



# Systemy Operacyjne

Automatyzacja skryptowa

Dr hab. inż. Krzysztof Rzecki, prof. AGH



# Sprawy organizacyjne

- Prowadzący: **Krzysztof Rzecki**, <http://rzecki.pl>
- Wykłady: 30h
- Laboratorium: 15h
- ECTS: 3.0
- Warunki zaliczenia:
  - Warunkiem zaliczenia wykładu jest pozytywna ocena z kolokwium teoretycznego.
  - Warunkiem zaliczenia laboratorium są pozytywne oceny z odpowiedzi oraz kolokwium praktycznego.
  - Warunkiem zaliczenia przedmiotu są trzy pozytywne, w/w oceny.
- Sposób obliczania oceny końcowej
  - Ocena wystawiana na podstawie średniej ważonej wyników uzyskanych z:
    - kolokwium teoretycznego z zakresu wykładów (30%),
    - kolokwium praktycznego z zakresu ćwiczeń laboratoryjnych (40%),
    - ocen[-a|-y] z odpowiedzi ustnej uzyskanej w trakcie ćwiczeń laboratoryjnych (30%).

# BIO

## Edukacja / Nauka

- **Informatyka:**
  - inżynier i magister - EAIiE AGH,
  - doktor - IITiS PAN,
  - habilitacja - RD ITiT AGH.
- **Zarządzanie i Marketing:**
  - magister - WZ AGH.

**Certyfikaty:** POWR.3.5, TOP 500 Innovators, ITIL Foundation, PRINCE2.

## Badania

- metody biometryczne, gesty,
- systemy wizyjne i bronchoskopia,
- radioterapia i promieniowanie kosmiczne,
- laser-induced breakdown spectroscopy,
- wirtualizacja,
- świadomość kontekstu,
- architektury zorientowane na usługi,
- protokoły sieciowe.

## Zawód / Przemysł / Biznes

- Profesor uczelni - KBiB, EAIIB, AGH.
- CEO - Live-Docs.com Sp. z o.o., Kraków.
- Kierownik projektu - KI, IMF, PK, Kraków.
- Kierownik B+R - EPL Sp. z o.o., Międzyrzecz.
- Recenzent w NCBiR.

## Doświadczenie

- ComArch Healthcare S.A., Kraków,
- VSoft SA, Kraków,
- CCNS SA, Kraków,
- Telekomunikacja Polska SA, Warszawa,
- Siemens AG, Monachium,
- Nokia Siemens Networks AG, Monachium.

**Programy:** Fulbright Cybersecurity, ISS on DL.

## Linked in:

<https://www.linkedin.com/in/krzysztof-rzecki/>



# Systemy operacyjne

1. Wprowadzenie i podstawy automatyzacji skryptowej.
2. Rodzaje i architektura systemów operacyjnych.
3. Program, proces, wątek, tworzenie i terminowanie, stany, współbieżność i równoległość.
4. Komunikacja międzyprocesowa, synchronizacja, zakleszczenia.
5. Pamięć operacyjna, pamięć główna, pamięć wirtualna, przestrzeń wymiany, zarządzanie pamięcią.
6. Magazyn danych i system plików.
7. Gniazda i komunikacja sieciowa.
8. Ochrona i bezpieczeństwo systemów operacyjnych.

Co w ramach tego przedmiotu byłoby interesujące, a nie mieści się na w/w liście?



# Literatura

- A. Silberschatz, P. B. Galvin, G. Gagne, Operating System Concepts Essentials, 10th ed.
- A. S. Tanenbaum, Bos H., Modern Operating Systems, Pearson, 4th ed.
- Cooper M., Advanced Bash-Scripting Guide. An in-depth exploration of the art of shell scripting, 2014, online: <https://www.tldp.org/LDP/abs/html/>
- W. Richard Stevens, Stephen A. Rago, Advanced Programming in the UNIX Environment, 3rd ed.
- W. R. Stevens, UNIX Network Programming, 2nd ed.
- R. Love, Linux System Programming: Talking Directly to the Kernel and C Library, 1st ed.
- P. Yosifovich, M. E. Russinovich, D. A. Solomon, A. Ionescu, Windows Internals, Part 1: System architecture, processes, threads, memory management, and more, 7th ed.



# Przygotowanie środowiska

- VPN do sieci AGH
  - <https://pomoc-it.agh.edu.pl/vpn-zdalny-dostep-do-sieci/>
- Konto na serwerze studenckim
  - <https://pomoc-it.agh.edu.pl/konta-unix/instrukcje/zakladanie-konta-unix/>
- Własna instalacja systemu Linux (2+ cores of CPU, 4+ GB RAM, 25+ GB HDD)
  - Natywna, czyli bezpośrednio na komputerze
  - Maszyna wirtualna:
    - VirtualBox: <https://www.virtualbox.org/>
    - QEMU: <https://www.qemu.org/>
    - VMware Player: <https://www.vmware.com/products/workstation-player.html>
- **Uwaga!** Rozwiązania w postaci terminala MacOS, PowerShell, czy Windows Subsystem for Linux(WSL) są niewystarczające do realizacji ćwiczeń.



# Przygotowanie środowiska do zajęć on-line

- **Sprawdź działanie swojego systemu audio** - połącz się przed zajęciami z Koleżanką/Kolegą za pomocą aplikacji, którą użyjesz do zajęć on-line i zapytaj, czy Cię słyszy, wyreguluj ustawienia mikrofonu tak, aby w czasie zajęć było Cię odpowiednio słychać, ale nie przesteruj wzmocnienia.
- **Sprawdź działanie kamery** - połącz się j.w. i zapytaj, czy Cię widać.
- **Sprawdź działanie udostępniania pulpitu i okna aplikacji** - połącz się j.w. i przetestuj działanie udostępniania całego pulpitu oraz poszczególnych okien.



