturę `task_struct`), więc jest to synonim wątku poziomu jądra
- w Windows `fibers` (włókna) są wątkami wykonywanymi całkowicie w trybie użytkownika, które nie są zarządzane bezpośrednio przez jądro systemu ale działają w kontekście wątku jądra - są to lekkie wątki użytkownika

Tworzenie procesu w UNIX/Linux

Funkcja systemowa `fork()` tworzy nowy proces

- przydziela nowemu procesowi pozycję w tablicy procesów
- przydziela procesowi potomnemu unikatowy identyfikator PID
- tworzy logiczną kopię procesu macierzystego (ew. zapewniając współdzielenie segmentów instrukcji, itp.);
- mechanizm kopiowania przy zapisie (*copy-on-write*, COW) odkłada w czasie tworzenie rzeczywistej kopii, aż do momentu gdy nastąpi zmiana w pamięci procesu macierzystego lub potomnego
- zwiększa plikom związanym z tym procesem liczniki w tablicy plików i i-węzłów
- przekazuje identyfikator potomka procesowi macierzystemu i wartość zero procesowi potomnemu

Tworzenie procesu w UNIX/Linux

Wywołanie systemowe `exec()` umieszcza w obszarze pamięci bieżącego procesu kopię pliku wykonywalnego

- aktualny proces zaczyna wykonywać nowy program
- zawartość kontekstu poziomu użytkownika staje się niedostępna, z wyjątkiem parametrów `exec()`, które jądro kopiuje ze starej do nowej przestrzeni adresowej
- w bibliotece systemowej wywołanie systemowe `exec()` realizowane jest przez funkcję `execve()` lub inne odmiany tej funkcji

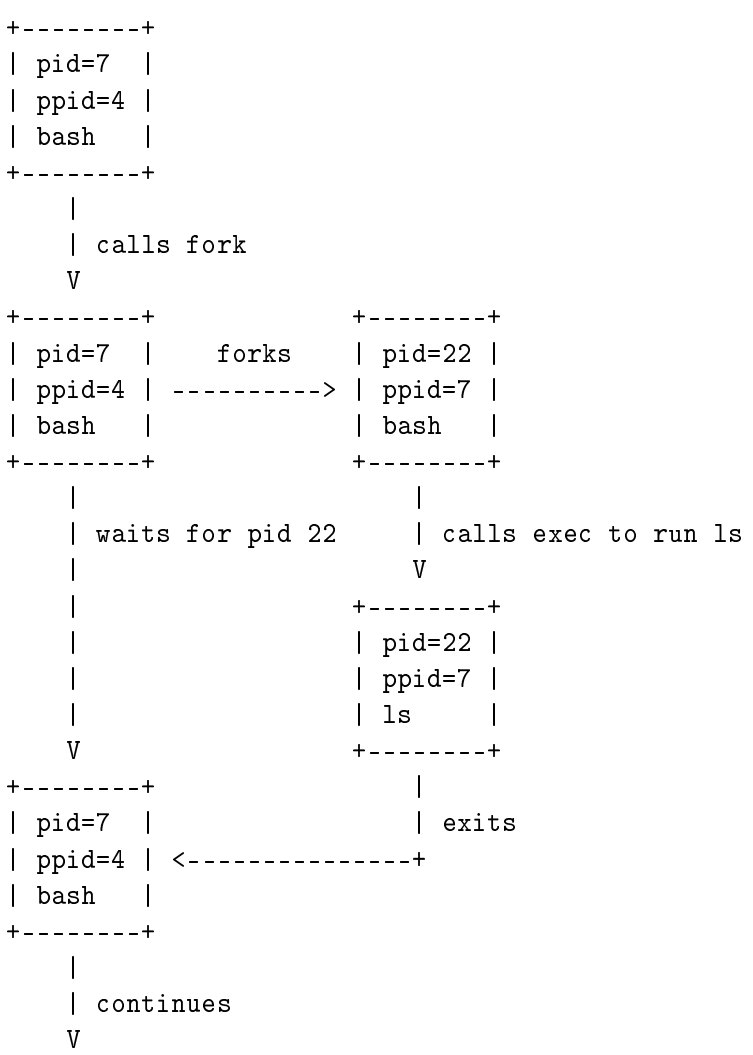
Procesy: tworzenie

Implementacja prostej powłoki

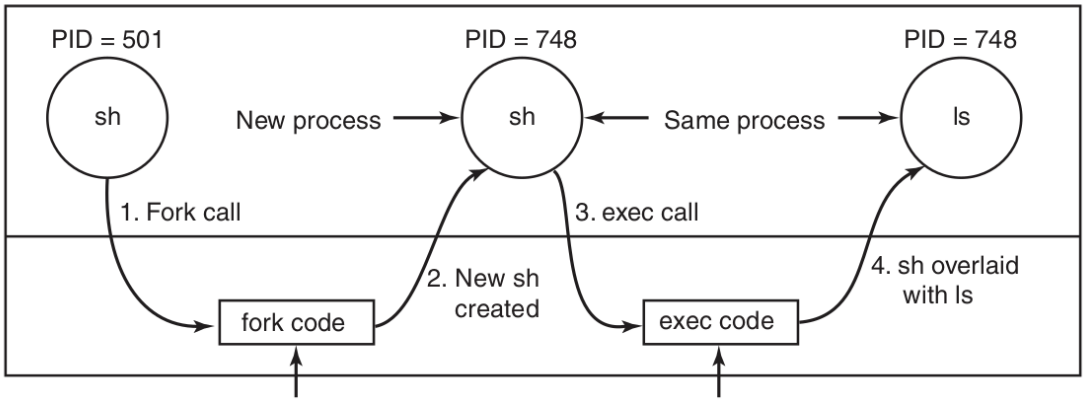
```
while(TRUE) {
   type_command();
   read_command(command, params);

   pid = fork();
   if (pid < 0) {
      printf("Unable to fork()");
      continue;
   }

   if (pid != 0) {
      /* rodzic czeka na potomka */
      waitpid(-1. &status, 0);
   } esle {
      /* potomek wykonuje pracę */
      execve(command, params, 0);
   }
}
```



Uruchomienie procesu w powłoce



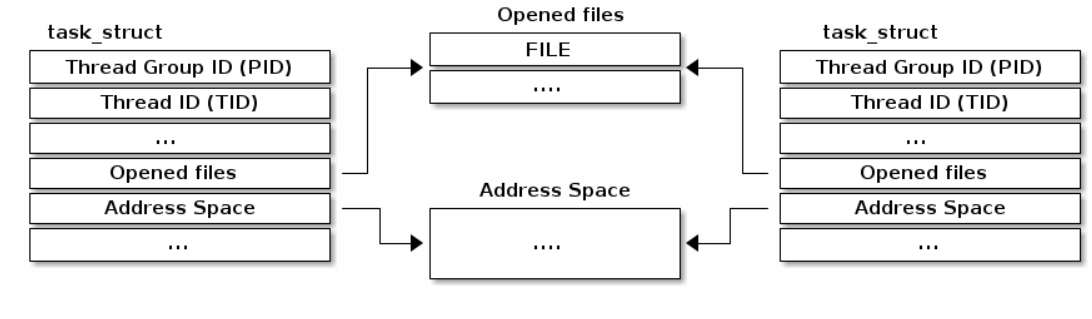
1. przydziela pamięć dla struktury procesu potomnego
2. wypełnia strukturę procesu potomnego danymi rodzica
3. przydziela pamięć dla stosu i przestrzeni użytkownika procesu potomnego
4. wypełnia przestrzeń użytkownika procesu potomnego danymi procesu rodzica
5. przydziela procesowi potomnemu PID
6. ustawia współdzielenie tekstu z rodzicem
7. kopiuje tablice stron danych i stosu
8. ustawia współdzielenie plików z rodzicem
9. kopiuje rejestry rodzica do dziecka

1. odnajduje program wykonywalny
2. sprawdza uprawnienia wykonania
3. odczytuje i weryfikuje nagłowek programu
4. kopiuje argumenty i łańcuchy środowiska do jądra
5. zwalnia starą przestrzen adresową
6. przydziela nową przestrzen adresową
7. kopiuje argumenty i łancuchy środowiska na stos
8. zeruje sygnały
9. inicjuje rejestry

Procesy: zarządzanie w systemie GNU/Linux

W jądrze linuksa procesy (i wątki) reprezentowane są w postaci **zadań** (*task*) przez strukturę danych (`task_struct`)

- struktura zadania reprezentuje jednostkę wykonawczą (proces lub wątek)
- zasoby zadania nie są osadzone w strukturze, struktura posiada wskaźniki do zasobów współdzielonych z innymi zadaniami
- jednowątkowy proces użytkownika to pojedyncze zadanie, wielowątkowy proces to wiele zadań (po jednym na wątek) ale dzielących wspólne zasoby



Najważniejsze atrybuty struktury zadań task_struct

- informacje o zadaniu (*task info*): jednoznaczny identyfikator zadania (TID), informacje o związkach rodzic/dziecko pomiędzy zadaniami, identyfikator grupy zadań (TGID), gdzie zadania z pojedynczej grupy TGID reprezentują tradycyjny proces wielowątkowy
- przestrzeń adresowa (*address space*): definiuje przestrzeń wirtualną zadania
- informacje o przydziale procesora (*scheduling info*): status zadania, politykę przydziału procesora, parametry tej polityki (priorytet dynamiczny), flaga *need_resched* informująca, czy zadanie powinno zostać wywłaszczone
- informacja o programie wykonywalnym (*executable info*): identyfikuje plik, który jest obrazem wykonywanego zadania
- informacja o sygnałach (*signal info*): zawiera dane związane z sygnałami i ich obsługą (*signal handler table, pending signal mask, signal queue*)

- dane uwierzytelniające (*credentials*): dane określające prawa i przywileje zadania (UID, GID, maska określająca prawa dostępu do urządzeń)
- informacje księgowe (*accounting info*): czas utworzenia zadania, czas zużyty w trybie jądra i użytkownika, liczba błędów stron, itp.
- ograniczenia zasobów (*resource limits*): parametry określające maksymalną wielkość pliku *core*, czas wykonywania się zadania, wykorzystywaną pamięć, liczbę otwartych plików, itp.
- informacja o systemie plików (*filesystem info*): domyślna maska określająca prawa dostępu przy tworzeniu plików (*umask*), bieżący katalog
- tablica otwartych plików (*open file table*): tablica plików otwartych przez zadanie
- różne (*miscellaneous*): dodatkowe atrybuty zadania potrzebne do jego prawidłowego wykonywania się, np. terminal sterujący

Jądro systemu GNU/Linux: tworzenie zadania

Wywołanie systemowe `clone()` tworzy nowe zadania (wątki lub procesy)

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

Uruchamia funkcję w kontekście nowego wątku, który współdzieli (lub nie) zasoby zadania zależnie od wartości `sharing_flags`

flagi	współdzielone zasoby	gdy ustawiona	gdy nie ustawiona
CLONE_VM	przestrzen adresowa	powstaje nowy wątek	powstaje nowy proces
CLONE_FS	infomracje o systemie plików katalog roboczy, główny, umask	wspólny katalog roboczy	niezależny katalog roboczy
CLONE_FILES	tablica otwartych plików	współdzielenie deskryptorów	koipie deskryptoów
CLONE_SIGHAND	tablica obsługi sygnałów	współdzieli	kopiuje tablicę
CLONE_PARENT	informacje o rodzicu	nowy wątek współdzieli rodzica	nowy wątek staje się potomkiem
CLONE_PID	PID	wątek ma ten sam PID	proces uzyskuje nowy PID

