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 (2026/27)

Ostatnia aktualizacja: 25 listopada 2026

Program wykładu

1. Wprowadzenie
   - Elementy systemu komputerowego
   - Systemy operacyjne i ich zadania
   - Historia i rozwój komputerów i systemów operacyjnych
2. Architektura współczesnych komputerów
   - Architektura i działanie procesora
   - Rodzaje i hierarchia pamięci
   - Magistrale i urządzenia peryferyjne
3. Działanie systemu komputerowego
   - Monitor prosty, buforowanie, spooling
   - Wieloprogramowość i podział czasu
   - Systemy rozproszone, systemy czasu rzeczywistego

4. Przerwania
    - Systemy z obsługą przerwań
    - Struktura wejścia-wyjścia
    - Dualny tryb pracy
    - Funkcje systemowe
5. Procesy
    - Model procesu i jego implementacja
    - Stany procesów
    - Zarządzanie procesami
    - Wątki poziomu użytkownika, wątki jądra, procesy lekkie
6. Zarządzanie procesami
    - Algorytmy planowania procesora
    - Polityki planowania
    - Planista O(1) i CFS

7. Synchronizowanie procesów
   - Komunikacja międzyprocesowa (IPC)
   - Sekcje krytyczne i wyścigi
   - Wzajemne wyłączanie z aktywnym czekaniem
   - Semafory, muteksy, monitory
   - Zakleszczenia
   - Klasyczne problemy synchronizacji
8. Zarządzanie pamięcią (P. Ablewski)
   - Wymiana pamięci i stronicowanie
   - Pamięć wirtualna i segmentacja pamięci
   - Algorytmy wymiany stron
   - Zarządzanie pamięcią w systemach GNU/Linux i Windows

9. Zarządzanie przestrzenią dyskową (P. Ablewski)
    - Fizyczna organizacja przestrzeni dyskowej
    - Systemy plików
    - Zarządzanie logicznymi wolumenami
10. Wirtualizacja i konteneryzacja (P. Ablewski)
    - Maszyny wirtualne
    - Wirtualizacja pełna i parawirtualizacja
    - Kontenery i ich zastosowania
11. Struktura systemów operacyjnych (P. Ablewski)
    - Systemy monolityczne, warstwowe i hybrydowe
    - Współdzielenie systemów operacyjnych
    - Ładowanie systemu operacyjnego
12. Przykłady systemów operacyjnych (P. Ablewski)
    - Windows, GNU/Linux, Unix, macOS

Literatura

Dowolna pozycja z poniższych (dowolne wydanie)

- A. S. Tanenbaum, H. Bos, *Systemy operacyjne*, Helion, 2015.
- William Stallings, *Systemy operacyjne. Architektura, funkcjonowanie i projektowanie*, Helion, 2018.
- A. Silberschatz, J. L. Peterson, P. B. Galvin, *Podstawy systemów operacyjnych*, t. 1–2, WNT, 2021.

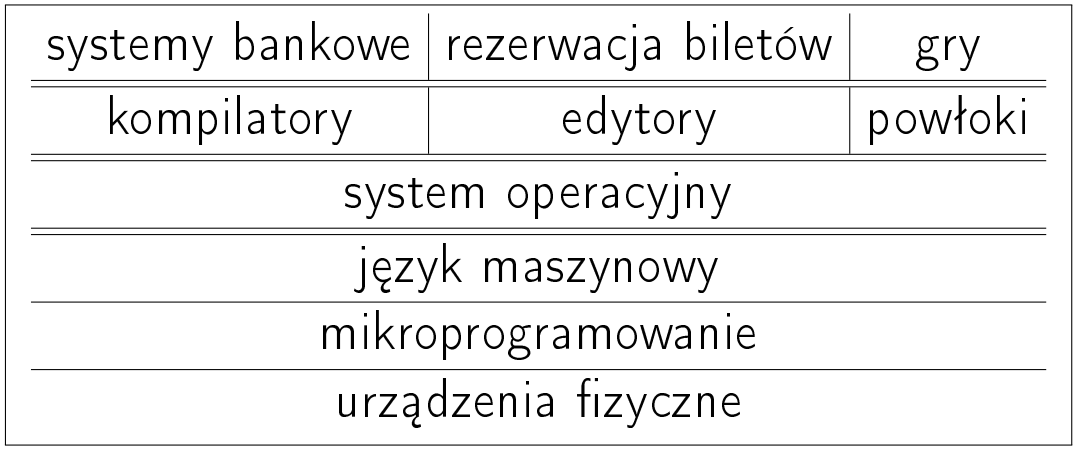
Literatura uzupełniająca:

- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, Pearson, 2022.
- K. Stencel, *Systemy operacyjne*, Wydawnictwo PJWSTK, 2004.
- A. M. Lister, R. D. Eager, *Wprowadzenie do systemów operacyjnych*, WNT, 1994.
- L. Null, J. Lobur, *Struktura organizacyjna i architektura systemów komputerowych*, Helion, 2004.
- R. Love, *Linux kernel. Przewodnik programisty*, Helion, 2004.

Elementy systemu komputerowego

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

System komputerowy wg Tanenbauma



System komputerowy

Poziomy abstrakcji współczesnych systemów komputerowych
(wg Null i Lobur)

6	użytkownik	programy wykonywalne
5	języki wysokiego poziomu	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 to **oprogramowanie**, które zarządza zasobami komputera i udostępnia programom usługi oraz abstrakcje sprzętu.
- Działa jako pośrednik między użytkownikiem komputera a sprzętem komputerowym
- System operacyjny **nadzoruje i koordynuje** przydział procesora, pamięci i urządzeń wejścia-wyjścia między programami.
- System operacyjny jest odpowiedzialny za
  - **zarządzanie zasobami** komputera
  - udostępnianie programom **warstwy abstrakcji** sprzętu, które ukrywają szczegóły jego obsługi

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

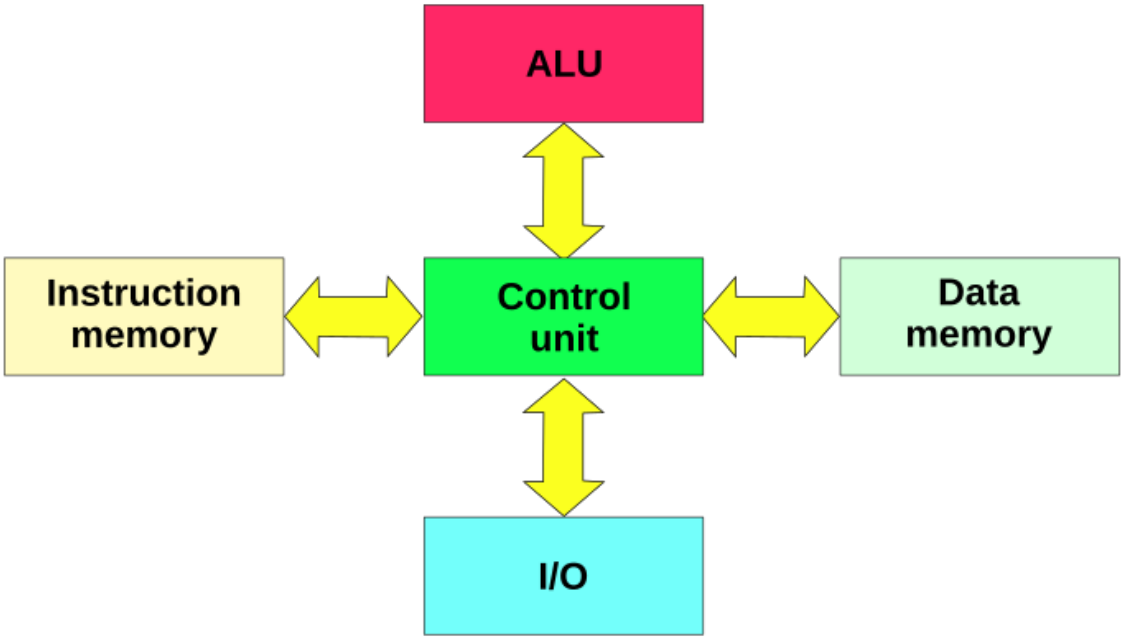
LSI (*Large Scale Integration*) duża skala integracji, 1000-10000 elem./układ

VLSI (*Very 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)
  - ASCC (*Automatic Sequence Controlled Calculator*), Harvard Mark I, Howard Aiken, IBM (1944) - architektura harwardzka
- 1945-1953 (1. generacja): komputery na lampach próżniowych
  - bezpośrednie programowanie w języku maszynowym
  - przeprowadzanie wyłącznie obliczeń numerycznych
  - zastosowanie kart dziurkowanych do wprowadzania danych (początek lat 1950)
- brak systemów operacyjnych i języków programowania

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
- niezależna organizacja pamięci instrukcji i danych
- możliwość doboru różnych parametrów pamięci instrukcji i danych

Wady:

- większa złożoność i koszt implementacji
- mniejsza elastyczność i trudniejsze współdzielenie pamięci przez dane i instrukcje

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

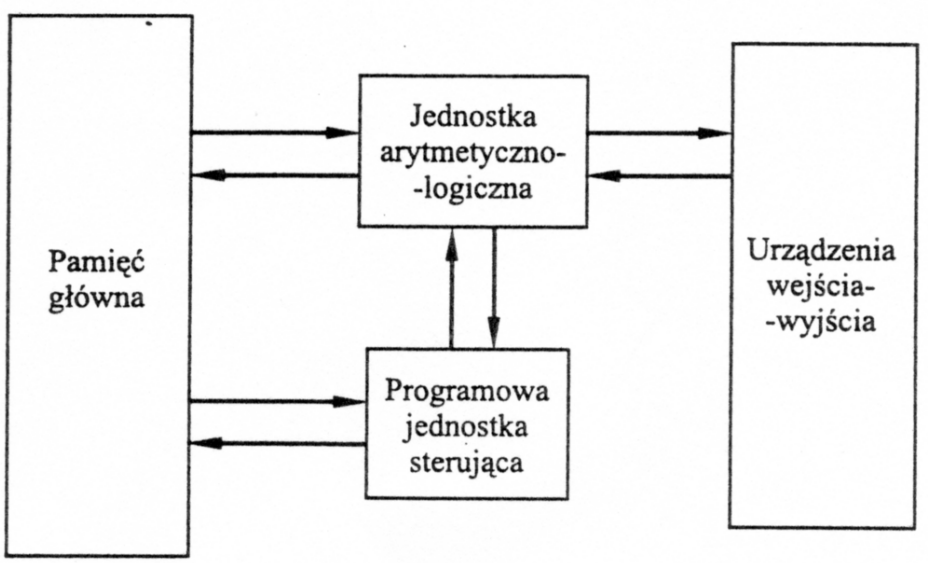
  1000 słów 40 bitowych (po 2 instrukcje (20-to bitowe) w słowie)

Pierwsze komputery komercyjne:

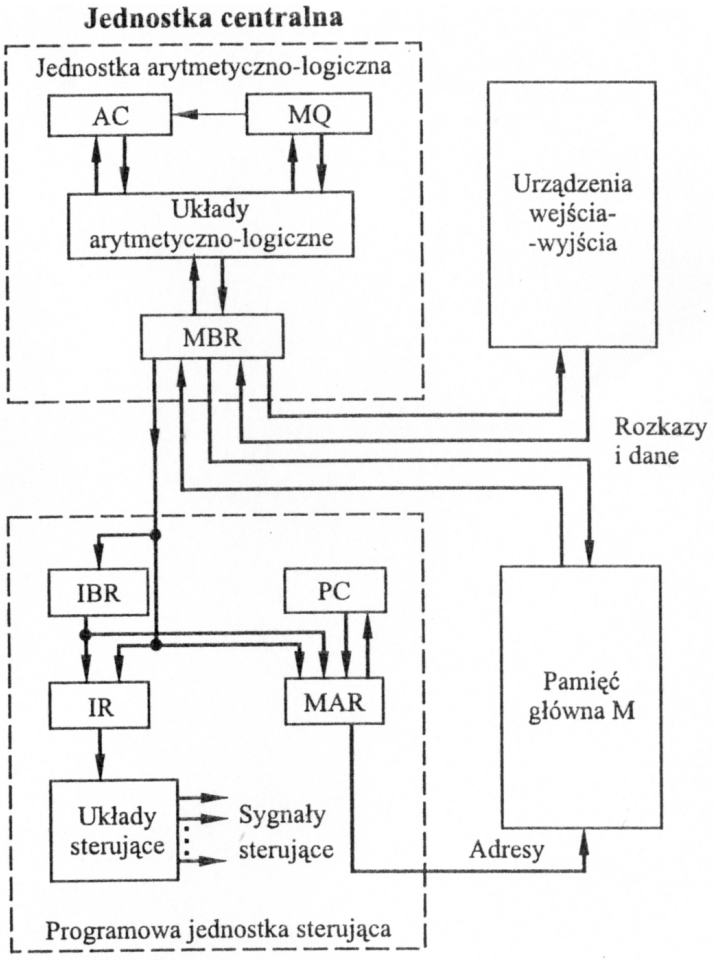
- **IBM 701** (1952) z przechowywanym programem do zastosowań naukowych, model 702 (1955) do zastosowań biznesowych

[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

- **rejestr buforowy pamięci** (MBR, *memory buffer register*) – przechowuje słowo odczytane z pamięci lub przygotowane do zapisu w 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*) – przechowuje drugi rozkaz z pobranego słowa podczas wykonywania pierwszego
- **licznik programu** (PC, *program counter*) – adres słowa pamięci z kolejną parą 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 pobrania 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

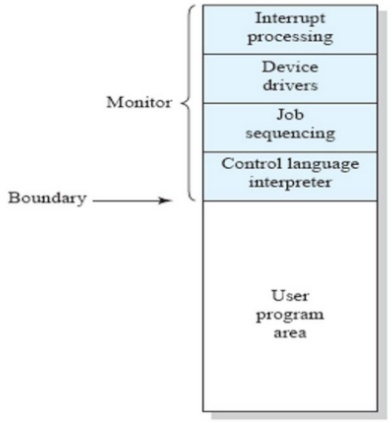
  Komputer IAS ma 21 rozkazów:

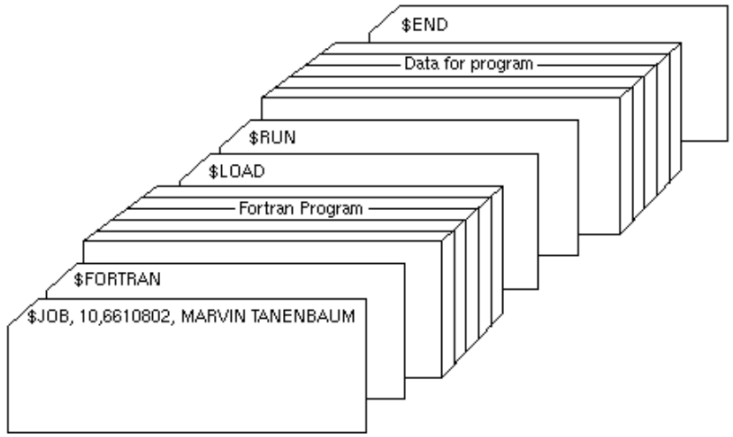
  – przenoszenie danych
  – rozgałęzienie bezwarunkowe
  – rozgałęzienie warunkowe
  – arytmetyka
  – modyfikowanie adresu

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

- zwiększona niezawodność
- bardziej złożone jednostki ALU oraz sterujące
- zadania wykonywane po jednym na raz
- 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

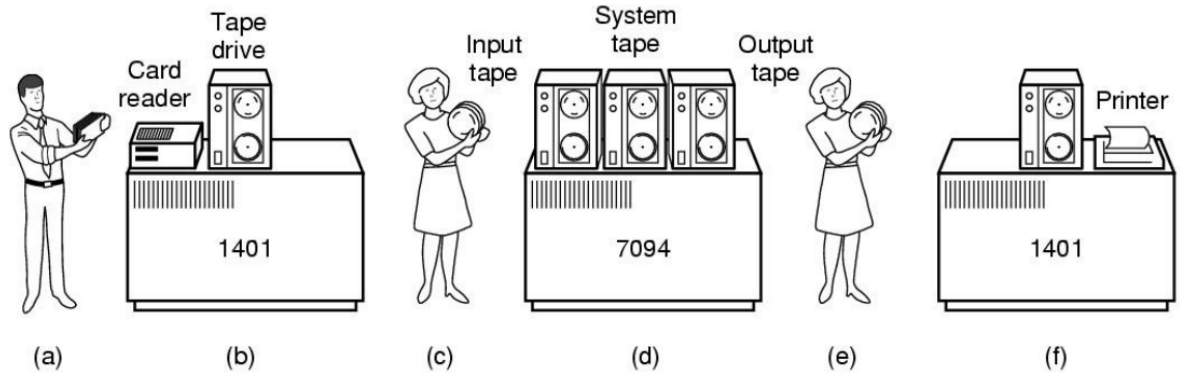
Monitor rezydujący w pamięci





- 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 pośrednia** (*off-line*) - 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, wyspecjalizowane urządzenia, a następnie przekazywane do komputera w blokach (np. na taśmie magnetycznej)
- procesor nie traci czasu na oczekiwanie na wolne urządzenia we/wy

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 w pamięci masowej. Specjalny program systemowy (*spooler*) pobiera dane z dysku i przesyła je do urządzenia w tle, we własnym, wolnym tempie. Wyeliminowało to małe komputery do obsługi kart
- pojawiają 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)

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

- stacje robocze, komputery osobiste (architektura minikomputerów),
- graficzne interfejsy użytkownika (GUI)
- sieciowe systemy operacyjne
- rozproszone systemy operacyjne (układy masywnie równoległe)
- wirtualizacja

5. generacja: systemy mobilne (1990-)

- telefony łączące funkcje telefonu i PDA (*Personal Digital Assistant*), np. Nokia 9000 Communicator (1996)
- 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]

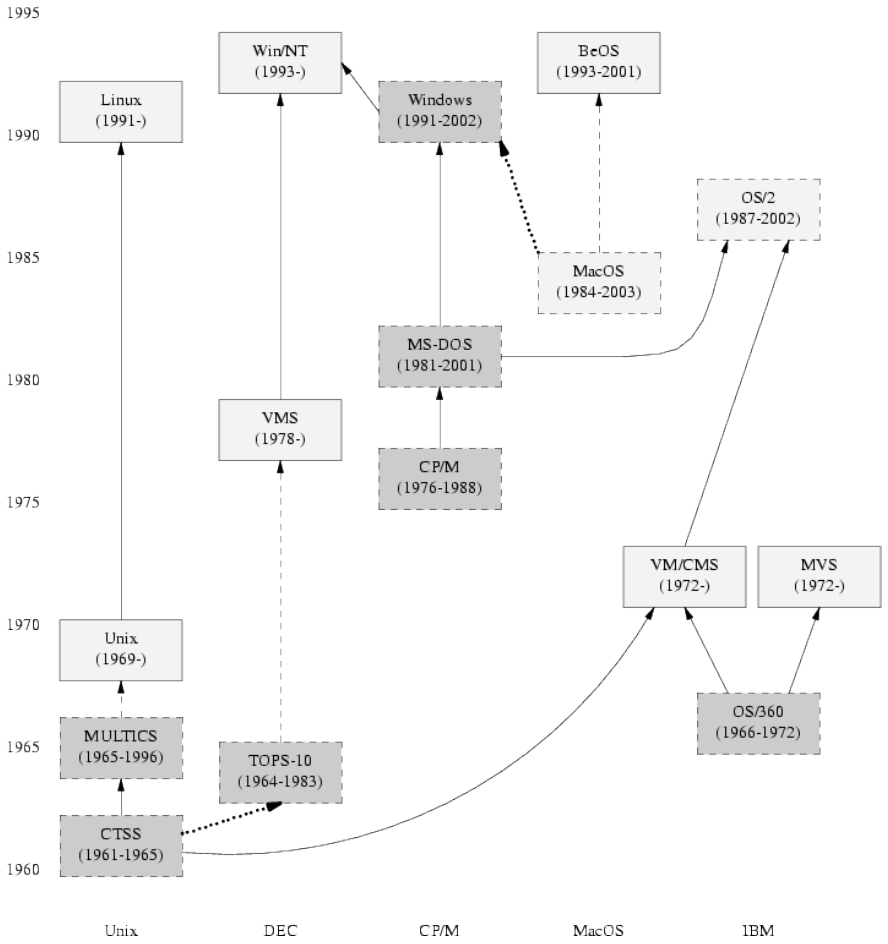
[2] Zob. System Mera 400

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]

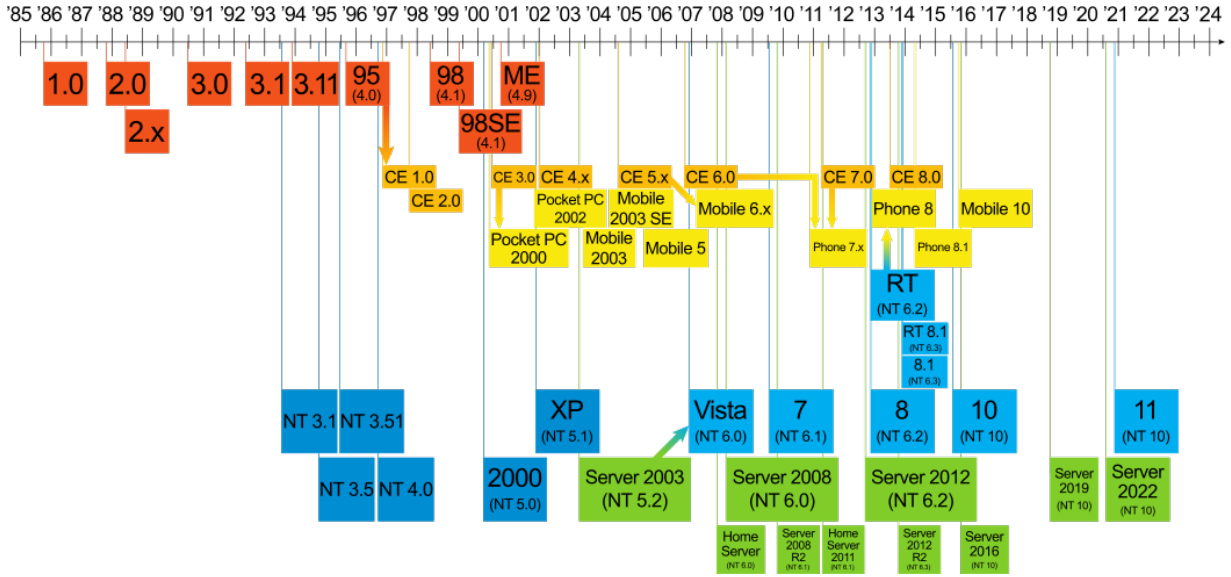


[4] *UNIX Sztuka programowania*, Wydawnictwo Helion, Gliwice, 2004. Także http://en.wikipedia.org/wiki/Timeline_of_operating_systems

MS-DOS, Windows, Mac OS

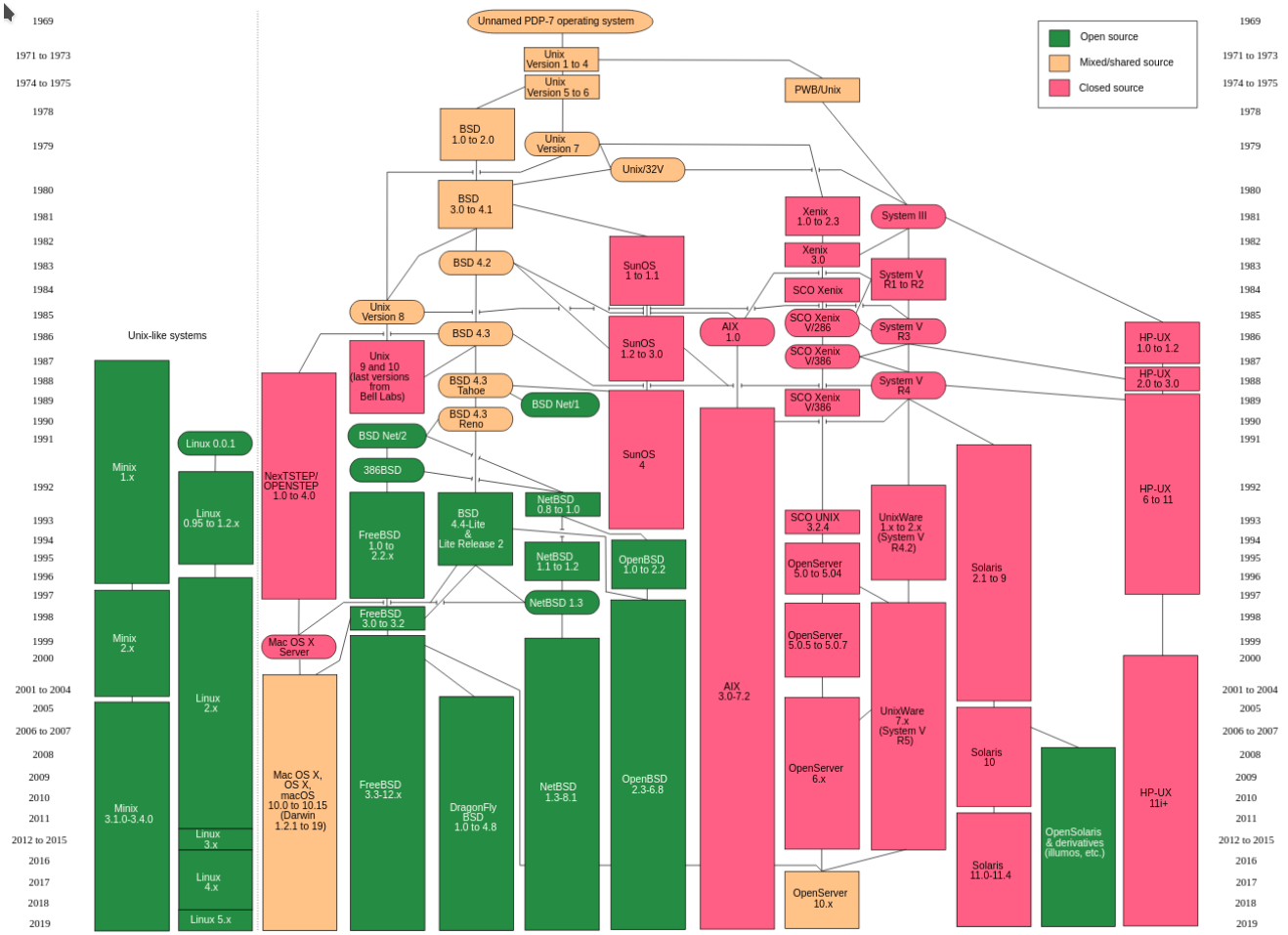
- 1974 pierwszy 8-mio bitowy procesor 8080 ogólnego przeznaczenia,
- system CP/M (*Control Program for Microcomputers*), rozwijany przez Digital Research pod kierunkiem Gary'ego Kildalla, szeroko stosowany w mikrokomputerach lat 70.
- 1981 IBM PC debiutuje z systemem PC DOS; Microsoft rozwijał jego wersję dla innych komputerów jako MS-DOS
- 1984 IBM PC/AT umacnia pozycję MS-DOS
- 1984 Apple wprowadza komputer Macintosh z GUI i myszą; wcześniej graficzne interfejsy rozwijano m.in. w Xerox PARC
- 1985 Windows 1.0 - początkowo graficzna nakładka na MS-DOS
- 1995 Windows 95: system z graficznym interfejsem, zachowujący obsługę MS-DOS

- Windows NT (*New Technology*): odrębna linia 32-bitowa systemów Windows z własnym jądrem, bezpieczna, z pełnym wywłaszczaniem procesów i ochroną pamięci.
- od 2000 dwie linie:[5]
  - serwerowa: Windows Server 2003, Windows Server 2008, ...
  - kliencka: oparte o Windows XP, Windows 7, 8, 10, 11, ...



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

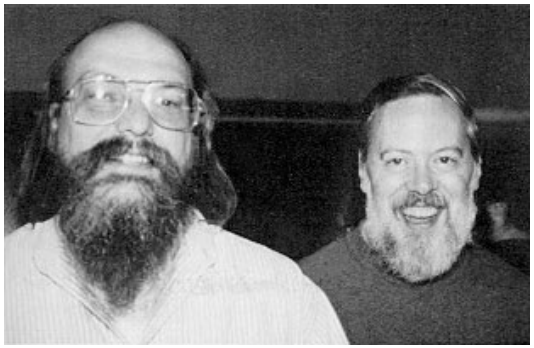
Historia systemu operacyjnego Unix



Wikipedia: History of Unix

Historia Unix

- 1969 Ken Thompson (Bell Labs, AT&T) tworzy na PDP-7 własny system operacyjny, wykorzystując doświadczenia z porzuconego MULTICS
- początek lat 70.: Dennis Ritchie tworzy język C; w 1973 r. Unix zostaje w nim przepisany
- 1974 Thompson i Ritchie publikują opis Uniksa, co przyczynia się do rozpowszechnienia systemu
- od 1977 r. Bill Joy i zespół z Uniwersytetu Kalifornijskiego w Berkeley rozwijają BSD, dodając m.in. obsługę 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 Richard Stallman ogłasza projekt GNU (*GNU's Not Unix*), którego celem jest stworzenie wolnego systemu operacyjnego zgodnego z Uniksem
- 1985 powstają Free Software Foundation i *The GNU Manifesto*
- 1989 ukazuje się pierwsza wersja licencji GNU GPL
- 1991 Linus Torvalds publikuje pierwszą wersję jądra Linux; w 1992 r. jądro zostaje objęte licencją GNU GPL
- 2007 powstaje The Linux Foundation





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*) to projekt wolnego systemu operacyjnego zgodnego z Uniksem
- GNU/Linux to system złożony z programów GNU i jądra Linux; projekt GNU rozwija także własne jądro, GNU Hurd
- GNU dostarcza m.in. kompilator GCC, bibliotekę glibc, powłokę Bash i pakiet GNU Coreutils (programy 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
  Copyleft w licencjach GNU odwróciła tradycyjne prawo autorskie w celu zapewnienia wolności użytkownikom oprogramowania
- dystrybucje Linuksa: Slackware (1993), Debian (1993), Ubuntu (2004), Red Hat, ...

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		
2026/08	7.0	ponad 40M	wersja stabilna

https://kernel.org/

- 1998 Eric S. Raymond i Bruce Perens zakładają Open Source Initiative (OSI), promując określenie *open source* obok terminu *free software*.
- Na początku lat 2000. firmy Oracle, Informix, Sybase, IBM, Dell i HP deklarują wsparcie dla ruchu open source.
- 2000: SUSE Linux Enterprise Server obsługuje m.in. architekturę IBM S/390 i x86.
- 2001: IBM zapowiada inwestycje w rozwój Linuksa o wartości 1 mld USD i angażuje w ten obszar swoich programistów.
  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 wnosi do jądra Linuksa ponad 20 tys. linii kodu związanych z obsługą Hyper-V
- 2014 Satya Nadella (CEO firmy Microsoft): *Microsoft loves Linux*
- 2016: Microsoft udostępnia Windows Subsystem for Linux i dołącza do Linux Foundation jako członek platynowy.

Rozwój jądra Linux (raport z 2020 r.)