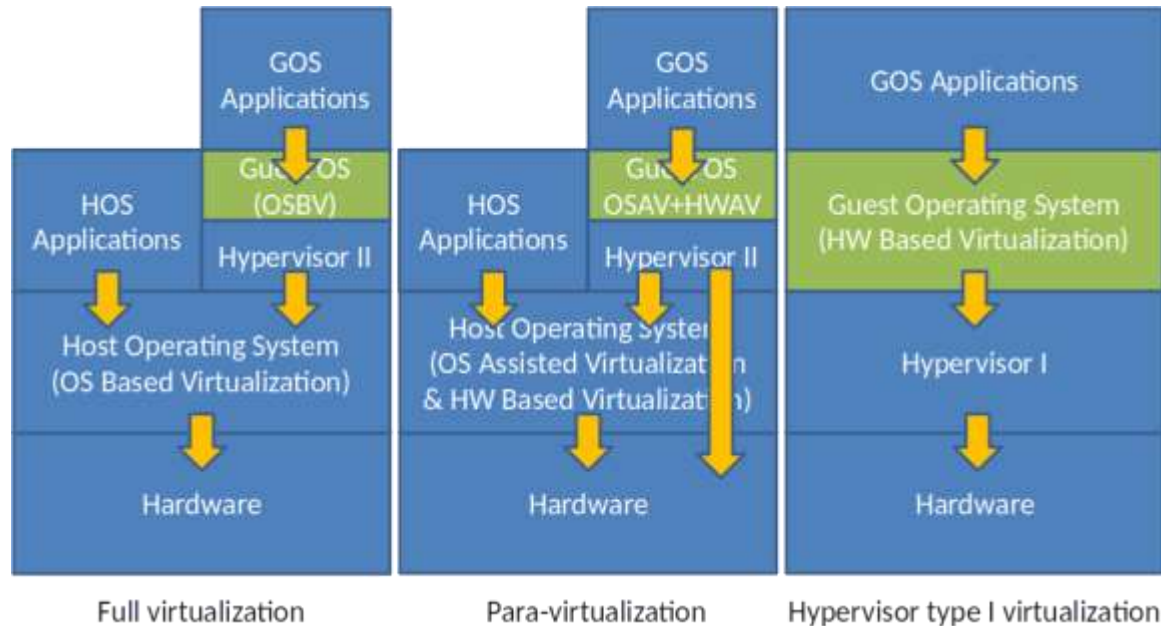
# Własna instalacja systemu Linux

Najpopularniejsze dystrybucje:

- Debian
- Ubuntu
- **Mint Cinnamon**
- Red Hat / CentOS
- Fedora
- openSUSE
- Mandriva
- Slackware



# Wirtualizacja





# Implementacje wirtualizacji

Pełna wirtualizacja Hipernadzorca II typu	Para-wirtualizacja	Bare Metal Hypervisor Hipernadzorca I typu
<ul style="list-style-type: none"><li>• Wirtualizacja sprzętowa BT (Binary Translation):<ul style="list-style-type: none"><li>◦ VirtualBox, 32bit host,</li><li>◦ VMware Workstation, 32bit,</li><li>◦ VMware Server,</li><li>◦ Parallels Desktop.</li></ul></li><li>• Wirtualizacja sprzętowa VT (Virtualization Technology): VT-x, AMD-V<ul style="list-style-type: none"><li>◦ VMware Workstation, 64bit,</li><li>◦ VirtualBox, 64bit host,</li><li>◦ QEMU.</li></ul></li></ul>	<ul style="list-style-type: none"><li>• VirtualBox (tryb parawirtualizacji)</li><li>• Xen (tryb parawirtualizacji)</li><li>• Oracle VM (baza: Xen)</li><li>• IBM LPAR</li></ul>	<ul style="list-style-type: none"><li>• KVM (moduł jądra Linux)</li><li>• Microsoft Hyper-V</li><li>• VMware ESXi</li><li>• Xen</li><li>• Oracle VM Server (baza: Xen)</li></ul> <p>Również implementacje wspierane sprzętowo VT, z pierwszej kolumny:</p> <ul style="list-style-type: none"><li>◦ VMware Workstation, 64bit,</li><li>◦ VirtualBox, 64bit host.</li></ul>



# Wirtualizacja

- Pełna wirtualizacja – binarna translacja instrukcji z systemu gościa do sprzętu poprzez system hosta:
  - Wirtualizowany system operacyjny nie wymaga żadnych zmian/modyfikacji.
  - Większe bezpieczeństwo w izolacji awarii.
- Parawirtualizacja – część instrukcji i odwołań systemu gościa jest tłumaczonych, a część jest przekazywana natywnie do sprzętu:
  - Efektywniejsze wykorzystanie współdzielonych zasobów.
  - Mniejszy narzut obliczeniowy na hosta.



# Podstawowe polecenia - przypomnienie

`pwd, cd, ls, cat, cp, mv, mkdir, rm, touch, locate,`

`find, grep, df, du, head, tail, diff, tar, chmod, chown,`

`id, jobs, kill, ping, wget, history, man, echo, zip, unzip,`

`hostname, useradd, userdel, curl, df, diff, echo, exit, finger, free,`

`grep, groups, less, passwd, ping, shutdown, ssh, reboot, sudo, top,`

`uname, w, whoami`



# Hello world...

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

```
echo "Hello world..."
```

Advanced Bash-Scripting Guide:

<https://tldp.org/LDP/abs/html/>

# Uprawnienia do plików

```
krz@zinc:~/abc$ ls -al
razem 44
drwxrwxr-x  2 krz krz  4096 paź  9 18:40 .
drwx----- 98 krz krz 28672 paź  9 18:40 ..
-rw-rw-r--  1 krz krz    0 paź  9 18:40 file.txt
krz@zinc:~/abc$
```

- Katalog bieżący: ~/abc oraz .
- Katalog nadrzędny: ..

## Ustawianie uprawnień:

```
$ chmod uprawnienia plik
$ chmod 644 file.txt
$ chmod a+rx,a-w directory
```

## Uprawnienia, przykład:

```
drwxr-x---
0123456789
```

Pozycja 0: d (dir), l (link), b (block), c (character)

Pozycja 1, 2 i 3: uprawnienia właściciela 'u'

Pozycja 4, 5 i 6: uprawnienia grupy 'g'

Pozycja 7, 8 i 9: uprawnienia pozostałych 'o'

Pozycje 1..9: uprawnienia wszystkich 'a'

rwX - read, write, eXecute

421 - zapis binarny, np. r-x = 5, rw- = 6, r-- = 4.



# Uprawnienia do plików

0	---	brak uprawnień	blokada
1	--x	wykonywanie	
		katalog bez podglądu	
2	-w-	zapis	
		zbieranie sekretnych logów	
3	-wx	zapis i wykonywanie	
		nieprzydatne	
4	r--	odczyt	
		stała konfiguracja	
5	r-x	odczyt i uruchamianie	pliki
		wykonywalne, katalogi	
6	rw-	odczyt i zapis	pliki
		edytowalne	



# Uprawnienia do uruchomienia

Podstawowe uprawnienie do wczytania i uruchomienia skryptu:

```
$ chmod 555 skrypt
```

lub:

```
$ chmod +rx skrypt
```

Sprawdź:

```
$ chmod u+s skrypt
```





# Skrypty - podstawy

Przypisanie wartości do zmiennej:

```
$ a=5  
$ b=$a
```

Przypisanie wyniku działania polecenia do zmiennej:

```
$ a=`ls /var/log | wc -l`      lub nowsze:      a=$(ls /var/log | wc  
-l)
```