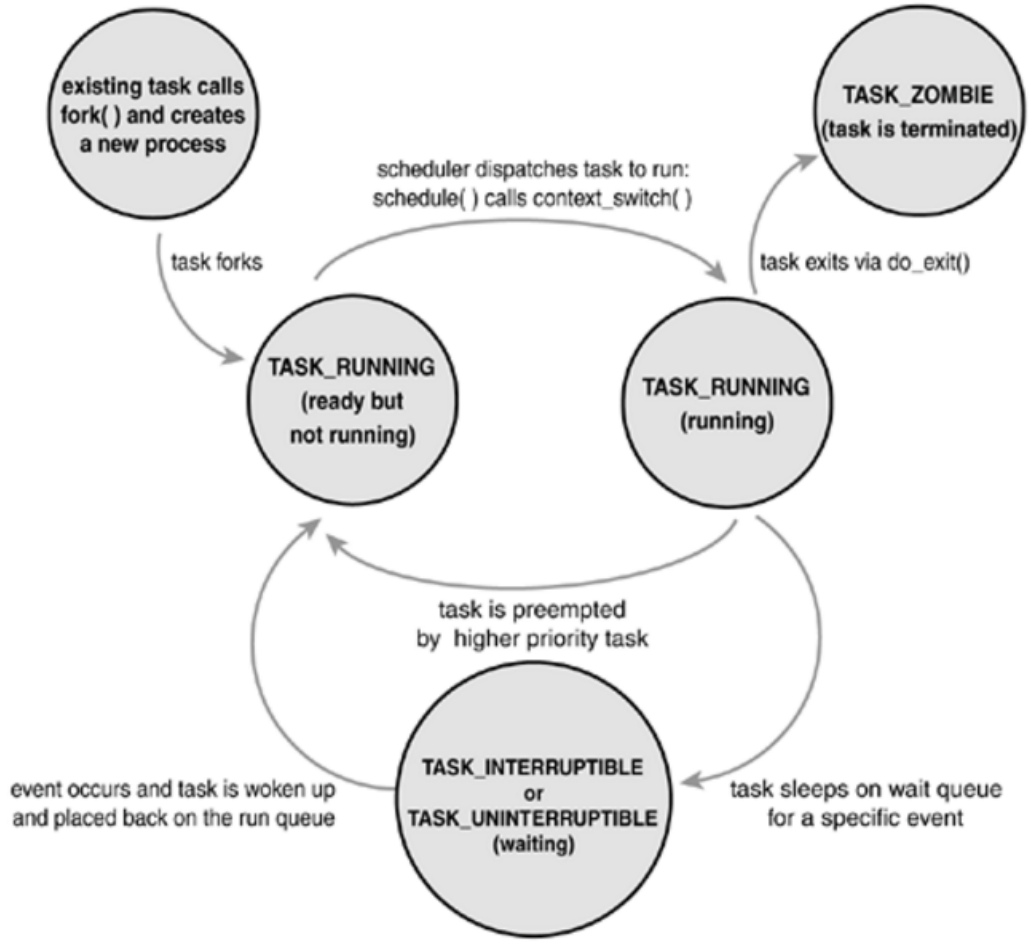
Zarówno wywołanie systemowe `fork()`, jak i funkcja `pthread_create()` korzystają z implementacji `clone()` do tworzenia procesów i wątków.

Stany procesów wg *Linux kernel*

- `TASK_RUNNING` – proces albo się wykonuje, albo czeka na wykonanie w kolejce procesów gotowych
- `TASK_INTERRUPTIBLE` – proces jest wstrzymany do czasu zajścia określonego zdarzenia. Proces może zostać zbudzony przez przerwania sprzętowe, zwolnienie zasobu, na który proces czeka, albo dostarczenie sygnału
- `TASK_UNINTERRUPTIBLE` – wybudzenie może nastąpić po uzyskaniu zasobu, na który proces czekał. Sygnały są ignorowane [14]
- `TASK_ZOMBIE` – wykonanie procesu zostało zakończone, ale proces rodzica nie odebrał jeszcze informacji o jego zakończeniu (rodzic nie użył jeszcze wywołania systemowego `wait()` lub `waitpid()`)
- `TASK_STOPPED` – wykonanie procesu zostało zatrzymane (wskutek odebrania sygnału `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`)

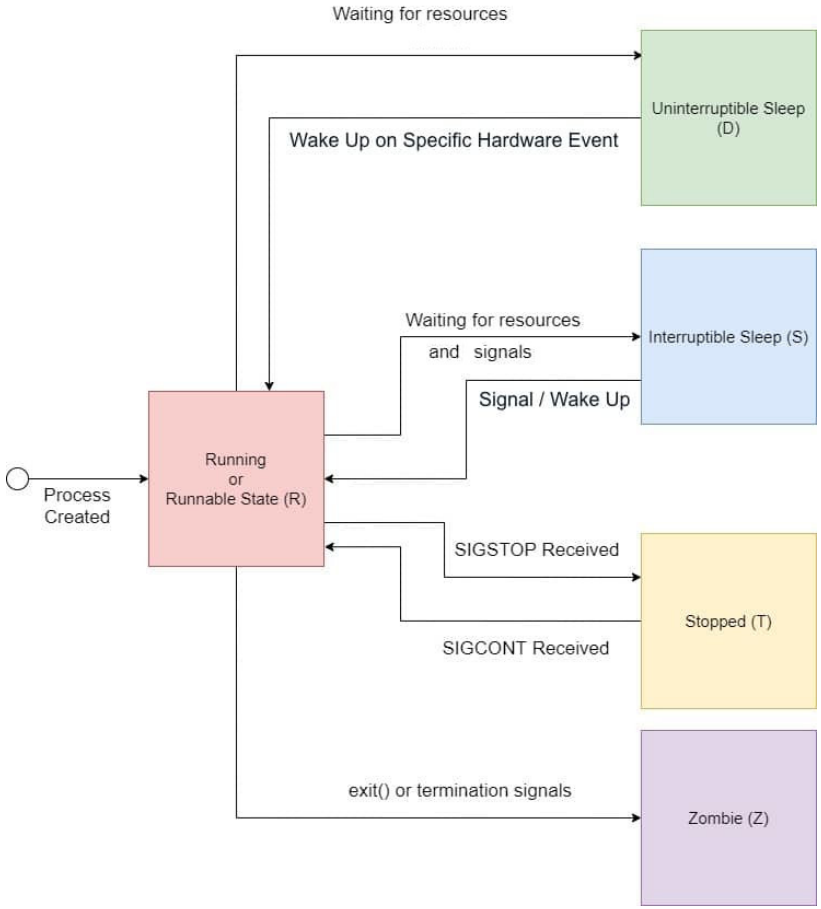
[14] TASK_KILLABLE, alternatywny stan `TASK_UNINTERRUPTIBLE` reagujący na krytyczne sygnały wprowadzony w łatce do jądrza 2.6.25

Stany procesów



http://learnlinuxconcepts.blogspot.com/2014/03/process-management.html

Stany procesów



Obserwowanie stanów: polecenie ps

```
$  ps a
    PID TTY      STAT   TIME COMMAND
   5270 tty2     SNsl+   0:00 /usr/libexec/gdm-x-session --run-script env GNOME_SHELL_SESSION_MODE=pop /usr/bin/gnome-s
   5272 tty2     S<l+   59:28 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -nolisten tcp -bac
   5924 pts/0    SNs    0:37 /usr/bin/zsh
1680036 pts/7    TNs+   0:00 zsh -Z -g
1871214 pts/5    TN     0:00 pdflatex so05-procesy.tex
1934828 pts/5    SNLl+  0:36 gimp process-finite-state-machine.jpeg
1940687 pts/0    SN     0:00 /usr/bin/zsh
1940688 pts/0    RN+    0:00 ps a
```

Stany procesów wg `man ps`

R (running or runnable) – wykonujący się lub zdolny do pracy (oczekujący w kolejce)

S (interruptible sleep) – śpiący (blocked)

D (uninterruptable sleep) – nieprzerywalny sen (zwykle I/O)

T – zatrzymany przez sygnał sterujący

t – zatrzymany przez program śledzący (debugger)

Z (defunct) "zombie" – zakończony, ale nie zebrany przez proces rodzica

X (dead) – (should never be seen)

< (high priority process) – proces o wysokim priorytecie

N (low priority process) – proces o niskim priorytecie

L (pages locked into memory) – proces ze stronami uwięzionymi w pamięci

s – leader sesji

l – proces wielowątkowy

\+ – proces pierwszego planu

Procesy i wątki: obserwowanie procesów wielowątkowych

```
$ ps -eo stat,pid,ppid,tgid,tid,lwp,nlwp,vsz,rsz,cmd
STAT     PID    PPID    TGID     TID     LWP NLWP    VSZ   RSZ CMD
...
SNl  1217879 1217875 1217879 1217879 1217879    4 321996 29420 /usr/bin/gnome-terminal.real --wait
...
```

```
$ ps  -m -o stat,pid,ppid,tgid,tid,lwp,nlwp,vsz,rsz,cmd -p 1217879
STAT     PID    PPID    TGID     TID     LWP NLWP    VSZ   RSZ CMD
-    1217879 1217875 1217879       -       -    4 321996 29420 /usr/bin/gnome-terminal.real --wait
SNl        -       -       - 1217879 1217879    -      -     - -
SNl        -       -       - 1217880 1217880    -      -     - -
SNl        -       -       - 1217881 1217881    -      -     - -
SNl        -       -       - 1217882 1217882    -      -     - -
```

Z dokumentacji `ps`:

`tgid` - identyfikator grupy wątków, do której należy zadanie (alias `pid`)

`tid` - unikalny numer reprezentujący jednostkę wykonywalną (*dispatchable entity*) (alias `lwp`)

`lwp` - ID lekkiego procesu (wątku) jednoski wykonywalnej (alias tid)

`rsz` - rozmiar zestawu rezydentnego (*resident set size*), czyli nieswapowana pamięć fizyczna, którą zadanie wykorzystało w kilobajtach (alias rsz)

`vsz` - rozmiar pamięci wirtualnej procesu w KiB

Procesy systemowe w Linux - wyłączne wątki jądra

Jądro nie jest procesem w tradycyjnym sensie, ale zarządcą procesów.

Wyłączne wątki jądra (`Kernel Threads`) wykonują zadania krytyczne dla działania systemu

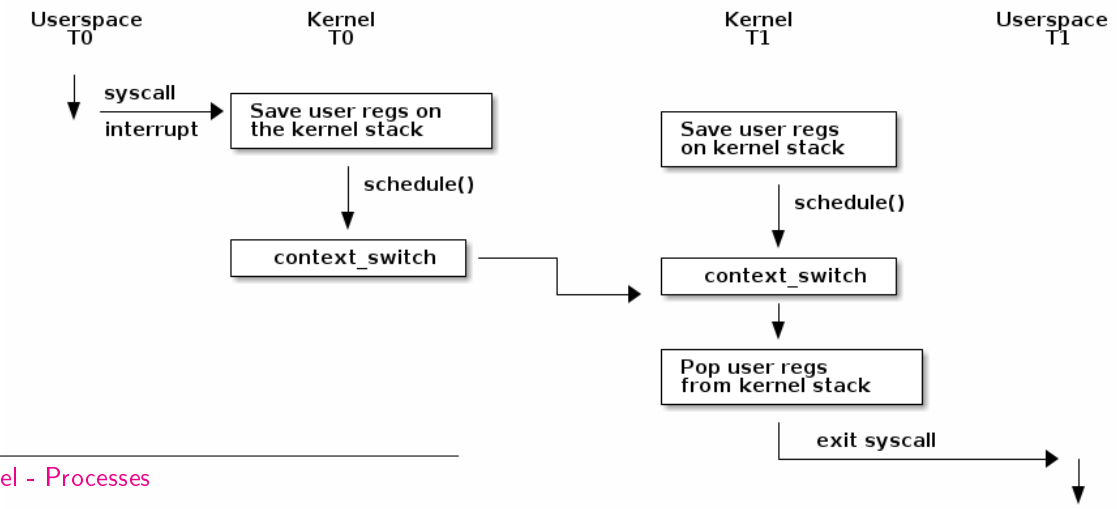
- działają wyłącznie w trybie jądra (w przestrzeni adresowej jądra), nie mają przypisanej przestrzeni adresowej użytkownika, nie mogą wykonywać kodu użytkownika
- działają w kontekście procesu (posiadają unikatowy `PID`), podlagają planowaniu, moga wykonywać operacje blokujące i być wywłaszczane
- nie komunikują się z użytkownikami (nie trzeba terminali)
- tworzone wyłącznie przez jądro (funkcja `kthread_create`) lub podczas startu systemu
- nie można ich zabić

Wątki jądra Linux

```
$ ps l -p 2 --ppid 2
F   UID     PID    PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
1     0       2       0  20   0      0     0 -      S    ?          0:00 [kthreadd]
1     0       3       2  20   0      0     0 -      S    ?          0:00 [pool_workqueue_release]
1     0       4       2   0 -20      0     0 -      I<   ?          0:00 [kworker/R-rcu_g]
1     0       5       2   0 -20      0     0 -      I<   ?          0:00 [kworker/R-slub_]
1     0       6       2   0 -20      0     0 -      I<   ?          0:00 [kworker/R-netns]
1     0       8       2   0 -20      0     0 -      I<   ?          0:00 [kworker/0:0H-events_highpr
...
```