- jeden z największych projektów programistycznych rozwijanych przez społeczność
- regularne wydania co 8-10 tygodni, każde z istotnymi zmianami i usprawnieniami
- w raporcie z 2020 r. szacowano około 12 tys. zmian na wydanie i około 10,7 commitów na godzinę
- 1600 programistów z ponad 200 korporacji
- około 12% zmian pochodziło od niezależnych programistów
- około 52% zmian pochodziło od programistów firm, m.in. Intel, Red Hat, IBM, SUSE, Linaro, Google, Samsung, AMD, Renesas, Texas Instruments i Oracle
- łącznie od 2005 r. ponad 15 600 programistów z 1400 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

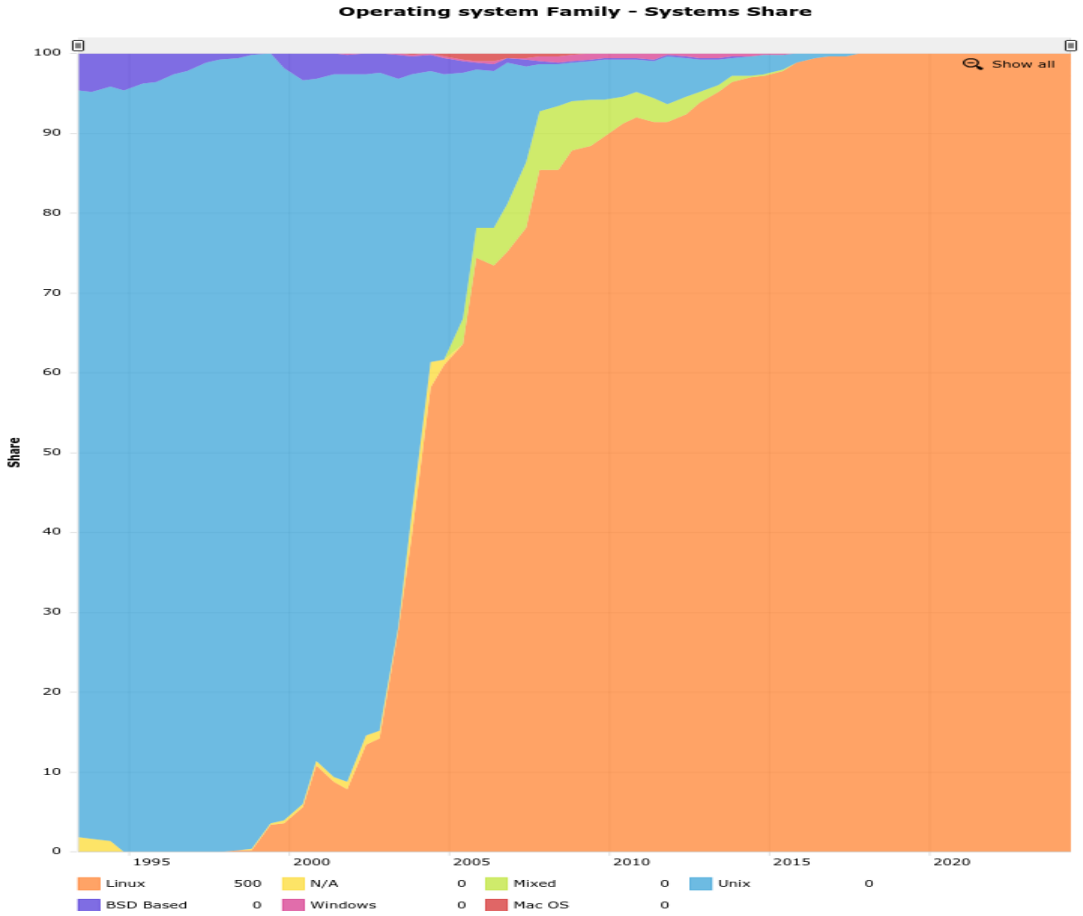
- iOS 55%
- Android 44%

Mobilne

- Android 72%
- iOS 27%

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

TOP 500: udział systemów operacyjnych (6-2024)

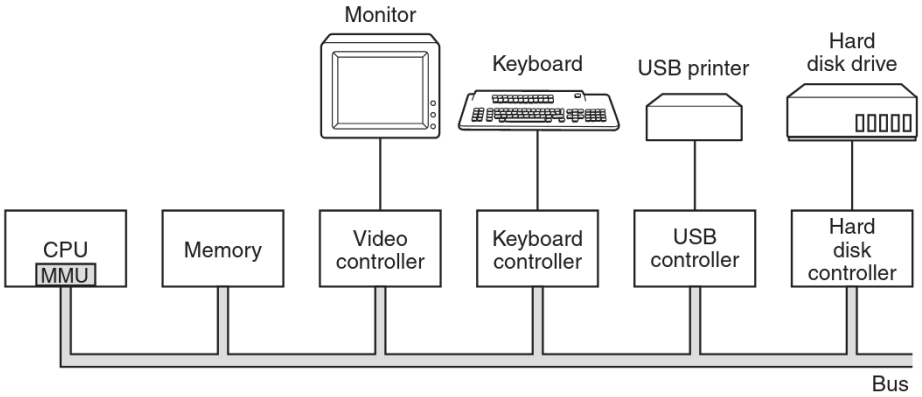


https://www.top500.org/statistics/overtime/

Pytania sprawdzające

1. Jakie są dwa podstawowe zadania systemu operacyjnego?
2. Co to jest jądro systemu operacyjnego?
3. Jakie są różnice między architekturą harwardzką a architekturą von Neumanna?
4. Na egzaminie nie będzie pytań dotyczących historii ale należy znać pojęcia takie jak, np.: spooling, praca wsadowa, monitor, wieloprogramowość, POSIX, ...

Architektura komputerów



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

Procesor

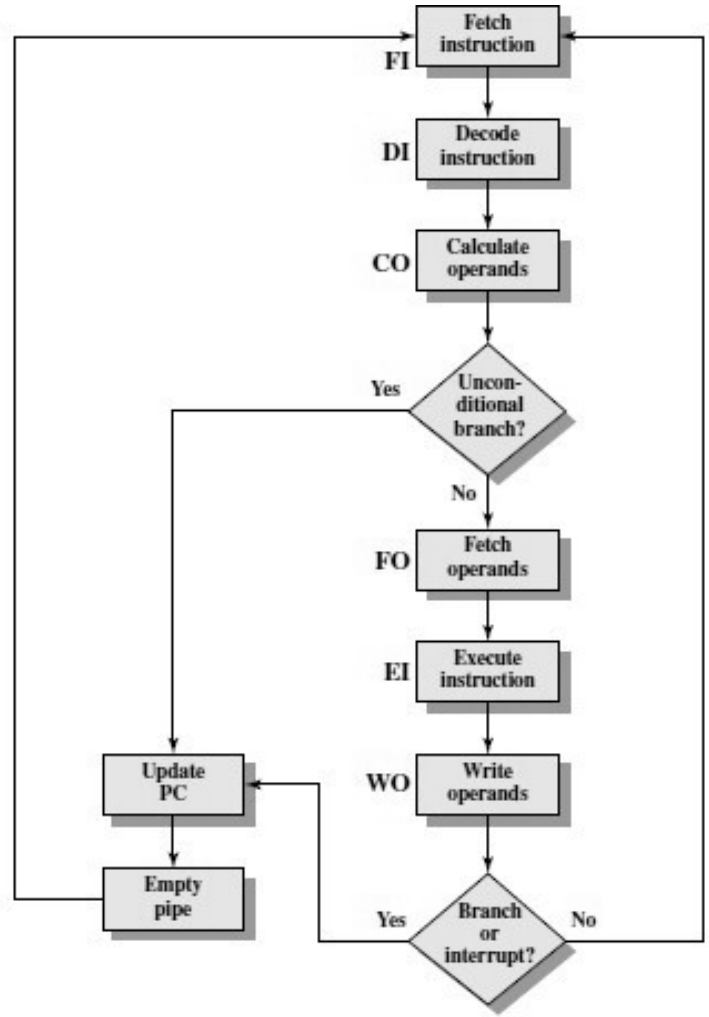
- rejestry
  - rejestry ogólnego zastosowania (x86: EAX, EBX, ECX, EDX)
  - licznik programu (*program counter*) - zawiera adres następnej instrukcji (x86: EIP)
  - wskaźnik stosu (*stack pointer*) wskazuje najwyższy element stosu wywołań w pamięci (x86: ESP)
- słowo statusu programu (PSW *program status 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 **RISC** - 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

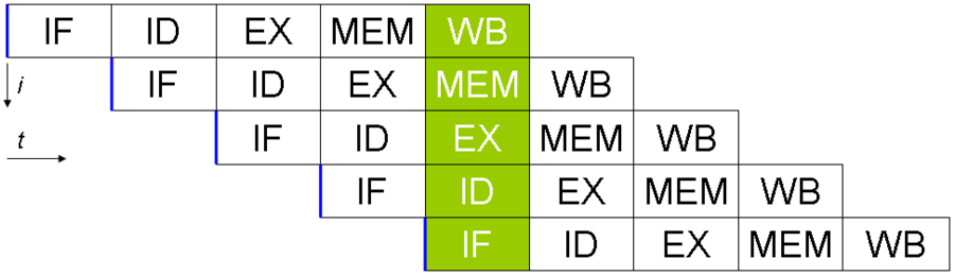
FO: pobierz dane z pamięci jeżeli nie ma ich w rejestrach

EI: wykonaj instrukcję

WO: zapisz wynik w pamięci

Rozgałęzienie lub przerwanie powoduje opróżnienie potoku i ponowne załadowanie odpowiednich instrukcji.

Przetwarzanie potokowe



Klasyczny potok RISC:

IF=Instruction Fetch

ID=Instruction Decode

EX=Execute

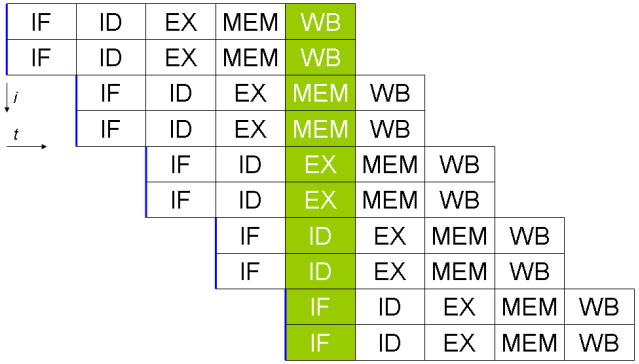
MEM=Memory access

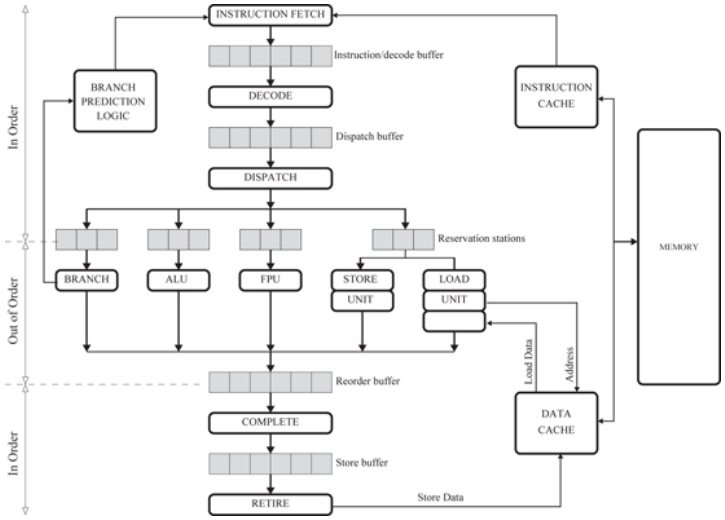
WB=Register write back

- 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 dostępie do wspólnych zasobów

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

Superskalarność





- zrównoleglenie wykonania rozkazów w obrębie jednego procesora poprzez zwielokrotnienie jednostek wykonawczych
- procesor wykrywa instrukcje niezależne od siebie; zależne muszą poczekać w na potrzebny wynik
- wykonanie poza kolejnością (*out-of-order*) - kolejność wykonania rozkazow jest inna niż w programie w celu maksymalizacji wykorzystania CPU

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

Ryzyka przetwarzania potokowego

Zatory w potoku poleceń pojawiają się, gdy skok warunkowy nie pozwala na sekwencyjne wykonanie instrukcji.

Możliwe ryzyka wykonania kodu poza kolejnością:

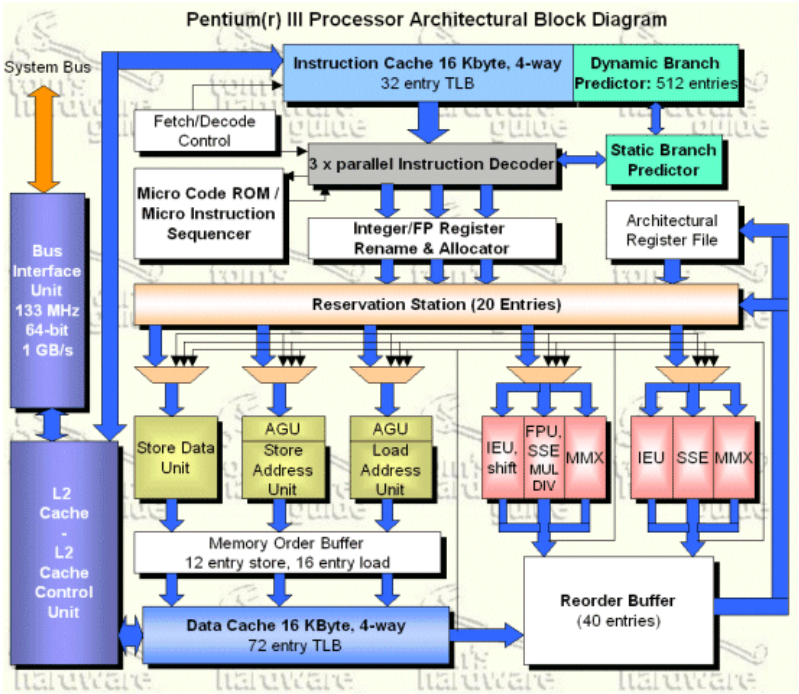
- **strukturalne** - gdy instrukcje równocześnie potrzebują tych samych zasobów (np. pamięci)
- **danych** - gdy instrukcje potrzebuje wyniku innej instrukcji, który nie jest jeszcze dostępny
- **sterowania** - wykonanie instrukcji po rozwidleniu, ale przed wykonaniem instrukcji warunkowej. Instrukcje z błędnej ścieżki muszą zostać odrzucone z potoku.

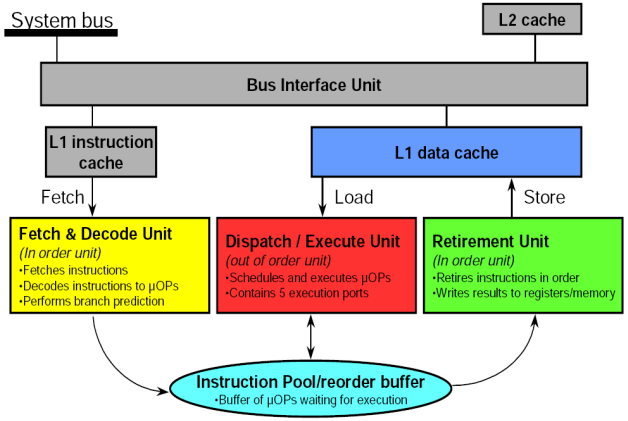
Ryzyka przetwarzania potokowego

Jak radzić sobie z konfliktami?

- wykonywanie instrukcji obu ścieżek jednocześnie
- przewidywanie rozgałęzień (np. na podstawie statystyki poprzednich wykonań). Kod jest wykonywany spekulatywnie. W przypadku błędnej predykcji wynik tego wykonania jest odrzucany
- *branch-free-code* technika programowania polegająca na ograniczaniu rozgałęzień w kodzie w celu zmniejszenia ryzyka wystąpienia zatorów w potoku
- 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

Architektura procesorów Pentium II i Pentium III





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

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, SSD)
- pamięć taśmowa, dyski optyczne

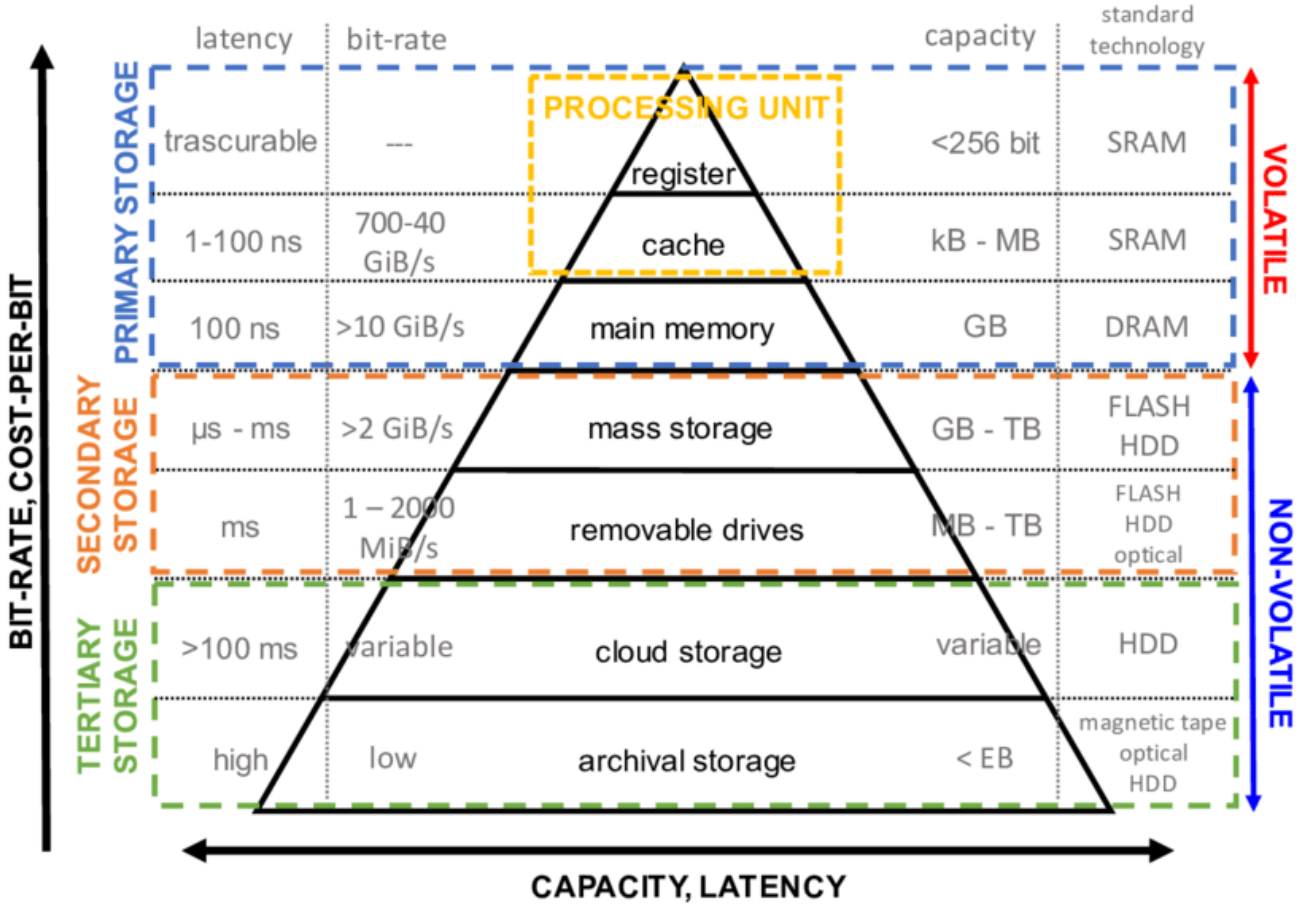
Koszt trafienia/chybienia (w cyklach) dla procesora Pentium M:

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

Zob.: Interactive latency

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

Hierarchia pamięci



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

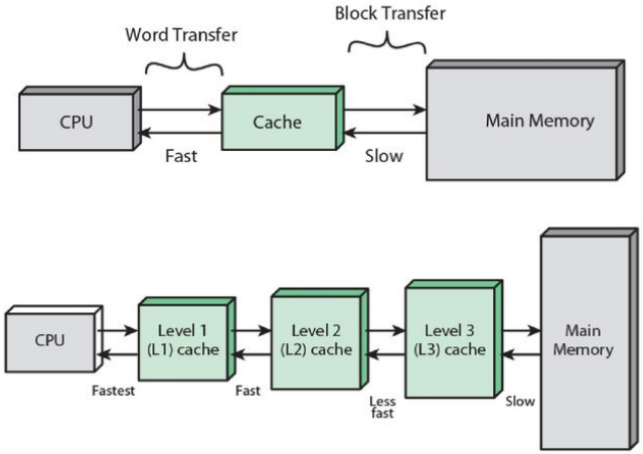
- malejąca cena/bit
- rosnąca pojemność
- rosnący czas dostępu
- malejąca częstość dostępu przez procesor

Rodzaj dostępu do pamięci:

- dostęp sekwencyjny
  - odczyt/zapis wyłącznie w określonej sekwencji liniowej (np. pamięć taśmowa)
  - czas dostępu zależy od położenia danych
- dostęp swobodny
  - pozycja w pamięci dostępna za pomocą unikatowego adresu (pamięć RAM)
  - czas dostępu jest stały i nie zależy od lokalizacji danych
- dostęp bezpośredni
  - pamięć podzielona na bloki (lub rekordy) o unikatowych adresach (np. pamięć dyskowa)
  - najpierw następuje dostęp (swobodny) do najbliższego otoczenia (wybór bloku), a następnie sekwencyjne poszukiwanie lokalizacji końcowej
- 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 ultraszybkim procesorem a wolną pamięcią główną



- przechowuje kopie fragmentów pamięci głównej
- procesor najpierw sprawdza, czy dane znajdują się w pamięci podręcznej
- jeśli danych brakuje (chybienie), do pamięci podręcznej wczytywany jest odpowiedni blok pamięci głównej

Lokalność przestrzenna

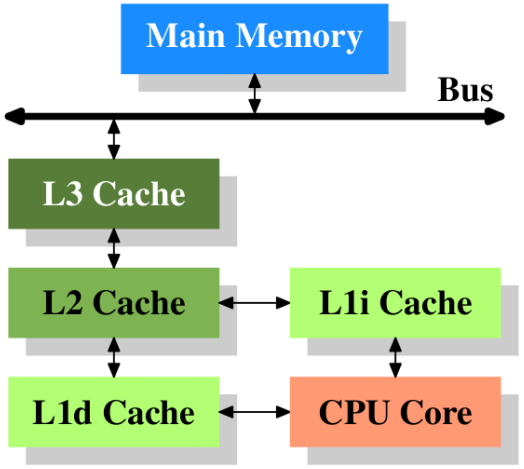
- tendencja do częstych odwołań do danych położonych blisko siebie w pamięci
- takie dane mają szansę trafić do tego samego bloku w pamięci podręcznej

Lokalność czasowa

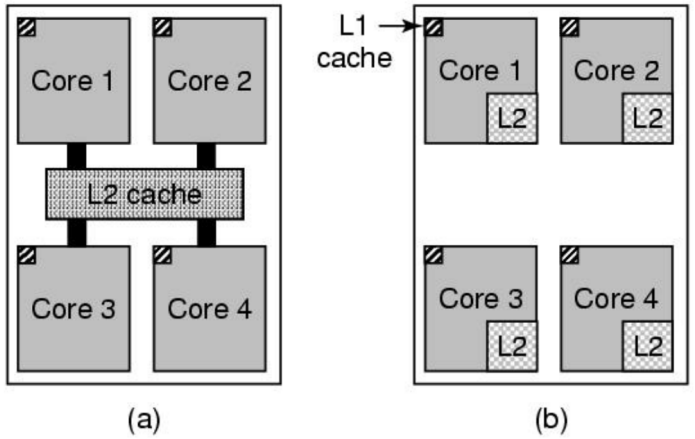
- tendencja do częstych odwołań do tych samych danych w krótkich odstępach czasu
- często wykorzystywane dane mają szansę pozostać w pamięci podręcznej między kolejnymi żądaniami

Procesor i pamięć

Architektura jednoprocesorowa



Architektura wieloprocesorowa

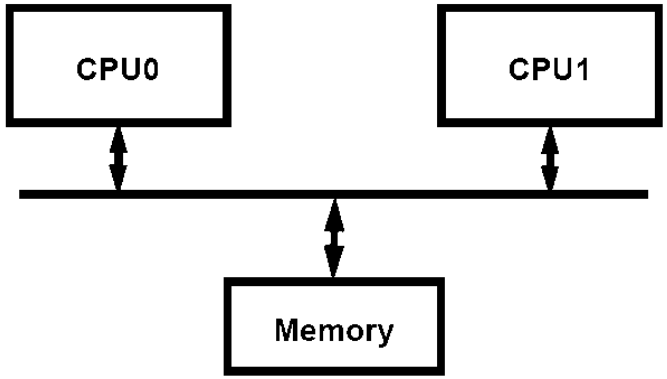


a) współdzielony L2 przez 4 procesory
b) niezależne L1 i L2 w układzie 4 procesorów

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

Wieloprocesorowość symetryczna (SMP)

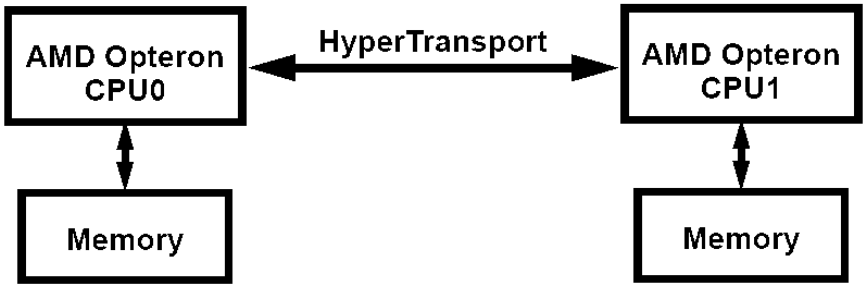
Symmetric Multiprocessing



- 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
- system operacyjny zarządza przydziałem zadań do procesorów oraz dostępem do pamięci i urządzeń wejścia/wyjścia

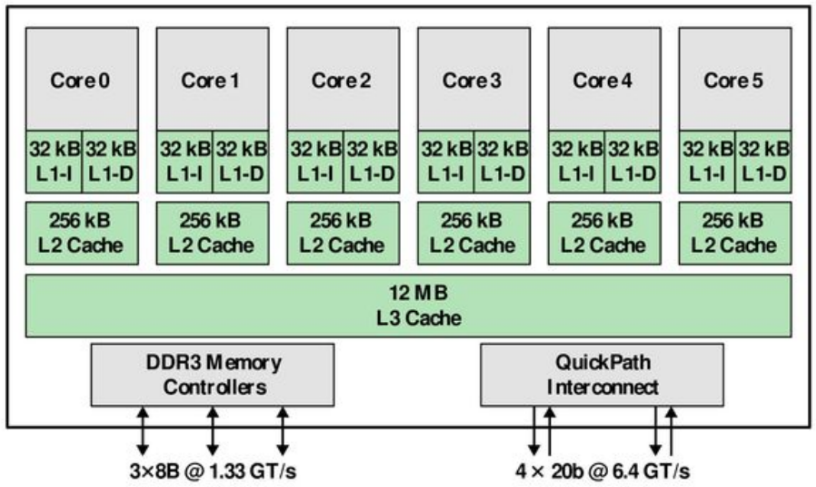
Niejednolity dostęp do pamięci (NUMA)

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

Architektury wieloprocesorowe



Rysunek 1: Intel Core-i7 990X

- wiele procesorów (rdzeni) na jednej kości
- każdy rdzeń ma jednostkę wykonawczą, rejestry i lokalną pamięć podręczną; może też obsługiwać wiele wątków
- 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 jest przechowywany w układzie zbudowanym z 6 tranzystorów (mniejsza gęstość danych niż DRAM)

Dynamic RAM (DRAM) - pamięć dynamiczna

- pamięć główna komputera
- ulotna, wymaga okresowego odświeżania zawartości

Pamięci dynamiczne

Synchronous DRAM (SDRAM) - pamięć 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

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

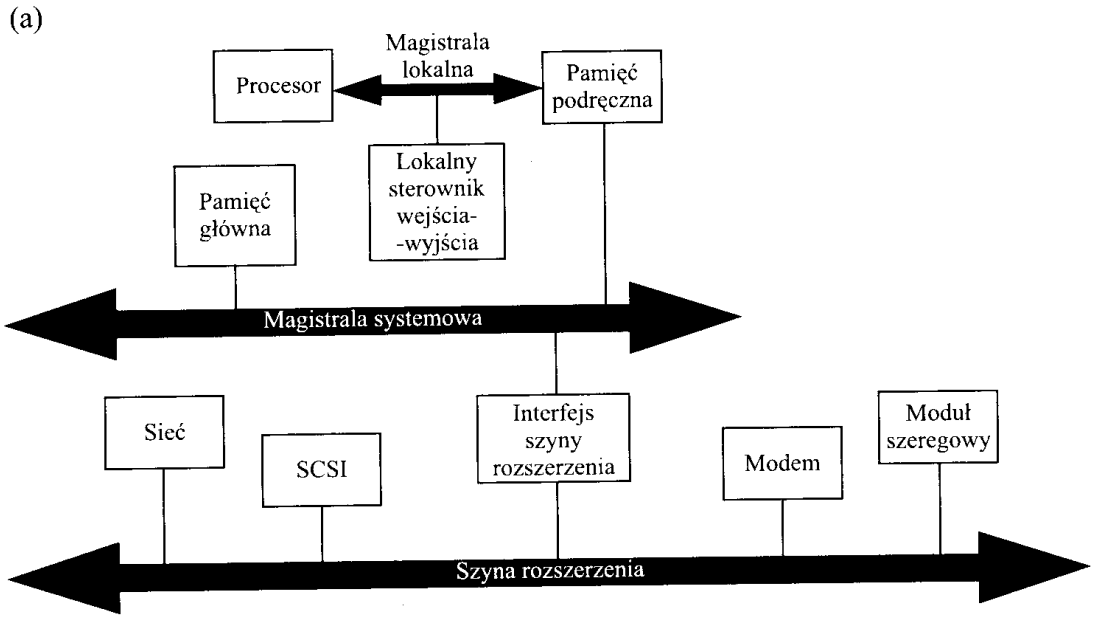
DDR Generations: Memory Density and Speed

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 (zastosowanie: podtrzymywanie ustawień BIOS)[6]
- **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

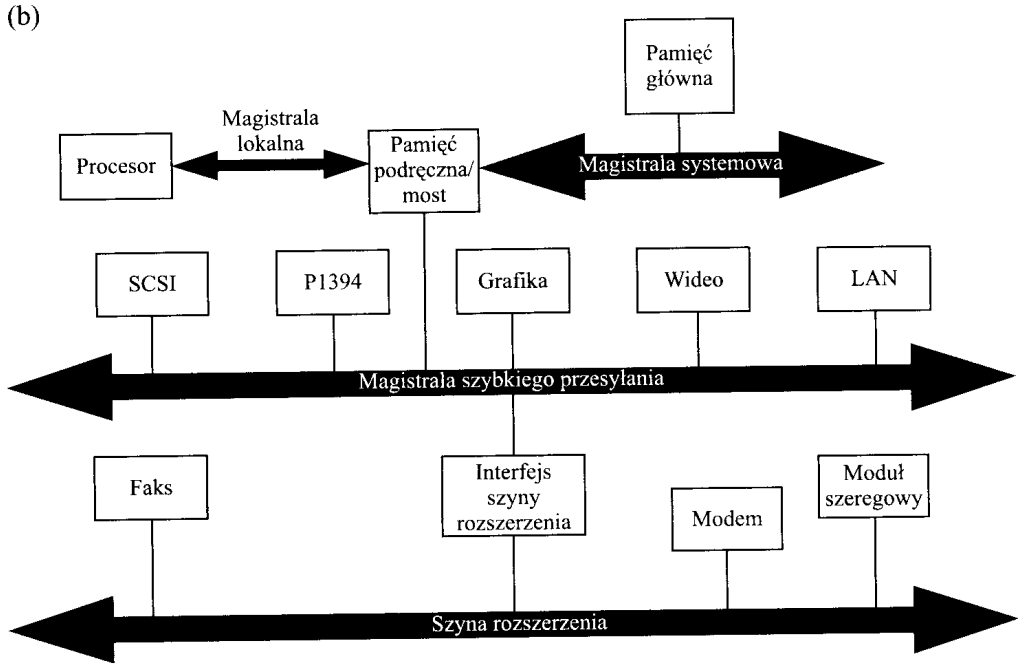
[6] Także EEPROM (*Electrically Erasable and Programmable ROM*) wymazywalna i programowalna pamięć stała (na poziomie bajtu) – głównie odczyt, wymazywanie/zapisywanie elektryczne; szybki odczyt, wolny zapis.

Magistrale (szyny) systemowe



- komunikacja między szyną systemową a szyną rozszerzeń może następować bez angażowania CPU

Magistrale (szyny) systemowe



- hierarchiczna magistrala o wysokiej wydajności
- urządzenia o większych wymaganiach są bliżej 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 konfiguracji 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* (PnP) - automatyczne wykrycie i konfiguracja urządzenia przez system operacyjny podczas startu, np. przydzielenie zasobów i załadowanie sterownika.

Ograniczenia:

- brak wsparcia w nowoczesnych systemach operacyjnych (np. Windows 10, Linux nie obsługuje ISA natywnie)
- przez problemy z konfliktami IRQ nazywany *plug and pray*
- przestarzała, zastąpiona przez PCI.