Wypisanie wartości zmiennej na ekran:

```
$ echo "Wartość a wynosi $a"
```



# Wybrane znaki specjalne

**\*** wildcard  
\$ ls /usr/\*bin/

**?** test operator  
\$ variable = a<10?5:7

**{a,b,c}** rozwijanie  
\$ file.{bin,txt} == file.bin file.txt

**{a..z}** lub **{1..9}**

**\$[...]** obliczenie  
\$ a=5; c=\$[a+5]    lub:    \$ let c=\$a+5

**\$** oznaczenie zmiennej  
\$ a=5; echo \$a



# Operacje na zmiennych

## Podstawienie

```
$ a='Ala ma kota'  
$ b=${a/Ala/Piotr}  
$ echo $b
```

## Wyzerowanie zmiennej:

```
$ a=''      lub      a=      lub a=""
```

## Ćwiczenie:

```
$ a=5  
$ a+=6      vs.      $ let a+=6  
$ echo $a
```



# Warunek if

```
if [ condition ]  
then  
    command  
    ...  
fi
```

```
if [ condition ]  
then  
    command  
    ...  
elif [ condition ]  
    command  
    ...  
else  
    command  
    ...  
fi
```

```
if [ condA ] && [ condB ]  
then  
    command  
    ...  
elif [ condC ] || [ condD ]  
    command  
    ...  
else  
    command  
    ...  
fi
```



# Petla for

```
for arg [list]
do
    command
    ...
done
```

```
for n in one two
do
    command
    ...
done
```

```
for n in {1..5}
do
    command
    ...
done
```

```
for n in "a b" "c d"
do
    Set -- $n
    cmd with $1
    cmd with $2
    ...
done
```



# Pęta while

```
while [ condition ]  
do  
    command  
    ...  
done
```

```
while [ $a -lt $b ]  
do  
    command  
    ...  
done  
  
lt, le, gt, ge
```

```
cond()  
{ if ... return 0;  
  else return 1;}  
  
while cond  
do  
    command  
    ...  
done
```



# Pętla until

Czy jest sterowana odwrotnym warunkiem niż while ?



# Podnoszenie uprawnień: su, sudo (doas)

Zalogowany jako 'root':

```
# id -> uid=0(root) gid=0(root) grupy=0(root)
```

Zalogowany jako użytkownik:

```
$ id -> uid=1000(krz) gid=1000(krz)
```

```
grupy=1000(krz),...
```

```
$ su
```

zmiana id użytkownika na 0 (superużytkownik, root)

wymagane: hasło użytkownika root

wykonanie pojedynczego polecenia: `$ su -c <polecenie>`

```
$ sudo
```

wykonanie polecenia z uprawnieniami root'a

wymagane: hasło bieżącego użytkownika + uprawnienia w





## Podnoszenie uprawnień: su, sudo - c.d.

```
$ sudo id                                -> uid=0(root) gid=0(root) grupy=0(root)
$ su - -c "id"                          -> uid=0(root) gid=0(root) grupy=0(root)
```

Aby możliwe było użytkowanie polecenia `su` należy ustawić hasło dla `root`:

- Typowa instalacja Linux: hasło ustawione jest podczas instalacji systemu.
- Instalacja Ubuntu i pochodnych: hasło trzeba ustawić poprzez `sudo`:

```
$ sudo passwd root
```

Aby możliwe było korzystanie z polecenia `sudo` należy ustawić uprawnienia (`visudo`):

- Typowa instalacja Linux: jest konto 'root', brak ustawień `sudoers`.
- Instalacja Ubuntu i pochodnych: użytkownik konfigurowany w trakcie instalacji ma uprawnienia.



## Podnoszenie uprawnień: su, sudo - c.d. 2

Połączenie poleceń:

```
$ sudo su          - mając w /etc/sudoers uprawnienia do sudo bez hasła przejdziemy do 'root'
```

```
$ sudo -i
```

Taki sposób **nie wywołuje** wykonania `.bashrc` konta root:

```
$ su -c <polecenie>
```

```
$ sudo polecenie
```

Taki sposób **wywołuje** wykonanie `.bashrc` konta root (odpowiednik: `$ source .bashrc`):

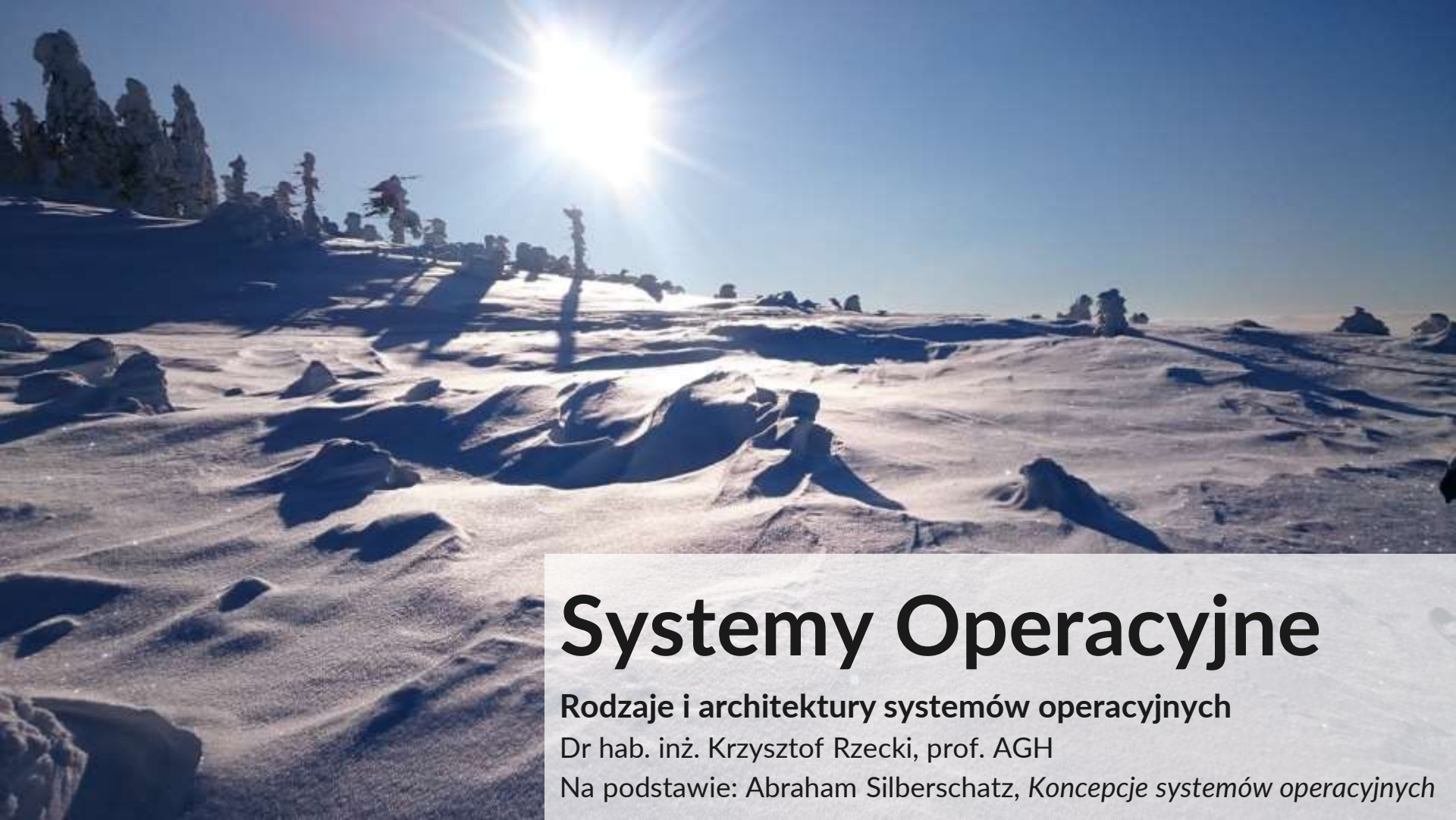
```
$ su - -c <polecenie>
```

```
$ sudo su - -c <polecenie> !!!
```