Przełączenie kontekstu zadania

- przed zmianą kontekstu musi dojść do zmiany trybu pracy na tryb jądra
- przejście do trybu jądra zachodzi w wyniku wywołania systemowego lub przerwania sprzętowego
- rejestry przestrzeni użytkownika są zapisywane na stosie jądra wskazywanym przez strukturę zadania
- w tym momencie może dojść do przełączenia kotekstu z innym wątkiem jądra, w którym odtwarzany jest jego stan wykonania



Wywłaszczanie procesu

Algorytm szeregujący zwany **planistą** lub **dyspozytorem** może wstrzymać aktualnie wykonywane zadanie (proces lub wątek), aby umożliwić działanie innemu zadaniu - zadanie zostaje **wywłaszczone**

Do wywłaszczenia może dojść gdy:

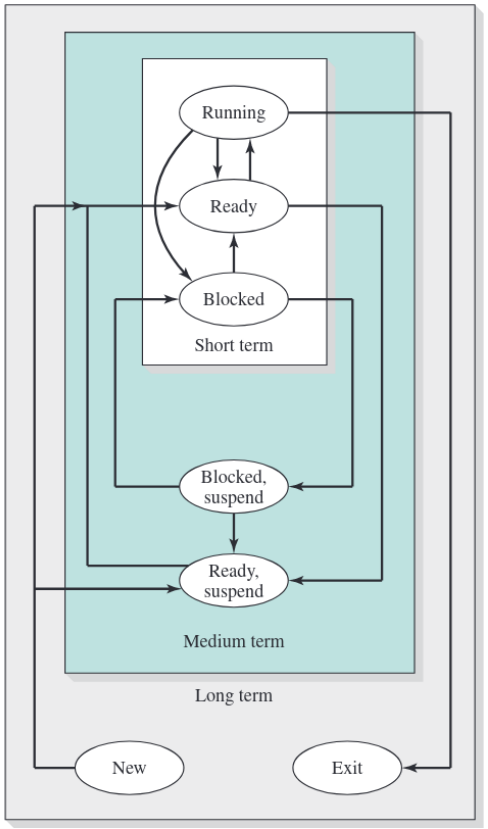
- proces wchodzi w stan gotowy (`TASK_RUNNING`) a jego priorytet jest wyższy od priorytetu procesu właśnie zawłaszczającego procesor
- kiedy proces wyczerpie swój kwant czasu
- wywłaszczenie procesu użytkownika następuje może nastąpić przy powrocie do przestrzeni użytkownika z wywołania systemowego lub z procedury obsługi przerwania
- może zajść w dowolnym momencie, kiedy jednostka centralna wykonuje kod w trybie użytkownika

Czy proces może być wywłaszczony, jeśli przebywa w trybie jądra?

Wywłaszczenie jądra

- jądra współczesnych systemów są **wielobieżne** (wielowejściowe, *reentrant*) i **wywłaszczalne** (*preemprive*): kilka wątków jądra może być bezpiecznie wykonywanych w tym samym czasie (równolegle) i podlegają wywłaszczeniu
- w systemie jednoprocesorowym, gdy tylko jeden proces może działać, inne będą czekać na CPU lub na zakończenie operacji wej/wyj (będąc zablokowanymi w trybie jądra)
- wystąpienie przerwania sprzętowego pozwala jądru wielobieżnemu na zatrzymanie procesu, nawet jeśli znajduje się on w trybie jądra. Wpływa to na zwiększenie szybkości obsługi urządzeń zewnętrznych.
- Linux (z jądrem w wersji $\leqslant$ 2.4) jest systemem operacyjnym z wywłaszczaniem procesów, ale bez wywłaszczania jądra. Jądro $\geqslant$2.6 jest już wielowejściowe, umozliwia wywłaszczenie zadania działajacego w trybie jądra i wykonującego wywołanie systemowe

Planowanie długoterminowe, krótkoterminowe i średnioterminowe



- **planowanie długoterminowe** (*long-term scheduling*) – decyduje o tym, które nowe procesy zostaną załadowane do pamięci głównej i będą mogły konkurować o zasoby systemowe (CPU, pamięć, urządzenia wejścia-wyjścia)
- **planowanie krótkoterminowe** (*short-term scheduling*) – decyduje o tym, który z gotowych do wykonania procesów otrzyma czas procesora (CPU) na wykonanie swoich instrukcji
- **planowanie średnioterminowe** (*medium-term scheduling*) – zarządza wymianą procesów między pamięcią główną a pamięcią wtórną (swapowaniem), co pozwala na lepsze wykorzystanie zasobów systemowych i kontrolę nad liczbą aktywnych procesów w pamięci

Algorytmy przydziału procesora

Planista krótkoterminowy (ang. scheduler) lub **dyspozytor** – część systemu operacyjnego odpowiedzialna za przydzielanie czasu procesora w ramach przełączania zadań. Jeśli dwa lub więcej procesów znajduje się w stanie *gotowy do wykonania*, to o kolejności przydziału CPU decyduje planista wykonując **algorytm szeregowania** realizujący politykę szeregowania.

Polityka szeregowania określa ogólne zasady i cele, jakimi kieruje się system operacyjny przy przydzielaniu czasu procesora procesom.

Planista powinien gwarantować:

- sprawiedliwość przydziału CPU
- dobrą wydajność CPU
- mały czas odpowiedzi
- mały czas przetwarzania (*turnaround time*)
- dużą przepustowość (*throughput*)

Procesy ograniczone obliczeniowo i operacjami wej-wyj

Klasyfikacja procesów ze względu na charakterystykę wykorzystania CPU:

- zorientowane na wejście-wyjście (*I/O-bound*)
  Wymagają szybkiego przydziału czasu procesora i częstszych przełączeń kontekstu, aby efektywnie obsługiwać operacje wejścia-wyjścia.
- zorientowane na obliczenia (*CPU-bound*)
  Wymagają dłuższego czasu procesora i rzadszych przełączeń kontekstu, aby zmaksymalizować wydajność obliczeniową

Wraz ze wzrostem szybkości CPU procesy mają tendencję stawać się bardziej ograniczone przez operacje wej-wyj

Szeregowanie bez wywłaszczenia

Algorytm szeregowania **bez wywłaszczania** (*non-preemptive*) wybiera proces do uruchomienia a następnie pozwala mu działać do czasu zablokowania (np. operacja we-wyj) lub do momentu, gdy proces dobrowolnie zwolni CPU

- prosty w implementacji, małe narzuty związane z przełączaniem kontekstu
- przerwanie zegarowe nie powoduje podjęcia decyzji o szeregowaniu
- wykorzystywany w systemach wsadowych, nie wymagających interakcji z użytkownikiem
- może prowadzić do mniejszej responsywności systemu dla procesów interaktywnych
- ryzyko głodzenia procesów o niższym priorytecie

Szeregowanie z wywłaszczeniem

Algorytm szeregowania **z wywłaszczaniem** (*preemptive*) wybiera proces i pozwala mu działać maksymalnie przez ustalony czas (kwant czasu)

- procesy mogą być przerwane w dowolnym momencie, zazwyczaj przerwaniami zegara, aby umożliwić planiście podjęcie decyzji o szeregowaniu
- umożliwiają sprawiedliwsze przydzielanie czasu procesora w systemach wielozadaniowych
- krótkie kwanty czasu faworyzują procesy interaktywne, poprawiając responsywność systemu
- większe narzuty związane z częstymi przełączeniami kontekstu, zbyt krótkie kwanty mogą prowadzić do spadku wydajności systemu
- dłuższe kwanty czasu zmniejszają narzuty związane z przełączaniem kontekstu, co jest korzystne dla procesów obliczeniowych, kosztem responsywności systemu
- trudniejsze w implementacji

Cele algorytmów szeregowania w różnych środowiskach

Wszystkie systemy

- **sprawiedliwość** - przydział procesom uczciwego dostępu do CPU
- **równowaga** - wszystkie części systemu powinny być zajęte

Systemy interaktywne

- **czasu odpowiedzi** (*response time*) - maksymalizacja responsywności procesów interaktywnych
- **proporcjonalność** - spełnienie oczekiwań użytkowników dotyczących czasu wykonania

Systemy wsadowe

- **przepustowość** - maksymalizacja liczby wykonanych zadań na godzinę
- **czas cyklu przetwarzania** - minimalizacja czasu pomiędzy rozpoczęciem i zakończeniem procesu
- **wykorzystanie CPU** - maksymalizacja zajętości

Systemy czasu rzeczywistego

- **dotrzymywanie terminów** - unikanie utraty danych
- **przewidywalność** - unikanie degradacji jakości w systemach multimedialnych

Planowanie przydziału procesora - mechanizmy

- pierwszy nadszedł, pierwszy obsłużony (FCFS, *First-Come, First-Served*)
- najkrótsze zadanie najpierw (SJF, *shortest job first*)
- rotacyjne, cykliczne (*Round-Robin*)
- priorytety: statyczne i dynamiczne
- klasy szeregowania: dwupoziomowe, wielopoziomowe

Pierwszy nadszedł, pierwszy obsłużony (FCFS)

First-Come, First-Served

- pojedyncza kolejka gotowych procesów
- procesy są wykonywane w kolejności, w jakiej zgłaszają się do systemu, bez względu na ich czas trwania czy priorytet
- proces jest wywłaszczany w momencie zablokowania, kolejny proces zostaje wybrany do wykonania
- odblokowany proces trafia na koniec kolejki gotowych

FCFS: wady i zalety

Zalety:

- prosty w implementacji
- sprawiedliwy, każdy proces jest obsługiwany w kolejności zgłoszeń
- minimalne narzuty związane z przełączaniem kontekstu
- przydatny w systemach wsadowych, gdzie zadania są długotrwałe i nie wymagają interakcji

Wady:

- długie procesy mogą nadmiernie blokować CPU, na niekorzyść krótkich zadań, tzw. zjawisko **konwoju** (*convoy effect*)
- nie jest optymalny dla systemów interaktywnych, gdzie responsywność jest kluczowa

Najkrótsze Zadanie Najpierw (SJF, *Shortest Job First*)

- procesy gotowe są sortowane i wykonywane w kolejności ich przewidywanego czasu wykonania, od najkrótszego do najdłuższego
- proces o najkrótszym przewidywanym czasie wykonania jest wybierany jako pierwszy do uruchomienia
- SJF minimalizuje średni czas oczekiwania procesu, ponieważ krótsze zadania są obsługiwane szybciej, co zmniejsza opóźnienia w kolejce
- w praktyce trudno jest dokładnie oszacować czas wykonania każdego zadania
- dłuższe procesy mogą być stale opóźniane (głodzenie), jeśli nowe, krótsze zadania ciągle się pojawiają w kolejce
- działa efektywne w systemach, gdzie czas wykonania procesów jest znany lub może być oszacowany z dużą dokładnością (systemy wsadowe)

Najkrótsze Zadanie Najpierw (SJF, *Shortest Job First*)



Figure 2-41. An example of shortest-job-first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest job first order.

Czas oczekiwania w oryginalnym porządku:
8, 12, 16, 20, średnio 14

Czas oczekiwania w kolejności od najkrótszego zadania:
4, 8, 12, 20, średnio 11

Algorytm **najkrótszego pozostałego czasu** (SRT, *Shortest Remaining Time*)

- wariant SJF z wywłaszczeniem
- proces o najkrótszym pozostałym czasie wykonania jest wybierany do uruchomienia
- SRT stale monitoruje pozostały czas wykonania procesów w kolejce gotowych
- gdy nowy proces o krótszym czasie wykonania pojawia się w kolejce gotowych, bieżący proces jest wywłaszczany i nowy proces jest uruchamiany
- minimalizuje średni czas oczekiwania procesu

Algorytm **najkrótszego następnego procesu** (SPN, *Shortest Process Next*)

- odmiana SJF, która używa historii uruchomień i zachowania procesów do estymacji czasu wykonywania
- Przykładowa estymacja czasu wykonywania procesu (*aging*) oparta na średniej wykładniczej:

$$T_{n+1} = aT_n + (1 - a)T_{n-1}$$

gdzie $0 \leqslant a \leqslant 1$.