PCI (*Peripheral Component Interconnect*, Intel 1992)

- obsługuje standard „podłącz i używaj" (PnP, *Plug and Play*) i automatyczną konfigurację urządzeń
- 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 256 GB/s w wersji 6.0), idealna dla kart graficznych, dysków NVMe i sieciowych kart wysokiej wydajności.
- niskie opóźnienia i skalowalność dzięki architekturze *point-to-point*
- wsparcie dla *hot-plugging*, podłączania urządzeń „na gorąco", gdy komputer jest włączony i system działa
- pełna kompatybilność z nowoczesnymi systemami operacyjnymi
- obsługa wielu linii (x1, x4, x8, x16), zależnie od potrzeb urządzenia
- zastępuje PCI i AGP (*Advanced Graphics Port*, PCI dla kart graficznych)
- brak linii przerwań IRQ, używa sygnałów MSI (Message Signaled Interrupts)

Ograniczenia:

- 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 wyłączenia zasilania
- łatwość rozmnażania portów i wydłużania połączenia poprzez zastosowanie maksymalnie pięciu koncentratorów (*USB hubs*)
- uniwersalne złącze obsługujące szeroką gamę urządzeń (klawiatury, myszy, drukarki, skanery, kamery wideo, dyski)
- szeroka kompatybilność z systemami operacyjnymi (Windows, Linux, macOS, Android)

Ograniczenia:

- wymaga obecności dokładnie jednego kontrolera po stronie hosta (nie można bezpośrednio połączyć dwóch komputerów)

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

- szyna do łączenia dysków HDD i napędów CD-ROM (40 pinów)
- 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)

Szeregowy ATA (*Serial ATA*, SATA)

- kolejkowanie zadań w celu minimalizacji liczby skoków głowicy
- powielacz portów pozwala podłączyć kilka dysków do jednego portu kontrolera SATA
- selektor portu umożliwia podłączyć jedno urządzenie z dwoma portami hosta (nadmiarowa ścieżka na wypadek awarii)
- połączenia punkt-punkt
- możliwość podłączania wewnętrznych i zewnętrznych urządzeń
- CRC (*Cyclic Redundancy Check*) - suma kontrolna do wykrywania błędów transmisji
- łatwość przyłączenia (wsparcie dla *hot swapping*)
- szerokie wsparcie we współczesnych systemach operacyjnych

SCSI (*Small Computer Systems Interface*)

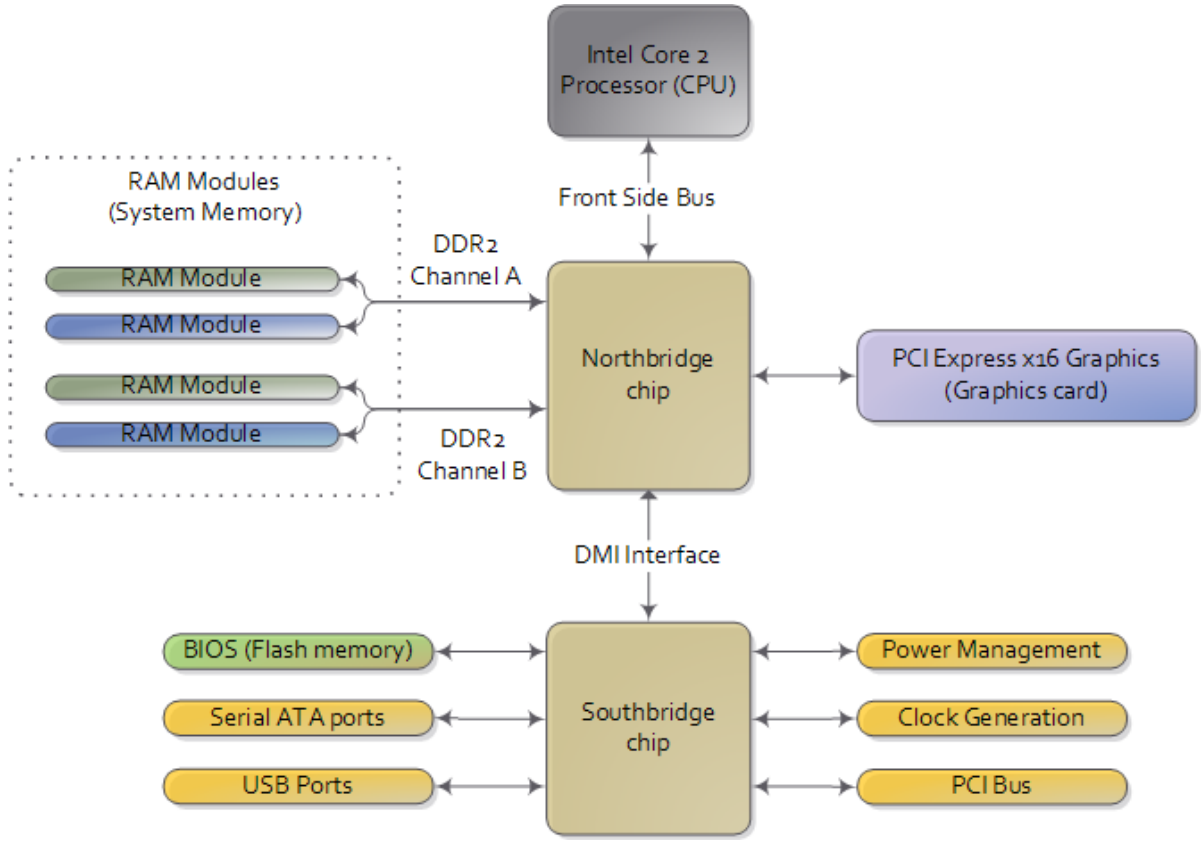
- obsługa urzadzeń pamięci masowej (dyski, napędy CD-ROM, napędy taśmowe (*streamer*), skanery)
- wysoka wydajność i niezawodność, wykorzystanie w stacjach roboczych i serwerach
- jeden IRQ, obsługa do 16 urządzeń, łącze wielopunktowe (*multidrop*)
- możliwość podłączania urządzeń o różnej prędkości na tym samym łańcuchu, maksymalna długość do 25 m

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

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

Architektura płyty głównej



Chipset - zestaw układów i kontrolerów organizujących komunikację między procesorem, pamięcią i urządzeniami

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 o dużej szybkości: RAM, GPU, PCIe, Gigabit Ethernet
- **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,
- współcześnie wiele funkcji mostka północnego, takich jak kontroler pamięci, znajduje się bezpośrednio w procesorze

Podsumowanie i pytania sprawdzające

- Co to jest przetwarzanie potokowe w architekturze procesora?
- Jakie są przyczyny głodzenia procesora?
- Na czym polega lokalność przestrzenna i lokalność czasowa w pamięci komputerowej?
- Czym różni się architektura SMP od NUMA?

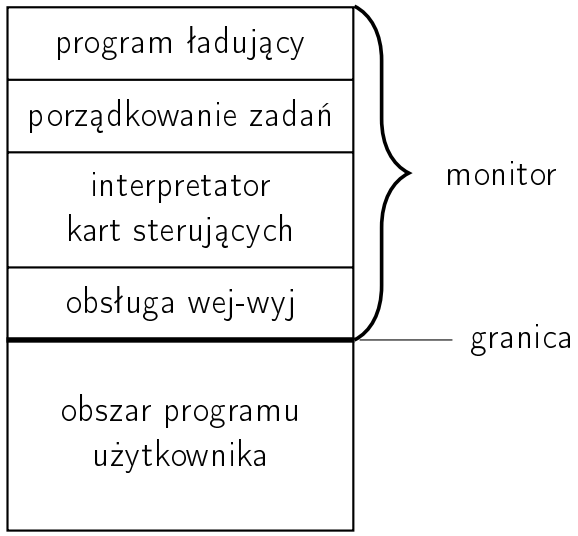
Jak system wykonuje zadania?

- Od monitora prostego po systemy wielodostępowe
- Wieloprogramowość
- Wielozadaniowość, 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; automatycznie przekazuje sterowanie od zadania do zadania oraz monitoruje i zarządza określonymi zasobami lub zdarzeniami
- automatyczne porządkowanie zadań – przydzielanie czasu procesora poszczególnym zadaniom sekwencyjnie 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 mogą być wykonywane wyłącznie przez monitor (tryb jądra)

Wolne operacje wejścia-wyjścia

- Urządzenia wejścia-wyjścia są zwykle znacznie wolniejsze od procesora. W systemie wykonującym jedno zadanie naraz procesor może pozostawać bezczynny, gdy zadanie czeka na zakończenie operacji I/O.
- Celem jest oddzielenie tempa pracy procesora od tempa urządzeń, aby ograniczyć czas oczekiwania.
- Przykłady rozwiązań stosowanych we wczesnych systemach przetwarzania wsadowego:
  - **praca pośrednia (off-line, historycznie)** – obsługa wejścia i wyjścia przez odrębne urządzenia; dane przekazywano do komputera w blokach, np. na taśmie magnetycznej.
  - **spooling** – kolejka danych lub zadań przechowywana na dysku, np. zadań drukowania; podczas gdy urządzenie obsługuje kolejkę, procesor może wykonywać inne zadanie.

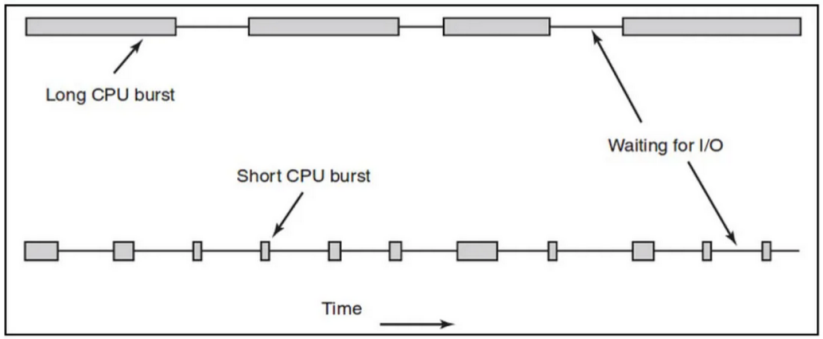
Buforowanie

Buforowanie usprawnia wykonywanie obliczeń i operacji wej-wyj w ramach zadania.

- polega na tymczasowym przechowywaniu danych w pamięci (buforze) przed ich dalszym przetwarzaniem lub przesłaniem
- zmniejsza liczbę operacji I/O i przyspiesza dostęp do danych
- procesor może wykonywać inne zadania, gdy dane są ładowane do bufora, co pozwala lepiej wykorzystać jego zasoby i skrócić czas oczekiwania na dane
- przykłady: pamięć podręczna procesora, bufor dyskowy, buforowanie w systemach plików, buforowanie w sieciach komputerowych (DNS cache)

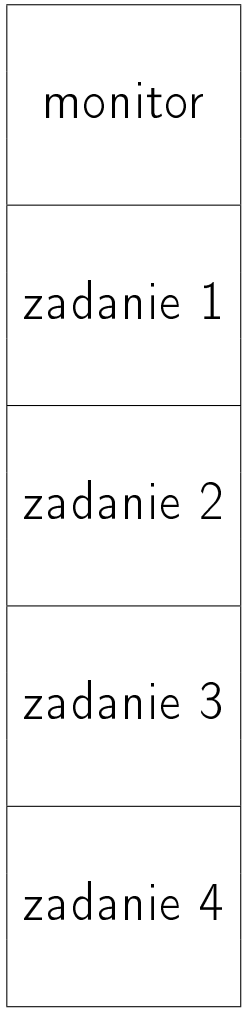
Zadania ograniczone przez CPU i I/O

- Zadania ograniczone operacjami wej-wyj (*I/O bound*)
  - szybkość przetwarzania zależy od szybkości urządzeń wej-wyj
  - buforowanie może zwiększyć przepustowość przez grupowanie operacji I/O
- Zadania ograniczone procesorem (*CPU bound*)
  - prędkość przetwarzania jest ograniczona szybkością CPU (bufor wejściowy jest zawsze pełny, a bufor wyjściowy – pusty)
  - buforowanie może zwiększyć wydajność przez gromadzenie często używanych danych i skrócenie czasu dostępu



Wieloprogramowość

- **wieloprogramowość** – w pamięci rezyduje więcej niż jeden program użytkownika, ale procesor wykonuje w danym momencie tylko jeden z nich
- po zablokowaniu zadania (np. podczas operacji I/O) system przydziela procesor innemu zadaniu aby maksymalizować wykorzystanie CPU
- **asynchroniczne operacje I/O** – procesor może kontynuować wykonywanie innych zadań, podczas gdy operacje wejścia-wyjścia są realizowane w tle



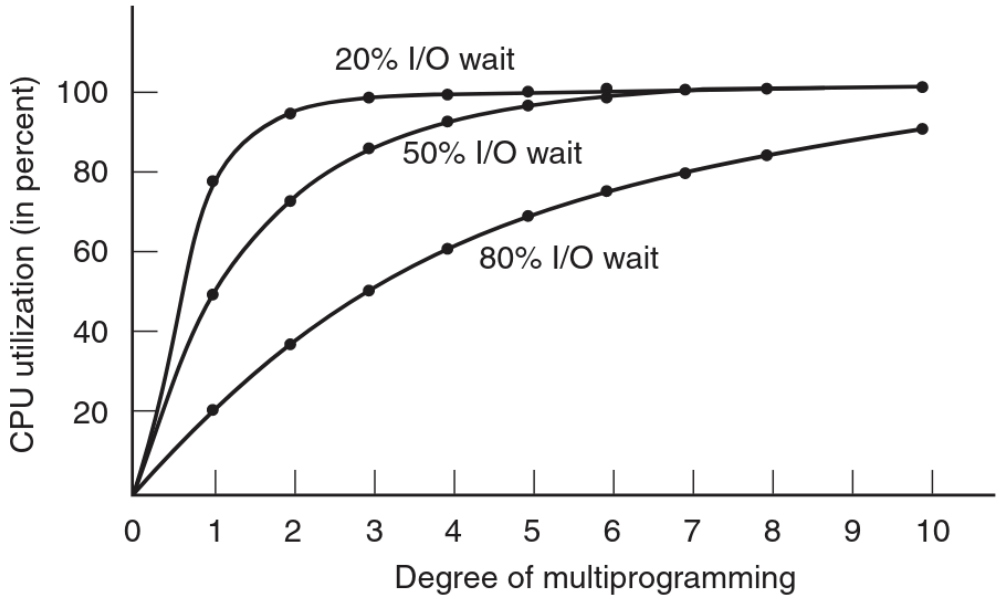
Realizacja wieloprogramowości wymaga systemu zapewniającego:

- ochronę pamięci
- planowanie przydziału procesora, zarządzanie kolejką zadań
- zarządzanie przydziałem pamięci dla zadań
- ochronę zasobów i synchronizację zadań

Wieloprogramowość: wykorzystanie CPU

Prosty model wykorzystania CPU:

- n procesów, z których każdy spędza część 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 wszystkie n procesów będą w stanie I/O
- wykorzystanie CPU $= 1 - p^n$

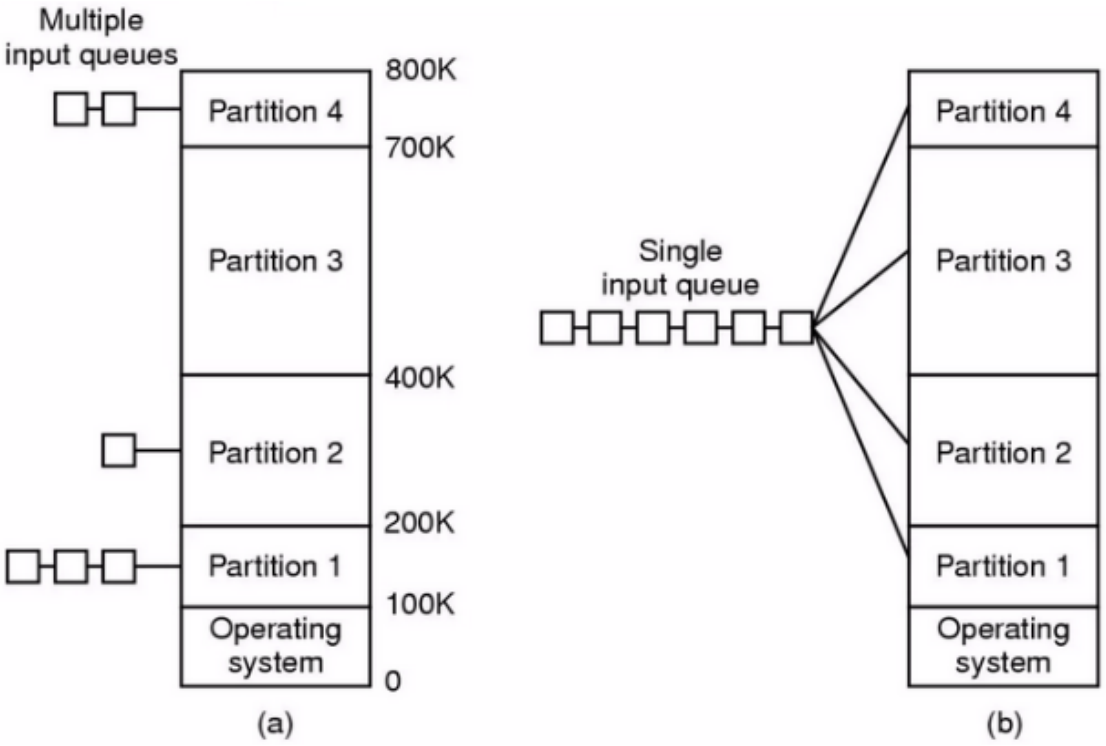


Wieloprogramowość: wykorzystanie pamięci

- podział pamięci:
  - partycje o **stałym** rozmiarze (jednakowej lub różnej wielkości)
  - podział **dynamiczny** – obszary pamięci dopasowane do rozmiaru zadania
- kolejki 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: przykład wykorzystania pamięci

- przykładowa sytuacja, w której zadania oczekują w kolejkach na przydział pamięci o stałym podziale
- pojedyncza kolejka vs. osobne kolejki dla każdej partycji



Wieloprogramowość ze stałym podziałem pamięci

a.) Osobne kolejki zadań do każdej partycji

- zadanie trafia do kolejki przypisanej najmniejszej pasującej partycji
- duże partycje mogą pozostawać puste, nawet gdy w innych kolejkach czekają zadania, które mogłyby się w nich zmieścić
- małe zadania mogą czekać

b.) Wspólna kolejka do wszystkich segmentów:

- zadanie z początku kolejki trafia do pierwszej wolnej partycji
- może powodować marnowanie pamięci na małe zadania
- potrzebny jest algorytm wyszukiwania zadań pasujących do wolnej partycji; może to jednak dyskryminować 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

Rozwiązania stosowane we współczesnych systemach operacyjnych:

- pamięć wirtualna
- segmentacja o stałym rozmiarze
- mechanizm stronicowania

Wielozadaniowość

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

Wielozadaniowość w systemach z podziałem czasu

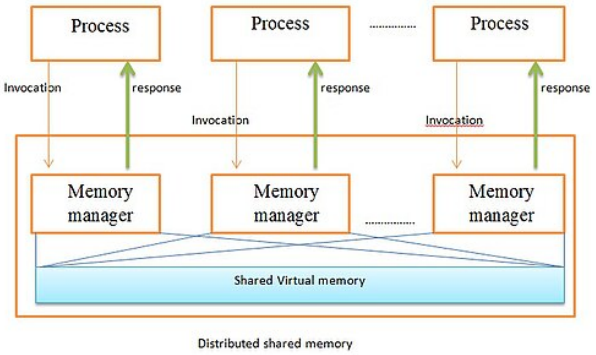
- **System z podziałem czasu** – system wielozadaniowy, w którym procesor jest przydzielany procesom na krótkie kwanty czasu.
- Po upływie kwantu przerwanie zegarowe przekazuje sterowanie jądru, które może wywłaszczyć bieżący proces i przydzielić procesor innemu gotowemu procesowi.
- Celem jest **interaktywność** – każdy użytkownik ma wrażenie ciągłej pracy systemu.

Systemy z podziałem czasu są skomplikowane, wymagają m.in.:

- realizacji mechanizmów działań współbieżnych
- zarządzania pamięcią i ochrony pamięci
- planowania przydziału CPU
- administrowania pamięcią dyskową, obsługi systemów plików
- obsługi systemu przerwań

Systemy HPC (*High Performance Computing*)

- **wieloprocesowość** (*multiprocessing*) – systemy wyposażone w więcej niż jeden procesor, umożliwiające 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 obsługę dużych baz danych
  - wolniejsza komunikacja między węzłami niż SMP



Systemy HPC (*High Performance Computing*)

- **MPP** (*Massively Parallel Processor*)
  - setki/tysiące procesorów, rozproszona pamięć
  - każdy CPU posiada własny system operacyjny i 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 ściśle 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

Podsumowanie i pytania sprawdzające

- Czym różni się wieloprogramowość od wielozadaniowości?
- Jakie są główne cechy systemów czasu rzeczywistego?
- Co to znaczy, że zadanie jest ograniczone przez procesor (*CPU bound*)?

Obsługa urządzeń wejścia-wyjścia

Jak ma współdziałać jednostka centralna z urządzeniami wejścia-wyjścia?

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

Sterowniki i moduły sterujące urządzeń

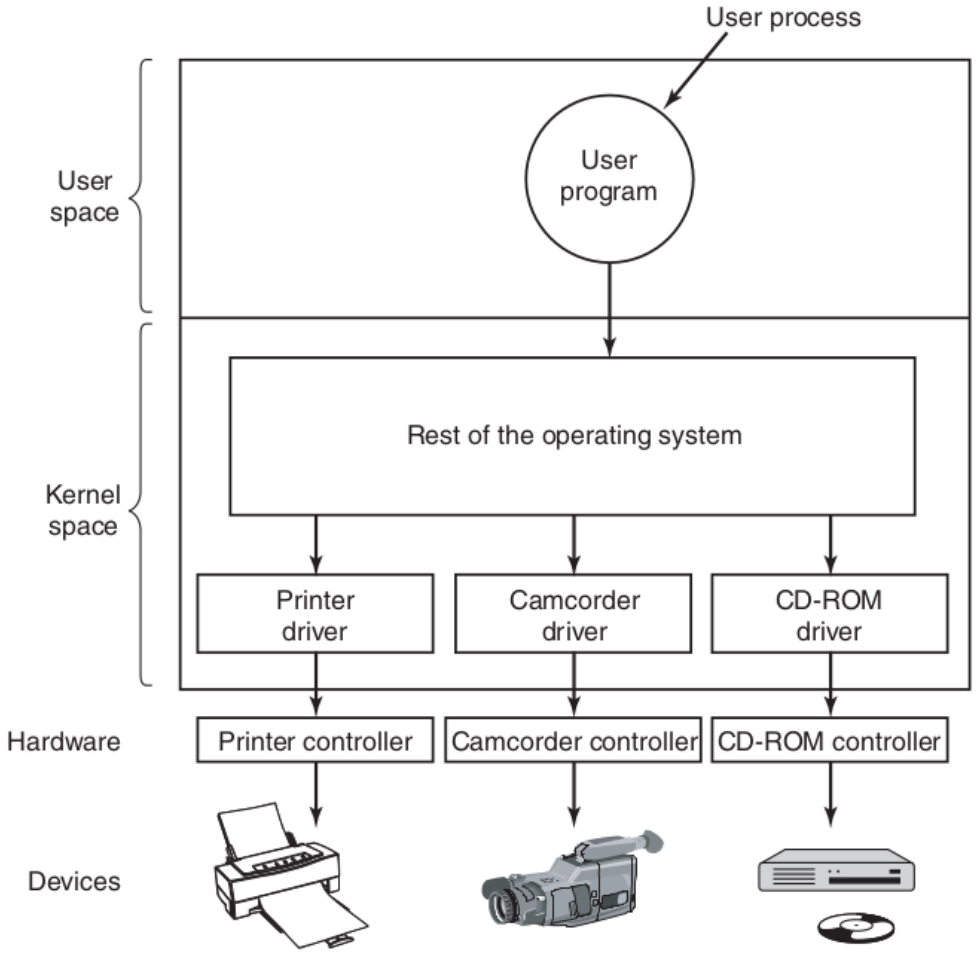


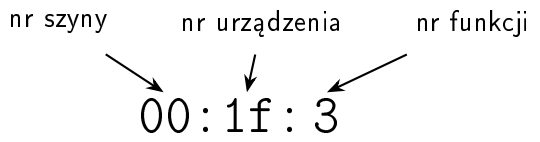
Figure 5-12. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

Sterowniki i moduły sterujące urządzeń

- **Sterownik urządzenia** (*device controller*) związany jest z konkretnym urządzeniem i dysponuje 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`

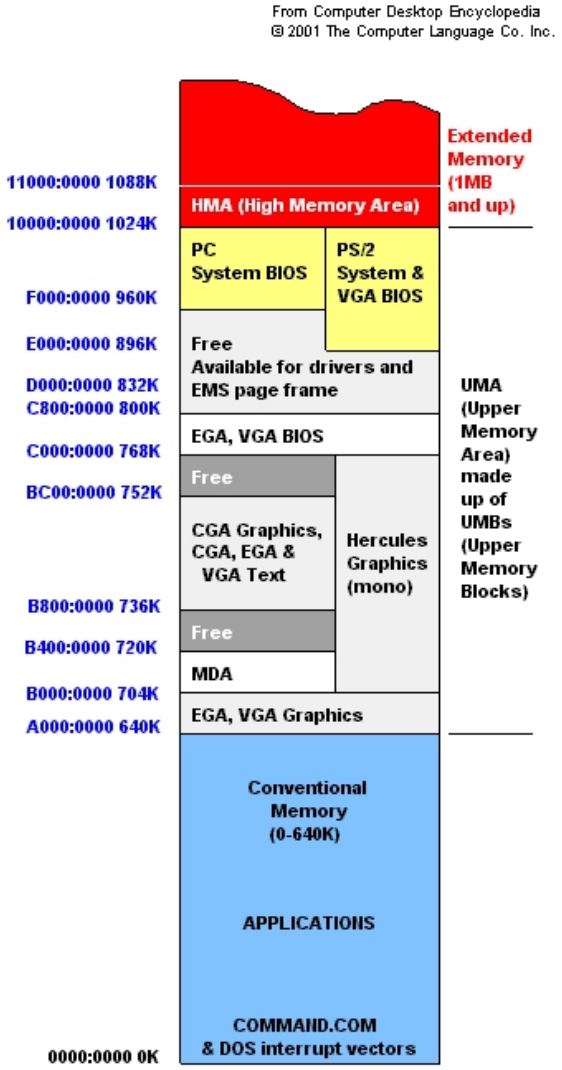


MMIO (memory-mapped input/output)

- 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[7]

[7] 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

Ograniczenia:

- marnowanie cykli procesora na oczekiwanie na dostępność zasobów
- brak reakcji systemu na inne zdarzenia w czasie oczekiwania

Odpytywanie (*polling*)

Procesor regularnie sprawdza stan kolejnych urządzeń (np. dysków, portów) 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.

Ograniczenia:

- marnowanie cykli procesora na regularne sprawdzanie stanu urządzeń
- opóźnienia w reakcji na zdarzenia w porównaniu do obsługi przerwań

Obsługa przerwań

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

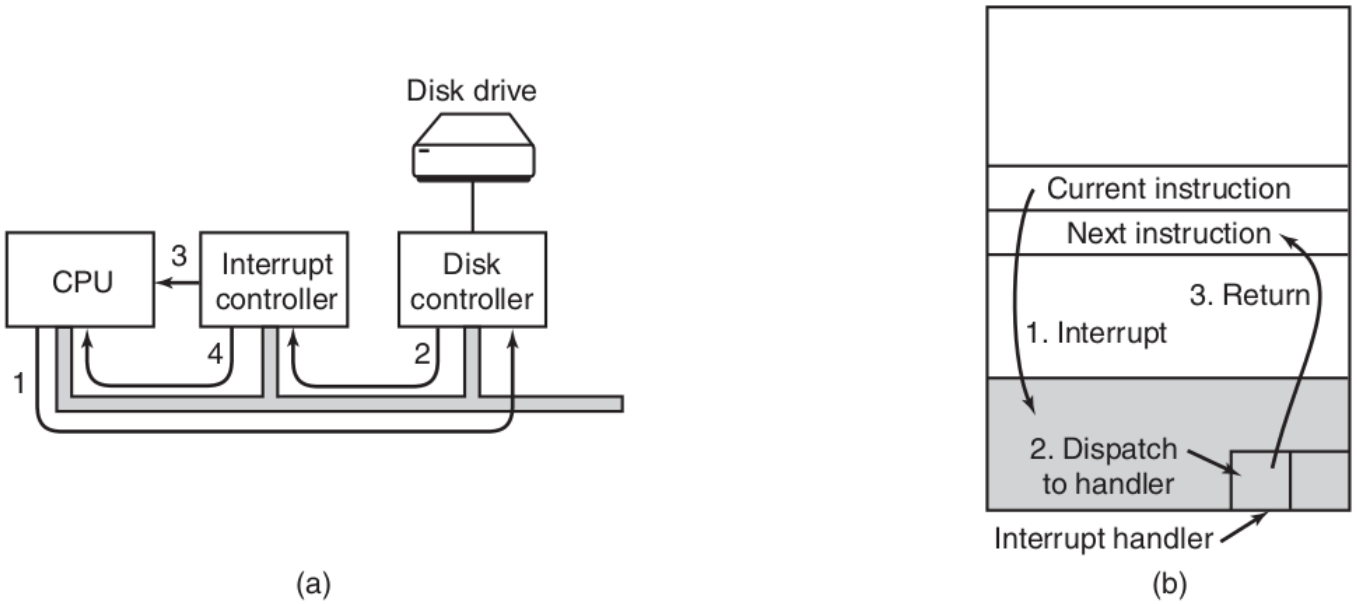


Figure 1-11. (a) The steps in starting an I/O device and getting an interrupt. (b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program.

Obsługa przerwań

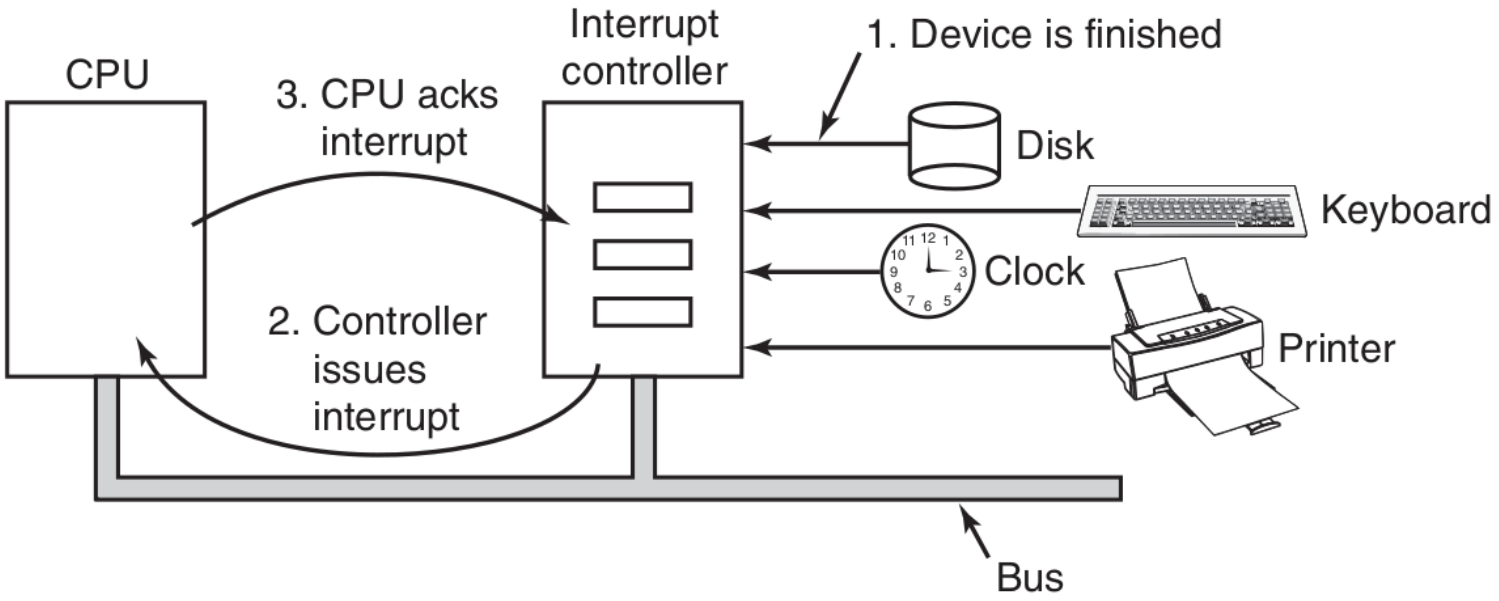


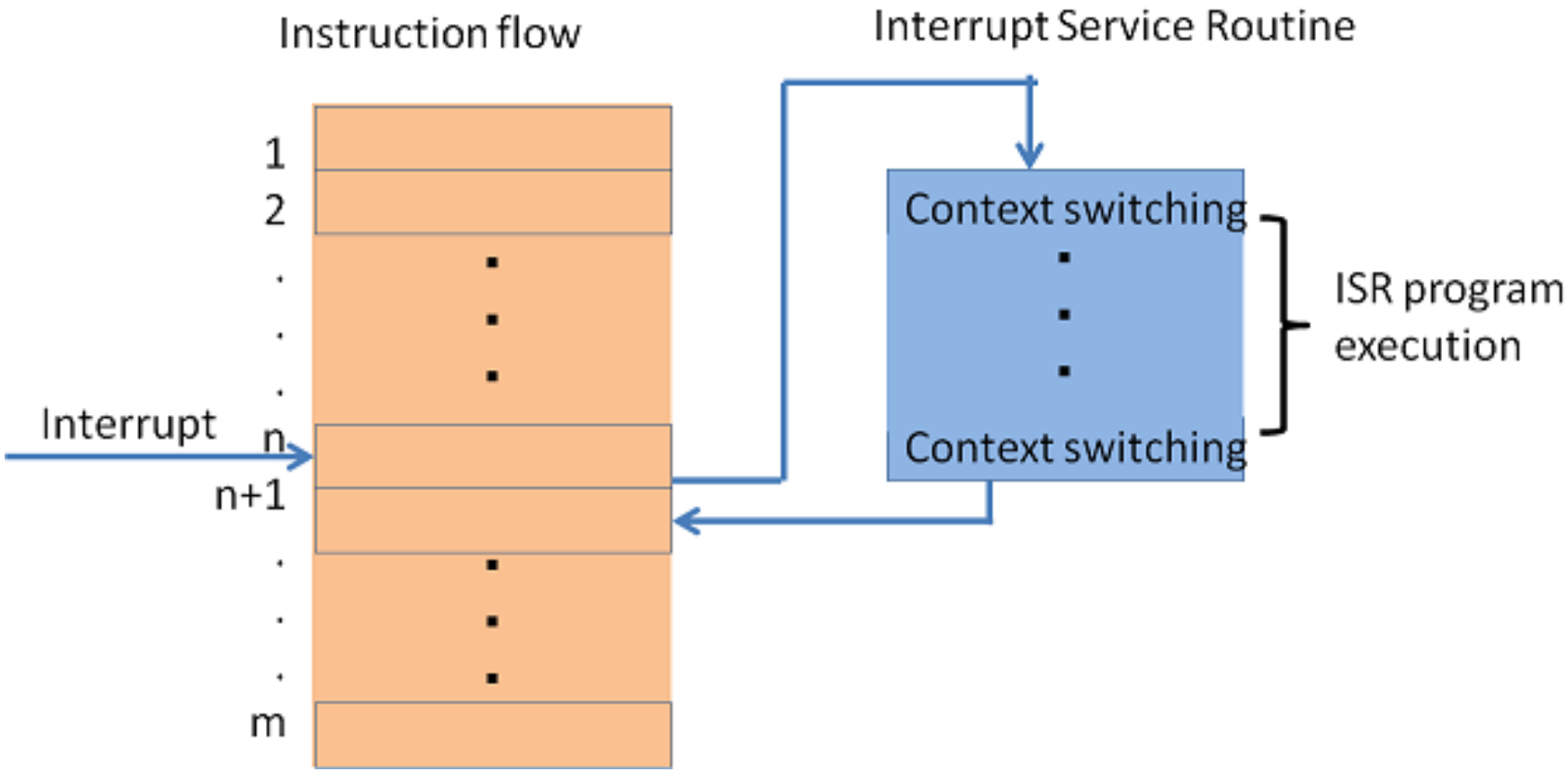
Figure 5-5. How an interrupt happens. The connections between the devices and the controller actually use interrupt lines on the bus rather than dedicated wires.