**Czerwone Wierchy**





# Systemy Operacyjne

Rodzaje i architektury systemów operacyjnych

Dr hab. inż. Krzysztof Rzecki, prof. AGH

Na podstawie: Abraham Silberschatz, *Koncepcje systemów operacyjnych*



# System operacyjny

**System operacyjny** - zarządza sprzętem komputerowym, jest pośrednikiem między użytkownikiem, a sprzętem komputerowym

**Sprzęt komputerowy** - CPU, RAM, I/O devices

Zadania systemu operacyjnego:

- dostarczenie środowiska do uruchamiania i zarządzania (ang. *control program*) programami użytkownika (**wygoda**),
- dystrybucja zasobów (ang. *resource allocator*) do efektywnej eksploatacji sprzętu komputerowego (**wydajność**).

Użytkownicy	
Programy użytkowe	
System operacyjny	Powłoka
	Jądro i moduły
BIOS - <i>Basic Input Output System</i>	
Sprzęt komputerowy	



# System operacyjny jako program sterujący

Program sterujący (ang. *control program*):

- Nadzorowanie działania programów użytkowych.
- Przechwytywanie i przeciwdziałanie błędom.
- Udostępnianie systemu komputerowego użytkownikom.
- Kontrola dostępu użytkowników i programów do zasobów.
- Obsługa i kontrola pracy urządzeń wejścia-wyjścia.

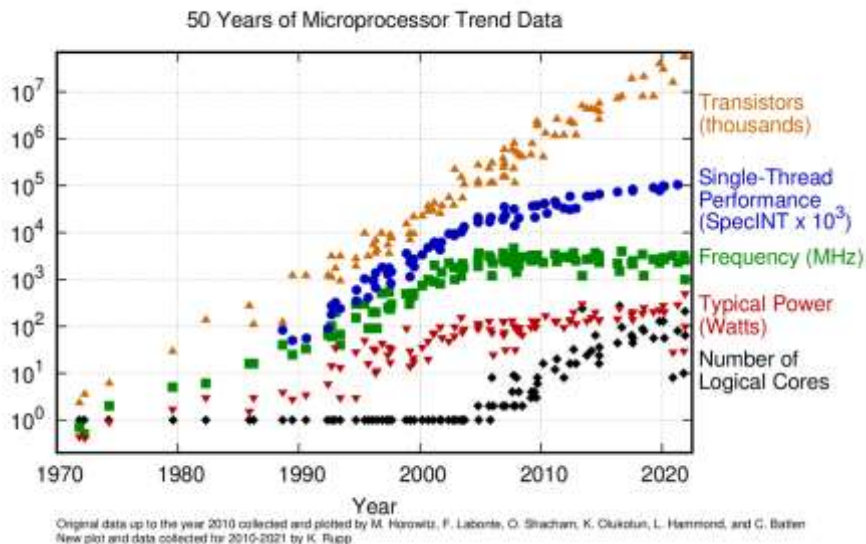


# Dystrybucja zasobów

Dystrybucja zasobów obejmuje:

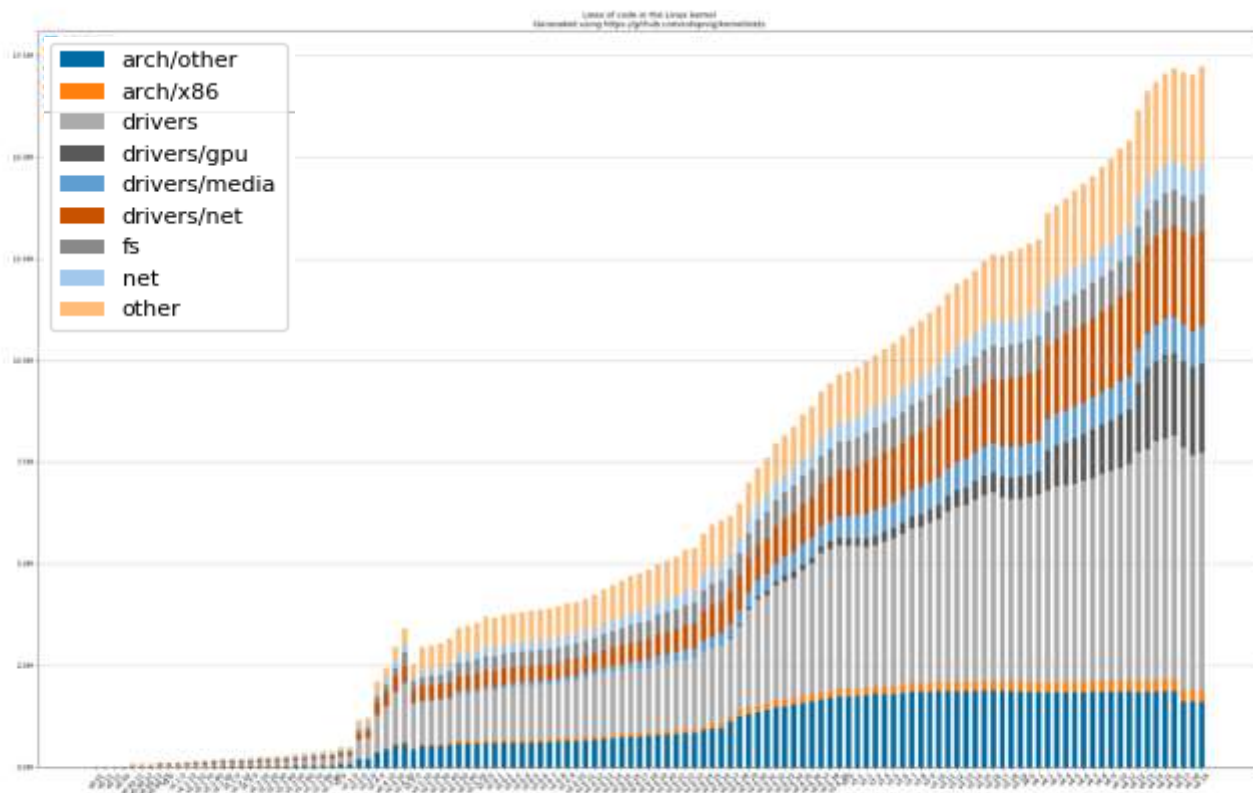
- Planowanie i przydział czasu procesora.
- Kontrola i przydział pamięci operacyjnej.
- Zarządzanie pozostałymi zasobami, jak oprogramowanie czy dostęp do sieci internet.
- Dostarczenie mechanizmów do synchronizacji zadań i komunikacji między zadaniami.