Algorytm rotacyjny (*Round-Robin*)

- procesy gotowe do wykonania są zorganizowane w postaci kolejki, każdy proces otrzymuje kwant czasu
- działa analogicznie do FCFS ale z wywłaszczeniem, wybierany jest zawsze pierwszy proces z kolejki
- gdy kwant czasu dobiegnie końca proces jest wywłaszczany i umieszczany na końcu kolejki
- jeśli proces zakończy się przed upływem kwantu czasu lub zostanie zablokowany to CPU jest przydzielane kolejnemu procesowi z kolejki
- kwant czasu typowo ok. 20-50 msec, wartość optymalna zależy od charakterystyki obciążenia systemu



Figure 2-42. Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

Planowanie priorytetowe

- decyzja planisty zależna od **priorytetów** przypisanych do zadań, ważniejsze zadania otrzymują wyższe priorytety i mają pierwszeństwo
- priorytety mogą być przydzielane **statycznie** lub **dynamicznie**
- statyczne priorytety są przypisywane na podstawie właściwości procesu i nie zmieniają się w trakcie jego życia
- dynamiczne priorytety mogą zmieniać się w zależności od zachowania procesu, np. procesy interaktywne (które często czekają na dane od użytkownika) mogą być promowane przez podwyższenie priorytetu aby zwiększyć responsywność
- procesy mogą zostać pogrupowane względem priorytetów tworząc **klasy priorytetów**. Planowanie między klasami odbywa się zgodnie z priorytetami, zaś w ramach konkretnej klasy wg. wybranego algorytmy szeregowania (np. cyklicznego)

Klasy priorytetów

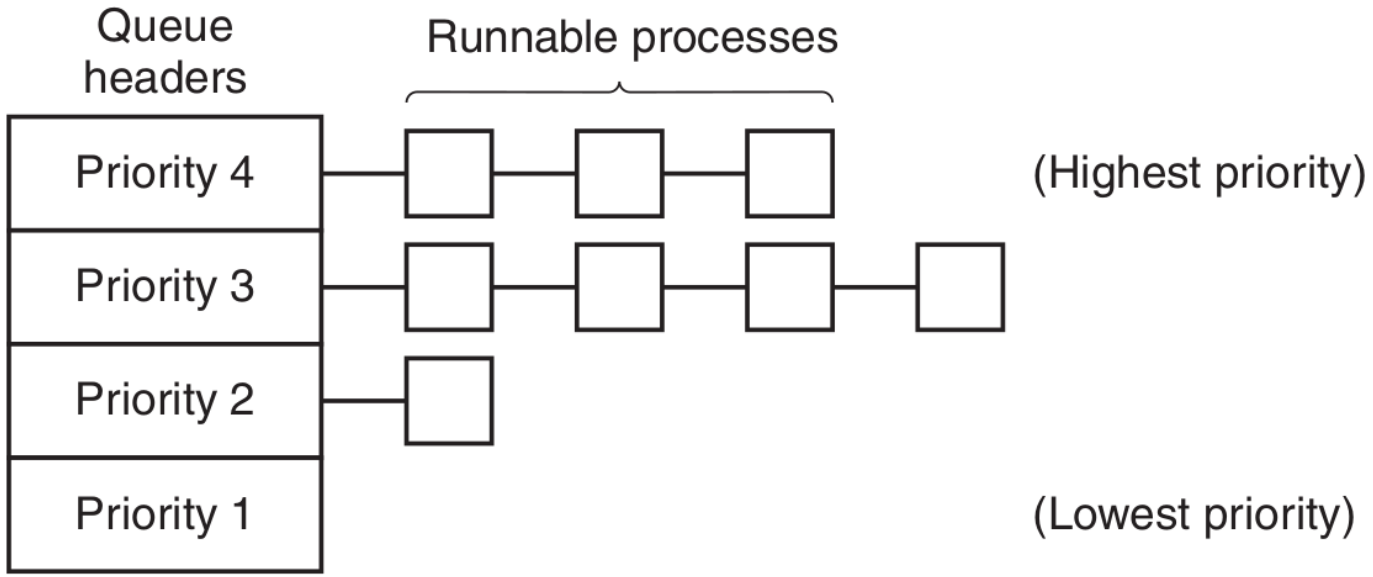


Figure 2-43. A scheduling algorithm with four priority classes.

Algorytm szeregowania w Linuksie

- szeregowaniu podlegają wątki jądra
- 140 priorytetów: [0,139], gdzie niższa wartość oznacza wyższy priorytet

Dwie główne klasy zadań:

- zadania czasu rzeczywistego: priorytety statyczne [0,99]
  polityki planowania: `SCHED_FIFO`, `SCHED_RR`
- zadania normalne: priorytety dynamiczne [100,139]
  polityki planowania: `SCHED_OTHER`

Algorytmy planowania:

- **planista O(1)** używany od wersji jądra 2.5 (11-2001)
- **planista całkowicie sprawiedliwy** (CFS, *Completely Fair Scheduler*) o złożoności O(log N) pojawił się w wersji 2.6.23 (10-2007)

,

Nikita Ishkov, *A complete guide to Linux*

nazwa nie odnosi się do systemów czasu rzeczywistego, nie oznacza spełniania rygorystycznych ograniczeń czasowych

Priorytety czasu rzeczywistego

- priorytety zadań czasu rzeczywistego są z zakresu 0-99, priorytety nie ulegają zmianie w czasie wykonywania
- zadania wykonywane są ściśle w kolejności malejących priorytetów, najpierw zadania czasu rzeczywistego, potem pozostałe zadania

Polityki szeregowania czasu rzeczywistego:

- `SCHED_FIFO` (*First-In, First-Out*) – szeregowanie bez wywłaszczenia analogicznie do FCFS. Do zadań, które muszą być wykonane jak najszybciej i nie mogą być przerwane przez inne procesy o tym samym priorytecie.
- `SCHED_RR` (*Round-Robin*) - szeregowanie rotacyjne (z wywłaszczeniem)

Procesy `SCHED_FIFO` mają zawsze pierwszeństwo przed procesami `SCHED_RR` o tym samym priorytecie.

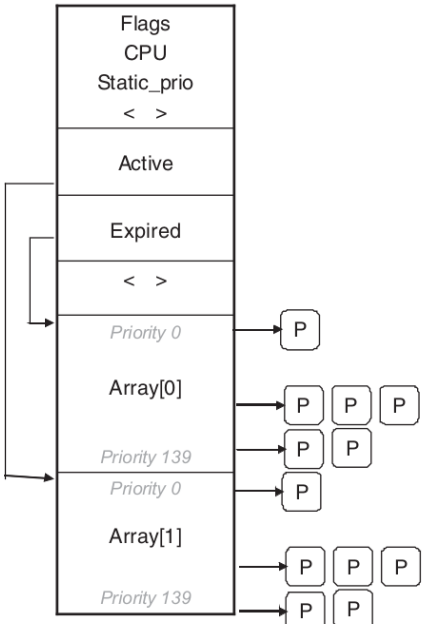
RHEL/CentOS/Fedora ograniczają maksymalne zużycie CPU dla wątków czasu rzeczywistego do 95%; zob. Real Time Scheduler Throttling

Priorytety dynamiczne w planiście O(1)

- klasa `SCHED_OTHER` - standardowe procesy użytkownika o priorytetach dynamicznych z zakresu 100-139
- początkowa wartość priorytetu dynamicznego jest określana przez wartość *poziomu uprzejmości*
- poziomy uprzejmości (*nice*) od -20 do +19 są mapowane na wartości od 100 do 139, domyślna wartość uprzejmości 0 to priorytet 120
- wartość priorytetu procesu jest zmieniana w granicach ± 5 w zależności od interaktywności zadania (stosunek czasu jaki zadanie spędza w zawieszeniu do czasu aktywności jest miarą uzależnienia zadania od operacji wejścia-wyjścia)
- wielkość kwantu czasu procesu zależy od priorytetu dynamicznego, procesy o wyższym priorytecie otrzymują dłuższy kwant czasu
- domyślnie proces otrzymuje kwant 100 ms. Ten kwant może być zwiększony do 800 ms (100) lub zmniejszony do 5 ms (139).
  Nowy proces dziedziczy kwant po swoim rodzicu.

Planista O(1)

- struktura zarządzania aktywnymi zadaniami (*runqueue*) zawiera dwie tablice o rozmiarze 140: zadania aktywne (*active*) i wygasłe (*expired*)
- każdy element tych tablic wskazuje na początek kolejki zadań o wspólnym priorytecie



(a) Per CPU runqueue in the Linux O(1) scheduler

- planista wybiera pierwsze zadanie z listy aktywnych o najwyższym priorytecie
- po wyczerpaniu kwantu czasu zadanie jest umieszczane na liście wygasłych (możliwa jest zmiana priorytetu)
- jeśli zadanie zostanie zablokowane przed upływem kwantu czasu to powraca do listy aktywnych z odpowiednio skróconym kwantem
- gdy wszystkie listy aktywnych zadań się wyczerpią, następuje zamiana tablic, zadania wygasłe stają się znów aktywne
- wybór zadania odbywa się w stałym czasie $O(1)$, niezależnie od liczby zadań
- algorytm gwarantuje, że wszystkie zadania, nawet o niskim priorytecie, mają szansę wykonania
- główną wadą algorytmu jest skomplikowana heurystyka ustalania priorytetów dynamicznych, promująca zadania interaktywne, kosztem zadań obliczeniowych

Co oznacza PRI?

proces/wątek	jądro	"top/htop"	"ps -eo pri"	"ps -eo rtprio"	"ps -el"	"nice"
migration	99	RT	139	99	-40	-
kworker	100	0	39	-	60	-20
bash	120	20	19	-	80	0
khugepaged	139	39	0	-	99	19

	priorytet wg jądra $\geqslant 100$	priorytet wg jądra < 100
top/htop	PRI='priorytet wg jądra' - 100	PRI=RT
ps -eo pri	PRI=139 - 'priorytet wg jądra'	PRI='priorytet wg jądra' + 40
ps -el	PRI='priorytet wg jądra' - 40	PRI='priorytet wg jądra' - 139

Obecne klasy szeregowania w jądrze Linux

Klasy zadań objętych wspólną polityką przydziału CPU:

- czasu rzeczywistego (*real time*): `SCHED_RR`, `SCHED_FIFO`
- sprawiedliwa (*fair*): `SCHED_NORMAL` (dawniej `SCHED_OTHER`) większość procesów użytkownika, zarządzana przez planistę CFS
- `SCHED_BATCH` dla zadań obciążających CPU nieinteraktywnych o niskim priorytecie
- bezczynna: `SCHED_IDLE` dla procesów o bardzo niskim priorytecie (niżej niż +19), które mogą być wykonywane tylko wtedy, gdy system jest bezczynny
- terminowa (*deadline*): `SCHED_DEADLINE` dla zadań o ścisłych wymaganiach czasowych (systemy czasu rzeczywistego)

Przykład: polityka szeregowania procesów w Linuxie

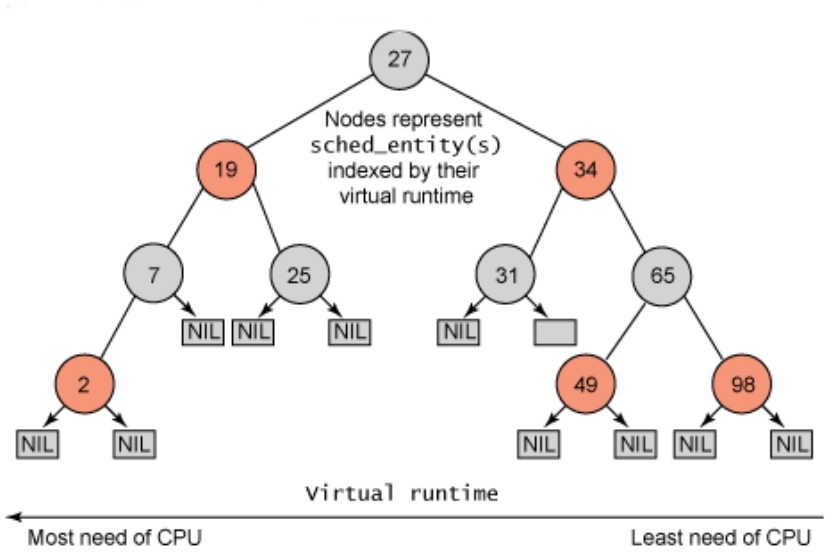
```
$ ps -e -o s,pid,policy,pri,ni,rtprio,command
S     PID POL PRI  NI RTPRIO COMMAND
S       1 TS   19   0      - /sbin/init splash
S       2 TS   19   0      - [kthreadd]
I       4 TS   39 -20      - [kworker/R-rcu_g]
...
S      19 FF  139   -     99 [migration/0]
S    4565 B     4  15      0 /usr/libexec/packagekitd
S    5670 TS    7  12      - /usr/bin/python3 /usr/lib/pop-transition/service.py
R   13056 IDL   0   -      0 /usr/libexec/tracker-miner-fs-3
S   30080 TS   13   6      - /usr/bin/python3 /usr/bin/gnome-terminal --wait
R  464202 TS   19   0      - ps -e -o s,pid,policy,pri,ni,rtprio,command
```