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 być krótka i wykonywana bez przerywania
  - wymagania urządzenia: jak najszybsza obsługa przerwania
  - wymagania systemu operacyjnego: jak najkrótsza obsługa
  - co zrobić, gdy podczas obsługi przerwania wystąpi następne żądanie?
- możliwe jest zagnieżdżanie obsługi przerwań lub odraczanie wykonania procedury obsługi (zależy od architektury i konfiguracji systemu)
- w Linuksie obsługa przerwania jest podzielona na etapy:
  - górna połówka (*top half*) - niezbędne operacje wykonane natychmiast po wystąpieniu przerwania
  - dolna połówka (*bottom half*) - instrukcje odłożone do późniejszej obsługi

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

Przerwania: podział

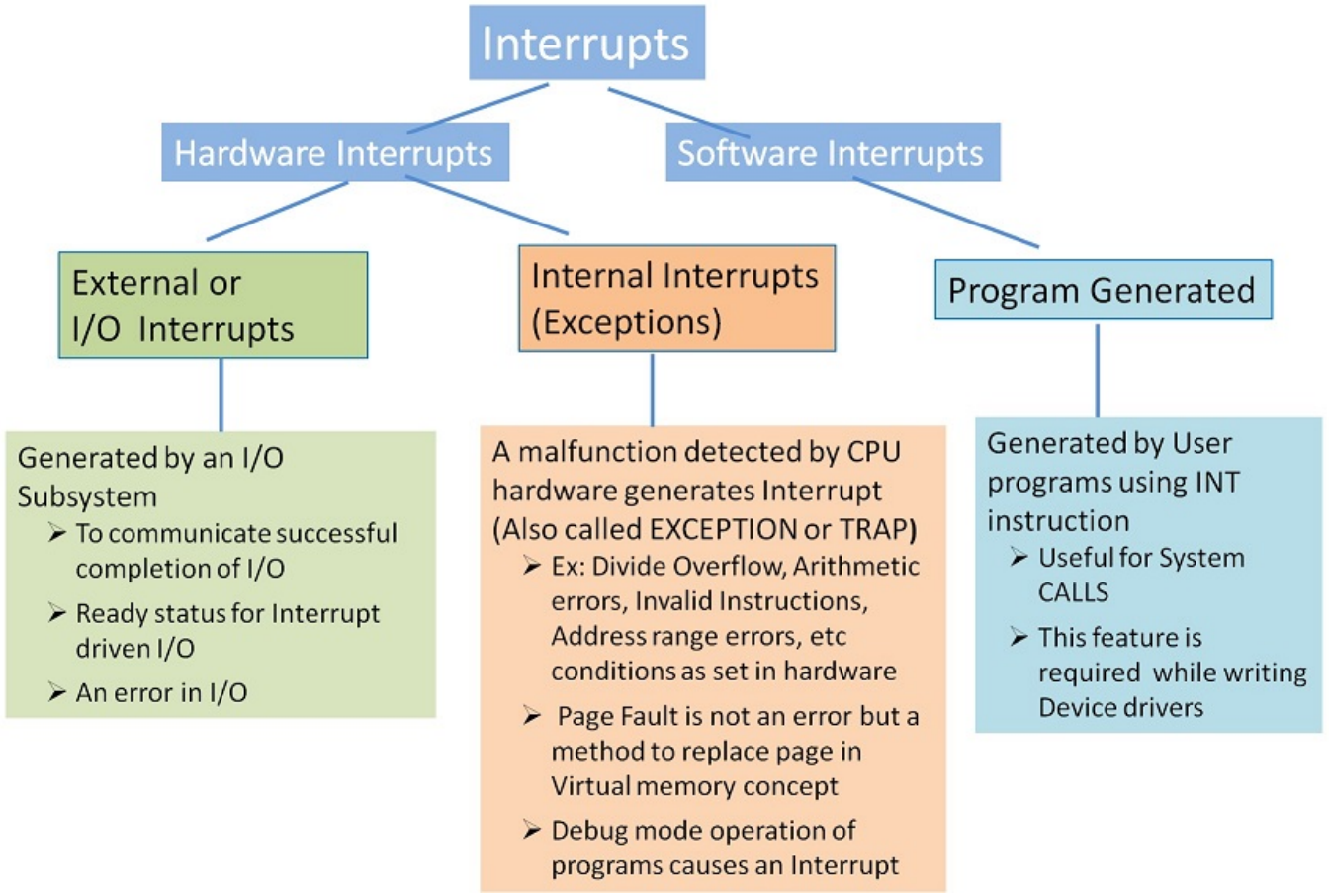
- **asynchroniczne** (przerwania) – generowane przez urządzenia sprzętowe w dowolnym czasie, niezależnie od sygnałów zegarowych procesora, np. przerwanie z karty sieciowej po odebraniu pakietu danych
- **synchroniczne** (wyjątki) – generowane przez procesor w wyniku wykonania instrukcji, tworzone przez 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), niezamaskowany
  - 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 powstają, gdy podczas wykonywania instrukcji procesor wykryje nieprawidłowość.

- **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 następnej 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 błąd sprzętu.

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

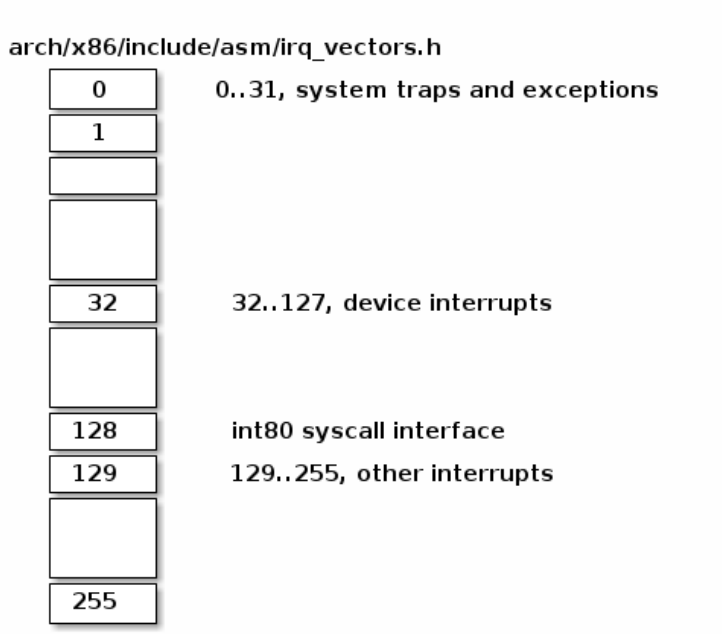
Przerwania: identyfikacja

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 znajdować się w dowolnym miejscu pamięci, pozycję tablicy wskazuje rejestr procesora `IDTR`

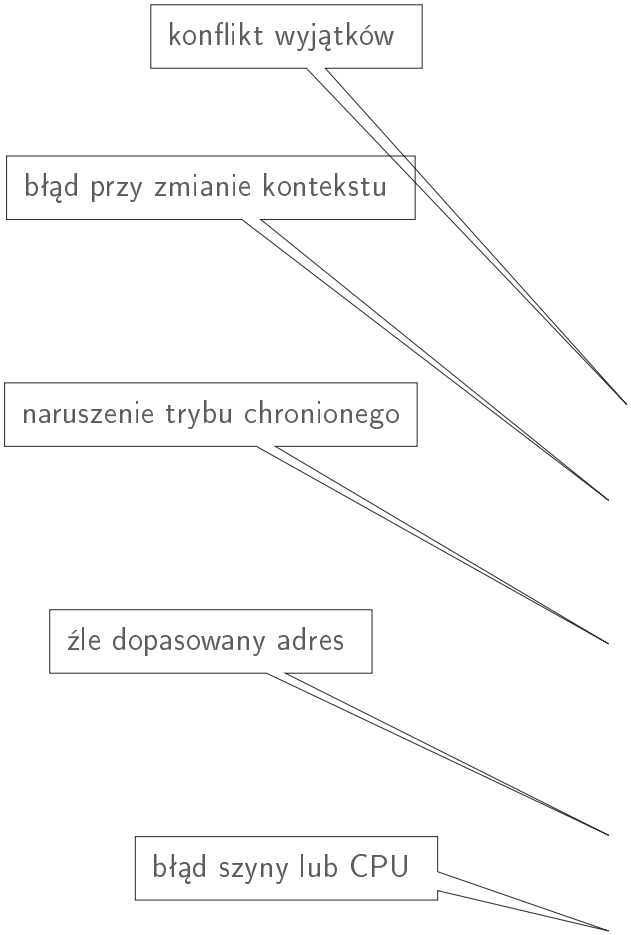


Tablica deskryptorów przerwań IDT

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 exception*	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 wystąpienie wyjątku

Sterownik przerwań PIC w systemach Intel PC

Przyporządkowanie linii IRQ do urządzeń I/O i wektora przerwań w układzie 8259A PIC

IRQ	wektor	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	podstawowy kanał IDA/ATA
15	47	wtórny kanał IDA/ATA

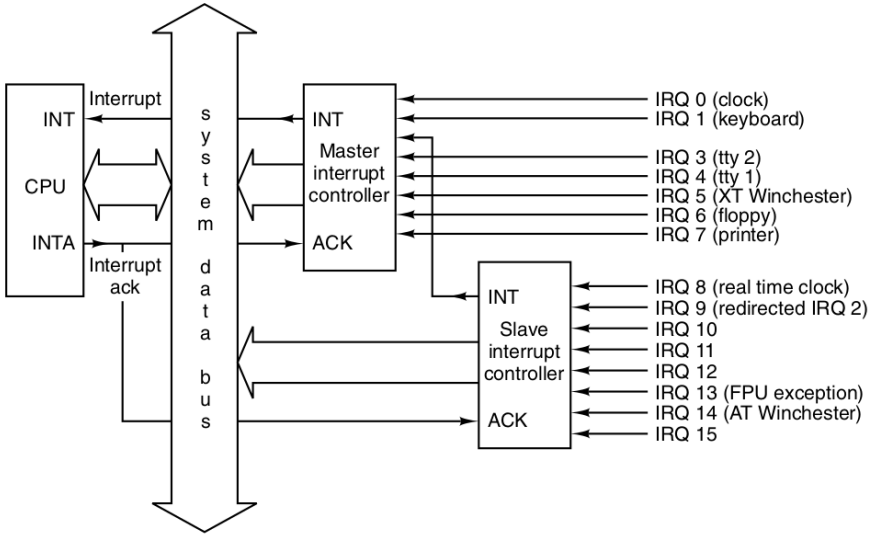


Figure 2-39. Interrupt processing hardware on a 32-bit Intel PC.

- PIC *Programmable interrupt controller* dla jednostek jednoprocesorowych
- układ 8259A obsługuje 8 linii IRQ; po połączeniu kaskadowym (przez pin 2) można obsłużyć do 15 linii IRQ

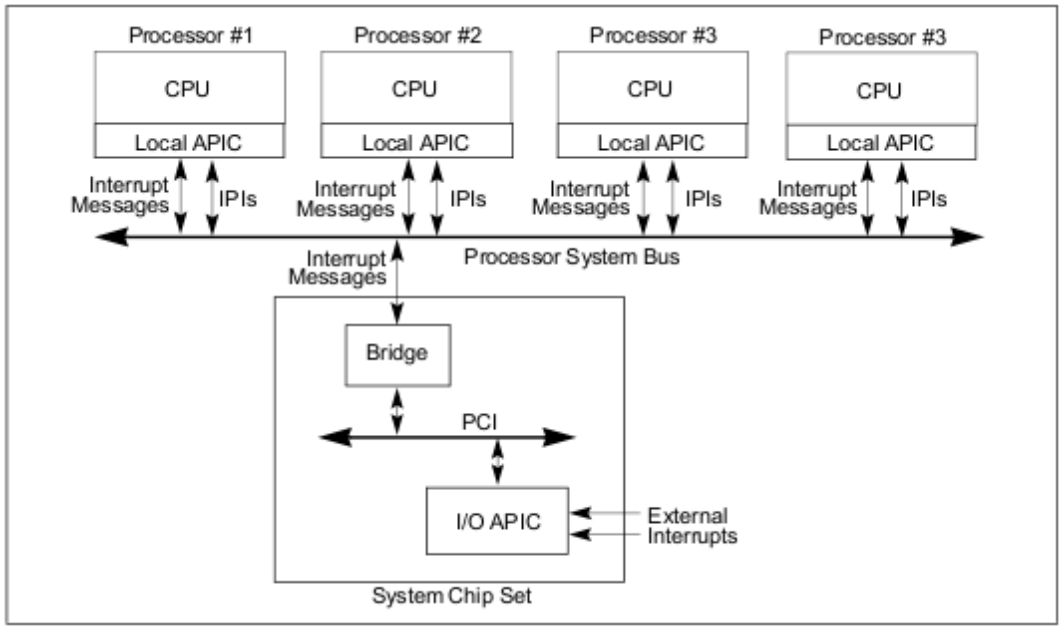
Sterownik przerwań

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

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

Kontroler APIC w systemach wieloprocesorowych SMP



Przerwania międzyprocesorowe (IPI, *Inter-Processor Interrupt*) - w systemie wieloprocesorowym (SMP) procesor może zażądać działania ze strony innego procesora (tzw. klepnięcie w ramię), np. unieważnianie wpisów TLB na innych CPU, wykonanie funkcji na innym CPU, zatrzymanie procesora.

Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide

APIC (*Advanced Programmable Interrupt Controller*)

- **Local APIC** znajduje się w każdym logicznym procesorze
  - odbiera przerwania skierowane do tego procesora
  - generuje i akceptuje przerwania międzyprocesorowe (IPIs)
  - może generować lokalne przerwania czasowe
- **I/O APIC** przy szynie systemowej
  - odbiera przerwania I/O i kieruje je komunikatami APIC do LAPIC
  - w wielu komputerach PC ma 24 wejścia pinowe (0-23)
  - tabela przekierowań IRT (*Interrupt Redirection Table*) mapuje wejście na wektor, docelowy procesor i tryb dostarczania; system konfiguruje ją przy inicjalizacji
  - obecnie komunikaty MSI/MSI-X omijają wejścia pinowe I/O APIC kierując przerwania bezpośrednio do LAPIC

Przerwania sygnalizowane komunikatami (MSI)

Message Signaled Interrupts

- mechanizm MSI (PCI 2.2) i MSI-X (PCI 3.0, stosowane także w PCIe) zgłaszają przerwania przez zapis pod specjalnie zarezerwowany adres pamięci (w x86 jest to adres rejestru LAPIC)
- komunikat jest małym pakietem danych zawierającym informacje o przerwaniu
- MSI-X umożliwia do 2048 wektorów na funkcję PCI; wektory można niezależnie konfigurować i kierować do wybranych procesorów
- MSI/MSI-X nie korzysta z dedykowanych linii przerwań; może ograniczyć współdzielenie przerwań i ułatwiać obsługę wielu kolejek
- jądro systemu konfiguruje MSI/MSI-X i przypisanie wektorów, rejestruje procedury obsługi, a po dostarczeniu przerwania uruchamia właściwą procedurę obsługi

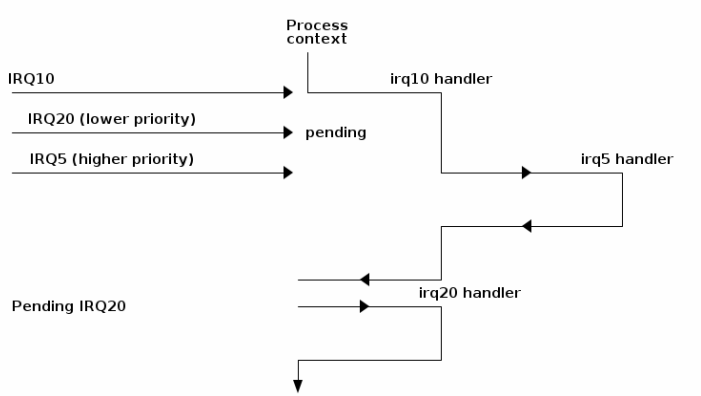
Maskowanie przerwań a synchronizacja

Synchronizacja dostępu do współdzielonych danych pomiędzy procedurą obsługi przerwań a innymi potencjalnymi współbieżnymi działaniami może wymagać włączenia i wyłączenia przerwań w kontrolowany sposób:

System może sterować przerwaniami na trzech poziomach:

- Urządzenie - sterownik może wyłączyć zgłaszanie przerwań przez rejestry urządzenia.
- Kontroler przerwań - PIC lub I/O APIC może zamaskować źródło i wstrzymać dostarczanie jego przerwań. To, czy zgłoszenie pozostanie oczekujące, zależy od urządzenia, kontrolera i trybu wyzwalania; nie każde zdarzenie musi zostać zapamiętane.
- Procesor - w architekturze x86 instrukcje `cli` i `sti` zmieniają flagę IF (*interrupt flag*) na bieżącym CPU, blokując lub dopuszczając maskowalne przerwania; nie blokują NMI ani wyjątków i nie wpływają na pozostałe CPU.

Priorytety przerwań



- IRQ o wyższym priorytecie może przerwać obsługę IRQ o niższym, jeśli kontroler i konfiguracja procesora dopuszczają zagnieżdżanie
- Niższy IRQ pozostaje oczekujący podczas obsługi wyższego i może zostać dostarczony później, o ile kontroler lub urządzenie zachowało zgłoszenie
- priorytet może być określany przez numer IRQ, konfigurację kontrolera lub ustawienia procesora
- priorytety przerwań są wspierane przez większość architektur, choć ich obsługa rzadko występuje w systemach ogólnego przeznaczenia
- priorytety są przydatne w systemach czasu rzeczywistego, pomagają ograniczać opóźnienia

Problemy operacji wejścia-wyjścia

Podczas programowej obsługi wejścia-wyjścia (aktywnego czekania lub odpytywania), a także podczas obsługi sterowanej przerwaniami, procesor jest zaangażowany w przesyłanie danych między 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*) albo przez funkcję zintegrowaną z chipsetem lub kontrolerem urządzenia wej-wyj (np. karty sieciowej albo dysku SSD).

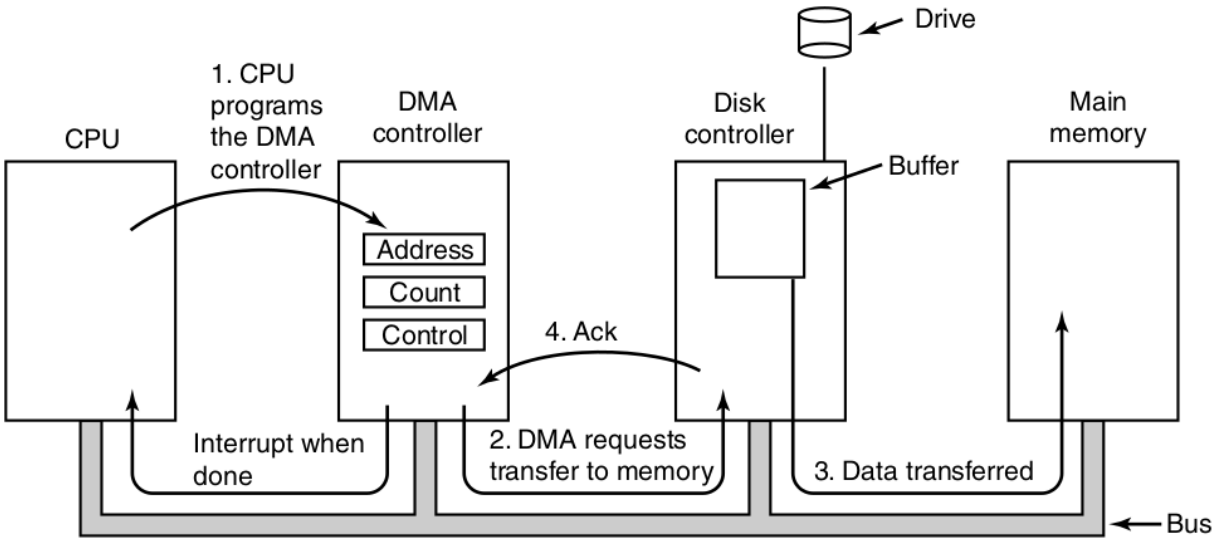


Figure 3-4. Operation of a DMA transfer.

Bezpośredni dostęp do pamięci (DMA)

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

- czy wymagany jest odczyt/zapis
- adres urządzenia wej-wyj
- adres początkowej komórki pamięci, z której dane mają być odczytane lub do której mają być zapisane
- liczbę słów do przesłania

Procesor inicjuje przesłanie danych i kontynuuje wykonywanie innych zadań do momentu nadejścia przerwania od modułu DMAC.

DMA ogranicza udział procesora w przesyłaniu danych, pozwala urządzeniom peryferyjnym przesyłać dane bezpośrednio między pamięcią operacyjną a urządzeniem, przyspieszając w ten sposób operacje wejścia/wyjścia.

Monitorowanie przerwań sprzętowych w Linuksie

```
$ 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 w Linux

```
# 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 Linuksie: obsługa bezpośrednia i odroczona

- **obsługa bezpośrednia** (*top half*)
  - wykonywana w kontekście sprzętowego przerwania; powinna być krótka
  - wykonuje pilne czynności, które mają kluczowe znaczenie dla funkcjonowania urządzenia (np. odczytuje stan urządzenia lub potwierdza przerwanie)
  - może zlecić wykonanie dalszej pracy w trybie odroczonym
  - może opóźnić obsługę innych przerwań na tym samym CPU
  - wykonywanie zbyt długich operacji przy zablokowanych przerwaniach powoduje opóźnienia, problemy z wydajnością i gubienie przerwań
- **obsługa odroczona** (*bottom half*)
  - wykonuje pracę, która nie musi być zrealizowana bezpośrednio w obsłudze sprzętowego IRQ, która może być odroczona do momentu mniejszego obciążenia systemu
  - zależnie od mechanizmu działa np. jako softirq albo w wątku jądra

Mechanizmy odroczania pracy w Linux

softirq

- rodzaje softirq są zdefiniowane statycznie; zgłaszają je podsystemy jądra, m.in. obsługa przerwań (dolna połówka)
- ten sam rodzaj softirq może być obsługiwany równolegle na różnych CPU

tasklet - szczególny rodzaj softirq

- tworzone dynamicznie (np. podczas ładowania modułu)
- ten sam tasklet jest wykonywany najwyżej na jednym CPU naraz

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 - softirq wysokiego priorytetu
TIMER - przerwania zegarowe
NET_TX - wysyłanie pakietów sieciowych
NET_RX - odbiór pakietów sieciowych
BLOCK - operacje blokowe (np. kończenie operacji I/O)
IRQ_POLL - odpytywanie urządzeń o zakończenie operacji
TASKLET - tasklety
SCHED - prace związane z planistą (np. bilansowanie obciążenia)
HRTIMER - wysokorozdzielcze timery
RCU - mechanizm synchronizacji jądra RCU (Read-Copy Update)

Monitorowanie przerwań sprzętowych: klawiatura i mysz

Fragment wpisu /var/log/syslog dotyczący przerwań klawiatury i myszy:

```
kernel: i8042: PNP: PS/2 Controller [PNP0303:KBD,PNP0f13:MOU] at 0x60,0x64 irq 1,12
kernel: serio: i8042 KBD port at 0x60,0x64 irq 1
kernel: serio: i8042 AUX port at 0x60,0x64 irq 12
```

Przytrzymanie klawisza IRQ=1

```
$ sar -I --int=1 1

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
...
09:38:24            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
```

Ruch gładzika (touchpad) IRQ=12

```
$ sar -I --intr=12 1

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
...
09:46:07           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

Wpis z `lspci` dotyczący karty sieciowej wlan0:

```
$ lspci -v
00:14.3 Network controller: Intel Corporation Cannon Point-LP CNVi [Wireless-AC] (rev 30)
        Subsystem: Intel Corporation Cannon Point-LP CNVi [Wireless-AC]
        Flags: bus master, fast devsel, latency 0, IRQ 173, IOMMU group 6
        Memory at b3938000 (64-bit, non-prefetchable) [size=16K]
        Capabilities: <access denied>
        Kernel driver in use: iwlwifi
        Kernel modules: iwlwifi, wl
```

Strumień pakietów na interfejsie wlan0

```
$ ping -i 0.002 -c 1000 192.168.1.1
```

Przerwania karty sieciowej wlan0

```
$ sar -I --intr=173 1

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
...
Średnia:          174    363,70
```

Wieloprogramowość i ochrona sprzętowa

- w systemach wieloprogramowych i z podziałem czasu operacje wejścia-wyjścia nakładają się na działanie jednostki centralnej (asynchroniczne wej-wyj).
- procesy współdzielą procesor, pamięć i urządzenia wejścia-wyjścia
- błąd programu nie może pozwolić mu zmienić kodu jądra ani pamięci innych procesów
- zarządzanie wieloma programami wymaga odpowiednich mechanizmów ochrony, aby zapewnić bezpieczeństwo i stabilność systemu operacyjnego
- system operacyjny i jego jądro są programami, dlatego potrzebny jest mechanizm odróżniający kod wykonywany przez użytkownika od kodu jądra systemu
- ochrona sprzętowa: izolacja zasobów, ochrona pamięci, tryb uprzywilejowany procesora

Przerwanie przekazuje sterowanie do jądra

- Procedury obsługi przerwań są wykonywane w trybie uprzywilejowanym i mają dostęp do zasobów jądra.
- Gdy procesor wykonuje kod użytkownika i przyjmuje przerwanie lub wyjątek, przekazuje sterowanie do procedury obsługi w trybie uprzywilejowanym.
- Procesor zachowuje stan potrzebny do obsługi zdarzenia i późniejszego wznowienia pracy.
- Po obsłudze jądro może wznowić przerwany kod, wybrać inne zadanie albo zakończyć proces, który spowodował wyjątek.
- Jeśli zdarzenie wystąpi podczas wykonywania kodu jądra, procesor pozostaje w trybie uprzywilejowanym.

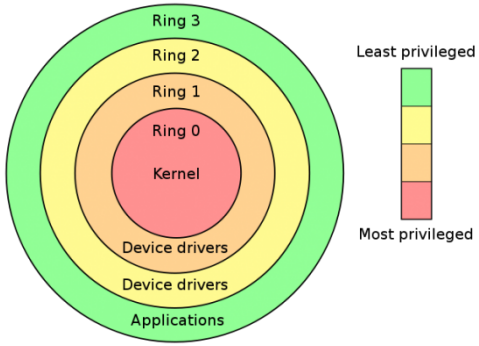
Sprzętowe mechanizmy ochrony

- Procesor rozróżnia poziomy uprawnień; kod użytkownika nie może swobodnie wykonywać operacji uprzywilejowanych
  $\longrightarrow$ dualny tryb pracy procesora: użytkownik i jądro.
- Dostęp do portów wejścia-wyjścia jest kontrolowany przez mechanizmy uprawnień procesora i system operacyjny. Rozkazy wejścia-wyjścia są uprzywilejowane, dlatego tylko system operacyjny może komunikować się z urządzeniami.
- W x86 są cztery poziomy Ring 0-3 - tzw. pierścienie uprzywilejowania. Systemy operacyjne (Linux/Windows) używają zwykle Ring 0 dla jądra i Ring 3 dla programów użytkownika; poziomy 1 i 2 są na ogół niewykorzystywane.

Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, oficjalna dokumentacja.

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
- pierścienie 1 i 2 - rzadko używane, czasem dla modułów urządzeń wejścia-wyjścia
- pierścień 0 - najwyższe uprawnienia; uprzywilejowane operacje może wykonywać tylko jądro systemu, niedostępne bezpośrednio dla programistów i użytkowników

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

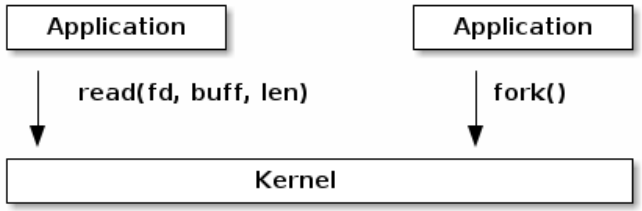
Dualny tryb pracy

- **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 działają z ograniczonymi uprawnieniami i mogą zlecić systemowi wykonanie zastrzeżonych operacji (np. dostępu do zasobów) za pośrednictwem wywołań systemowych (*system calls*).

  Wywołanie systemowe (*system call*) 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 lub programisty wywołania systemowe są „usługami" oferowanymi aplikacjom przez jądro. Przypominają funkcje API bibliotek: mają nazwę, parametry i wartość zwracaną.[8]



Wywołanie systemowe to żądanie usługi jądra. Programy zwykle korzystają z funkcji opakowujących udostępnianych przez bibliotekę C, które wykonują:

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

[8] Biblioteka C w Linuksie udostępnia wiele interfejsów zgodnych z POSIX (*Portable Operating System Interface*); Linux oferuje również interfejsy specyficzne dla systemu.

- biblioteki systemowe udostępniają funkcje implementujące wywołania systemowe, dzięki czemu można z nich łatwo korzystać w aplikacjach
- interfejs wywołań systemowych w Linuksie (podobnie jak w większości systemów Unix) jest częściowo udostępniany przez bibliotekę 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 wywoływać funkcji 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 wywołanie systemowe jest identyfikowane przez numer umieszczany w rejestrze `eax`
- argumenty wywołania systemowego umieszczane są w osobnych rejestrach (np. `ebx`, `ecx`, `edx`, `esi`, `edi` w architekturze 32-bitowej x86)

Przerwania programowe i wywołania systemowe

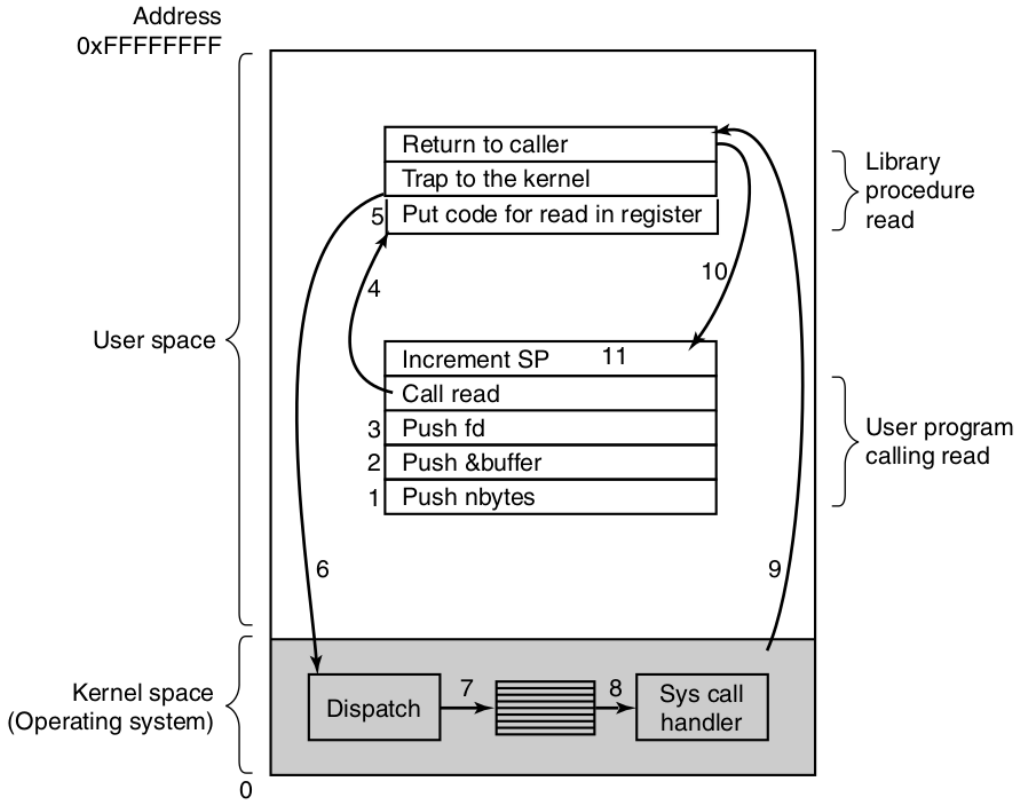


Figure 1-16. The 11 steps in making the system call read(fd, buffer, nbytes).

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_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_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
```

Opis funkcji systemowych Linux dostępny jest w podręczniku `man` (np. `man 2 read`).