# Rozwój sprzętu - prawo Moore'a

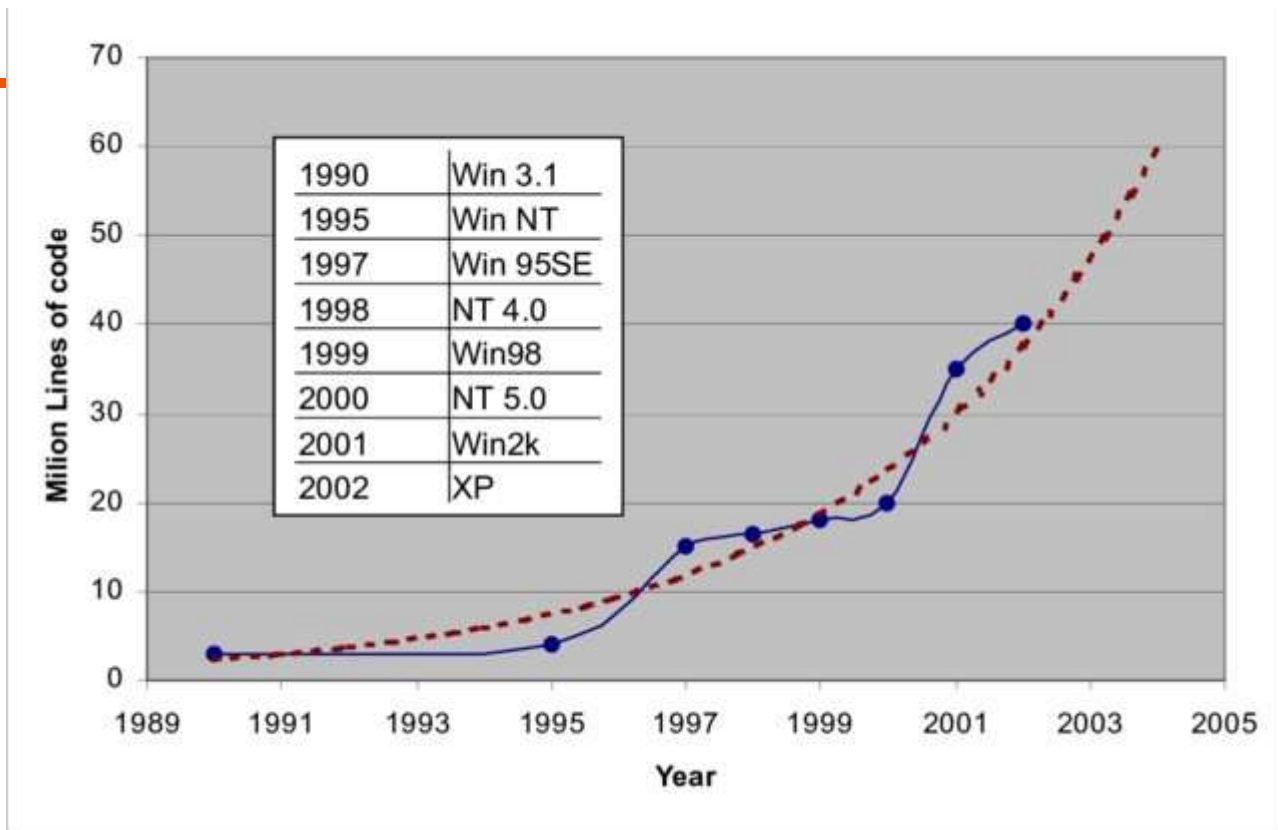


Źródło: <https://github.com/karlrupp/microprocessor-trend-data>





Źródło: <https://github.com/udoprogram/kernelstats>



Źródło: <https://github.com/udoprogram/kernelstats>



# Podział systemów komputerowych

- Ze względu na wielkość:
  - Mainframe
  - Minicomputer
- Ze względu na zasoby:
  - Server
  - Workstation



# Czy system operacyjny to program ?

- System operacyjny to jedyny program działający cały czas na komputerze
  - (uwaga: systemy wbudowane)
- Program ten zwykle nazywamy: **jądro** (ang. *kernel*)
- Pozostałe typy programów:
  - Programy systemowe (część z nich to polecenia systemowe)
  - Programy aplikacyjne (aplikacje użytkowe)



# Jednostka danych

- **bit** - podstawowa jednostka informacji, oznaczenie: b
- bit może przyjmować wartość 0 lub 1
- kombinacja bitów reprezentuje cyfry, litery, obrazy, wideo, dźwięki, dokumenty, programy, itp.
- **bajt** to 8 bitów, oznaczenie B
- **word** to natywna dla danej architektury komputera jednostka informacji
- np. w architekturze 64-bitowej słowo to 8 bajtów
- Mnożniki:
  - $1 \text{ KB} = 1024 \text{ B} = 2^{10} \text{ B}$
  - $1 \text{ MB} = 1024 \text{ KB} = 1024^2 \text{ B} = 2^{20} \text{ B}$  (małe k =  $10^3$ , wielkie K =  $2^{10}$ )
  - $1 \text{ GB} = 2^{30} \text{ B}$ , itd.
  - $1 \text{ Kib} = 1024 \text{ b}$  i analogicznie  $\text{Mib} = 2^{20} \text{ b}$ ,  $\text{Gib} = 2^{30} \text{ b}$ , itd.