Polecenie chrt ustawia lub odczytuje atrybuty szeregowania

```
$ chrt -p 19
pid 19's current scheduling policy: SCHED_FIFO
pid 19's current scheduling priority: 99
```

Planista kompletnie sprawiedliwy CFS

- struktura zarządzania zadaniami (`runqueue`) jest zorganizowana w postaci drzewa czerwono-czarnego
- zadania w drzewie są uporządkowane w kolejności względem ilości czasu wirtualnego przez jaki zajmowały CPU (wartość `vruntime`)



http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

Planista CFS

- planista zawsze wybiera do wykonania zadanie o najmniejszym czasie wirtualnym (skrajny węzeł z lewej)
- okresowo planista aktualizuje wirtualny czas działającego zadania porównując z wartością czasu kolejnego zadania znajdującego się w drzewie
- jeżeli w drzewie znajduje się węzeł o nizszym czasie to nastepuje wywłaszczenie i wykonywane zadanie zostaje umieszczone ponownie w drzewie we właściwym miejscu
- różnice w priorytetach zadań oraz uprzejmości (`nice`) są odzwierciedlone w postaci szybkości upływu wirtualnego czasu
- dla zadań o niskim priorytecie czas płynie szybciej, więc szybciej ulegają wywłaszczeniu, na korzyść zadań o wyższym priorytecie

Komunikacja międzyprocesowa (IPC)

Inter-Process Communication

- pamięć współdzielona (*Shared Memory*)
- potoki (*pipes*) - jednokierunkowe przesyłanie strumieni danych
  - potoki nienazwane (*unnamed pipes*) - bezpośrednie połaczenie (`stdout-stdin`)
  - potoki nazwane (*named pipes*) - powiązanie przez pliki `FIFO`
- sygnały
- gniazda (*sockets*)
- semafory
- pliki
- kolejki komunikatów (*Message Queues*) - przesyłanie asynchroniczne komunikatów za pomocą buforowanych kolejek
- zdalne wywołanie procedur (*remote procedure call*, RPC)

Problemy IPC

- wyścigi (*race conditions*) - wynik działania zależy od nieprzewidywalnego przeplatania się operacji wykonywanych przez różne procesy
- zakleszczenie (*deadlock*) - dwa (lub więcej) procesy czekają na zasoby, które są trzymane przez siebie nawzajem, co powoduje, że żaden z procesów nie może kontynuować pracy
- zablokowanie (*livelock*) procesy ciągle zmieniają swoje stany w odpowiedzi na działania innych procesów, nie wykonując jednak żadnej efektywnej pracy
- głodzenie (*starvation*) - proces nie otrzymuje zasobów, których potrzebuje do kontynuowania pracy, z powodu nieustannego przydzielania tych zasobów innym procesom
- problem odwróconych priorytetów (*priority inversion*) - proces o wyższym priorytecie jest blokowany przez proces o niższym priorytecie, który trzyma zasób

Sygnały

Sygnały są krótkimi wiadomościami, które można wysyłać do procesu, wątku lub grupy procesów.

- z każdym sygnałem jest związana jego nazwa i numer, np. `SIGINT` (2)
- są programowymi odpowiednikami przerwań sprzętowych, obsługa sygnału odbywa się w ramach procesu w trybie użytkownika
- informują proces lub wątek o wystąpieniu określonego zdarzenia
- zmuszają proces do przerwania aktualnej pracy i wykonania procedury obsługującej sygnał
- jeżeli proces nie zarejestrował funkcji obsługi sygnału to uruchamiana jest domyślna funkcja obsługi (zazwyczaj zamknięcie procesu)
- wyjątki procesora powodują dostarczenie sygnału do procesu (SIGBUS, SIGEMT, SIGFPE, SIGILL, SIGSEGV, SIGTRAP)
- zdefiniowane w normie POSIX (dostepne w Unix, Linux, macOS X, w ograniczony sposób w Windows)

Sygnały

Jądro używa sygnałów, żeby powiadomić proces (zadanie) o wystąpieniu błędów lub asynchronicznych zdarzeń.

Przykłady:

- naruszenie ochrony pamięci (SIGSEGV, *segmentation violation signal*)
- przerwanie klawiaturowe Ctrl-C (SIGINT), Ctrl-Z (SIGSTOP)
- funkcje systemowe `kill()`, `pthread_kill()`
- funkcje biblioteki standardowej C (`signal.h`): `signal()`, `raise()`
- polecenie `kill`

Obsługa sygnału

Działania podejmowane przez proces po otrzymaniu sygnału:

1. wyłapanie sygnału i uruchomienie procedury obsługi
2. zignorowanie sygnału
3. działanie domyślne, gdy nie zdefiniowano procedury obsługi
   - przerwanie (`Term`) - proces jest niszczony
   - zrzut (`Core`) - tworzony jest plik `core`[15] i proces jest niszczony
   - ignorowanie (`Ign`) - sygnał jest pomijany
   - zatrzymanie (`Stop`) - proces jest zatrzymywany (przechodzi w stan TASK_STOPPED)
   - kontynuacja (`Cont`) - proces ze stanu TASK_STOPPED przechodzi w stan TASK_RUNNING

[15] Także `core.$$` lub `/var/lib/systemd/coredump/core.sar.1000...`

Standardowe sygnały

Numer	Nazwa	Domyślnie	Opis
1	**SIGHUP**	`Term`	zamknięcie terminala kontrolującego
2	**SIGINT**	`Term`	przerwanie z klawiatury
3	SIGQUIT	`Core`	zamknięcie z klawiatury
4	SIGILL	`Core`	nieprawidłowa instrukcja
5	SIGTRAP	`Core`	pułapka dla programu śledzącego
6	SIGABRT, SIGIOT	`Core`	nieprawidłowe zakończenie
7	SIGBUS	`Core`	błąd szyny (np. niepoprawny adres fizyczny)
8	SIGFPE	`Core`	wyjątek operacji zmiennoprzecinkowej
9	**SIGKILL**	`Term`	wymuszone zakończenie procesu
10	SIGUSR1	`Term`	do wykorzystania przez proces
11	**SIGSEGV**	`Core`	nieprawidłowe odwołanie do pamięci
12	SIGUSR2	`Term`	do wykorzystania przez proces
13	SIGPIPE	`Term`	zapis do potoku, którego nikt nie czyta
14	SIGALRM	`Term`	alarm upłynięcia czasu z funkcji `alarm()`
15	**SIGTERM**	`Term`	zakończenie procesu

Zobacz: `man 7 signal` (`/usr/include/asm/signal.h`)

Numer	Nazwa	Domyślnie	Opis
16	SIGSTKFLT	`Term`	błąd stosu
17	SIGCHLD	`Ign`	proces potomny zakończony lub zatrzymany
18	**SIGCONT**	`Cont`	wznawianie zatrzymanego procesu
19	**SIGSTOP**	`Stop`	zatrzymanie wykonywania procesu
20	**SIGTSTP**	`Stop`	zatrzymanie procesu z terminala (`Ctrl+Z`)
21	SIGTTIN	`Stop`	program w tle żąda wejścia z terminala
22	SIGTTOU	`Stop`	program w tle żąda wyjścia na terminal
23	SIGURG	`Ign`	pilne dane w gnieździe
24	SIGXCPU	`Core`	przekroczenie limitu czasu procesora
25	SIGXFSZ	`Core`	przekroczenie limitu wielkości pliku
26	SIGVTALRM	`Term`	alarm zegara wirtualnego
27	SIGPROF	`Term`	alarm zegara programu profilującego
28	SIGWINCH	`Ign`	zmiana rozmiaru okna (terminala)
29	SIGIO, SIGPOL	`Term`	gotowość do operacji I/O
30	SIGPWR, SIGLOST	`Term`	awaria źródła zasilania
31	SIGUNUSED, SIGSYS	`Core`	nie używane (dla wstecznej kompatybilności)

Sygnały `SIGKILL` i `SIGSTOP` nie mogą zostać przechwycone, zablokowane ani ignorowane

Ryzyka sygnałów

- sygnały są asynchroniczne, obsługa sygnałów może powodować wystąpienie wyścigu (*race conditions*)
- inny sygnał może zostać dostarczony do procesu podczas wykonywania procedury obsługi sygnału
- sygnały mogą spowodować przerwanie trwającego (np. zablokowanego) wywołania systemowego
- procedury obsługi sygnałów powinny być napisane w sposób nie powodujący niepożądanych skutków ubocznych (np. zmiana globalnych zmiennych procesu, użycie funkcji niebezpiecznych z punktu widzenia asynchronicznego wykonania)
- z punktu widzenia jądra wykonanie kodu obsługi sygnału jest dokładnie takie samo, jak wykonanie dowolnego innego kodu przestrzeni użytkownika (odpowiedzialność programisty)

Maskowanie sygnałów

- proces ma możliwość zamaskować (zablokować) czasowo wybrane sygnały w celu wykonania krytycznych operacji (maska jest zasobem procesu)
- sygnały standardowe nie są kolejkowane, tylko jeden sygnał danego typu dociera po odblokowaniu
- przy odblokowaniu kilku sygnałów różnego typu kolejność obsługi jest nieokreslona
- proces może zablokować się w oczekiwaniu na sygnał (obsługa synchroniczna)

Sygnały czasu rzeczywistego w Linux

Linux używa 31 standardowych sygnałów i do 32 sygnałów czasu rzeczywistego (od 32 do 64)

Sygnały czasu rzeczywistego:

- nie mają zdefiniowanego zastosowania, programiści mogą ich używać według własnego uznania
- domyślne działanie to przerwanie wykonywania procesu
- moga być powiązane z danymi dostarczanymi do procesu (liczba lub wskaźnik)
- umożliwiają odczytanie identyfikatora procesu i użytkownika nadawcy
- są kolejkowanie, kolejność dostarczenia jest zagwarantowana, im niższy numer sygnału tym wyższy priorytet

Wysyłanie sygnałów z poziomu powłoki

- wysyłanie sygnałów do procesów
  `kill -SIGNAL PID`
- zatrzymywanie i wznawianie procesów
  `kill -STOP PID`
  `kill -CONT PID`
- zamknięcie procesu (domyślnie sygnał `SIGTERM`)
  `kill PID`
- zabicie procesu
  `pkill -9 top`
- zabicie wszystkiego co się da
  `kill -1 -9`

Sygnał `SIGKILL/9` nie trafia do wskazanego procesu, ale do procesu `init/systemd`, dlatego nie może być przechwycony ani zignorowany przez proces docelowy.

Zasoby podzielne i niepodzielne

Zasoby niepodzielne: zasoby systemowe, które nie mogą być współdzielone ani używane równocześnie przez więcej niż jeden proces w danym momencie

- większość urządzeń zewnętrznych, pliki do zapisu, obszary pamięci współdzielonej, które mogą być modyfikowane przez procesy (np. zmienne globalne), itp.

Zasoby podzielne: mogą być współdzielone przez wiele procesów jednocześnie bez ryzyka powstania błędów lub niespójności danych

- pliki tylko do czytania, obszary pamięci, gdzie znajdują się (dzielone) biblioteki, kody programów, potoki nazwane, itp.