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
```

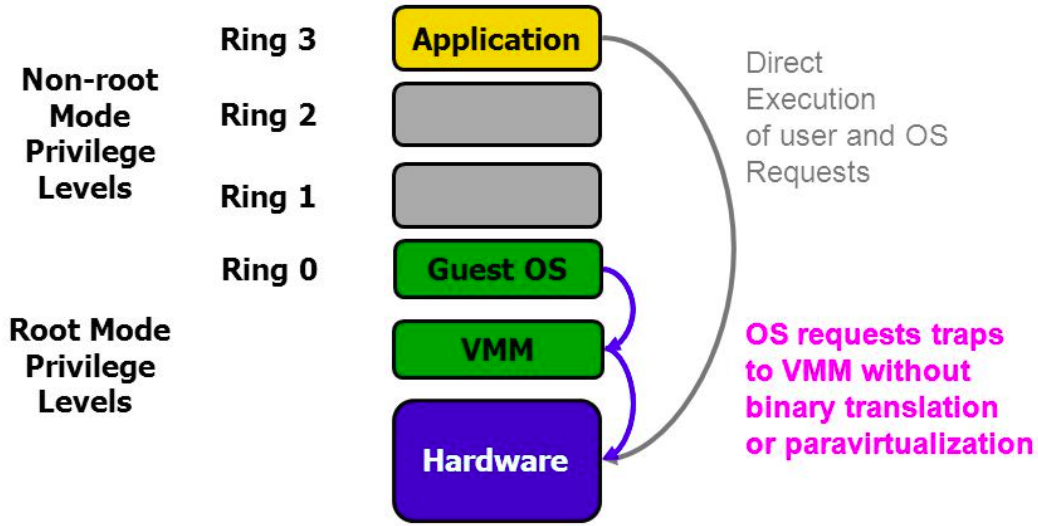
`time` - udział w łącznym czasie zarejestrowanym w wywołaniach, `seconds` - łączny zmierzony czas, `usecs/call` - średni czas jednego wywołania, `calls` - liczba wywołań, `errors` - zlicza wywołania zakończone błędem

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 wykonania polecenia
- `user` czas CPU spędzony w trybie użytkownika
- `sys` czas CPU spędzony w trybie jądra

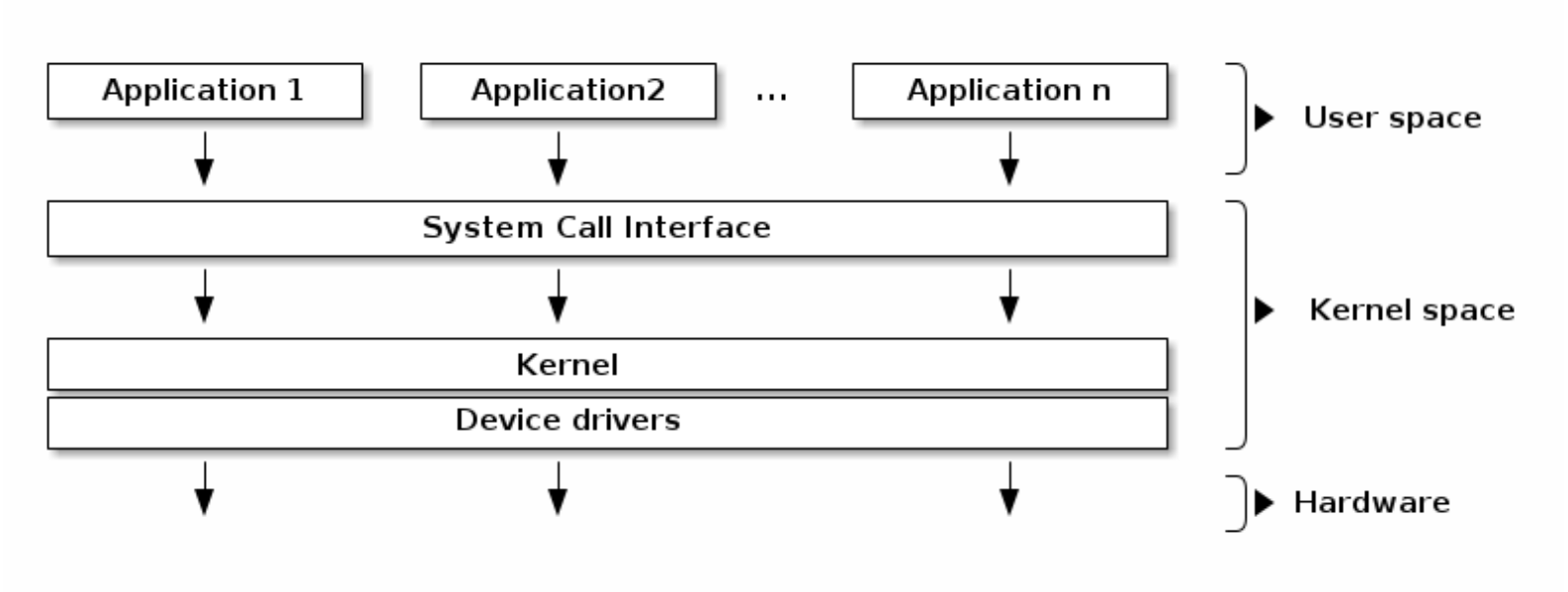
Pierścienie uprzywilejowania i wirtualizacja



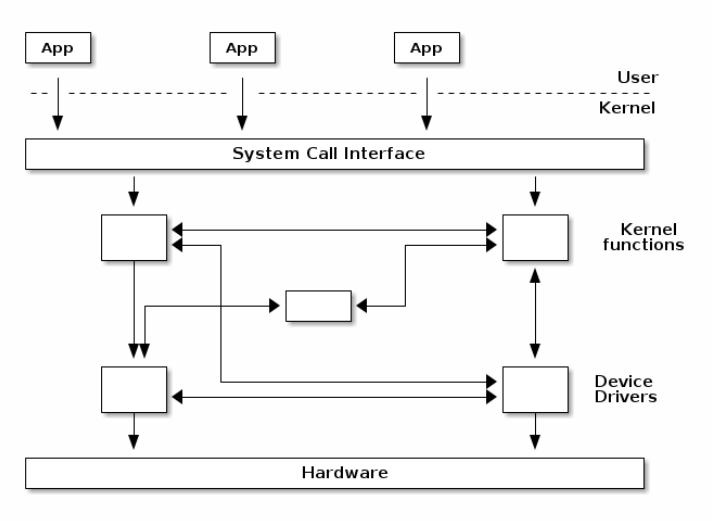
Mechanizmy sprzętowej wirtualizacji, takie jak Intel VT-x, wprowadzają tryby VMX root i VMX non-root. Hiperwizor (np. KVM) zarządza przejściami między trybem hosta i gościa; system gościa może działać w Ring 0 w trybie VMX non-root.

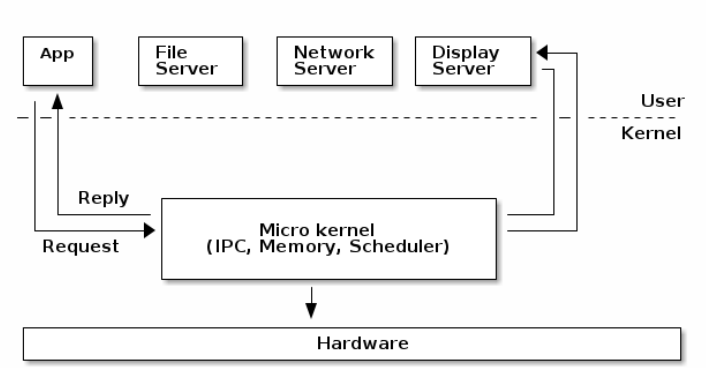
http://slideplayer.com/slide/5084764/

Typowa architektura systemu



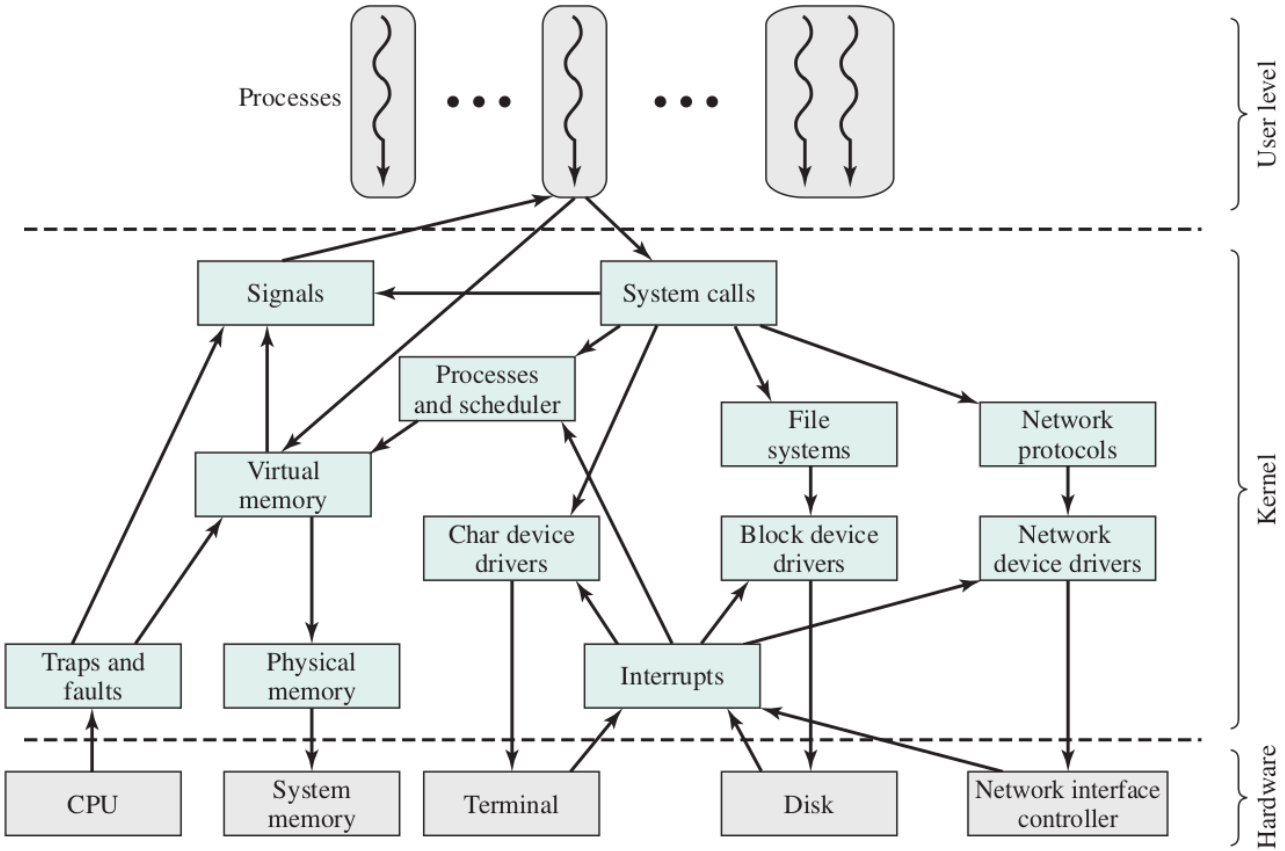
Monolityczne jądro vs. mikrojądro





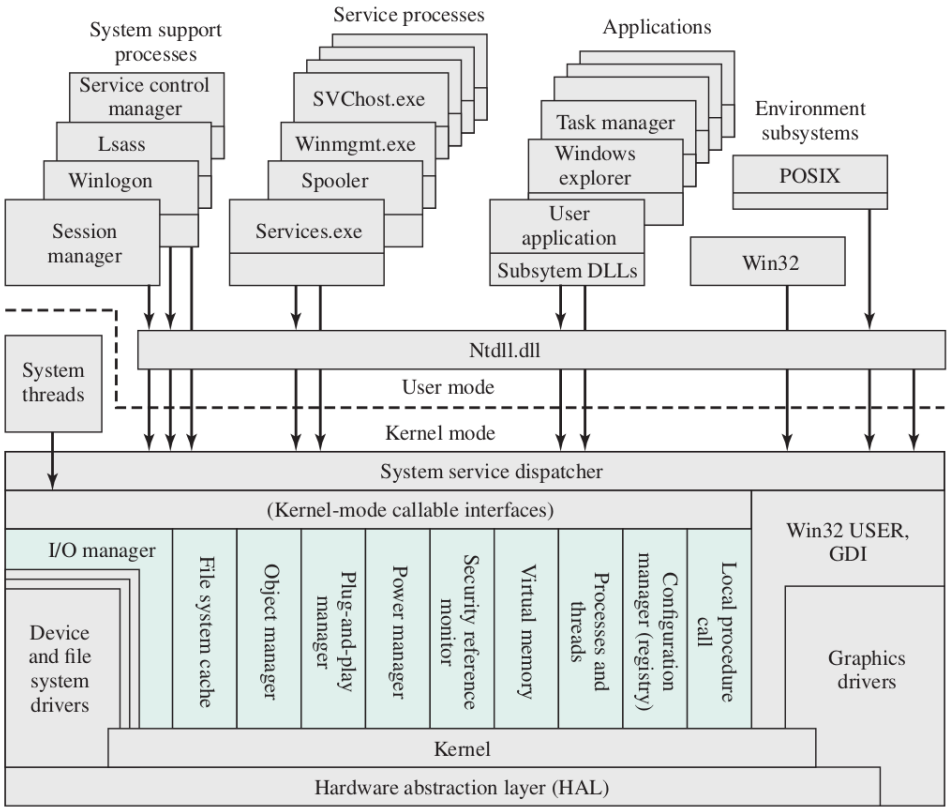
- **Jądro monolityczne** nie ma ochrony sprzętowej dostępu pomiędzy różnymi podsystemami jądra. Podsystemy mogą wywoływać swoje funkcje bezpośrednio, co ogranicza koszt komunikacji (np. Linux)
- **Mikrojądro** utrzymuje w trybie jądra tylko podstawowe mechanizmy. Wiele usług systemu działa w odizolowanych procesach w przestrzeni użytkownika. Taki podział zwiększa izolację: awaria usługi nie musi zakończyć się awarią całego systemu. Wymaga jednak komunikacji między procesami, co może zwiększać narzut i złożoność systemu (np. Minix)
- jądra hybrydowe - systemy Mac OS X i Windows wykonują znaczną część usług i sterowników w trybie jądra. Czasem kwalifikuje się je jako monolityczne (Mac OS X, Windows)
- 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 (zaznaczony na rysunku kolorem), który udostępnia API oprogramowaniu działającemu w trybie użytkownika.

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

Pytania sprawdzające

- Co to jest przerwanie sprzętowe i jak jest obsługiwane przez system operacyjny?
- Wyjaśnij różnicę między trybem uprzywilejowanym a trybem użytkownika.
- Porównaj jądro monolityczne i mikrojądro.

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

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

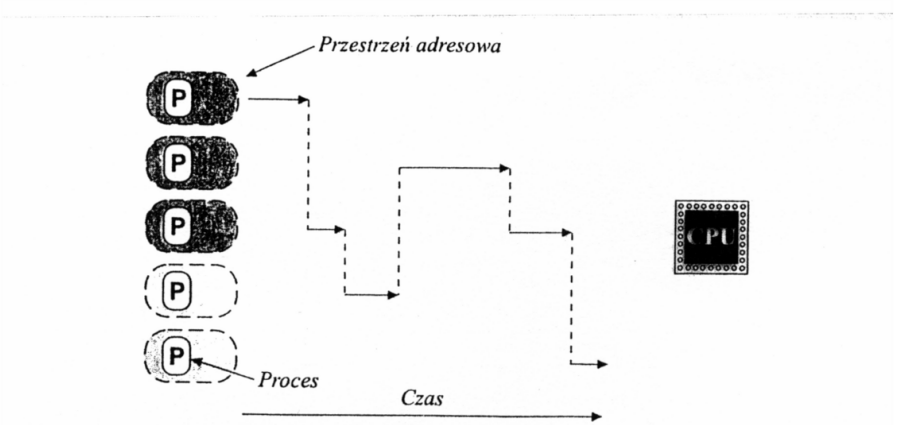
System operacyjny odpowiada za wykonywanie następujących czynności:

- 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
- zarządzanie przydziałem pamięci dla procesów, wirtualizacja pamięci i ochrona dostępu
- monitorowanie działania procesów i dostarczanie narzędzi do diagnostyki

Procesy: klasyfikacja

- **Rola:**
  - procesy systemowe wykonują kod systemu
  - procesy użytkowników wykonują programy użytkowników
- **Sposób interakcji:**
  - procesy interaktywne wymagają bieżącej interakcji z użytkownikiem (np. terminal, aplikacja GUI)
  - procesy wsadowe wykonują zadania bez bieżącej interakcji z użytkownikiem
- **Tryb działania:**
  - procesy pierwszoplanowe, związane z bieżącą interakcją
  - procesy działające w tle (np. demony, usługi)

Współbieżność (pseudo-paralelizm) - w jednoprocesorowym 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, czyli ich rzeczywiste równoczesne wykonywanie.

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:

Dobrowolne:

- normalne wyjście po zakończeniu pracy
  - Unix/Linux: `exit()`
  - Windows WinAPI: `ExitProcess()`
- wyjście zakończone błędem, gdy proces wykryje sytuację uniemożliwiającą dalsze wykonywanie

Niedobrowolne:

- krytyczny błąd, w wyniku wykonania niepoprawnej instrukcji
- zabicie przez inny (uprawniony) proces, poprzez wywołanie funkcji systemowej
  - Unix/Linux: `kill()`
  - Windows WinAPI: `TerminateProcess()`

Hierarchia 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 relacji rodzic-potomek (drzewo procesów), pierwszy proces (korzeń) startuje podczas inicjacji systemu (np. `init`, `systemd`), a wszystkie pozostałe są jego potomkami bezpośrednimi lub pośrednimi
- niezależnie od tej hierarchii proces należy do jednej **grupy procesów**, której członkowie mogą wspólnie odbierać sygnały. Każdy proces może mieć własną procedurę obsługi sygnału albo go zignorować. Domyślnie proces potomny należy do tej samej grupy procesów co jego rodzic
- nie wszystkie systemy stosują hierarchię w takim samym stopniu: w Windows proces nadrzędny uzyskuje **uchwyt** do potomka, za pomocą którego może nim zarządzać; uchwyt można jednak przekazać innemu procesowi.

Hierarchia procesów

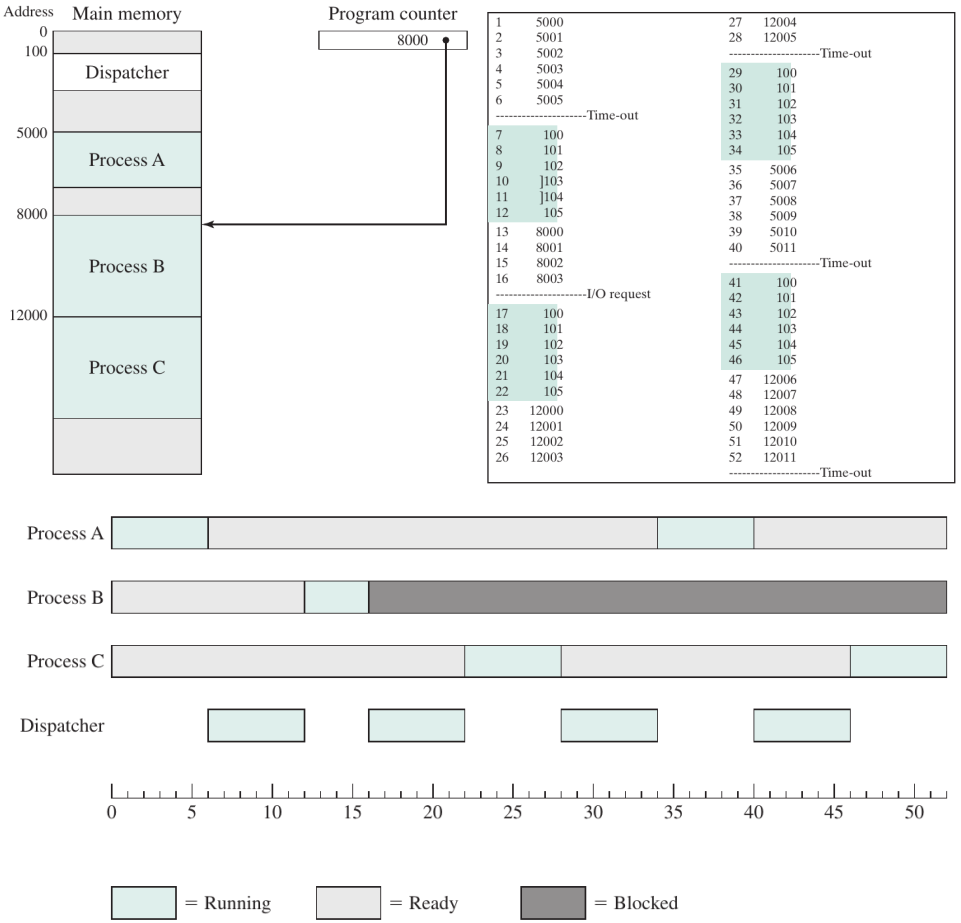
```
$ pstree
systemd-+-ModemManager---2*[{ModemManager}]
        |-NetworkManager---2*[{NetworkManager}]
        |-accounts-daemon---2*[{accounts-daemon}]
        |-acpid
        |-atop
        |-atopacctd
        |-avahi-daemon---avahi-daemon
        |-blueman-mechani---2*[{blueman-mechani}]
        |-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

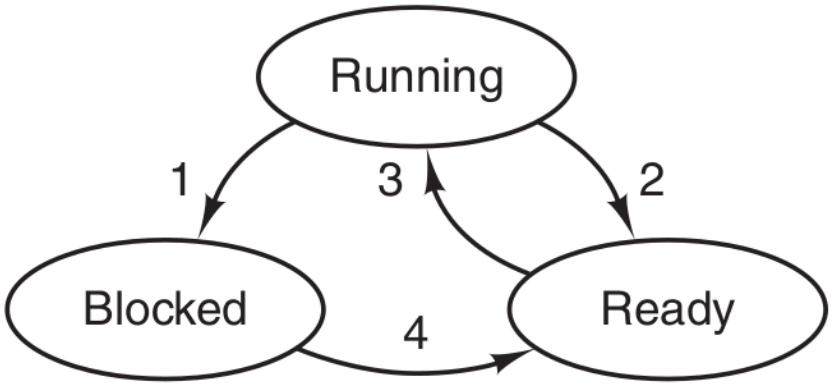
- **gotowy** (*ready*) – proces czeka na przydział procesora
- **bieżący** (*running*) – aktualnie używający CPU, są wykonywane instrukcje
- **zablokowany** (oczekujący, *blocked*) – czeka na wystąpienie jakiegoś zdarzenia (np. zakończenia operacji wejścia-wyjścia)



Przykładowy przebieg procesów w systemie

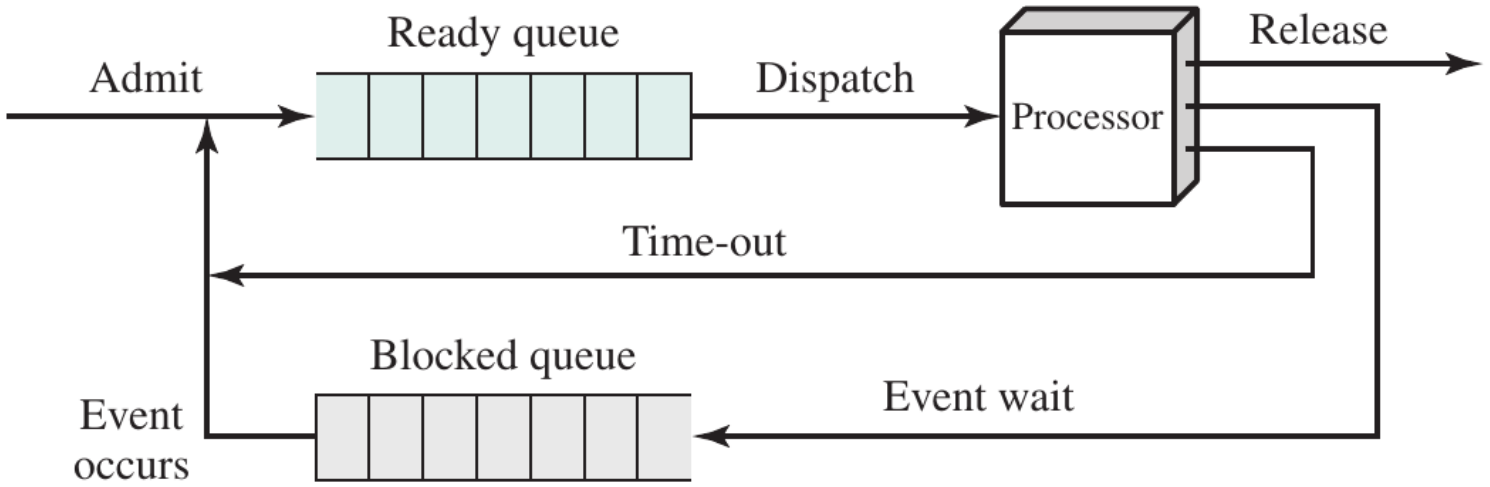


Procesy: zmiana stanu



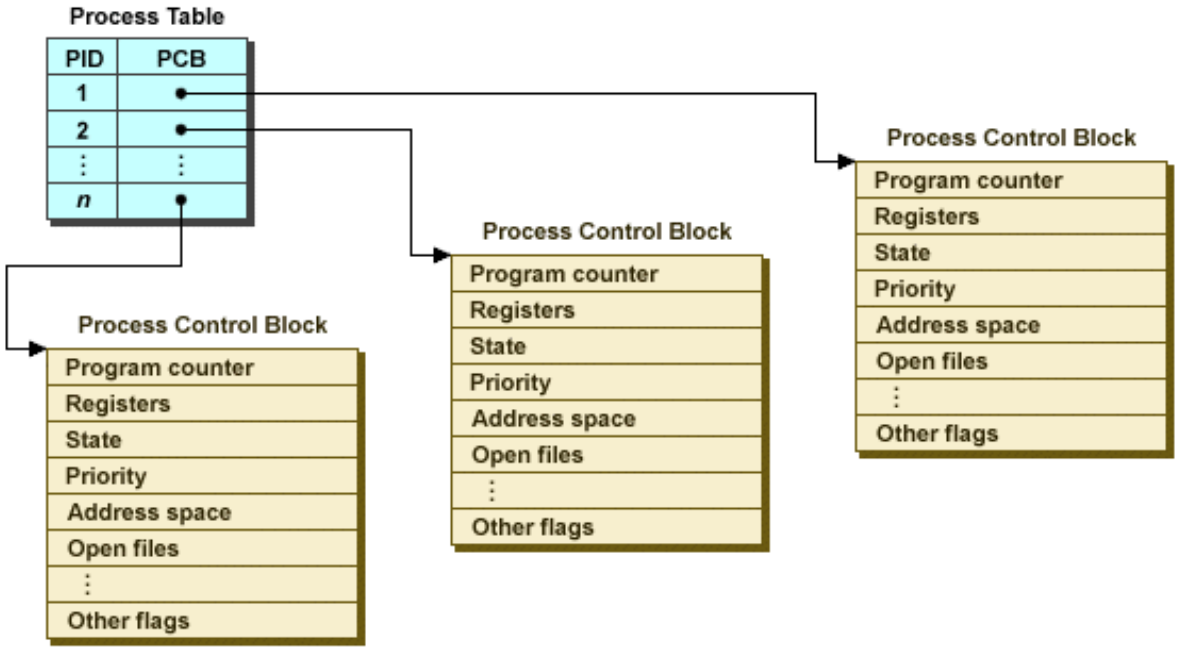
1. proces zablokowany w oczekiwaniu na zdarzenie (np. na zakończenie operacji wejścia)
2. planista wybiera inny proces do wykonania
3. planista wybiera ten proces do wykonania
4. zachodzi zdarzenie odblokowujące proces (wejście staje się dostępne)

Kolejki 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.

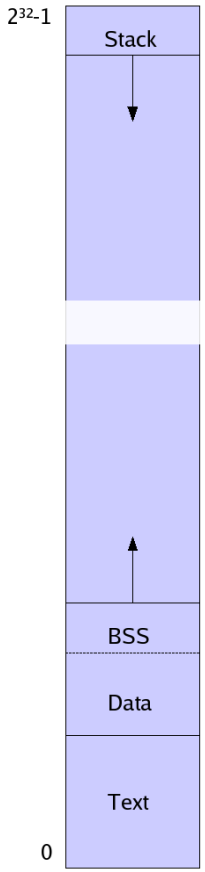
- rejestry procesora, licznik programu, słowo stanu programu, wskaźnik stosu
- informacje o wykorzystanych zasobach (wykaz otwartych plików, ograniczenia pamięci, itp.)
- pozostałe informacje wymagane do zarządzania procesem, planowania przydziału procesora, identyfikacji procesu itp.

Typowe pola bloku kontrolnego procesu

zarządzanie procesem	zarządzanie pamięcią	zarządzanie plikami
rejestry	wskaźnik do informacji segmentu tekstu	katalog główny
licznik programu	wskaźnik do informacji segmentu danych	katalog roboczy
słowo stanu programu	wskaźnik do informacji segmentu stosu	deskryptory plików
wskaźnik 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 wirtualna przestrzeń adresowa zawierająca segmenty:



- programu (*text*) – kod maszynowy programu, czyli sekwencja instrukcji wykonywanych przez procesor
- danych zainicjowanych (*data*) – zainicjowane zmienne globalne i statyczne
- danych niezainicjowanych (*bss*) – niezainicjowane zmienne globalne i statyczne
- sterty (*heap*) – pamięć alokowana dynamicznie, rośnie wraz z przydzielaniem nowych struktur
- stosu (*stack*) – ramki aktywnych procedur, w których zapisywane są zmienne lokalne oraz adresy powrotu z funkcji

Przykładowe rozmiary segmentów programu:

```
$ size /bin/ps
  text      data     bss     dec     hex filename
114103     16776  135888  266767   4120f  /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.

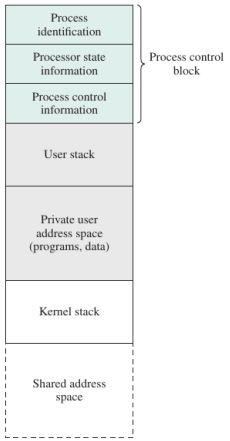
```
$ ps -elf
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root         1     0  0  80   0 - 55324 ep_pol Nov10 ?        00:00:15 /usr/lib/systemd/systemd
1 S root         2     0  0  80   0 -     0 kthrea Nov10 ?        00:00:00 [kthreadd]
1 S root         4     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 i stos użytkownika w modelu procesu

W typowym systemie komputerowym proces posiada dwa stosy:

- stos użytkownika używany do wykonywania funkcji programu w trybie użytkownika
- 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.

Wątek to jednostka wykonania działająca w ramach procesu.

Wątki tego samego procesu współdzielą jego przestrzeń adresową oraz zasoby, takie jak kod, dane globalne i otwarte pliki.

Wątki posiadają własny:

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

Wątki współdzielą w ramach procesu:

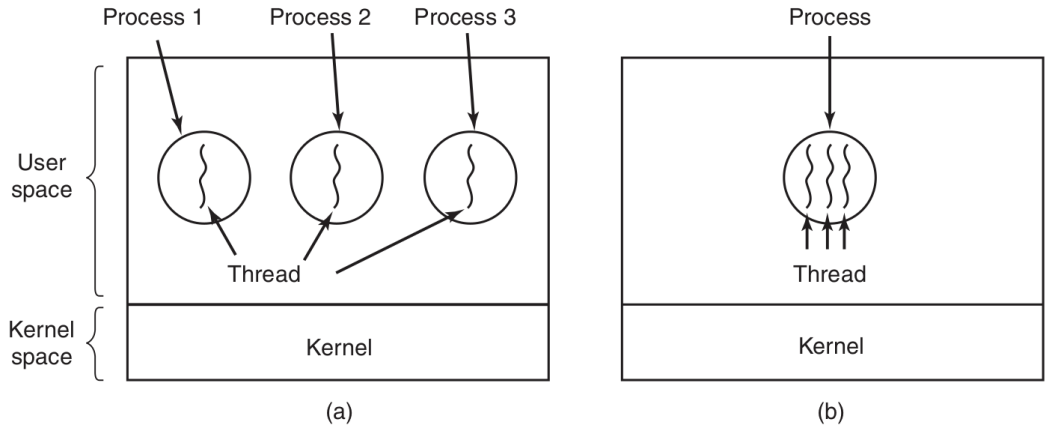
- przestrzeń adresową
- zmienne globalne
- otwarte pliki
- zasoby systemowe przypisane do procesu
- sygnały i procedury obsługi sygnałów przypisane do procesu
- uprawnienia przypisane do procesu

Procesy i wątki

- Procesy grupują zasoby, zaś wątki są jednostkami wykonawczymi pracującymi na CPU.
- Procesy są zwykle od siebie odizolowane, a ich współpraca wymaga mechanizmów komunikacji międzyprocesowej. Zarówno procesy, jak i wątki mogą współpracować lub konkurować o zasoby.
- Wątki czasem nazywane są lekkimi procesami, ponieważ posiadają pewne cechy procesu, ale są lżejsze pod względem zarządzania i przełączania kontekstu.

Wielowątkowość – możliwość występowania wielu wątków w procesie. Pozwala lepiej wykorzystywać zasoby procesora i ułatwia realizację zadań współbieżnych w ramach jednego procesu.