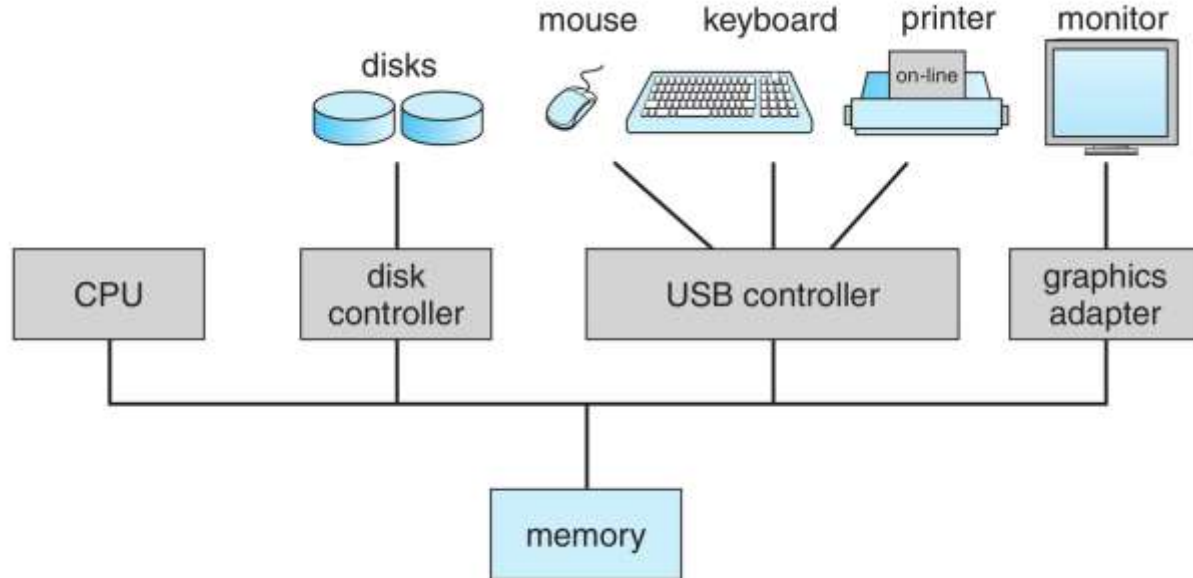
*Młody informatyk myśli, że 1 kilobajt to 1000 bajtów, stary informatyk jest przekonany, że 1 kilometr to 1024 metry...*



# Elementy systemu komputerowego

- Sprzęt: procesor CPU, pamięć RAM, urządzenia we-wy
- Systemy operacyjne: Linux/Unix, Windows, MacOS, itp.
- Programy użytkowe: aplikacje, systemy baz danych, gry komputerowe, oprogramowanie biurowe, środowiska programistyczne, itp.
- Użytkownicy: ludzie, programy, maszyny

# Organizacja systemu komputerowego





# Działanie systemu komputerowego

- Uruchamianie systemu komputerowego:
  - BIOS - Basic Input-Output System
  - Bootstrap program przechowywany w ROM - read-only memory lub EEPROM - erasable programmable read-only memory zwany zwykle firmware
  - Następuje znalezienie i załadowanie systemu operacyjnego oraz pierwszego procesu "init"
  - Oczekiwanie na zdarzenia
- Wystąpienie zdarzenia sygnalizowane jest przez przerwanie (ang. *interrupt*):
  - Sprzętowe przerwanie może być wyzwolone przez przesłanie sygnału przez szynę systemową do procesora
  - Programowe przerwanie może być wyzwolone specjalną operacją, tzw. *system call* lub *monitor call*
- W momencie wystąpienia przerwania, procesor przerywa aktualnie wykonywaną operację, wykonuje procedurę przewidzianą dla danego zdarzenia i wraca do przerwanej operacji.





# Struktura pamięci

- RAM - *random access memory* - pamięć, do której ładowane są programy do wykonania oraz dane dla tych programów
- Pamięci RAM to zwykle technologia półprzewodnikowa DRAM - *dynamic random-access memory*
- ROM - *read-only memory* - pamięć, w której przechowywane są niezmiennalne programy, np. wgrane gry do konsol video
- EEPROM - *erasable programmable read-only memory* - pamięć, która nie może być zbyt często nadpisywana, np. system operacyjny urządzeń mobilnych
- Rejestr - szybka, niewielka pamięć przy/w procesorze na czas wykonywania pojedynczej instrukcji
- Pamięć stała: dysk twardy HDD - *hard disk drive*, dysk optyczny, pen drive, itp.
- Pamięć NVRAM - *nonvolatile RAM* - odpowiedni pamięci RAM podtrzymywany bateryjnie



# Proste systemy wsadowe

- Pierwsze komputery:
  - Wejście: czytniki kart i przewijaki taśm
  - Wyjście: drukarki wierszowe, przewijaki taśm, perforatory kart
- Zadanie na karcie perforowanej: program, dane, karty sterujące
- Czas obliczeń: minuty+ (czasem dni)
- System operacyjny umieszczony na stałe w pamięci operacyjnej
- Grupowanie zadań o podobnych wymaganiach: wsad (ang. *batch*)
- Komputer obsługiwał operator, który pobierał i sortował programy
- Istotna różnica w szybkości działania procesora w porównaniu z we/wy
- Następstwem było wprowadzenie technologii dyskowej



# Wieloprogramowe systemy wsadowe

- Zastosowanie pamięci o dostępie swobodnym (dysków)
- Wczytywanie kart na dysk i zapamiętanie położenia danych => spooling
- Pula zadań (ang. *job pool*) - wczytanie pewnej liczby zadań na dysk
- Możliwość dobierania zadań z dysku tak, aby zwiększyć efektywność jednostki centralnej
- Planowanie zadań (ang. *scheduling*) - planowanie zadań i planowanie przydziału procesora



# Systemy z podziałem czasu (1960r.)

- Problemy systemów wieloprogramowych:
  - wielowariantowość ścieżek wykonywania
  - brak możliwości modyfikowania programu
  - długi czas od rozpoczęcia tworzenia programu do wyniku jego działania
- Podział czasu, wielozadaniowość, ang. *multitasking*
- Interakcyjność, ang. *hands-on* - wymiana danych z programem w ciągu jego trwania
- System plików (ang. *file*, *filesystem*):
  - zestaw powiązanych informacji
  - format, typ
  - organizacja w katalogi
- Bezpośredni dostęp użytkownika do komputera (bez operatora)
- Pamięć wirtualna - wspomaganie pamięci operacyjnej pamięcią dyskową
- Problem zakleszczenia - wzajemne oczekiwanie programów na zasoby



# Systemy dla komputerów osobistych ('70)

- Zmniejszenie cen sprzętu
- Rozwój linii komputerów PC (ang. *personal computer*)
- Początkowo systemy operacyjne dla pierwszych komputerów osobistych: nie wielostanowiskowe, nie wielozadaniowe, lecz z czasem je rozwinięto
- Microsoft: MS-DOS, później Microsoft Windows
- IBM: MS-DOS (od Microsoft), później OS/2
- Apple: Macintosh, później iOS
- Bell Laboratories: UNIX dla PDP-11 (wiele koncepcji z systemu MULTICS dla komputera GE645)
- Na bazie rozwiązań UNIX w latach '80 => Windows NT, IBM OS/2, Macintosh Operating System