Wyścigi i sekcje krytyczne

- wspólnie użytkowane zasoby wymagają ochrony przed współbieżnym dostępem, gdyż może to prowadzić do niespójności danych
- sytuacje, w których procesy (wątki) czytają oraz modyfikują pewne dzielone dane i rezultat końcowy zależy od tego, kiedy dokładnie każdy z tych procesów (wątków) będzie je czytał/modyfikował, nazywamy **wyścigiem** (*race condition*), hazardem lub przeplotem operacji.
- **sekcja krytyczna** (*critical section*) - część programu, w której następuje czytanie i modyfikowanie dzielonych danych
- aby zapobiec wyścigom musimy zapewnić **wzajemne wykluczenie**

Przykład: zmiana zmiennej współdzielonej

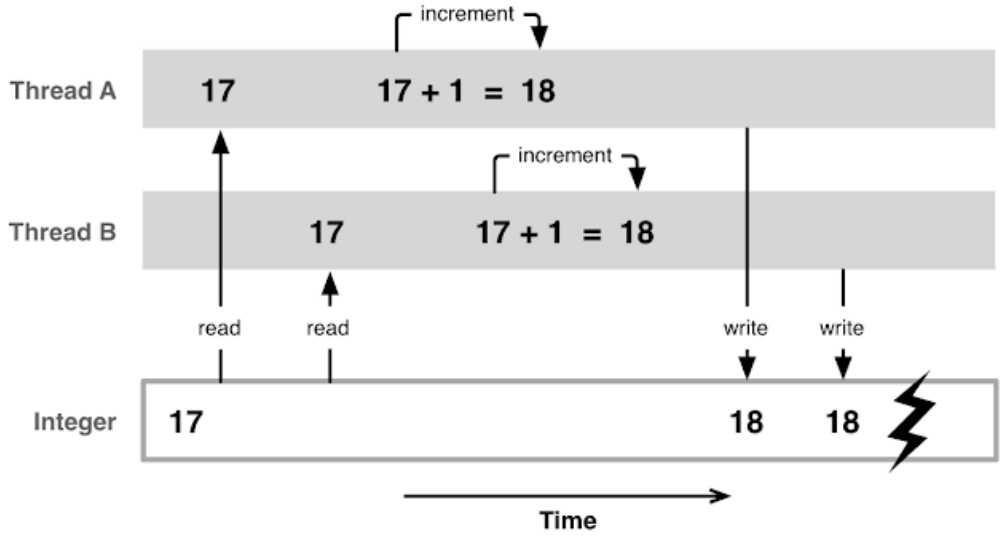
Dwa wątki współdzielą zmienną globalną typu całkowitego `n` i wykonują kod: `n++`, co oznacza

1. `pobierz aktualną wartość n i umieść ją w rejestrze`
2. `dodaj 1 do wartości w rejestrze`
3. `zapisz nową wartość n do pamięci`

```
Wątek A                          Wątek B
------------------------         ------------------------
pobierz n (17)                    ...
zwiększ wartość n (17->18)         ...
zapisz n (18)                     ...
...                              pobierz n (18)
...                              zwiększ wartość n (18->19)
...                              zapisz n (19)
```

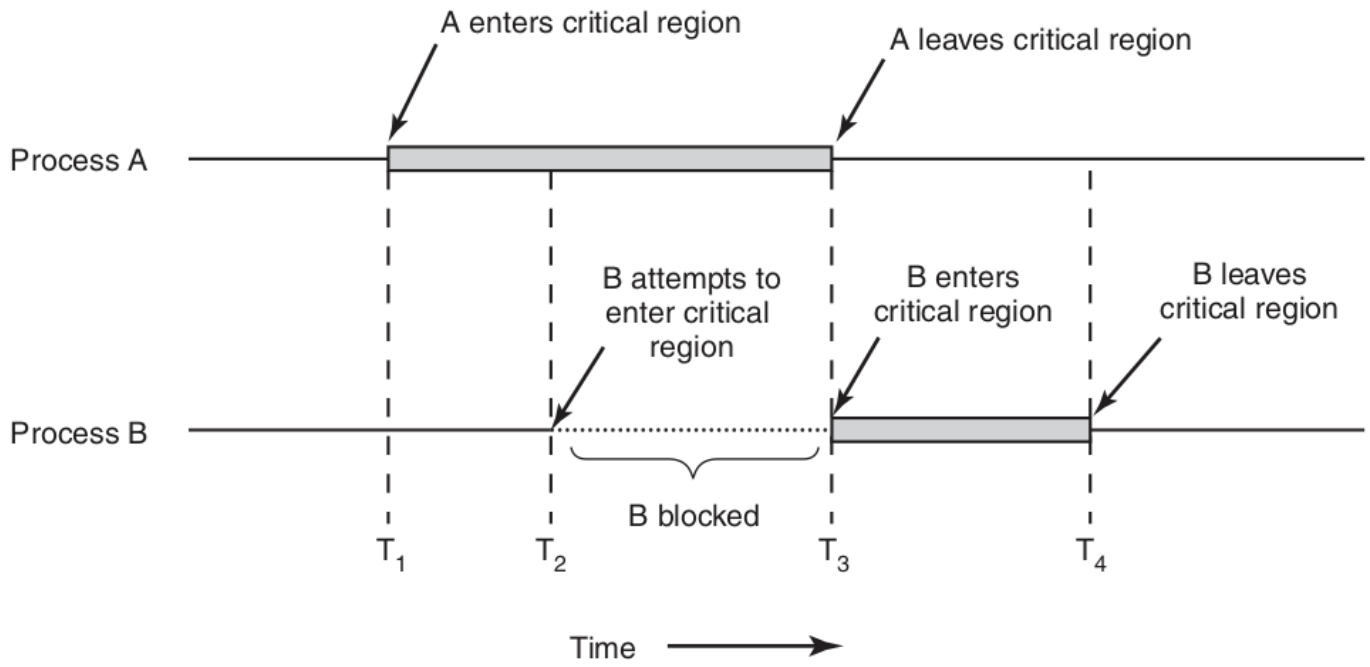
```
-----------------------------------------------------------
Wątek A                          Wątek B
------------------------         ------------------------
pobierz n (17)                    ...
...                              pobierz n (17)
zwiększ wartość n (17->18)         ...
...                              zwiększ wartość n (17->18)
zapisz n (18)                     ...
...                              zapisz n (18)
```

Wyścigi



- zwiększanie wartości zmiennej globalnej powinno być wykonywane w sposób niepodzielny (atomowy)
- **operacje atomowe** (*atomic operation*) to niepodzielna operacja, która jest wykonywana w całości albo wcale, zapewnia poprawne wykonanie operacji na zasobach współdzielonych

Wzajemne wykluczenie



Wzajemne wykluczenie (*mutual exclusion*) polega na zapewnieniu takich warunków działania systemu, by tylko jeden proces mógł korzystać z zasobów niepodzielnych, tj. tylko jeden proces mógł przebywać w sekcji krytycznej

Wymagania dotyczące wzajemnego wykluczenia

1. żadne dwa procesy nie mogą przebywać jednocześnie w swojej sekcji krytycznej
2. nie można czynić żadnych założeń, co do szybkości procesów i liczby procesorów
3. żaden proces wykonujący się poza sekcją krytyczną nie może blokować innych procesów
4. żaden proces nie może czekać w nieskończoność na wejście do swojej sekcji krytycznej
5. jeśli żaden proces nie przebywa w sekcji krytycznej, to dowolny proces, który chce do niej wejść, powinien otrzymać na to natychmiastową zgodę
6. proces pozostaje w sekcji krytycznej przez skończony czas

Jak zapewnić wzajemne wykluczanie?

- wsparcie sprzętowe dla operacji niepodzielnych
  - wyłączanie przerwań
  - atomowe instrukcje procesora: testuj i załóż rygiel TSL (*Test and Set Lock*), zamiana XCHG (*exchange*)
- aktywne czekanie
  - zmienne blokujące
  - algorytm Petersona
- bez aktywnego czekania (blokowanie procesów)
  - semafory, muteksy,
  - monitory
- inne: bariery, blokady wirujące (*spin locks*), sekcje krytyczne języków wysokiego poziomu, synchronizacja za pomocą komunikatów

Wyłączanie przerwań

Proces wyłącza przerwania (także zegarowe), kiedy wchodzi do swojej sekcji krytycznej i włącza je, kiedy ją opuszcza.

Wady:

- proces użytkownika nie powinien mieć możliwości wyłączania przerwań, technika przydatna tylko dla procesów jądra
- w systemach wieloprocesorowych wyłączanie przerwań dotyczy tylko jednego procesora

Zalety:

- łatwy sposób modyfikowania struktur jądra w spójny sposób

Aktywne czekanie (*busy waiting*) - wątek lub proces sprawdza w pętli, czy warunek synchronizacji został spełniony

Zalety:

- proste w implementacji
- niskie opóźnienia w dostępie do sekcji krytycznej, przydatne, gdy oczekiwanie na zdarzenie jest krótkie i nieopłacalne jest przełączanie kontekstu

Wady:

- procesor jest w pełni obciążony wykonaniem pętli, może być nieefektywne w kontekście zużycia zasobów procesora

Wzajemne wykluczenie z aktywnym czekaniem

- zmienne blokujące (*lock variables*) - proste w użyciu ale bez operacji niepodzielnych wciąż podatne na wyścigi
- ścisła zmienność (*strict alternation*) - proste rozwiązanie dla dwóch wątków
- algorytm Petersona - programowe rozwiązanie wzajemnego wykluczania

Ścisła zmienność

Procesy uzyskują dostęp naprzemiennie, wartość zmiennej `turn` określa, który proces może wejść do sekcji krytycznej

proces A

```
while (TRUE) {
  while (turn != 0) /* wait */;
  critical section();
  turn = 1;
  noncritical_section();
}
```

proces B

```
while (TRUE) {
  while (turn != 1)  /* wait */;
  critical section();
  turn = 0;
  noncritical_section();
}
```

Problem:

- wykonanie sekcji krytycznej jest naprzemienne - tempo zalezy od wolniejszego procesu
- w przypadku awarii jednego z procesów drugi proces może zostać zablokowany na zawsze

Algorytm Petersona dla dwóch procesów

```
#define FALSE 0
#define TRUE  1
#define N     2                       /* number of processes */
int turn;                             /* whose turn is it? */
int interested[N];                    /* all values initially zero (FALSE) */

void enter_region (int process)       /* process: who is entering (0 or 1) */
{
  int other = 1 - process;            /* number the opposite of process */
  interested[process] = TRUE;         /* show that you are interested */
  turn = process;                     /* set flag */
  while (turn == process && interested[other] == TRUE);
}

void leave_region(int process)        /* who is leaving (0 or 1) */
  interested[process] == FALSE;       /* indicate departure from critical region */
}
```

Algorytm Petersona

- spełnia wszystkie wymogi wzajemnego wykluczania
- globalna tablica `interested[]` wskazuje, czy dany proces chce wejść do sekcji krytycznej
- zmienna `turn` wskazuje, który proces ma pierwszeństwo wejścia do sekcji krytycznej
- zapobiega wzajemnemu blokowaniu się procesów
- algorytm może być uogólniony na dowolna liczbę procesów

Instrukcja maszynowa TSL

- rozkaz `tsl` kopiuje wartość zmiennej blokującej do rejestru i jednocześnie zmienia jej wartość, jest to gwarantowana sprzętowo **operacja niepodzielna**
- na czas wykonywania instrukcji procesor blokuje szynę pamięci aby żaden inny proces nie miał do niej dostępu

Blokada wirująca (*spin lock*) z użyciem instrukcji TSL:

```
enter_region:
  tsl register,lock   | copy lock to register and set lock to 1
  cmp register,#0     | was lock zero?
  jne enter_region    | if it was not zero, lock was set, so loop
  ret                 | return to caller; critical region entered

leave_region:
  mov lock,#0         | store a 0 in lock
  ret                 | return to caller
```

Wzajemne wykluczenie instrukcją TSL lub XCHG

- eliminuje problem wyścigu przy dostępie do zmiennej blokującej
- nadaje się dla dowolnej liczby procesów (maszyny jedno- i wieloprocesorowe (SMP))
- proste i łatwe w weryfikacji
- nadaje się do kontroli wielu sekcji krytycznych, każda z własną zmienną blokującą

Wady:

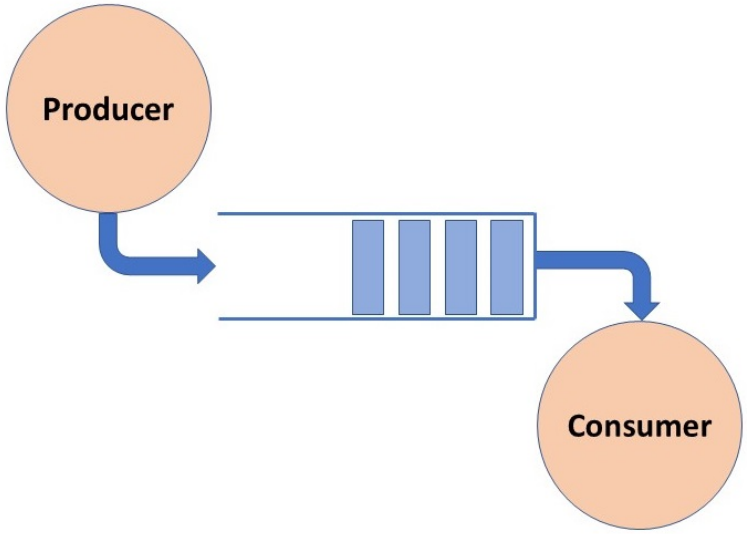
- wykorzystuje się aktywne czekanie a to marnuje czas CPU
- możliwość wystąpienia głodzenia (*starvation*), gdy procesy o wysokim priorytecie ciągle przejmują dostęp do sekcji krytycznej
- możliwość wystąpienia zakleszczeń
- problem odwróconego priorytetu - proces aktywnie czekający o wyższym priorytecie, może nie dopuścić do przełączenia CPU na proces o niższym priorytecie przebywającym w sekcji krytycznej, powodując zakleszczenie

Wzajemne wykluczenie bez aktywnego czekania

Można uniknąć marnowania czasu jednostki centralnej na jałowe sprawdzanie warunku jeśli system udostępnia dwie operacje do synchronizacji pracy procesów: `SLEEP` oraz `WAKEUP`.