Procesy i wątki

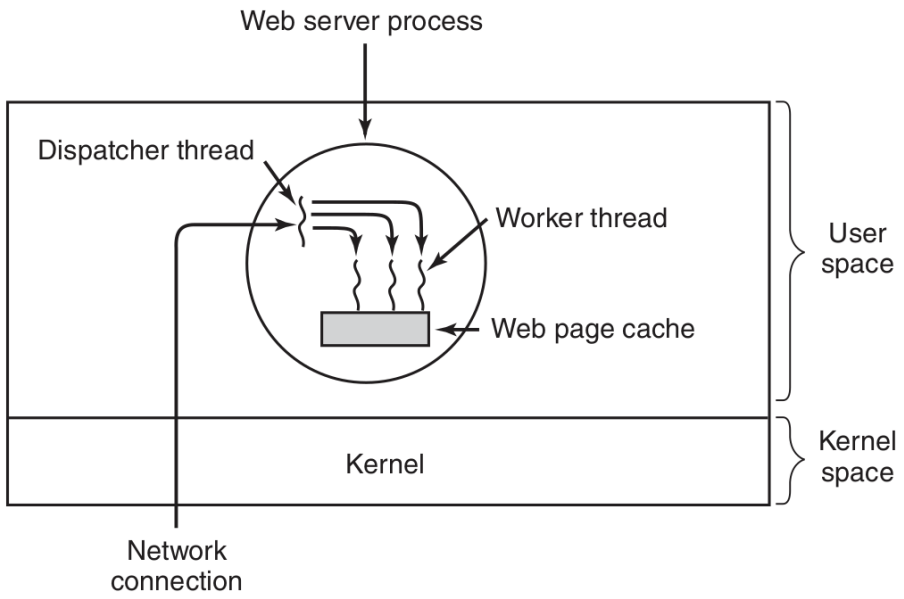


Proces	Wątek
przestrzeń adresowa	licznik programu
zmienne globalne	rejestry
otwarte pliki	stos
procesy potomne	stan
zaległe alarmy	
sygnały i procedury obsługi sygnałów	
informacje dotyczące statystyk	

Zalety wątków

- przełączanie procesora między wątkami w procesie trwa krócej niż przełączanie między procesami
- tworzenie i kończenie 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



Dyspozytor

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

(a)

Wątek roboczy

```
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)

Dyspozytor

- odbiera kolejne żądania i przekazuje je do wątków roboczych

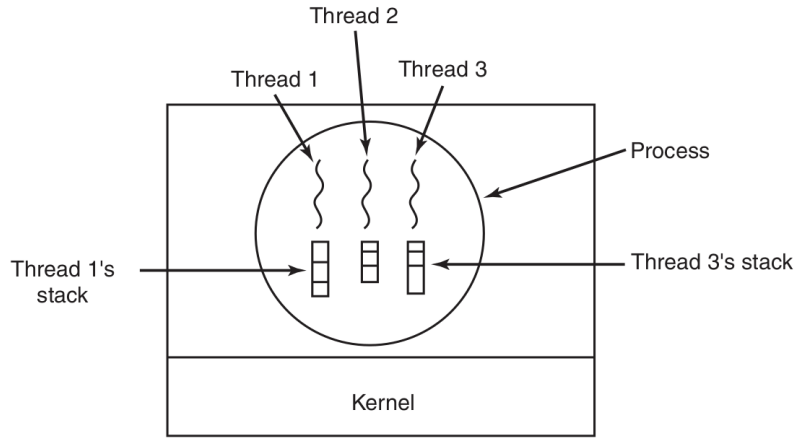
Wątek roboczy

- szuka strony w pamięci podręcznej; w razie braku odczytuje ją z dysku
- odsyła stronę klientowi

Zalety

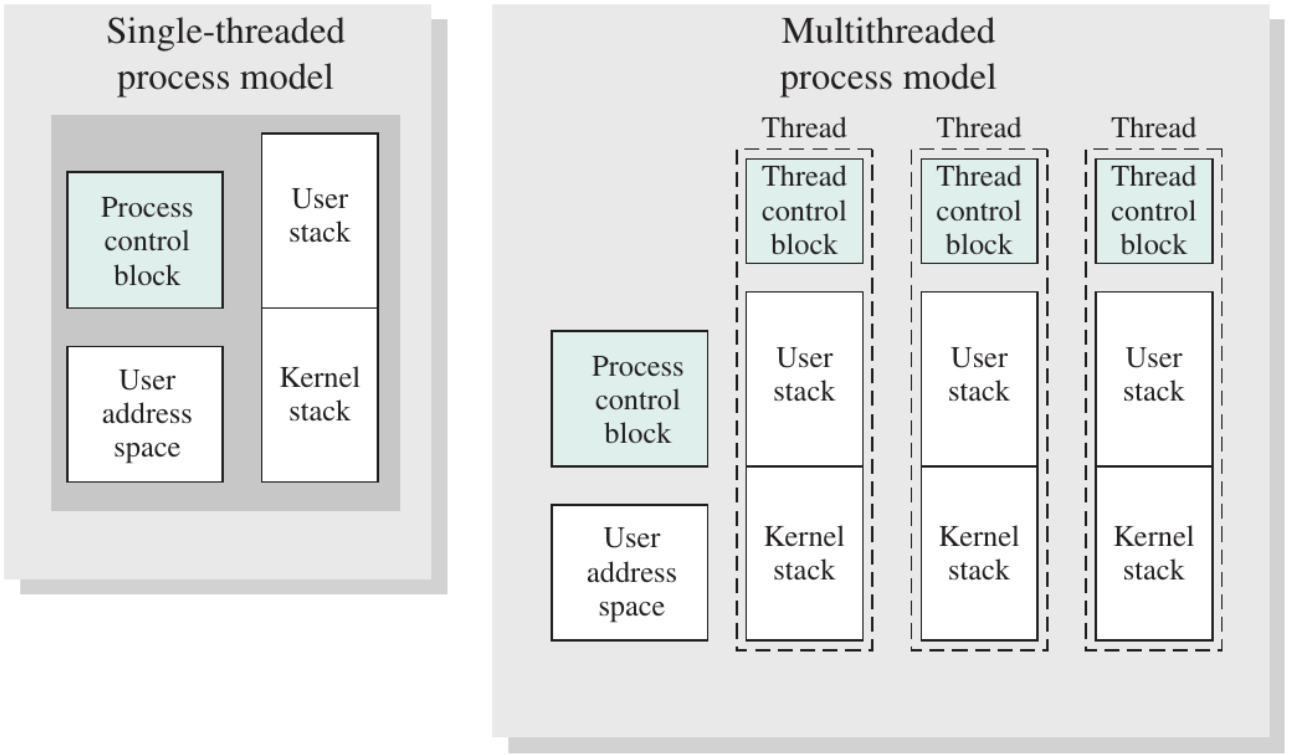
- współbieżna obsługa wielu żądań
- oczekiwanie na I/O jednego wątku nie zatrzymuje innych, jeśli są dostępne
- wątki współdzielą pamięć podręczną i są tańsze w obsłudze niż procesy

Stos procesu



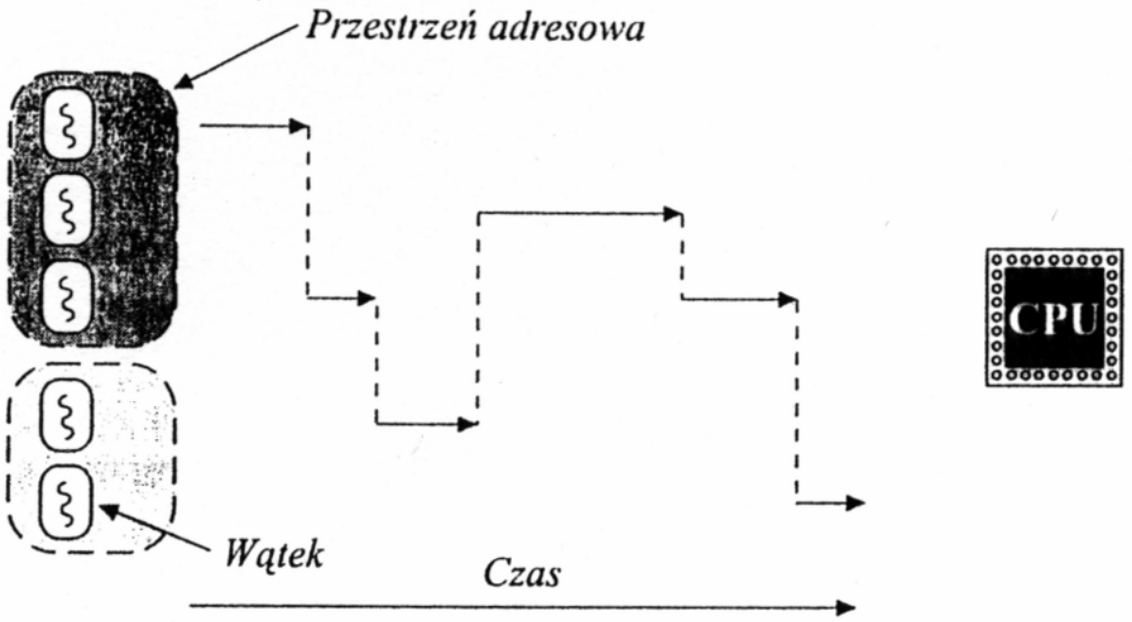
- każdy wątek posiada własny stos
- stos zawiera po jednej ramce dla każdej procedury, która została uruchomiona, a z której jeszcze nie nastąpił powrót
- ramka zawiera zmienne lokalne procedury i adres powrotu
- każdy wątek może uruchamiać różne procedury i posiadać odmienną historię wywołań

Model procesu jednowątkowego i wielowątkowego



Każdy wątek obsługiwany przez jądro ma własny stos jądra. W procesie wielowątkowym każdy wątek ma także własny stos użytkownika.

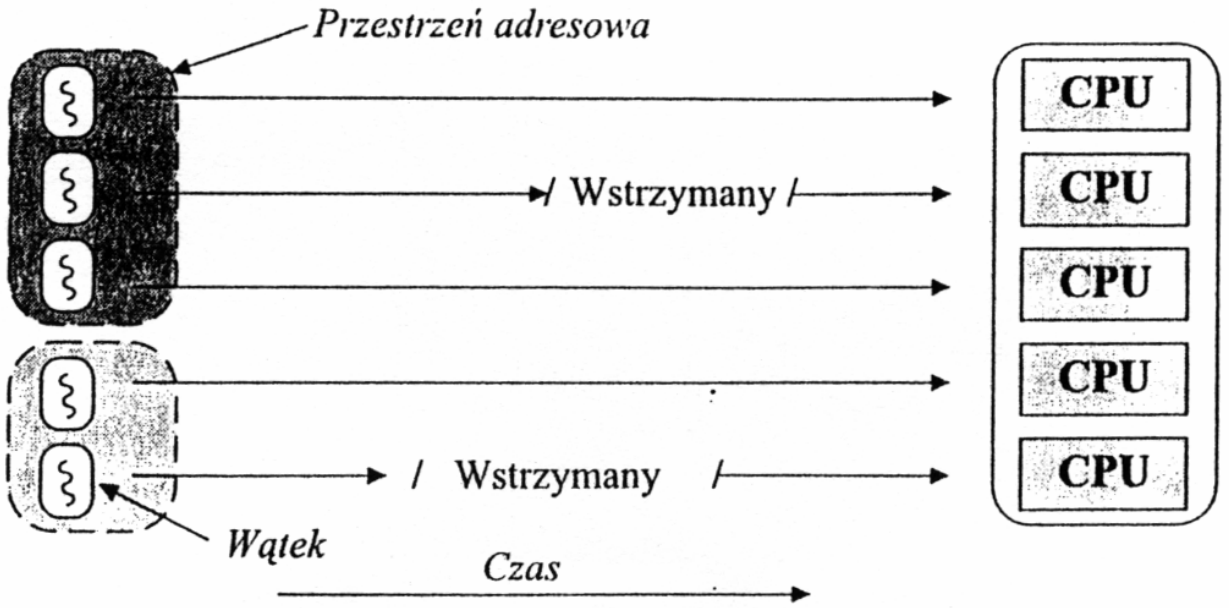
Procesy wielowątkowe w systemie z jednym CPU



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

Niektóre procesory obsługują sprzętową wielowątkowość (SMT), np. Intel Hyper-Threading, która pozwala jednemu rdzeniowi wykonywać współbieżnie instrukcje z dwóch wątków.

Procesy wielowątkowe w wieloprocesorze

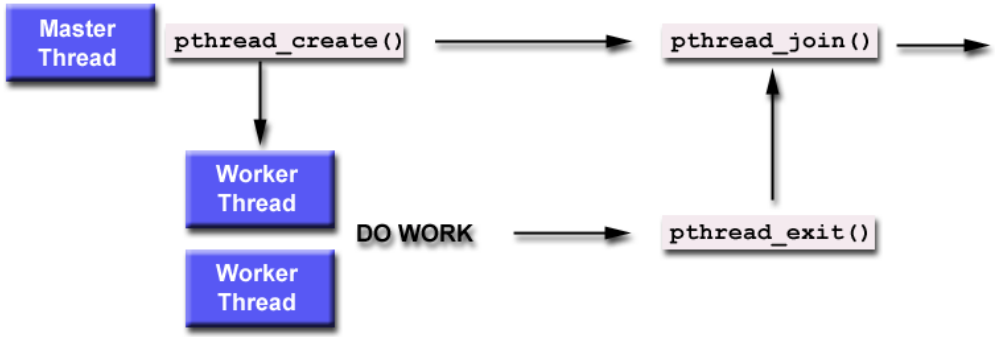


Rysunek 3-3. Procesy wielowątkowe w wieloprocesorze

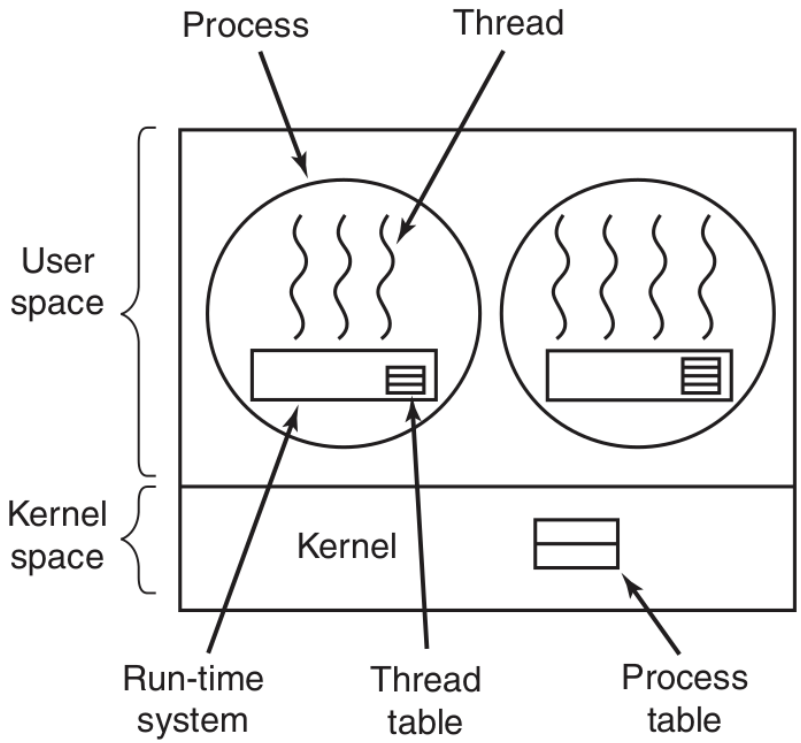
Nowoczesne procesory mają wiele rdzeni, co pozwala na jednoczesne wykonywanie wielu wątków w sposób rzeczywiście równoległy

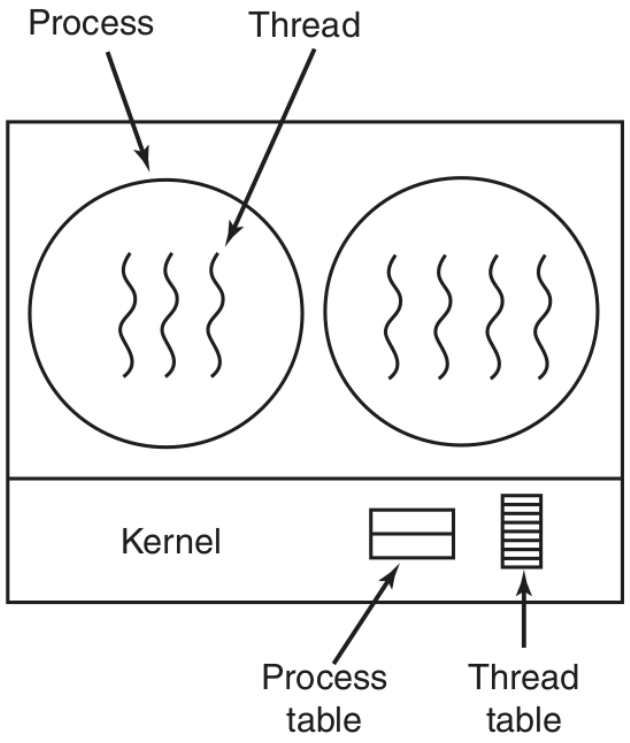
Zarządzanie wątkami

- 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_yield`), tak aby dać innym wątkom szansę na wykonanie



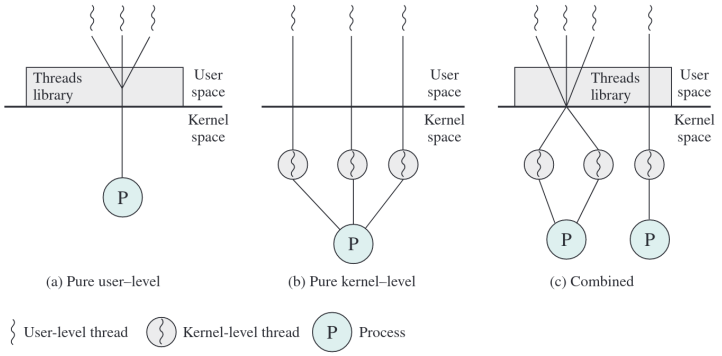
Wątki poziomu użytkownika i poziomu jądra





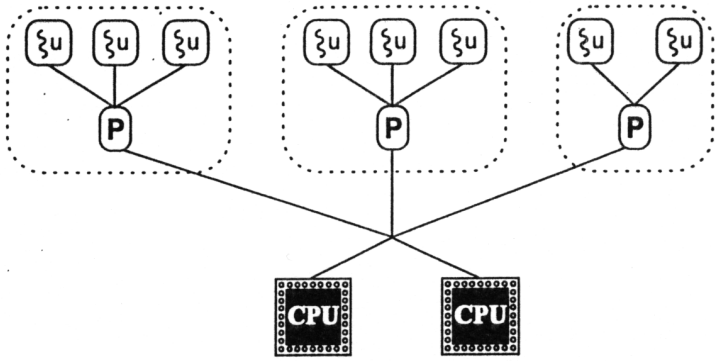
Rodzaje wątków

- **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*) – model, w którym biblioteka zarządza wieloma wątkami użytkownika, mapując je na pulę wątków poziomu jądra (model M:N)

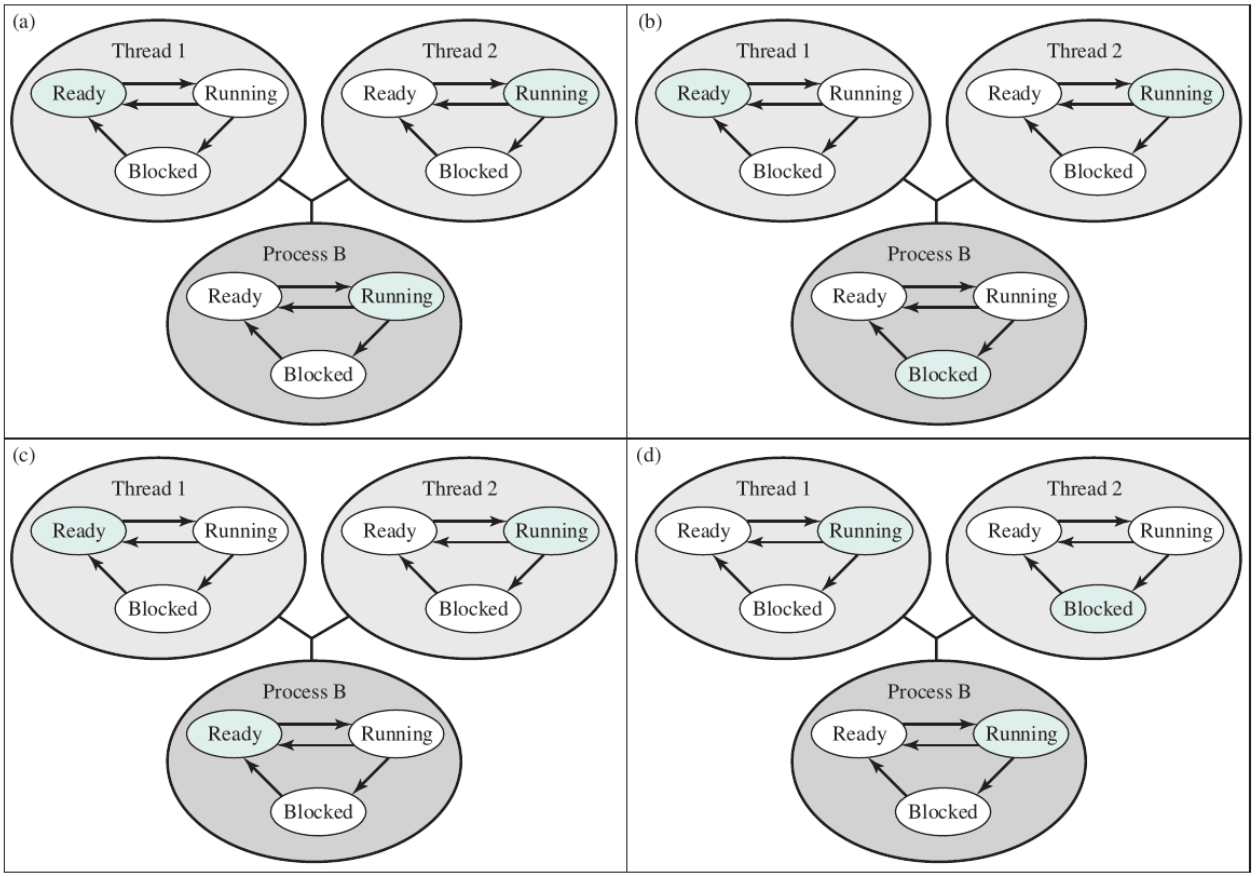


Wątki poziomu użytkownika

- wątki poziomu użytkownika (*user-level threads*, ULT) - 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 (biblioteka nie działa i nie aktualizuje stanu wątków)

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 planowanie procesu realizowane jest przez jądro systemu operacyjnego.

Zalety wątków użytkownika

- **Błyskawiczne przełączanie kontekstu:** przełączanie wątków nie wymaga przejścia do trybu jądra i odbywa się w przestrzeni użytkownika
- **Wydajność:** wątki użytkownika są wydajne, nie zużywają zasobów jądra, tworzenie wątku sprowadza się do alokacji małej struktury danych w przestrzeni użytkownika
- **Własny algorytm zarządzania:** każdy proces może mieć indywidualny schemat zarządzania wątkami, dostosowany do potrzeb aplikacji
- **Niezależność od systemu:** mogą być implementowane w systemach operacyjnych, które nie wspierają wielowątkowości na poziomie jądra

Problemy wątków użytkownika

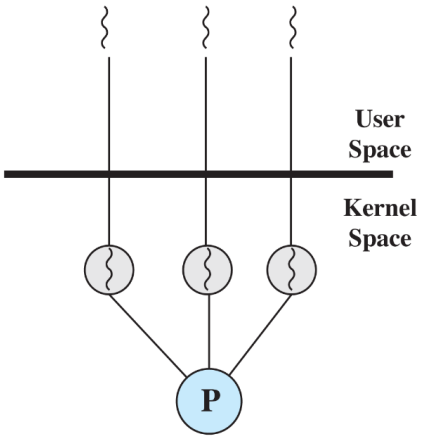
- **Ograniczenie zrównoleglenia:** biblioteka szereguje wątki, jądro procesy; tylko jeden wątek procesu może być wykonywany w danym momencie nawet na maszynie wieloprocesorowej
- **Blokowanie procesu:** operacje blokujące (wywołania systemowe, błędy strony) powodują zablokowanie całego procesu, więc i wszystkich wątków poziomu użytkownika w procesie
- **Kosztowne wywołania systemowe:** wywołanie systemowe powoduje synchroniczne przejście do trybu jądra; przejście i obsługa w jądrze wiążą się z narzutem
- **Monopolizacja CPU:** uruchomiony wątek zajmuje CPU, nie dając możliwości działania innym wątkom, dopóki nie zrzeknie się procesora (*yield*)

Osłanianie (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 (KLT, *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`



Zalety wątków poziomu jądra

- **Prawidłowe blokowanie:** w momencie zablokowania wątku system może wznowić wykonywanie innego, gotowego wątku (także z innego procesu)
- **Wykorzystanie wieloprocesowości:** system operacyjny może jednocześnie planować wiele wątków z jednego procesu na wielu procesorach
- **Precyzyjna kontrola:** 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 wątków poziomu jądra

- **Kosztowne przełączanie kontekstu:** przełączanie wątków przez jądro jest wolniejsze niż w przypadku wątków użytkownika, ponieważ wymaga przejścia z trybu użytkownika do trybu jądra i z powrotem
- **Większe zużycie zasobów:** tworzenie oraz niszczenie wątków odbywa się przez odwołania do jądra, jest to kosztowniejsze niż w przypadku wątków użytkownika

Procesy i wątki: porównanie

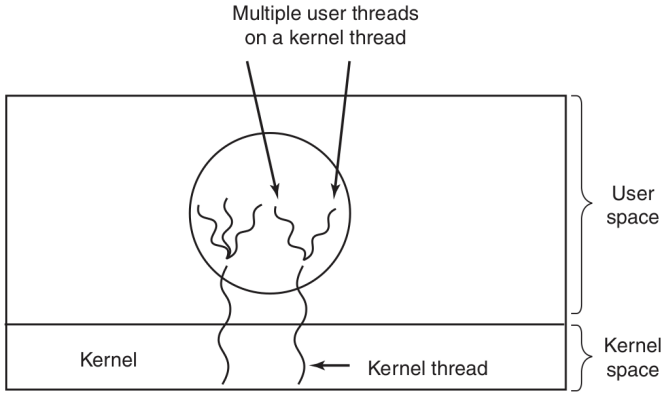
- **Null fork:** czas mierzony od utworzenia procesu lub wątku, przez jego szeregowanie i wykonanie, aż do jego zakończenia (wątku), który wykonuje pustą procedurę (miara obciążenia spowodowanego przez tworzenie procesu (wątku))
- **Signal-Wait:** czas potrzebny, aby proces (wątek) przesłał sygnał oczekiwania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

Czas oczekiwania (μs) na wykonanie operacji przez wątki poziomu użytkownika (ULT), wątki poziomu jądra (KLT) i procesy (P).[9]

	ULT	KLT	P
Null fork	34	948	11300
Signal-wait	37	441	1840

[9] VAX, system uniksopodobny

Hybrydowe implementacje wątków



- podejście mieszane, wykorzystuje wątki poziomu jądra i użytkownika (model M:N)
- aplikacja tworzy wiele wątków poziomu użytkownika (M) i są one mapowane na mniejszą lub równą liczbę wątków poziomu jądra (N)
- jądro systemu zarządza wyłącznie wątkami jądra
- zarządzanie wątkami użytkownika odbywa się w ramach procesu, analogicznie do tradycyjnego modelu wątków poziomu użytkownika

Zalety hybrydowych wątków

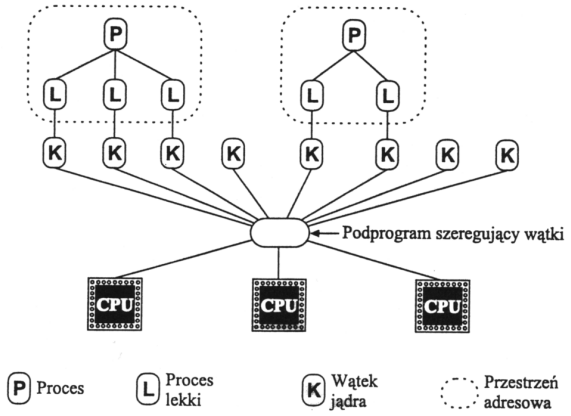
Wątki hybrydowe łączą zalety wątków poziomu użytkownika i wątków poziomu jądra.

- 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

Wadą hybrydowych wątków jest większa złożoność implementacji i zarządzania wątkami, zarówno na poziomie użytkownika, jak i jądra.

Procesy lekkie, LWP

- Proces lekki (*light-weight process*, LWP) jest wątkiem w przestrzeni użytkownika wspieranym przez jądro
- z procesem może być związanych wiele procesów lekkich, z których każdy jest wspierany przez osobny wątek jądra (relacja 1:1)
- procesy lekkie są niezależnie szeregowane, współdzielą przestrzeń adresową, mogą wywoływać blokujące funkcje systemowe bez blokowania procesu, mogą się wykonywać na różnych procesorach



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

Pojęcie procesów lekkich bywa używane w kilku znaczeniach:

- oryginalnie w Solarisie (Unix) oznacza mapowanie wątku użytkownika na wątek jądra
- w Linuksie proces lekki to jednostka wykonawcza (zadanie) planowana przez jądro. Może być procesem lub wątkiem (oba są reprezentowane przez strukturę `task_struct`), dlatego termin ten bywa używany jako synonim wątku poziomu jądra

Lekkie wątki użytkownika: w systemie Windows `fibers` (włókna) są wątkami wykonywanymi całkowicie w trybie użytkownika. Nie są zarządzane bezpośrednio przez jądro systemu, lecz działają w kontekście wątku jądra.

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.);
- zwiększa plikom związanym z tym procesem liczniki w tablicy plików i i-węzłów
- przekazuje identyfikator potomka procesowi macierzystemu i wartość zero procesowi potomnemu

Mechanizm **kopiowania przy zapisie** (*copy-on-write*, COW) odkłada w czasie tworzenie rzeczywistej kopii procesu, aż do momentu gdy nastąpi zmiana w pamięci procesu macierzystego lub potomnego. Jest realizowany niezależnie przez podsystem zarządzania pamięcią i system stronicowania (MMU, *Memory Management Unit*)

Tworzenie procesu w UNIX/Linux

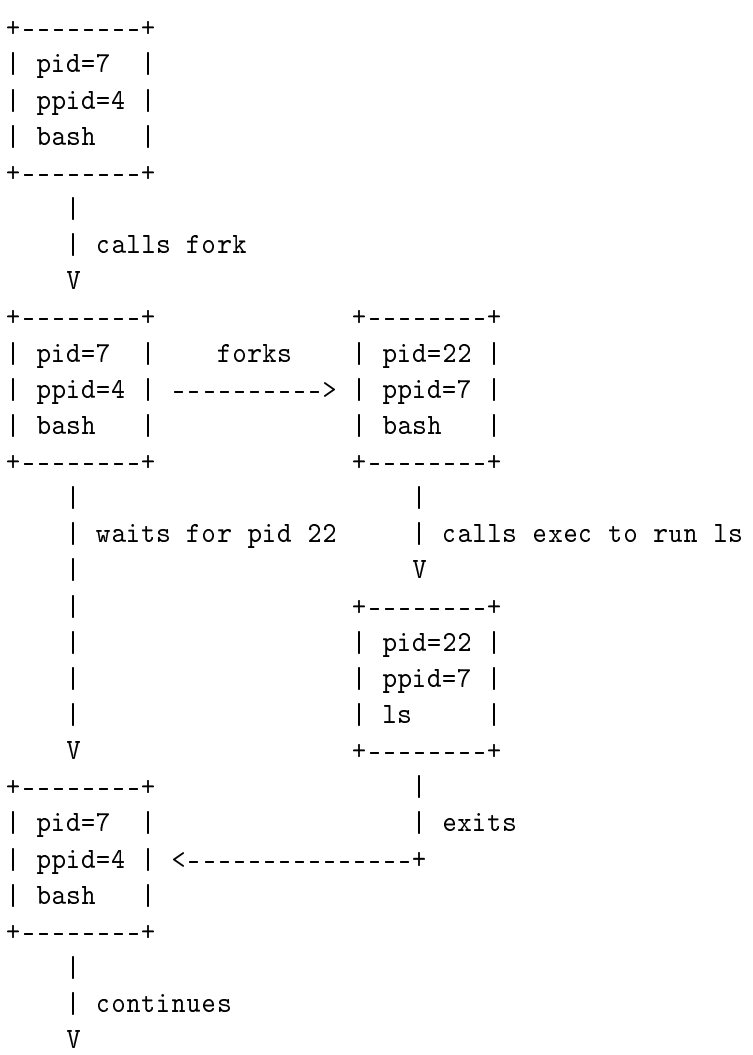
Wywołanie systemowe `exec()` umieszcza w obszarze pamięci bieżącego procesu kopię pliku wykonywalnego

- 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

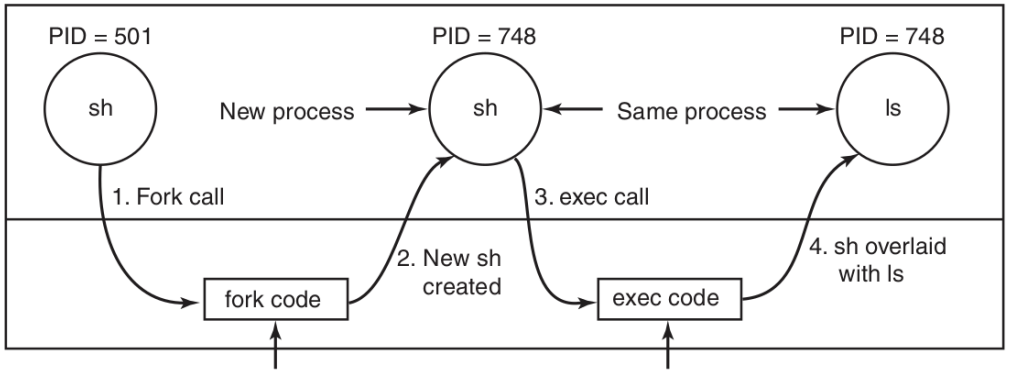
Procesy: tworzenie

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);
   } else {
      /* potomek wykonuje pracę */
      execve(command, params, 0);
   }
}
```



Uruchomienie procesu w powłoce



`fork()` – tworzenie potomka

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

Obowiązuje kopiowanie przy zapisie (*Copy on Write*, CoW)

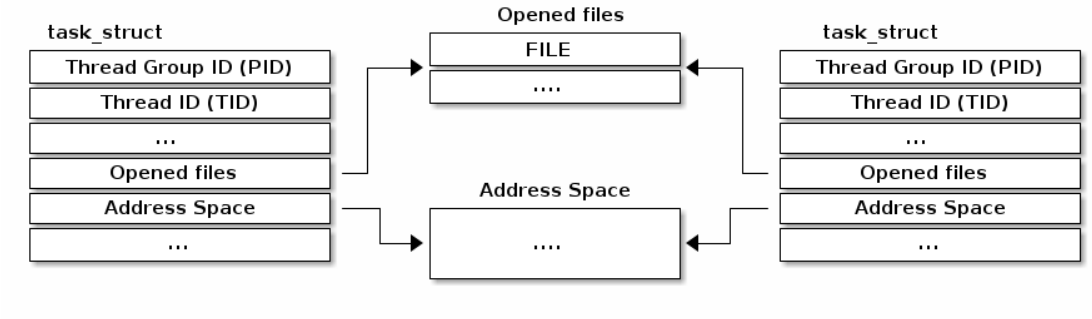
`exec()` – zastąpienie obrazu procesu nowym programem

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

Procesy: zarządzanie w systemie GNU/Linux

W jądrze Linuksa procesy i wątki są reprezentowane jako **zadania** (*task*) za pomocą struktury 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, polityka przydziału procesora, parametry tej polityki (priorytet dynamiczny)
- informacja o programie wykonywalnym (*executable info*): identyfikuje plik, który jest obrazem wykonywanego zadania
- informacja o sygnałach (*signal info*): 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*, limity czasu wykonania zadania, wykorzystania pamięci, liczby 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*)
- 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	przestrzeń adresowa	współdzielona z rodzicem	osobna kopia przestrzeni adresowej
CLONE_FS	informacje o systemie plików katalog roboczy, główny, umask	wspólny katalog roboczy	niezależny katalog roboczy
CLONE_FILES	tablica otwartych plików	współdzielenie deskryptorów	kopie deskryptorów
CLONE_SIGHAND	tablica obsługi sygnałów	współdzieli	kopiuje tablicę
CLONE_PARENT	informacje o rodzicu	współdzieli rodzica	staje się potomkiem
CLONE_THREAD	grupa wątków	potomek dołącza do grupy wątków rodzica	potomek tworzy nową grupę

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

Przykład wywołania tworzącego wątek:

```
int tid = clone(worker, stack_top,
    CLONE_VM | CLONE_SIGHAND | CLONE_THREAD, arg);
```

gdzie `stack_top` wskazuje na szczyt wcześniej zaalokowanego stosu nowego wątku.