# Systemy wieloprocessorowe, równoległe

- Zwiększona przepustowość
- Współczynnik przyspieszenia, który nie zawsze powiela wydajność
- Współużytkowanie urządzeń zewnętrznych
- Zwiększenie niezawodności (redundancja/nadmiarowość węzłów obliczeniowych)
- Wieloprzetwarzanie symetryczne - w każdym procesorze działa identyczna kopia
  - Działa N procesów na N egzemplarzach jednostki centralnej
  - Może się zdarzyć niebalansowanie obciążenia procesorów
  - Wersja Encore systemu UNIX dla komputera Multimax
  - SunOS w wersji 5 (Solaris 2) dla komputerów Sun
- Wieloprzetwarzanie asymetryczne - każdy procesor ma inne zadanie (+procesor główny)
  - Np. słabsze procesory obsługują komunikację (ang. *front-end*)
  - IBM i komputer IBM Series/1 jako procesor czołowy
  - SunOS w wersji 4 dla komputerów Sun

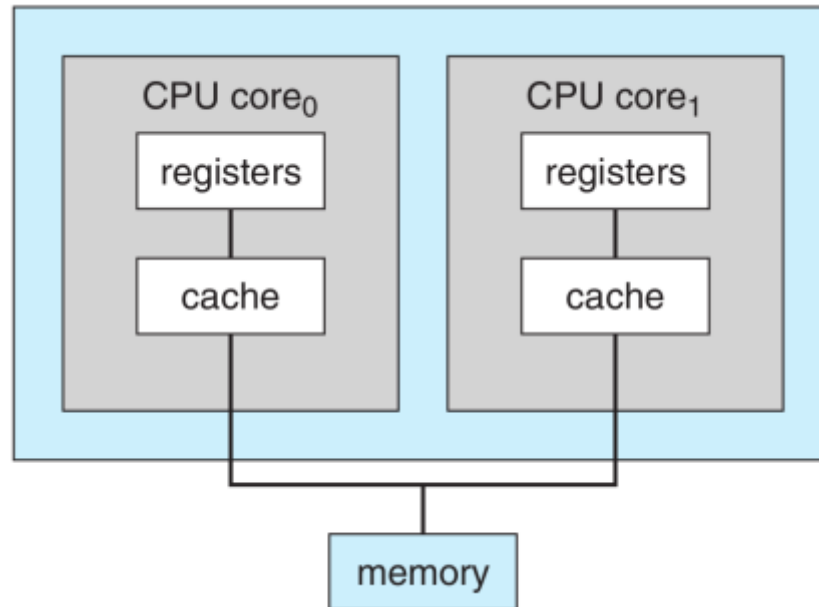


# UMA/NUMA

- UMA - *uniform memory access* - pamięć o jednorodnym czasie dostępu
- NUMA - *non-uniform memory access* - pamięć o niejednorodnym czasie dostępu

System operacyjny musi tak zarządzać dostępem do pamięci, aby minimalizować efekt NUMA.

# Procesor wielordzeniowy







# Systemy rozproszone, systemy klastrowe

- Systemy luźno powiązane (ang. *loosely coupled*), rozproszone (ang. *distributed systems*)
- Rozdzielenie geograficzne
- Połączenie przez linie telekomunikacyjne (internet, linie telefoniczne, itp.)
- Zróżnicowanie architektur poszczególnych węzłów (ang. *node*)
- Zróżnicowanie mocy obliczeniowych poszczególnych węzłów
- Cechy:
  - Podział zasobów
  - Przyspieszenie obliczeń
  - Niezawodność
  - Komunikacja