- `SLEEP` – odwołanie do systemu, które powoduje zablokowanie procesu wywołującego tę funkcję do czasu, aż nie nadejdzie sygnał budzenia
- `WAKEUP` – odwołanie do systemu służące do budzenia procesów

Problem producenta i konsumenta (ograniczonego buforu)



- proces producenta nie może tworzyć przedmiotu, jeśli współdzielony bufor jest pełny
- proces konsumenta nie może zużywać przedmiotu, jeśli współdzielony bufor jest pusty
- dostęp do współdzielonego bufora musi się wzajemnie wykluczać

Problem producenta i konsumenta: przykład rozwiązania

```
#define N     100                       /* number of slots in the buffer */

int count = 0;                          /* number of items in the buffer */

void producer(void)
{
  int item;

  while (TRUE) {                          /* repeat forever */
    produce_item(&item);                  /* generate next item */
    if (count == N) sleep();              /* if buffer is full go to sleep */
    enter_item(item);                     /* put item in buffer */
    count = count + 1;                    /* increment number of items in buffer */
    if (count == N) wakeup(consumer);     /* was buffer empty? */
  }
}
```

```
void consumer(void)
{
  int item;

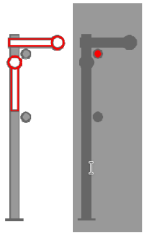
  while (TRUE) {                          /* repeat forever */
    if (count == 0) sleep();              /* if buffer is empty go to sleep */
    remove_item(&item);                   /* take item out of buffer */
    count = count - 1;                    /* decrement number of items in buffer */
    if (count == N-1) wakeup(producer); /* was buffer full? */
    consume_item(item);                   /* print item */
  }
}
```

Uwaga! Zmienna count odzwierciedlająca liczbę przedmiotów w buforze, nie jest chroniona, dostęp do niej nie jest niepodzielny. W przypadku wyścigu może dojść do zablokowania producenta i konsumenta

Semafory

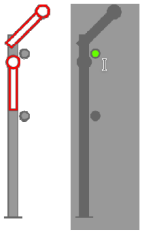
Semafor to specjalna zmienna całkowita, na której są wykonywane operacje `DOWN` i `UP`[16]

`SLEEP` ⟶ `DOWN`



Stój

`WAKEUP` ⟶ `UP`



Wolna droga

- UP (podniesienie, sygnalizacja) - zwiększa wartość semafora o 1 i budzi procesy oczekujące na semaforze
- DOWN (opuszczenie, oczekiwanie) - zmniejsza wartość semafora o 1, jeśli wartość semafora jest większa od 0, w przeciwnym razie blokuje proces wywołujący tę operację

[16] Oryginalne nazwy operacji używane przez Dijkstra to `P` (*Proberen*, sprawdź) i `V` (*Vorhogen*, podnieś). W literaturze używa się też nazw `wait` i `signal`.

- Wartość semafora oznacza liczbę procesów, które mogą wejść do sekcji krytycznej
- Operacje `UP` i `DOWN` muszą być **niepodzielne** (atomowe).
  Realizowane jest to w postaci wywołań systemowych blokujących przerwania i instrukcją TSL dla systemów wieloprocesorowych
- semafor jest wysokopoziomową abstrakcją opartą na niskopoziomowych mechanizmach elementarnych, które zapewniają niepodzielność i dostarczają mechanizmów wstrzymywania

Wady:

- wstrzymywanie i wznawianie wymaga przełączania kontekstu i zmian w kolejkach modułu szeregowania i kolejek wątków wstrzymanych; operacje na semaforze są **powolne**
- trudne do debugowania

Operacje UP i DOWN na semaforze

Definicje operacji opuszczania (DOWN) i podnoszenia (UP) semafora:

```
void DOWN(semaphore) {
   if (semaphore > 0 ) {
       semaphore = semaphore - 1;
       continue();
   } else {
       sleep();  /* add the process to semaphore queue and block it */
   }
}

void UP(semaphore) {
   semaphore = semaphore + 1;
   wakeup(); /* wakeup all blocked processes and make them runnable */
             /* scheduler selects one to run */
}
```

Problem producenta i konsumenta z semaforami

```
#define N     100                    /* number of slots in the buffer */
typdef int semaphore;                /* semaphores are a special kind of int */
semaphore mutex = 1;                 /* controles access to critical section */
semaphore empty = N;                 /* count empty buffer slots */
semaphore full  = 0;                 /* count full buffer slots */

void producer(void)
{
  int item;
  while (TRUE) {                     /* TRUE is the constant 1 */
    produce_item(&item);             /* generate next item */
    down(&empty);                    /* decrement empty count */
    down(&mutex);                    /* enter critical region */
    enter_item(&item);               /* put new item in buffer */
    up(&mutex);                      /* leave critical region */
    up(&full);                       /* increment count of full slots */
  }
}
```

```
void consumer(void)
{
  int item;
  while (TRUE) {                    /* infinite loop */
    down(&full);                    /* decrement full count */
    down(&mutex);                   /* enter critical region */
    remove_item(&item);             /* take item from buffer */
    up(&mutex);                     /* leave critical area */
    up(&empty);                     /* increment count of empty slots */
    consume_item(&item)             /* do sth with the item */
  }
}
```

- semafor `mutex` przyjmuje wartości 0 lub 1, jest używany przez dwa lub więcej procesów, aby zapewnić że tylko jeden z nich może w danej chwili przebywać w swojej sekcji krytycznej
- semafory `empty` oraz `full` służą do **synchronizacji**, zapewniają, że zdarzenia będą zachodziły we właściwej kolejności

Rodzaje semaforów

- **semafor zliczający (ogólny)** - wartość całkowita służąca do sygnalizacji między procesami (synchronizacja procesów, zarzadzanie ograniczonymi zasobami)
- **semafor binarny** - semafor, który przyjmuje tylko wartości 0 lub 1
- **mutex** - semafor binarny, który zapewnia wzajemne wykluczanie procesów (ang. *mutual exclusion*), proces zamykający mutex jest tym samym, który go odblokuje

Zarówno w semaforach binarnych jak i zliczających jest utrzymywana kolejka procesów oczekujących na semafor.

- **semafor silny** (Dijkstry) - kolejka FIFO, procesy oczekujące na semafor są budzone w kolejności ich przybycia(sprawiedliwy, zapobiega głodzeniu)
- **słaby semafor** - procesy mogą być budzone w dowolnej kolejności

Monitory

Monitor to mechanizm synchronizacji zapewniający bezpieczny dostęp do wspólnych danych poprzez enkapsulację danych i operacji w jednym module, ułatwia programowanie współbieżne poprzez ukrycie szczegółów synchronizacji (dostępne w Java, C#).

Główne cechy monitora

- Monitor **grupuje zmienne współdzielone i procedury** operujące na tych danych w jednej strukturze. Dostęp do danych jest możliwy tylko przez procedury monitora.
- W danym momencie **tylko jeden proces** może wykonywać procedurę monitora. Gwarancja wzajemnego wykluczania.
- Monitory używają **zmiennych warunkowych** (*condition variables*) do blokowania procesów
- Procesy mogą **czekać** (`wait(warunek)`) na spełnienie warunku lub **sygnalizować** (`signal(warunek)`) innym procesom, że warunek został spełniony

Przykład monitora w C#

```
using System.Threading;

class Bufor {
    private readonly object _lock = new object();
    private int[] _dane = new int[3];
    private int _licznik = 0;

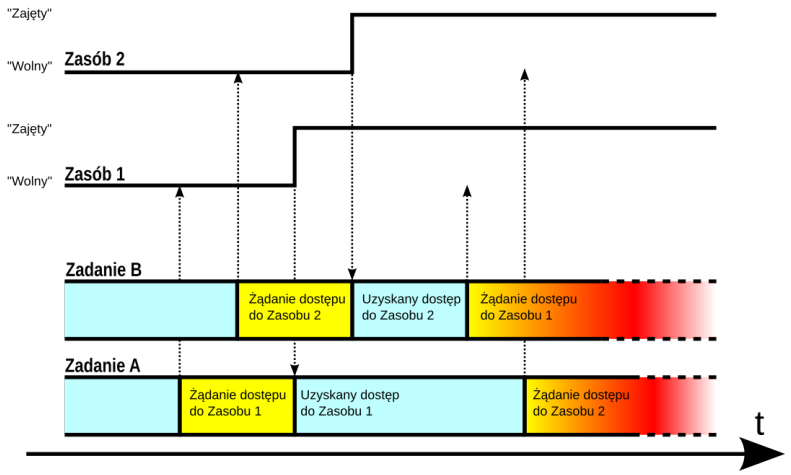
    public void Produkuj(int element) {
        lock (_lock) {
            while (_licznik == _dane.Length) Monitor.Wait(_lock); // Czekaj, jeśli bufor pełny
            _dane[_licznik++] = element;
            Monitor.Pulse(_lock); // Obudź konsumenta
        }
    }
    public int Konsumuj() {
        lock (_lock) {
            while (_licznik == 0) Monitor.Wait(_lock); // Czekaj, jeśli bufor pusty
            int element = _dane[--_licznik];
            Monitor.Pulse(_lock); // Obudź producenta
            return element;
        }
    }
}
```

```
class Program {
    static void Main() {
        Bufor bufor = new Bufor();

        // Uruchom producenta
        new Thread(() => {
            for (int i = 0; i < 5; i++) {
                bufor.Produkuj(i);
                Thread.Sleep(300); // Symulacja pracy
            }
        }).Start();

        // Uruchom konsumenta
        new Thread(() => {
            for (int i = 0; i < 5; i++) {
                bufor.Konsumuj();
                Thread.Sleep(500); // Symulacja pracy
            }
        }).Start();
    }
}
```

Zakleszczenia (*deadlocks*, *deadly embraces*)



Zakleszczenie to trwałe zablokowanie zbioru procesów, spowodowane wzajemnym oczekiwaniem na zasoby zajęte przez inne procesy z tego zbioru.

Przykład: proces A otrzymuje dostęp do drukarki, a proces B do dysku, a następnie proces A żąda dostępu do dysku, a proces B do drukarki.

https://pl.wikipedia.org/wiki/Zakleszczenie

Unikanie zakleszczeń

W odróżnieniu innych problemów zarządzania procesami współbieżnymi nie istnieje skuteczne i ogólne rozwiązanie problemu zakleszczeń.

Jak unikać zakleszczeń?

1. Jeśli to tylko możliwe, to **należy unikać stosowania blokad** (używaj operacji atomowych lub struktur danych nie wymagających zakładania blokad).
2. Jeśli blokada jest niezbędna, to **należy stosować blokadę pojedynczą**
3. Jeśli potrzebnych jest kilka blokad należy:
   - zdefiniować (małą) lokalną **hierarchię blokad** (blokowanie w ustalonej kolejności)
   - ustawić limity czasu blokady (`Monitor.TryEnter()` w C#)
   - zastosować **blokowanie stochastyczne** (losowa kolejność blokowania zmniejszy prawdopodobieństwo zakleszczenia)

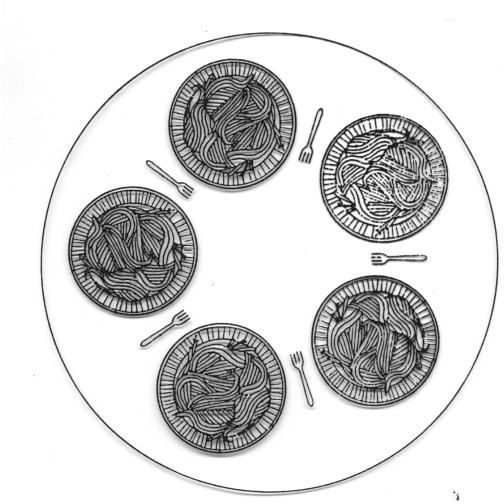
.