Przykład wywołania tworzącego proces:

```
int pid = clone(child_fn, stack_top, SIGCHLD, arg);
```

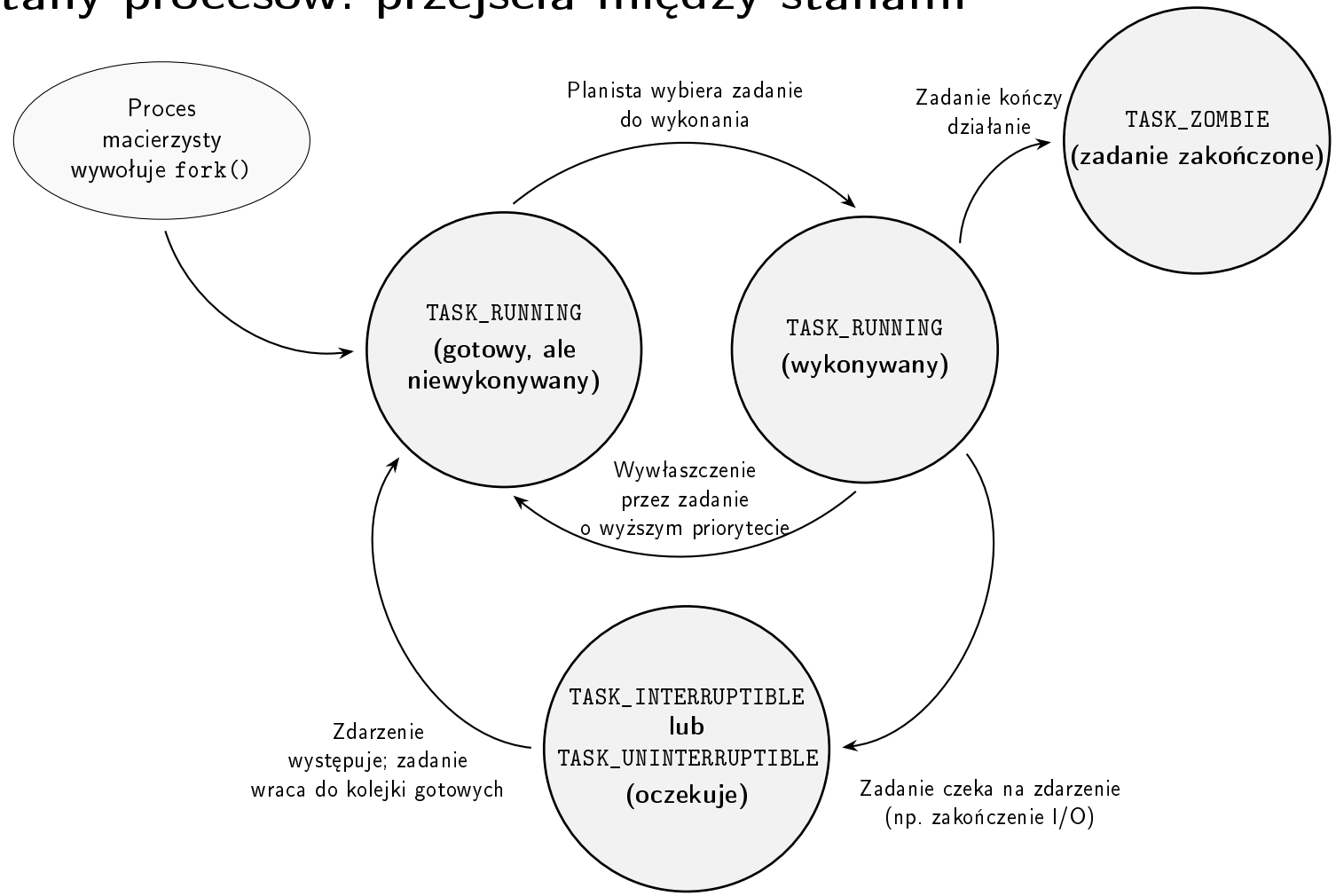
gdzie nie ustawiono flag `CLONE_*`, zaś `SIGCHLD` zajmuje najniższy bajt argumentu `flags` i określa sygnał wysyłany do procesu rodzica po zakończeniu procesu potomnego.

Stany procesów według jądra Linuksa

- `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 nie wybudzają zadania lecz mogą pozostać oczekujące do czasu jego wybudzenia. [10]
- `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`)

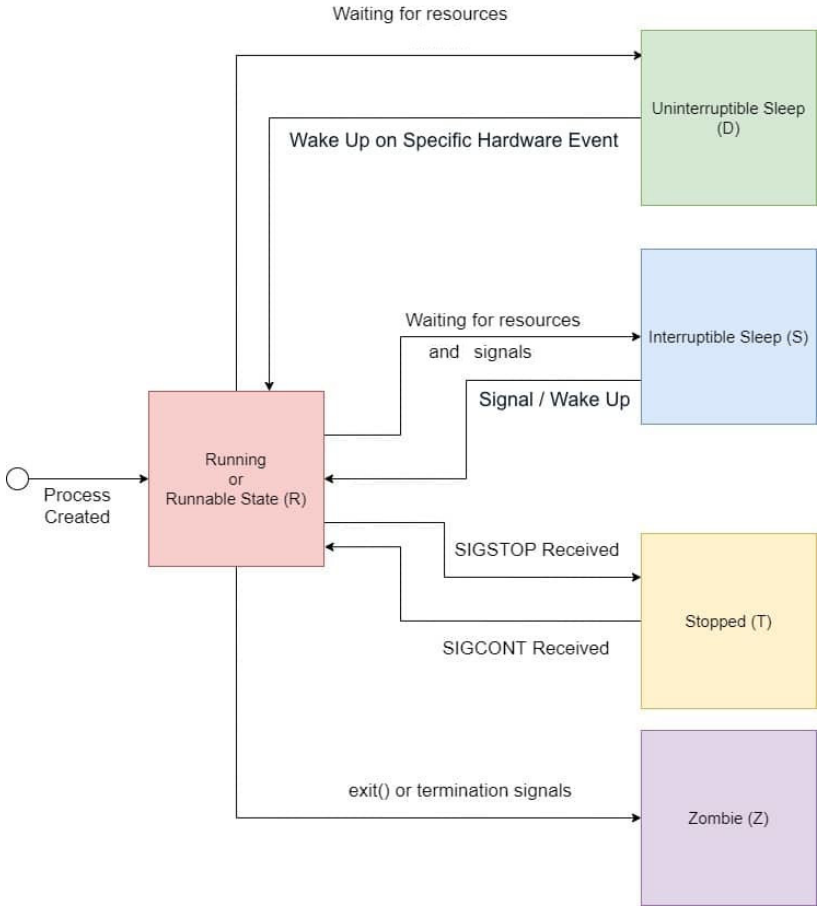
[10] TASK_KILLABLE, alternatywny stan `TASK_UNINTERRUPTIBLE` reagujący na krytyczne sygnały wprowadzony w łatce do jądra 2.6.25

Stany procesów: przejścia między stanami



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 gotowy do wykonania

S (*interruptible sleep*) śpiący, czeka na zdarzenie, można wybudzić sygnałem

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

T zatrzymany przez sygnał sterujący (np. `SIGSTOP`)

t zatrzymany przez program śledzący (debugger)

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

X (*dead*) – *should never be seen*

< proces o wysokim priorytecie (nie jest „uprzejmy")

N proces o niskim priorytecie

L (pages locked into memory) – proces ze stronami zablokowanymi 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) jednostki wykonywalnej (alias tid)

`rsz` - (*resident set size*), przybliżony rozmiar pamięci fizycznej procesu, która obecnie znajduje się w RAM-ie, podany w KiB.

`vsz` - rozmiar przestrzeni pamięci wirtualnej procesu w KiB, także mapowanie pamięci, których strony nie muszą obecnie znajdować się w RAM-ie

Procesy systemowe w Linuksie – 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`), podlegają planowaniu, mogą 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 Linuksa

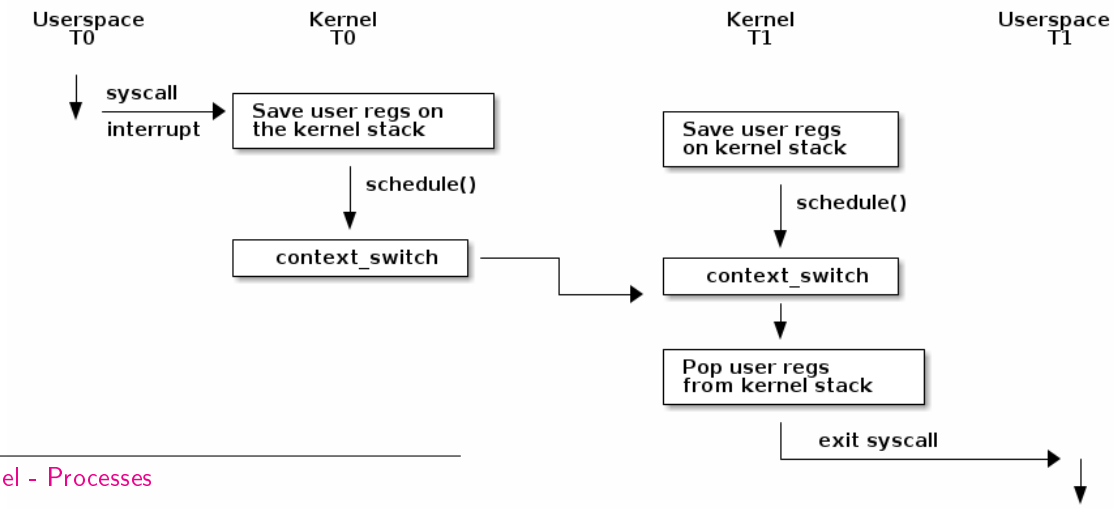
Informacje o procesie PID 2 ([kthreadd]) i jego wątkach potomnych:

```
$ 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
...
```

- wątki jądra mają numer PID (identyfikator procesu) tak jak zwykłe procesy użytkownika
- VSZ i RSS wynoszą 0, co oznacza, że wątki jądra nie zajmują pamięci użytkownika.
- wątki jądra nie mają przypisanego terminala (TTY)
- wątek jądra [kthreadd] ma PPID=0 i nie ma procesu nadrzędnego
- nawiasy kwadratowe, np. [kthreadd], są typowym sposobem prezentowania wątków jądra

Przełączenie kontekstu zadania z trybu użytkownika

- przed zmianą kontekstu musi dojść do zmiany trybu pracy z trybu użytkownika na tryb jądra w wyniku wywołania systemowego lub przerwania (wejście T0 do jądra)
- rejestry użytkownika zostają zapisane na stosie jądra bieżącego zadania
- kod jądra (planista) wybiera zadanie do dalszego wykonania (`schedule()`)
- podczas przełączania kontekstu zapisywany jest kontekst jądra bieżącego zadania i odtwarzany zapisany stan innego zadania (`context_switch`), w tym jego stos
- wybrane zadanie wznawia wykonanie w jądrze, odtwarza swoje rejestry użytkownika ze stosu jądra i wraca do przestrzeni użytkownika



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:

- nowe zadanie o wyższym priorytecie staje się gotowe do wykonania
- zadanie wyczerpie swój kwant czasu
- po wywołaniu systemowym lub obsłudze przerwania przełączenie może nastąpić przed powrotem do przestrzeni użytkownika
- przerwanie może nadejść asynchronicznie podczas wykonywania kodu użytkownika; samo jego nadejście nie oznacza jednak, że zadanie zostanie wywłaszczone

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

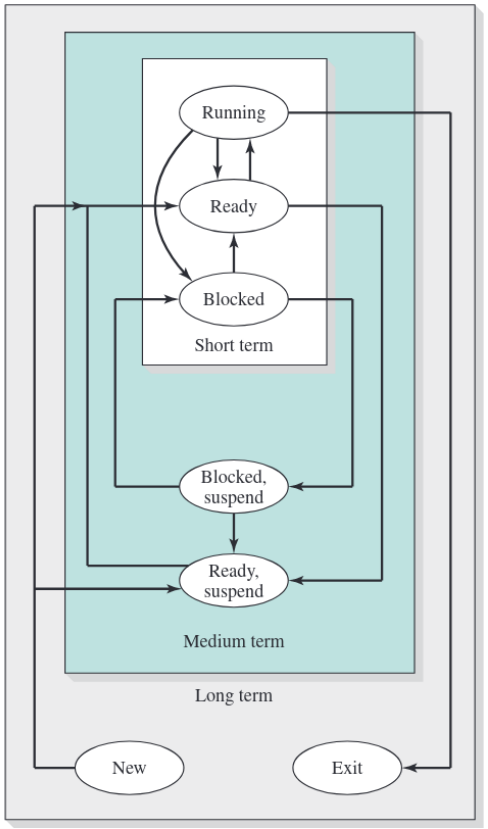
Wywłaszczenie jądra

- jądra współczesnych systemów są **wielobieżne** (*reentrant*) i **wywłaszczalne** (*preemptive*)
- **Wielobieżność** (*reentrancy*) pozwala na równoległe wykonywanie kodu jądra na wielu procesorach
- **Wywłaszczalność** pozwala wstrzymać wykonywanie kodu jądra i uruchomić inne gotowe zadanie. Na jednym procesorze nie zapewnia równoległości, ale może skrócić czas reakcji na pilniejsze zadanie
- przełączenie następuje w bezpiecznym punkcie: w sekcjach krytycznych preempcja jest wyłączona lub odraczana.
- przerwanie sprzętowe może wywołać decyzję planisty o wywłaszczeniu aktualnie wykonywanego zadania, nawet wykonywanego w trybie jądra, skracając czas reakcji na zdarzenie urządzenia
- w Linuksie zakres preempcji zależy od konfiguracji jądra: możliwy jest brak wymuszonej preempcji, preempcja dobrowolna lub pełna

Pytania sprawdzające

- Wymień trzy podstawowe stany procesu w systemie operacyjnym i wskaż, w jakich sytuacjach może nastąpić przejście między nimi
- Co to jest wywłaszczenie procesu i kiedy może do niego dojść?
- Jaka jest różnica między wątkami poziomu użytkownika i wątkami poziomu jądra?

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 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
- **planowanie krótkoterminowe** (*short-term scheduling*) – decyduje o tym, który z gotowych do wykonania procesów otrzyma czas procesora (CPU) na wykonanie swoich instrukcji

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 *gotowości*, o kolejności przydziału CPU decyduje planista, wykonując **algorytm szeregowania** realizujący politykę szeregowania.

Polityka szeregowania określa ogólne zasady i cele, którymi 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 wejścia-wyjścia

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 do coraz większego ograniczenia przez operacje wejścia-wyjścia.

Cele algorytmów szeregowania w różnych środowiskach

Wszystkie systemy

- **sprawiedliwość** – zapewnienie procesom uczciwego dostępu do CPU
- **równowaga** – dążenie do zajęcia wszystkich części systemu

Systemy interaktywne

- **krótki czas odpowiedzi** (*response time*) – maksymalizacja responsywności procesów interaktywnych
- **proporcjonalność** – spełnianie oczekiwań użytkowników dotyczących czasu wykonania

Systemy wsadowe

- **przepustowość** – maksymalizacja liczby wykonanych zadań na godzinę
- **czas cyklu przetwarzania** – minimalizacja czasu między rozpoczęciem a zakończeniem procesu
- **wykorzystanie CPU** – maksymalizacja jego zajętości

Systemy czasu rzeczywistego

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

Planowanie przydziału procesora - mechanizmy

- **Algorytmy niewywłaszczające** (*Non-preemptive*)
  - **FCFS** (*First-Come, First-Served*) pierwszy nadszedł, pierwszy obsłużony
  - **SJF** (*Shortest Job First*) najkrótsze zadanie najpierw
- **Algorytmy wywłaszczające** (*Preemptive*)
  - **RR** (*Round-Robin*) rotacyjne, karuzelowe, cykliczne
  - **SRTF** (*Shortest Remaining Time First*) najpierw najkrótszy pozostały czas
  - **Priorytetowy**: priorytety statyczne i dynamiczne
- **Algorytmy Zaawansowane** stosowane we współczesnych systemach
  - **Kolejki wielopoziomowe** - zestaw kolejek o różnych priorytetach, każdy poziom może mieć inną politykę planowania
  - **Kolejki Wielopoziomowe ze Sprzężeniem Zwrotnym** (MLFQ, *Multi-Level Feedback Queue*) – procesy mogą przemieszczać się między kolejkami w zależności od swojego zachowania
    Przykład: planista Linux (O(1), CFS), planista Windows NT, 10, 11

Szeregowanie bez wywłaszczenia

Algorytm szeregowania **bez wywłaszczania** (*non-preemptive*) wybiera proces do uruchomienia, a następnie pozwala mu działać do czasu zablokowania (np. podczas operacji wejścia-wyjścia) 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, niewymagających interakcji z użytkownikiem
- może prowadzić do mniejszej responsywności systemu dla procesów interaktywnych
- ryzyko głodzenia procesów o niższym priorytecie

Szeregowanie z wywłaszczeniem

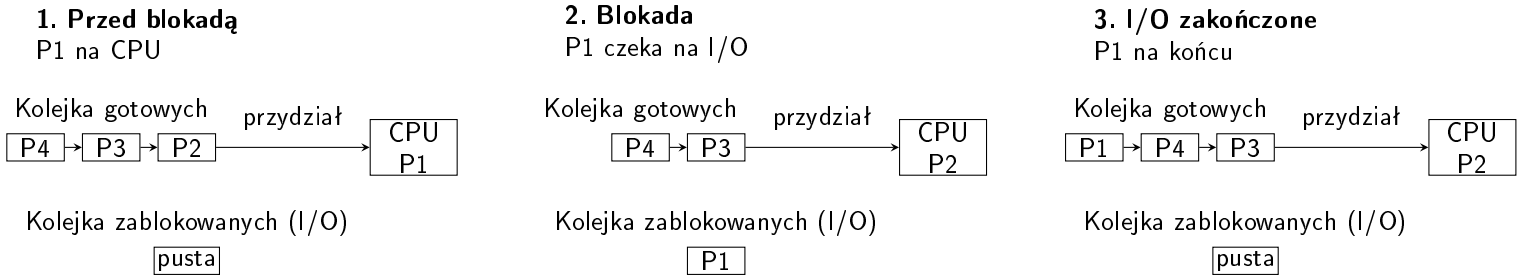
Algorytm szeregowania **z wywłaszczaniem** (*preemptive*) przydziela procesowi CPU na ustalony czas (kwant).

- proces może zostać wywłaszczony, zwykle wskutek przerwania zegarowego, co pozwala planiście ponownie zdecydować o przydziale CPU
- sprawiedliwiej dzieli czas procesora w systemach wielozadaniowych
- krótki kwant sprzyja procesom interaktywnym i poprawia responsywność systemu
- częste przełączenia kontekstu zwiększają narzut, a zbyt krótki kwant może obniżyć wydajność systemu
- dłuższy kwant ogranicza narzut związany z przełączaniem kontekstu i sprzyja procesom obliczeniowym kosztem responsywności
- wymaga dodatkowych mechanizmów synchronizacji, więc jest trudniejszy w implementacji

Pierwszy nadszedł, pierwszy obsłużony (FCFS)

First-Come, First-Served

- pojedyncza kolejka gotowych procesów
- procesy są wykonywane w kolejności, w jakiej trafiają do kolejki gotowych, bez względu na czas trwania czy priorytet (kolejka FIFO)
- gdy proces się blokuje, zwalnia CPU, a planista wybiera kolejny proces z kolejki gotowych
- odblokowany proces trafia na koniec kolejki gotowych



Źródło: John Bell, *CPU Scheduling*, sekcja 5.3.1, University of Illinois Chicago, dostęp 9 listopada 2026 r.

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 w kolejce gotowych są uporządkowane według 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
- sprawdza się w systemach, w których czas wykonania procesów jest znany lub można go dokładnie oszacować (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 lub SRTF, *Shortest Remaining Time First*)

- 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 wykorzystująca historię uruchomień i zachowanie procesów do szacowania czasu wykonania
- przykładowa estymacja czasu wykonania procesu (*aging*) oparta na średniej wykładniczej:

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

  gdzie $0 \leqslant a \leqslant 1$.
- czas T_n oznacza rzeczywisty czas wykonania procesu podczas n-tego uruchomienia

Algorytm rotacyjny (*Round-Robin*)

- procesy gotowe do wykonania są zorganizowane w kolejkę, a każdy proces otrzymuje kwant czasu
- algorytm działa podobnie do FCFS, ale z wywłaszczaniem; zawsze wybierany jest pierwszy proces z kolejki
- gdy kwant czasu dobiegnie końca, proces zostaje wywłaszczony i umieszczony na końcu kolejki
- jeśli proces zakończy się przed upływem kwantu czasu lub zostanie zablokowany, CPU zostaje przydzielone kolejnemu procesowi z kolejki
- kwant czasu wynosi typowo około 20–50 ms; 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 zależy 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
- priorytety dynamiczne 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 można pogrupować według priorytetów, tworząc **klasy priorytetów**. Planowanie między klasami odbywa się zgodnie z priorytetami, a w ramach danej klasy – według wybranego algorytmu szeregowania (np. cyklicznego)

Klasy priorytetów

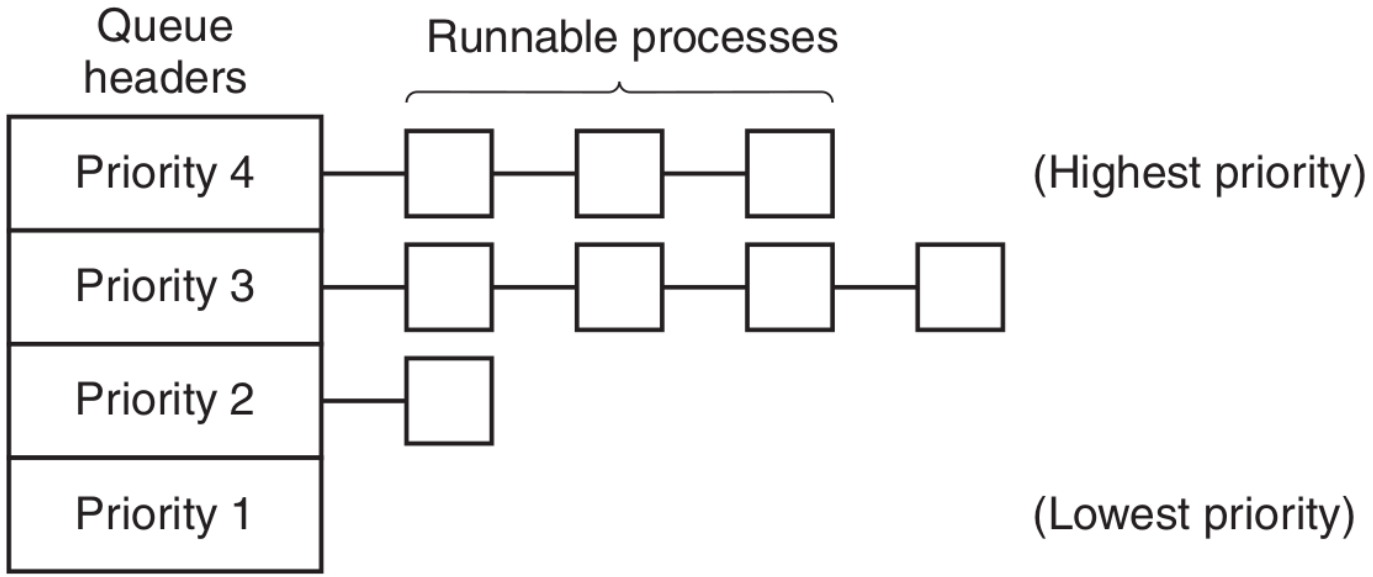


Figure 2-43. A scheduling algorithm with four priority classes.

Algorytm szeregowania w Linuksie

- szeregowane są 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), od wersji 2.6.23 (10.2007)
- **algorytm EEVDF** (*Earliest Eligible Virtual Deadline First*) od jądra 6.6 (10.2023)

Nikita Ishkov, *A complete guide to Linux*

Nazwa nie odnosi się do systemów czasu rzeczywistego i nie oznacza spełniania rygorystycznych ograniczeń czasowych

Priorytety czasu rzeczywistego

- priorytety zadań czasu rzeczywistego mieszczą się w zakresie 0–99 i nie zmieniają się w trakcie wykonywania
- zadania są wykonywane ściśle według malejących priorytetów: najpierw zadania czasu rzeczywistego, a następnie pozostałe

Polityki szeregowania czasu rzeczywistego:

- `SCHED_FIFO` (*First-In, First-Out*) – szeregowanie bez wywłaszczenia, analogiczne do FCFS. Stosowana w przypadku 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.

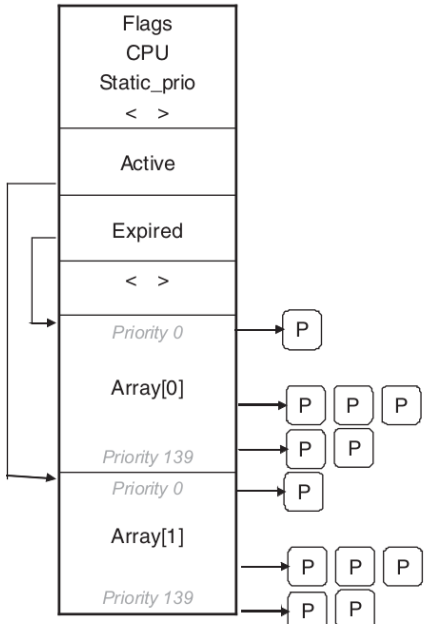
RHEL/CentOS/Fedora ograniczają maksymalne zużycie CPU dla wątków czasu rzeczywistego do 95%; zob. Real Time Scheduler Throttling

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 na podstawie poziomu uprzejmości (*nice*)
- poziomy uprzejmości od -20 do $+19$ są mapowane na wartości od 100 do 139; domyślna wartość uprzejmości 0 odpowiada priorytetowi 120
- wartość priorytetu procesu zmienia się dynamicznie w granicach ± 5 w zależności od interaktywności zadania wyznaczanego jako stosunek czasu, jaki zadanie spędza w zawieszeniu, do czasu aktywności
- wielkość kwantu czasu procesu zależy od priorytetu dynamicznego; procesy o wyższym priorytecie otrzymują dłuższy kwant czasu
- domyślnie proces otrzymuje kwant 100 ms. Może on zostać zwiększony do 800 ms (100) lub zmniejszony do 5 ms (139).
  Nowy proces dziedziczy kwant po rodzicu.

Planista O(1)

- struktura zarządzania aktywnymi zadaniami (*runqueue*) zawiera dwie tablice o rozmiarze 140: tablicę zadań aktywnych (*active*) i tablicę zadań wygasłych (*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, powraca do listy aktywnych z odpowiednio skróconym kwantem
- gdy wszystkie listy aktywnych zadań się wyczerpią, następuje zamiana tablic i zadania wygasłe stają się 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 Linuksa

Klasy zadań objęte wspólną polityką przydziału CPU w kolejności priorytetu, od najwyższego do najniższego:

1. wstrzymania (*stop*): kluczowe wątki jądra, które musza być wykonane natychmiast
2. terminowa (*deadline*): `SCHED_DEADLINE` dla zadań o ścisłych wymaganiach czasowych
3. czasu rzeczywistego (*real time*): `SCHED_RR`, `SCHED_FIFO`
4. sprawiedliwa (*fair*): `SCHED_NORMAL` (dawniej `SCHED_OTHER`); obejmuje większość procesów użytkownika zarządzanych algorytmem EEVDF (wcześniej CFS)
5. wsadowa (*batch*): `SCHED_BATCH` dla nieinteraktywnych zadań obciążających CPU o niskim priorytecie
6. bezczynna (*idle*): `SCHED_IDLE` dla procesów o bardzo niskim priorytecie (niższym niż +19), które mogą być wykonywane tylko wtedy, gdy system jest bezczynny

Przykład: polityka szeregowania procesów w Linuksie

```
$ 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
```

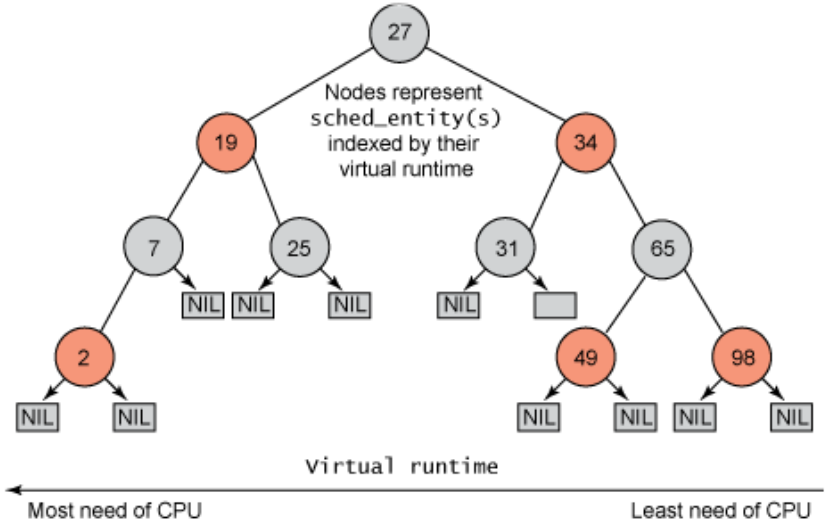
Polityki szeregowania: `TS` (SCHED_NORMAL), `FF` (SCHED_FIFO), `RR` (SCHED_RR), `B` (SCHED_BATCH), `IDL` (SCHED_IDLE), `DL` (SCHED_DEADLINE)

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 całkowicie sprawiedliwy CFS

- struktura zarządzania zadaniami (`runqueue`) jest zorganizowana w postaci drzewa czerwono-czarnego, w którym zadania gotowe do wykonania są uporządkowane według **czasu wirtualnego** (wartość `vruntime`)
- planista wybiera do wykonania zadanie o najmniejszym czasie wirtualnym (skrajny lewy węzeł)



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

Planista CFS

- **Czas wirtualny** to rzeczywisty czas wykonania podzielony przez wagę zadania
- waga zadania zależy od priorytetu (`nice`), niższa uprzejmość oznacza wyższą wagę
- zadania o wyższej wadze (mniej uprzejme) mają wirtualny czas płynący wolniej, co pozwala im na dłuższe uruchomienie
- dla zadań o niskim priorytecie (o niskiej wadze) czas płynie szybciej, dlatego są one szybciej wywłaszczane na rzecz zadań o wyższym priorytecie
- planista okresowo aktualizuje wirtualny czas wykonywanego zadania, porównując go z czasem kolejnego zadania w drzewie
- jeżeli w drzewie znajduje się węzeł o niższym czasie wirtualnym, następuje wywłaszczenie zadania i umieszczenie go ponownie w drzewie we właściwym miejscu (według nowego czasu wirtualnego)

Algorytm EEVDF (Earliest Eligible Virtual Deadline First)

Najwcześniejszy kwalifikowany termin wykonania w pierwszej kolejności

- planista wyznacza **czas wirtualny** dla każdego zadania w ten sam sposób jak w CFS
- średni czas wirtualny wszystkich procesów wyznacza **uczciwą linię** (*fair line*) V, która reprezentuje idealne, sprawiedliwe rozdzielenie CPU
- wirtualne **opóźnienie** (*lag*) zadania to różnica między jego czasem wirtualnym a uczciwą linią

$$lag = vruntime - V$$

- **Kwalifikowalność** (*eligibility*) – zadanie jest kwalifikowane do uruchomienia tylko wtedy, gdy jego czas wirtualny nie przekracza uczciwej linii, czyli gdy

$$lag \leqslant 0 \qquad vruntime \leqslant V$$

- procesy z dodatnim opóźnieniem (które już wiele dostały) są wstrzymywane aż do wyrównania

- **Wirtualny termin wykonania** (*virtual deadline*) – każdy proces otrzymuje kawałek czasu CPU (*slice*), a z niego obliczany jest wirtualny termin wykonania:

$$deadline = vruntime + (slice/weight)$$

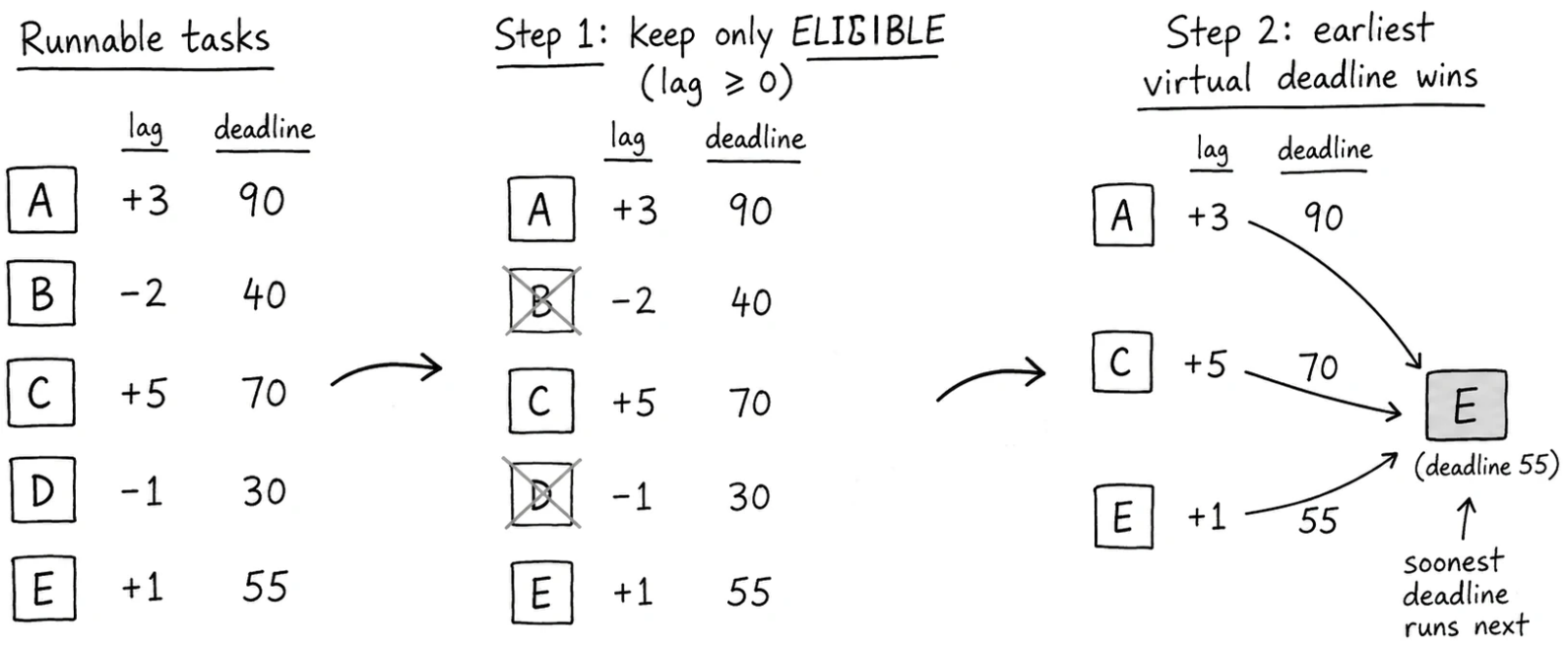
  - procesy o krótszym żądanym slice otrzymują wcześniejszy termin
  - procesy o wyższej wadze mają wcześniejszy termin dla tego samego slice

- Algorytm wyboru: EEVDF wybiera spośród **kwalifikowanych** procesów ten z **najwcześniejszym terminem wykonania**

- `runqueue` zorganizowana jest w drzewie czerwono-czarnym sortowanym po terminie wykonania, co pozwala na wybór w czasie $O(\log n)$

- drzewo jest wzmocnione informacją o minimalnym `vruntime` w każdym poddrzewie, co pozwala na efektywne filtrowanie niekwalifikowanych procesów

Algorytm wyboru EEVDF



Porównanie EEVDF z CFS

- CFS wybierał proces z najmniejszym `vruntime`, co mogło prowadzić do niestabilnych opóźnień (*jitter*) dla procesów interaktywnych
- EEVDF gwarantuje bardziej deterministyczne opóźnienia poprzez koncepcję wirtualnego terminu wykonania
- EEVDF lepiej obsługuje procesy o różnych żądanych slice'ach, preferując krótkie żądania (bardziej interaktywne procesy)
- zarówno CFS jak i EEVDF zapewniają sprawiedliwość poprzez wagę i kwalifikowalność

W algorytmie EEVDF waga zapewnia priorytet, kwalifikowalność zapewnia sprawiedliwość, a wirtualny termin wykonania kontroluje opóźnienie.

Pytania sprawdzające

- Jakie jest zadanie planisty krótkoterminowego?
- Jakie cele realizują algorytmy planowania procesów w systemach operacyjnych ogólnego przeznaczenia?
- Jak wpływa wielkość kwantu czasu na efektywność działania algorytmu rotacyjnego (*round robin*)?
- Jaka jest różnica między algorytmem planowania FCFS i algorytmem rotacyjnym (*round robin*)?
- Jaka jest domyślna wartość priorytetu statycznego procesu uruchomionego przez użytkownika w systemie Linux?

Komunikacja międzyprocesowa (IPC)

Inter-Process Communication

- pamięć współdzielona (*Shared Memory*)
- potoki (*pipes*) - jednokierunkowe przesyłanie strumieni danych
  - potoki nienazwane (*unnamed pipes*) - bezpośrednie połączenie (`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 (dostępne 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`[11] 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

[11] 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`	nieużywane (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 nieokreślona
- 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
- mogą być powiązane z danymi dostarczanymi do procesu (liczba lub wskaźnik)
- umożliwiają odczytanie identyfikatora procesu i użytkownika nadawcy
- są kolejkowane, 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, które w danym sposobie użycia wymagają wyłącznego dostępu, w danej chwili korzysta z nich co najwyżej jeden proces lub wątek

- niektóre urządzenia zewnętrzne (np. drukarka), pliki do zapisu, kolidujące obszary pamięci współdzielonej (np. zmienne globalne aplikacji wielowątkowej), itp.

Zasoby podzielne: zasoby, z których wiele procesów lub wątków może korzystać współbieżnie bez ryzyka powstania błędów lub niespójności danych

- pliki tylko do odczytu, niemodyfikowalny kod biblioteki współdzielonej, kody programów, itp.

Wyścigi i sekcje krytyczne

- Współbieżny dostęp do wspólnych danych lub zasobów może wymagać synchronizacji, jeżeli operacje procesów lub wątków mogą ze sobą kolidować
- **Wyścig** (*race condition*) występuje, gdy poprawność działania zależy od niekontrolowanej kolejności lub czasu wykonania współbieżnych operacji. Np. dwa procesy próbujące jednocześnie zaktualizować tę samą zmienną współdzieloną.
  Wyścigi nazywane są również **hazardem** lub **przeplotem** operacji.
- **Sekcja krytyczna** (*critical section*) - fragment programu, który korzysta ze współdzielonych danych lub zasobów i którego wykonanie wymaga ochrony przed kolidującymi operacjami innych procesów lub wątków
- Aby zapobiec wyścigom musimy zapewnić **wzajemne wykluczenie**

Przykład: zmiana zmiennej współdzielonej

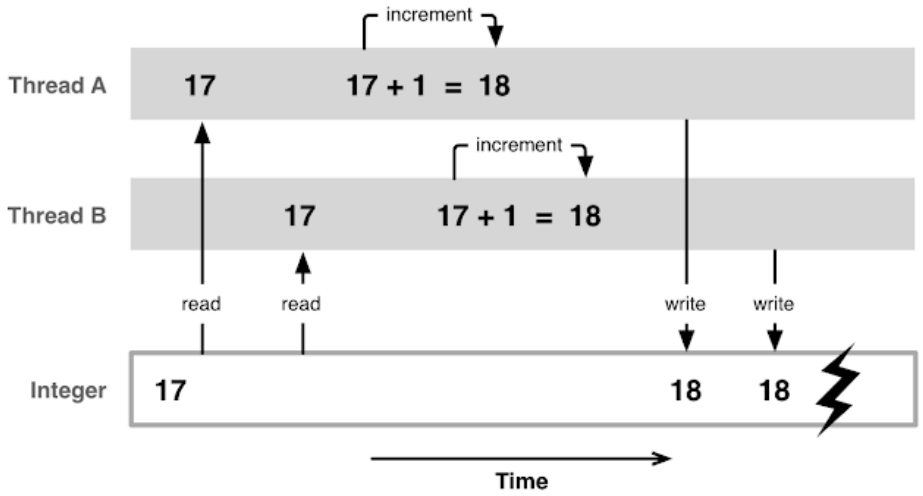
Dwa wątki współdzielą zmienną globalną typu całkowitego `n` i wykonują kod `n++`, co oznacza

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)
```

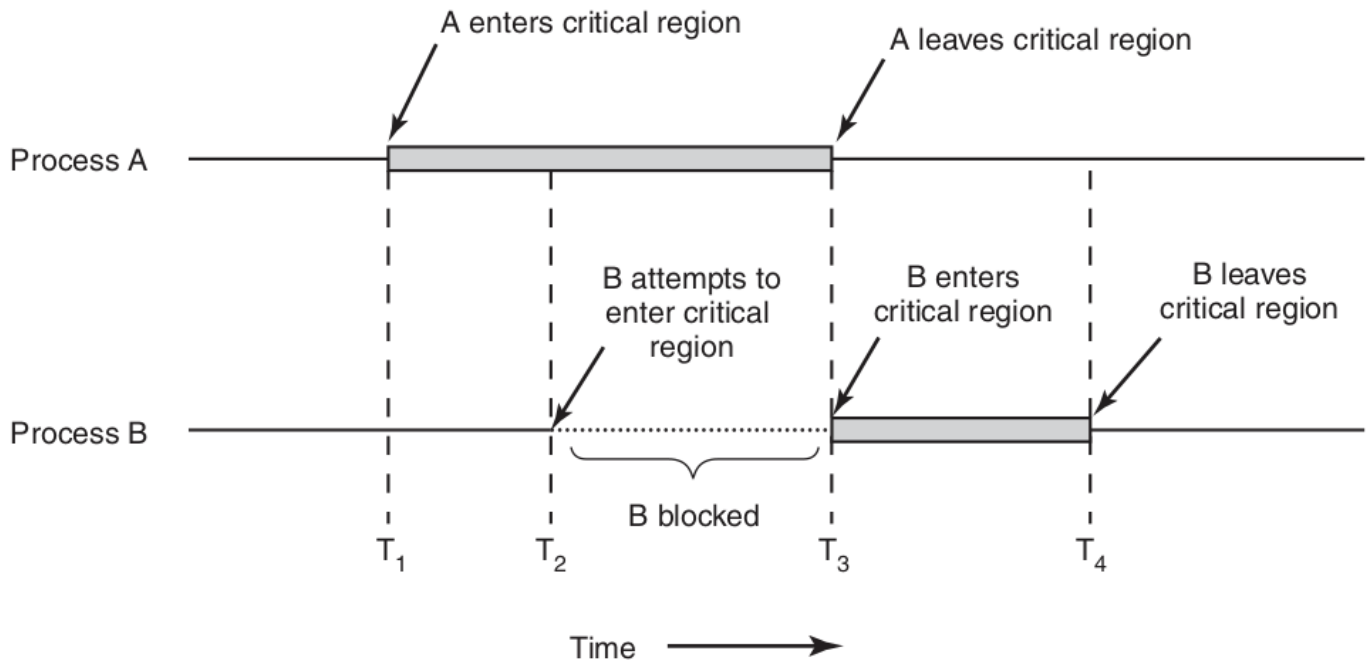
Wyścigi i operacje atomowe



- Zwiększanie wartości zmiennej globalnej powinno być wykonywane w sposób niepodzielny (atomowy) względem innych operacji
- **Operacja atomowa** (*atomic operation*) to niepodzielna operacja, która jest wykonywana w całości, nie można zaobserwować jej częściowego wykonania ani wstawić kolidującej operacji pomiędzy jej etapy. Zapewnia poprawne wykonanie operacji na zasobach współdzielonych

Race Condition, Synchronization, atomic operations and Volatile keyword

Wzajemne wykluczenie



Wzajemne wykluczenie (*mutual exclusion*) polega na zapewnieniu takich warunków działania systemu, by w danej chwili co najwyżej jeden proces lub wątek 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 sekcji krytycznej chroniącej dany zasób
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
  - blokady wirujące (*spin locks*)
  - algorytm Petersona
- **blokowanie oczekujących procesów** (bez aktywnego czekania)
  - semafory, muteksy,
  - monitory
- inne: synchronizacja za pomocą komunikatów, koordynowanie dostępu przez właściciela zasobu

Wyłączanie przerwań

Kod jądra może na krótko wyłączyć lokalne przerwania maskowalne przed sekcją krytyczną, a po jej zakończeniu przywrócić ich poprzedni stan. Zapobiega to przerwaniu tej sekcji przez lokalną obsługę takich przerwań.

Ograniczenia:

- proces użytkownika nie powinien mieć możliwości wyłączania przerwań, technika przydatna tylko dla kodu jądra
- w systemach wieloprocesorowych wyłączanie przerwań dotyczy tylko jednego procesora, nie chroni przed dostępem z innych procesorów
- opóźnia obsługę przerwań, co może wpływać na czas reakcji systemu
- wymaga ostrożności wobec wywłaszczania i blokowania procesów w sekcji krytycznej

Aktywne czekanie (*busy waiting*) - proces lub wątek wielokrotnie sprawdza w pętli, czy warunek synchronizacji został spełniony, zamiast blokować się na czas oczekiwania.

Zalety:

- proste w implementacji
- możliwość szybkiej reakcji bez kosztu uśpienia i wznowienia wątku, przydatne, gdy oczekiwanie na zdarzenie jest krótkie i nieopłacalne jest przełączanie kontekstu

Wady:

- oczekujący wątek zużywa czas procesora i energię, co jest szczególnie niekorzystne przy długim oczekiwaniu

Wzajemne wykluczenie z aktywnym czekaniem

- **zmienne blokujące** (*lock variables*) - wskazują, czy blokada jest wolna, czy zajęta (np. przez inny proces w sekcji krytycznej); proste w użyciu, ale bez operacji niepodzielnych wciąż podatne na wyścigi
- **blokada wirująca** (*spinlock*) to blokada zapewniająca wzajemne wykluczanie, na której zwolnienie wątek oczekuje aktywnie, sprawdzając w pętli stan blokady. Po uzyskaniu dostępu wątek atomowo przejmuje blokadę
- **ścisła zmienność** (*strict alternation*) - proste rozwiązanie wzajemnego wykluczenia dla dwóch wątków z użyciem zmiennej określającej dostęp do sekcji krytycznej na zmianę
- **algorytm Petersona** - programowe rozwiązanie wzajemnego wykluczania

Ścisła zmienność

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

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

Problem:

- wykonanie sekcji krytycznej jest naprzemienne - tempo zależy od wolniejszego procesu
- gdy A zakończy sekcję i ustawi `turn = 1`, nie może wejść ponownie, dopóki B nie wykona swojej sekcji, nawet jeśli B nie chce z niej korzystać, a sekcja jest wolna.
- 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;
  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, w sytuacji konfliktu, gdy oba procesy chcą wejść jednocześnie
- zapobiega wzajemnemu blokowaniu się procesów
- algorytm może być uogólniony na dowolną liczbę procesów

Instrukcja maszynowa TSL

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

Przykład blokady wirującej (*spin lock*) z użyciem instrukcji TSL:

```
enter_region:
  tsl register,lock   | skopiuj lock do rejestru i ustaw lock na 1
  cmp register,#0     | czy lock było równe 0?
  jne enter_region    | jeśli nie, blokada była zajęta - ponów próbę
  ret                 | powrót; wejście do sekcji krytycznej

leave_region:
  mov lock,#0         | ustaw lock na 0
  ret                 | powrót do procedury wywołującej
```

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 blokadą
- wykorzystywane w implementacjach blokad w systemach operacyjnych (np. semaforów) i mechanizmów synchronizacji procesów

Ograniczenia:

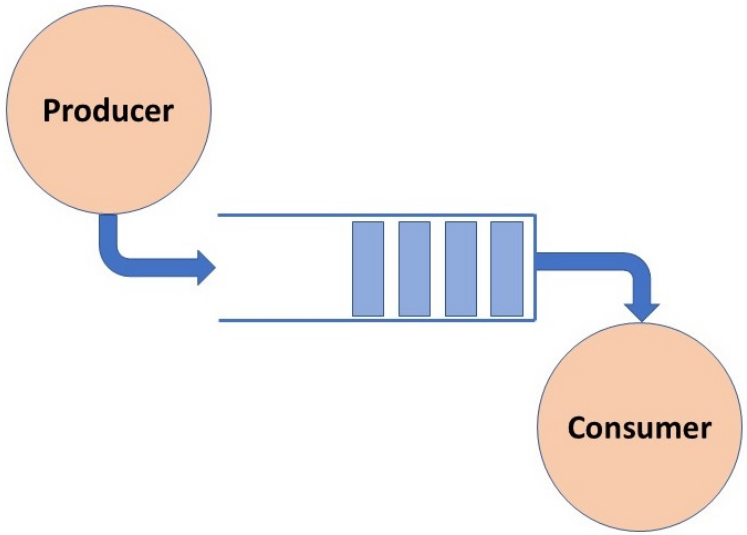
- blokada wirująca wykorzystuje aktywne czekanie, a to marnuje czas CPU
- możliwość wystąpienia głodzenia (*starvation*), np. gdy procesy o wysokim priorytecie ciągle przejmują dostęp do sekcji krytycznej
- możliwość wystąpienia zakleszczeń (*deadlock*) oraz problemu odwróconego priorytetu

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 wstawić przedmiotu, jeśli współdzielony bufor jest pełny
- proces konsumenta nie może pobrać przedmiotu, jeśli współdzielony bufor jest pusty
- dostęp do współdzielonego bufora musi być zabezpieczony wzajemnym wykluczeniem

Próba rozwiązania problemu producenta i konsumenta

```
#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 */
    while (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 == 1) wakeup(consumer);     /* was buffer empty? */
  }
}
```

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

  while (TRUE) {                          /* repeat forever */
    while (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 */
  }
}
```

Problemy:

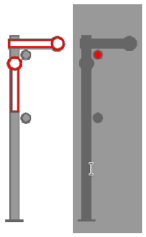
- zmienna `count` nie jest chroniona, w przypadku wyścigu może dojść do zablokowania producenta i konsumenta
- możliwe jest utracone przebudzenie: konsument sprawdza `count == 0`, producent wstawia przedmiot i wywołuje `wakeup`, zanim konsument zdąży zasnąć. Konsument następnie zasypia mimo niepustego bufora.

Potrzebne są mechanizmy synchronizacji, takie jak semafory.

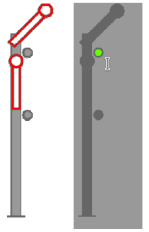
Semafory

Semafor to specjalna zmienna całkowita, na której są wykonywane operacje niepodzielne `DOWN` i `UP`[12]

`SLEEP` ⟶ `DOWN`



`WAKEUP` ⟶ `UP`



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

[12] Oryginalne nazwy operacji używane przez Edsgera Dijkstrę 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 (liczba pozwoleń)
- 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 */
                                     /* semaphores are a special kind of int */
semaphore mutex = 1;                 /* controls 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 */
  }
}
```

- semafory `empty` oraz `full` służą do **synchronizacji**, zapewniają, że zdarzenia będą zachodziły we właściwej kolejności
- 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
- `empty` i `full` reprezentują dostępne pozwolenia na wstawienie i pobranie przedmiotu; oczekiwanie na `empty` lub `full` odbywa się przed zajęciem `mutex`

Rodzaje semaforów

- **semafor zliczający (ogólny)** - wartość całkowita służąca do sygnalizacji między procesami (synchronizacja procesów, zarządzanie 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

W semaforach 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

Monitor to mechanizm synchronizacji, który łączy dane współdzielone z operacjami na tych danych i zapewnia wzajemne wykluczanie podczas ich wykonywania.

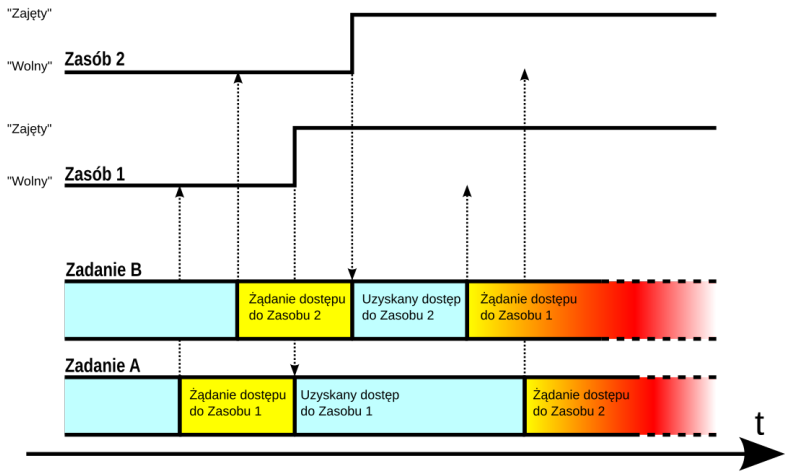
- Dane są dostępne przez procedury monitora; **tylko jeden wątek** może wykonywać jego kod, posiadając blokadę monitora.
- **Zmienne warunkowe** (*condition variables*) umożliwiają oczekiwanie na zmianę stanu monitora.
- **Oczekiwanie** (`wait`) zwalnia blokadę i usypia wątek; przed powrotem wątek ponownie przejmuje blokadę.
- **Sygnalizowanie** (`signal`) powiadamia oczekujący wątek o możliwej zmianie stanu. Po wznowieniu należy ponownie sprawdzić warunek.
- Implementowane często w postaci klas lub obiektów, które enkapsulują dane i metody synchronizujące dostęp do nich.
- W języku Java każdy obiekt ma powiązany monitor udostępniający metody `wait`, `notify` i `notifyAll`, którego blokadę można przejąć za pomocą `synchronized`.

Przykład monitora w C# (producent-konsument)

```
using System.Threading;
class Bufor {
    private readonly object _lock = new object();    // Obiekt używany do synchronizacji
    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.PulseAll(_lock);                               // Powiadom oczekujące wątki
        }
    }
    public int Konsumuj() {
        lock (_lock) {
            while (_licznik == 0) Monitor.Wait(_lock);             // Czekaj, jeśli bufor pusty
            int element = _dane[--_licznik];
            Monitor.PulseAll(_lock);                               // Powiadom oczekujące wątki
            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 od innych problemów zarządzania procesami współbieżnymi nie istnieje skuteczne i ogólne rozwiązanie problemu zakleszczeń.

Jak unikać zakleszczeń?

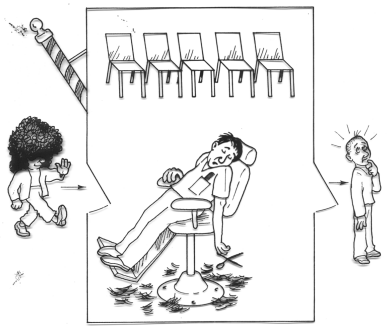
1. Jeśli to możliwe, **należy unikać stosowania blokad** (używaj operacji atomowych lub struktur danych nie wymagających zakładania blokad).
2. Jeśli blokada jest niezbędna, to **należy stosować blokadę pojedynczą**
3. Jeśli potrzebnych jest kilka blokad należy:
   - zdefiniować (małą) lokalną **hierarchię 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 – modelowanie 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 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 dostać dostępu do fryzjera)

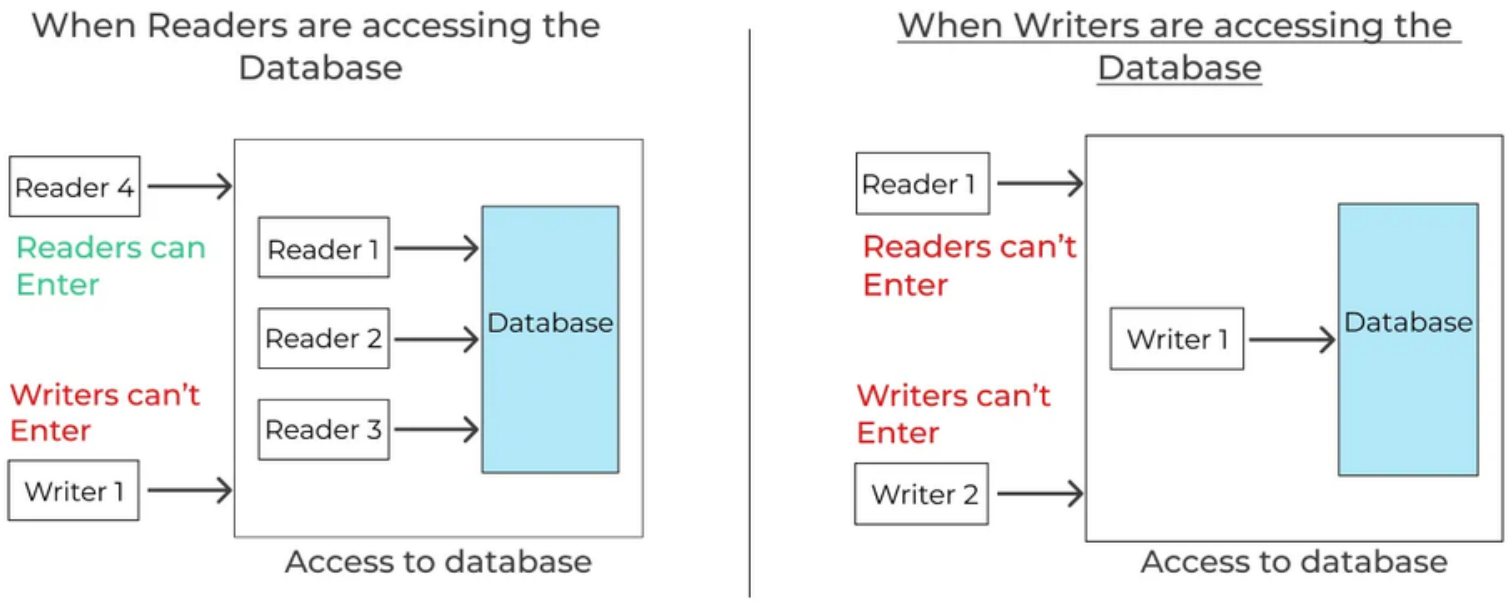
Problem czytelników i pisarzy

- 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)

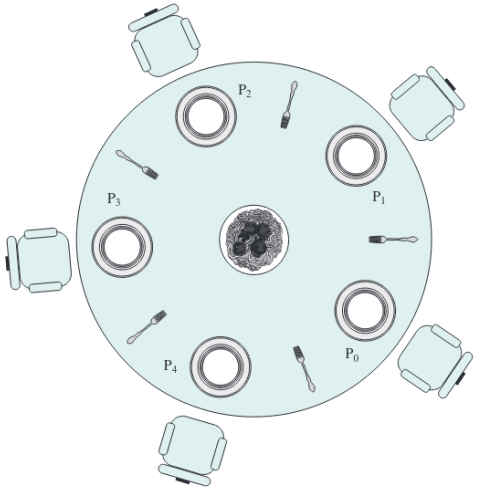
Problem czytelników i pisarzy



https://prepinsta.com/operating-systems/readers-writers-problem/

Problem ucztujących filozofów

- Pięciu filozofów siedzi przy okrągłym stole. Każdy filozof może myśleć lub jeść.
- Między sąsiadującymi 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).



Przykład próby rozwiązania problemu semaforami

```
semaphore fork[5] = {1};

void philosopher(int i) {
    while (true) {
        think();
        down(fork[i]);
        down(fork[(i+1)%5]);
        eat();
        up(fork[i]);
        up(fork[(i+1)%5]);
    }
}
```

Problem: możliwe zakleszczenie, gdy każdy filozof trzyma jeden widelec i czeka na drugi.

Rozwiązanie problemu filozofów semaforami

```
semaphore fork[5] = {1};
semaphore room = 4;

void philosopher(int i) {
    while (true) {
        think();
        down(room);
        down(fork[i]);
        down(fork[(i+1)%5]);
        eat();
        up(fork[i]);
        up(fork[(i+1)%5]);
        up(room);
    }
}
```

Semafor `room` ogranicza liczbę filozofów, którzy mogą jednocześnie próbować podnieść widelce. Dzięki temu zawsze przynajmniej jeden filozof może zdobyć oba widelce i rozpocząć jedzenie.

Pytania sprawdzające

- Jaką sytuację nazywamy wyścigiem (*race condition*)?
- Co to jest semafor? Wymień operacje, które można wykonać na semaforach.
- Do czego służy algorytm Petersona?
- Jaka jest różnica między mutexem a semaforem binarnym?
- Na czym polega aktywne czekanie? Podaj główną wadę takiego rozwiązania.

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 Attachment*) 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
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
CTSS (*Compatible Time-Sharing System*) kompatybilny system z podziałem czasu
DAC (*discretionary access control*) uznaniowa kontrola dostępu

DDR (*Double Data Rate*) podwójna szybkość transmisji danych
DMA (*Direct Memory Access*) bezpośredni dostęp do pamięci
DMAC (*Direct Memory Access Controller*) sterownik bezpośredniego dostępu 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
EEVDF (*Earliest Eligible Virtual Deadline First*) algorytm planowania z najwcześniejszym dopuszczalnym wirtualnym terminem zakończenia
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 Foundation*) Fundacja Wolnego Oprogramowania
GDT (*Global Descriptor Table*) globalna tablica deskryptorów
GNU (*Gnu is Not Unix*) Gnu nie jest Uniksem
GPL (*General Public License*) 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
IRT (*Interrupt Redirection Table*) tablica przekierowań przerwań
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 (*Journaled File System*) system plików z kroniką
KLT (*Kernel-Level Thread*) wątek poziomu jądra
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
MLFQ (*Multi-Level Feedback Queue*) wielopoziomowa kolejka ze sprzężeniem zwrotnym
MMIO (*Memory-Mapped Input/Output*) wejście/wyjście mapowane w pamięci
MMU (*Memory Management Unit*) jednostka zarządzania pamięcią
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
OSI (*Open Source Initiative*) inicjatywa otwartego oprogramowania
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
RCU (*Read-Copy-Update*) odczyt-kopiowanie-aktualizacja - mechanizm synchronizacji wątków w systemach wieloprocesorowych
ROM (*Read-Only Memory*) pamięć stała/tylko do odczytu
RPC (*Remote Procedure Call*) zdalne wywołanie procedury
RR (*Round Robin*) planowanie rotacyjne
RSS (*Resident Set Size*) rozmiar zestawu rezydentnego
SATA (*Serial AT Attachment*) 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
ULT (*User-Level Thread*) wątek poziomu użytkownika
USB (*Universal Serial Bus*) uniwersalna szyna szeregowa
VFS (*Virtual File System*) wirtualny system plików
VLSI (*Very Large Scale Integration*) bardzo duża skala integracji