Klasyczne problemy komunikacji międzyprocesowej

- problem producenta-konsumenta – modelowanie procesów synchronizacji między procesami (problem ograniczonego buforu)
- problem ucztujących filozofów – modelowania procesów, które współzawodniczą w dostępie do ograniczonych zasobów, np. urządzeń wej/wyj
- problem pisarzy i czytelników – zagadnienie zapewnienia prawidłowego dostępu do bazy danych (w trybie odczytu i zapisu)
- problem śpiącego fryzjera – modelowanie procesów synchronizacji między procesami (problem ograniczonej liczby zasobów usługowych)

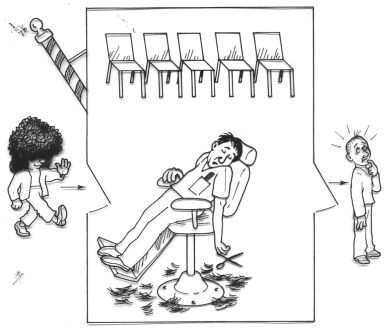
Problem ucztujących filozofów

- Pięciu filozofów siedzi przy okrągłym stole. Każdy filozof może myśleć lub jeść.
- Między sąsiadującycmi filozofami znajduje się jeden widelec. Aby jeść, filozof musi mieć oba widelce znajdujące się po jego lewej i prawej stronie.
- Problem polega na zaprojektowaniu protokołu, który pozwoli filozofom jeść i myśleć bez ryzyka zakleszczenia (gdy każdy filozof trzyma jeden widelec i czeka na drugi) lub głodzenia (gdy jeden lub więcej filozofów nigdy nie dostaje szansy na jedzenie).



Problem śpiącego fryzjera

- W salonie fryzjerskim jest jeden fryzjer, jedno krzesło do strzyżenia i kilka krzeseł dla oczekujących klientów.
- Jeśli nie ma klientów, fryzjer zasypia na swoim krześle.
- Gdy przychodzi klient, budzi fryzjera, jeśli ten śpi. Jeśli fryzjer jest zajęty, a są wolne krzesła dla oczekujących, klient siada i czeka. Jeśli nie ma wolnych krzeseł, klient odchodzi.
- Fryzjer strzyże jednego klienta naraz i po skończeniu strzyżenia sprawdza, czy są jacyś oczekujący klienci. Jeśli tak, woła następnego klienta. W przeciwnym razie zasypia.



Problemy do rozwiązania:

- Zapewnienie wzajemnego wykluczania dostępu do krzesła fryzjera
- Zapobieganie głodzeniu klientów (jeśli ciągle przychodzą nowi klienci, ci którzy już czekają mogą nigdy nie dostaać dostępu do fryzjera)

Problem czytelników i pisarzy

- Obszar danych współdzielonych (pamięć, plik, baza danych) jest użytkowany przez procesy czytające (czytelników) i procesy modyfikujące (pisarzy).
- Dane mogą być jednocześnie odczytywane przez wielu czytelników.
- Dane mogą być modyfikowane tylko przez jednego pisarza w danym czasie.
- Jeśli pisarz modyfikuje bazę danych, żaden czytelnik nie może jej czytać.

Problemy do rozwiązania:

- Zapewnienie wzajemnego wykluczania dostępu do bazy danych
- Zapobieganie głodzeniu pisarzy (jeśli ciągle przychodzą czytelnicy, pisarz może nigdy nie dostać dostępu do bazy danych)

Słownik skrótów

3DNow (*3D NO Waiting*) bez stanów oczekiwania 3D
AC (*accumulator*) akumulator
AGP (*Accelerated Graphics Port*) przyspieszony port graficzny, port akcelerowanej grafiki
ALU (*Arithmetic-Logic Unit*) jednostka arytmetyczno-logiczna
API (*Application Programming Interface*) interfejs programistyczny aplikacji
APIC (*Advanced Programmable Interrupt Controller*) zaawansowany programowalny kontroler przerwań
ATA (*AT Attachement*) końcówka AT
ASCII (*American Standard Code for Information Interchange*) amerykański standard kodowania na potrzeby wymiany informacji
BIOS (*Basic Input/Output System*) podstawowy system wejścia/wyjścia
BSD (*Berkeley Software Distribution*) dystrybucja oprogramowania z Berkeley
Btrfs (*B-tree File System*) system plików oparty o B-drzewo
CD-ROM (*Compact Disk Read-Only Memory*) pamięć tylko do odczytu na płycie kompaktowej
CHS (*Cylinder/Head/Sector*) cylinder/głowica/sektor
CFS (*Completely Fair Scheduler*) planista całkowicie sprawiedliwy
CIFS (*Common Internet File System*) powszechny internetowy system plików
CISC (*Complex Instruction Set Computer*) komputer o pełnej liście rozkazów
CLI (*Command Line Interface*) interfejs wiersza poleceń
COW (*Copy On Write*) kopiowanie przy zapisie
CPU (*Central Processing Unit*) jednostka centralna
CRC (*cyclic redundancy check*) cykliczna kontrola nadmiarowa
DAC (*discretionary access control*) uznaniowa kontrola dostępu

DDR (*Double Data Rate*) podwójna szybkość transmisji danych
DMA (*Direct Memory Access*) bezpośredni dostęp do pamięci
DMI (*Direct Media Interface*) bezpośredni interfejs multimedialny, magistrala łącząca mostek północny z południowym
DRAM (*Dynamic RAM*) dynamiczna pamięć RAM
EDVAC (*Electronic Discrete Variable Computer*) elektroniczny komputer automatyczny o dyskretnych zmiennych
EEPROM (*Electrically Erasable and Programmable ROM*) elektrycznie wymazywalna i programowalna pamięć stała
EIDE (*Extended Integrated Device Electronics*) ulepszona zintegrowana elektronika napędów
EISA (*Extended Industry Standard Architecture*) rozszerzona standardowa architektura przemysłowa
EMS (*Expanded Memory Specification*) specyfikacja rozszerzonej pamięci
EPROM (*Erasable and Programmable ROM*) wymazywalna i programowalna pamięć stała
EXT2 (*Second Extended File System*) drugi rozszerzony system plików
EXT3 (*Third Extended File System*) trzeci rozszerzony system plików
EXT4 (*Fourth Extended File System*) czwarty rozszerzony system plików
FAT (*File Allocation Table*) tablica alokacji plików
FCFS (*First-Come, First-Served*) pierwszy nadszedł, pierwszy obsłużony
FDD (*Floppy Disk Drive*) stacja dyskietek
FLOPS (*Floating-point operations per second*) liczba operacji zmiennopozycyjnych na sekundę
FIFO (*First In, First Out*) pierwszy nadszedł, pierwszy obsłużony
FSB (*Front System Bus*) przednia magistrala systemowa
FSF (*Free Software Fundation*) Fundacja Wolnego Oprogramowania
GDT (*Global Descriptor Table*) globalna tablica deskryptorów

GNU (*Gnu is Not Unix*) Gnu nie jest Uniksem
GPL (*General Public Licence*) Ogólna Licencja Publiczna
GPU (*Graphics Processing Unit*) procesor graficzny
GUI (*Graphical User Interface*) graficzny interfejs użytkownika
HDD (*Hard Disk Drive*) dysk twardy
HPC (*High-Performance Computing*) wysokowydajne obliczenia
HT (*Hyper-Threading*) hiperwątkowość
IBR (*instruction buffer register*) rejestr bufora rozkazów
IDT (*Interrupt Descriptor Table*) tablica deskryptorów przerwań
I/O (*Input/Output*) wejście-wyjście
IPC (*InterProcess Communication*) komunikacja międzyprocesowa
IPI (*Inter-Processor Interrupt*) przerwanie międzyprocesorowe
IR (*instruction register*) rejestr rozkazów
IRQ (*Interrupt ReQuest*) linia/żądanie przerwania
ISA (*Instruction Set Architecture*) architektura zestawu instrukcji (model programowy procesora)
ISR (*Interrupt Service Routine*) procedura obsługi przerwania
JCL (*Job Control Language*) język kontroli zadań
JFS (*Journaling File System*) system plików z kroniką
KVM (*Kernel-based Virtual Machine*) maszyna wirtualna realizowana przez jądro systemu operacyjnego
LBA (*Logical Block Addressing*) logiczne adresowanie bloków
LDT (*Local Descriptor Table*) lokalna tablica deskryptorów
LSI (*Large Scale Integration*) duży stopień scalenia
LWP (*Light Weight Process*) lekki proces
LVM (*Logical Volume Manager*) menadżer woluminów logicznych

MAC (*Mandatory Access Control*) obowiązkowa kontrola dostępu
MAR (*Memory Address Register*) rejestr adresowy pamięci
MBR (*Memory Buffer Register*) rejestr buforowy pamięci
MBR (*Master Boot Record*) główny rekord rozruchowy
MIMD (*Multiple Instruction Multiple Data*) wiele instrukcji wiele danych
MMIO (*Memory-Mapped Input/Output*) wejście/wyjście mapowane w pamięci
MMX (*MultiMedia Extensions*) rozszerzenia multimedialne
MPP (*Massively Parallel Processing*) przetwarzanie masywnie równoległe
MS-DOS (*Microsoft Disk Operating System*) dyskowy system operacyjny Microsoft
MSI (*Message Signaled Interrupts*) przerwania sygnalizowane komunikatami
MTBF (*Mean Time Between Failures*) średni czas międzyawaryjny
NMI (*NonMaskable Interrupts*) przerwania niemaskowalne
NPTL (*Native POSIX Thread Library*) natywna biblioteka wątków POSIX
NVRAM (*Non-Volatile RAM*) nieulotna pamięć RAM
NTFS (*New Technology File System*) system plików nowej technologii
NUMA (*Non-Uniform Memory Access*) niejednolity dostęp do pamięci
PATA (*Parallel AT Attachement*) równoległa końcówka AT
PC (*Program Counter*) licznik programu (rejestr procesora)
PC (*Personal Computer*) komputer osobisty
PCB (*Process Control Block*) blok kontrolny procesu
PCI (*Peripheral Component Interconnect*) połaczenie komponentów peryferyjnych
PIC (*Programmable Interrupt Controller*) programowalny kontroler przerwań
PID (*Process IDentifier*) identyfikator procesu
PPID (*Parent Process IDentifier*) identyfikator procesu macierzystego

PnP (*Plug and Play*) podłącz i używaj
POSIX (*Portable Operating System Interface for UNIX*) przenośny interfejs systemu operacyjnego dla Uniksa
PROM (*Programmable ROM*) programowalna pamięć stała
RAID (*Redundant Array of Inexpensive Disks*) nadmiarowa macierz niedrogich dysków lub (*Redundant Array of Independent Disks*) nadmiarowa macierz niezależnych dysków
PSW (*Program Status Word*) słowo stanu programu
RAM (*Random Access Memory*) pamięć swobodnego dostępu
RISC (*Reduced Instruction Set Computer*) komputer o zredukowanej liście rozkazów
ROM (*Read-Only Memory*) pamięć stała/tylko do odczytu
RPC (*Remote Procedure Call*) zdalne wywołanie procedury
SATA (*Serial AT Attachement*) szeregowa końcówka AT
SAS (*Serial Attached SCSI*) szeregowy interfejs SCSI
SCSI (*Small Computer System Interface*) interfejs małych systemów komputerowych
SIMD (*Single Instruction Multiple Data*) pojedyncza instrukcja wiele danych
SJF (*Shortest Job First*) najkrótsze zadanie najpierw
SMB (*Server Message Block*) blok komunikatów serwera
SMP (*Symmetric MultiProcessing*) symetryczne przetwarzanie wieloprocesorowe
SPN (*Shortest Process Next*) najkrótszy proces następny
SPOOL (*Simultaneous Peripheral Operations On-Line*) jednoczesna bezpośrednia praca urządzeń
SRAM (*Static RAM*) statyczna pamięć RAM
SRT (*Shortest Remaining Time*) najkrótszy pozostały czas
SSD (*Solid State Drive*) dysk półprzewodnikowy
SSE (*Streaming SIMD Extensions*) strumieniowe rozszerzenia SIMD

TGID (*Thread Group IDentifier*) identyfikator grupy wątków
TLB (*Translation Lookaside Buffer*) bufor translacji adresów
TID (*Thread IDentifier*) identyfikator wątku
TSL (*Test and set Lock*) testuj i załóż rygiel
TSS (*Task Status Segment*) segment statusu zadania
USB (*Universal Serial Bus*) uniwersalna szyna szeregowa
VFS (*Virtual File System*) wirtualny system plików
VLSI (*Very Large Scale Integration*) bardzo duża skala integracji