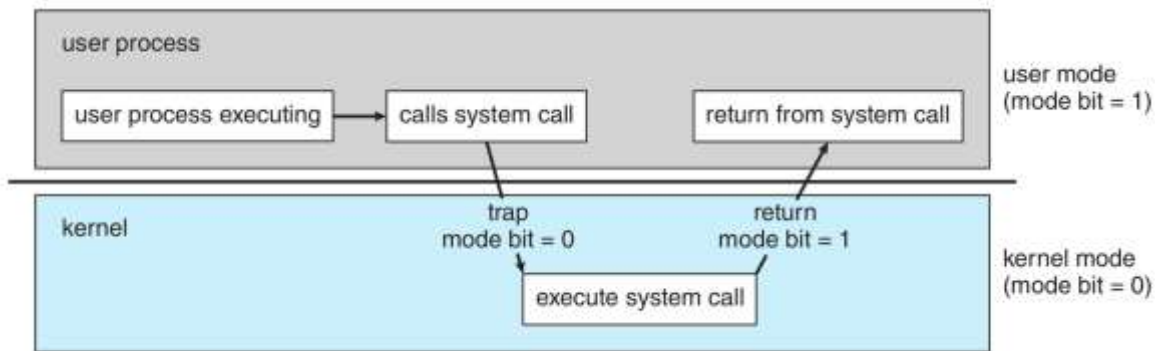


# Systemy czasu rzeczywistego (ang. *real-time*)

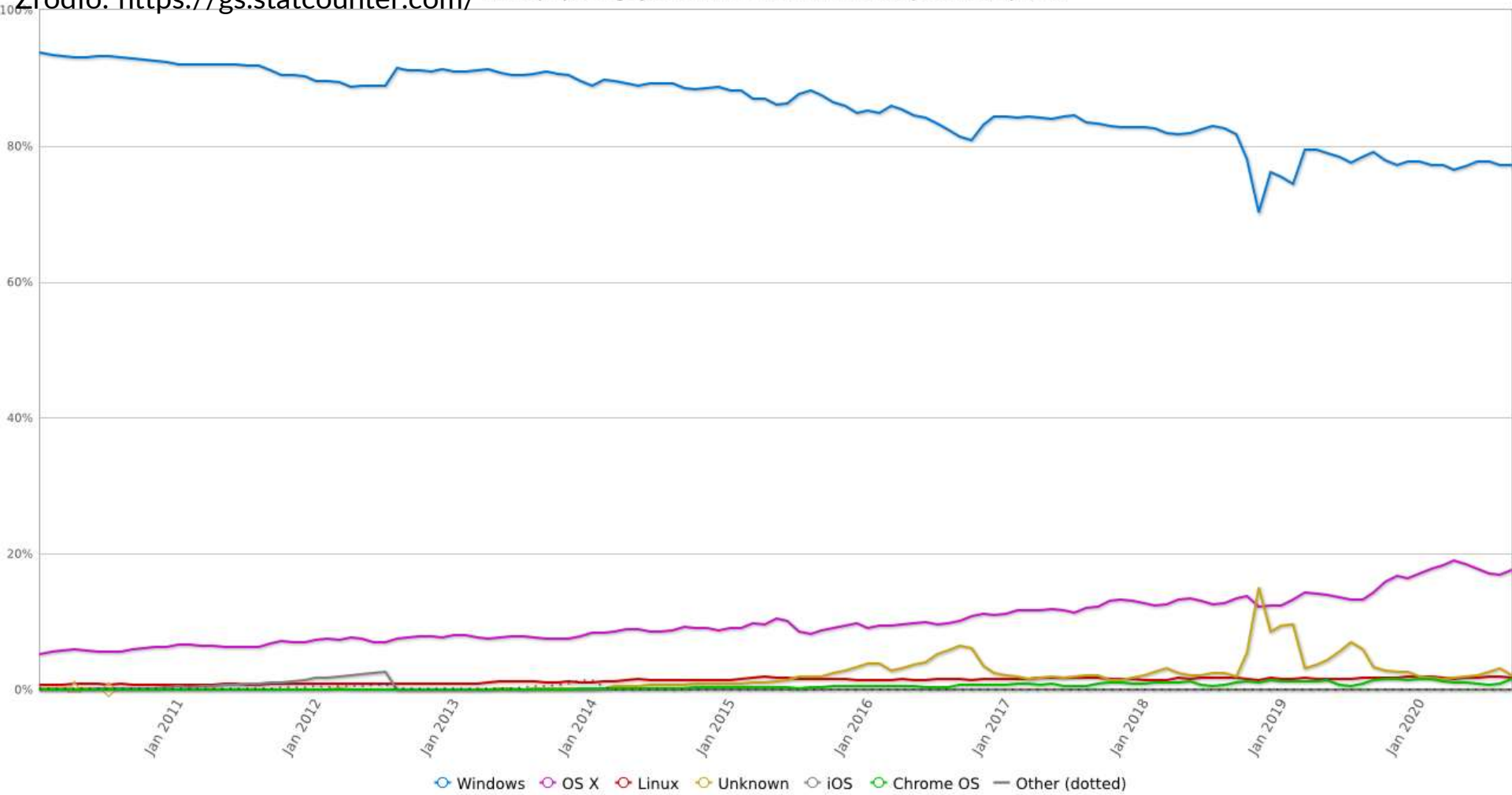
- Zastosowanie: surowe wymagania na czas wykonania operacji lub przepływu danych
- Przykłady:
  - Jednokierunkowe sterowanie maszyną według zadanego programu
  - Odczytywanie wartości czujników
  - Analiza odczytanych wartości czujników i adekwatna reakcja robota
  - Analiza otoczenia, przetwarzanie danych (sygnałów, obrazów) i podejmowanie decyzji
- Odmiany systemów czasu rzeczywistego:
  - Rygorystyczny (ang. *Hard real-time system*) - terminowe wypełnienie krytycznych zadań
  - Łagodny (ang. *Soft real-time system*) - krytyczne zadanie do obsługi otrzymuje pierwszeństwo

# Tryby pracy systemu operacyjnego

- Tryb użytkownika (ang. *user mode*)
- Tryb jądra (ang. *kernel mode*)
- Zaimplementowany w sprzęcie *mode bit*: 0 - kernel mode, 1 - user mode

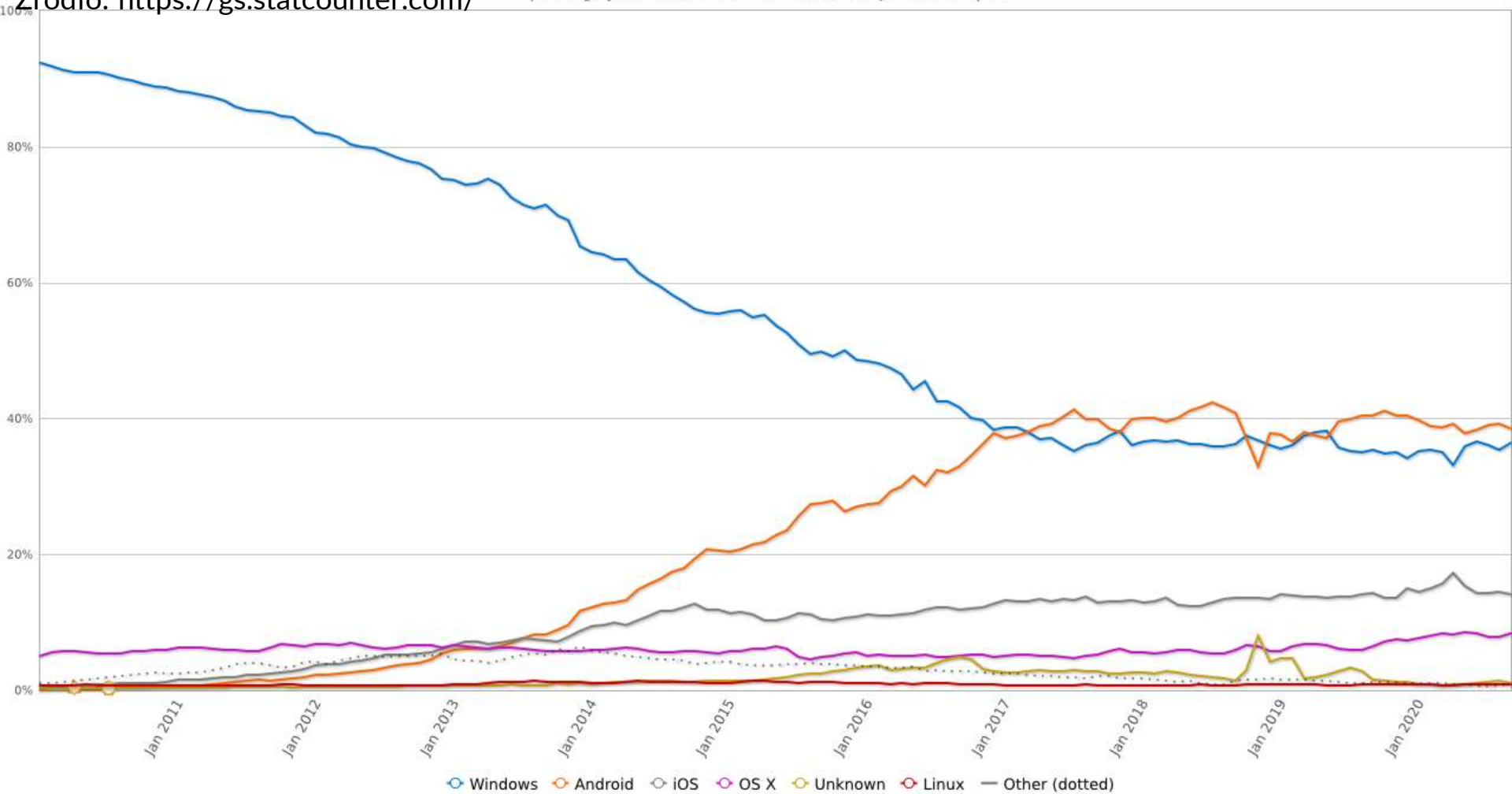


**StatCounter Global Stats**  
Desktop Operating System Market Share Worldwide from Jan 2010 - Sept 2020

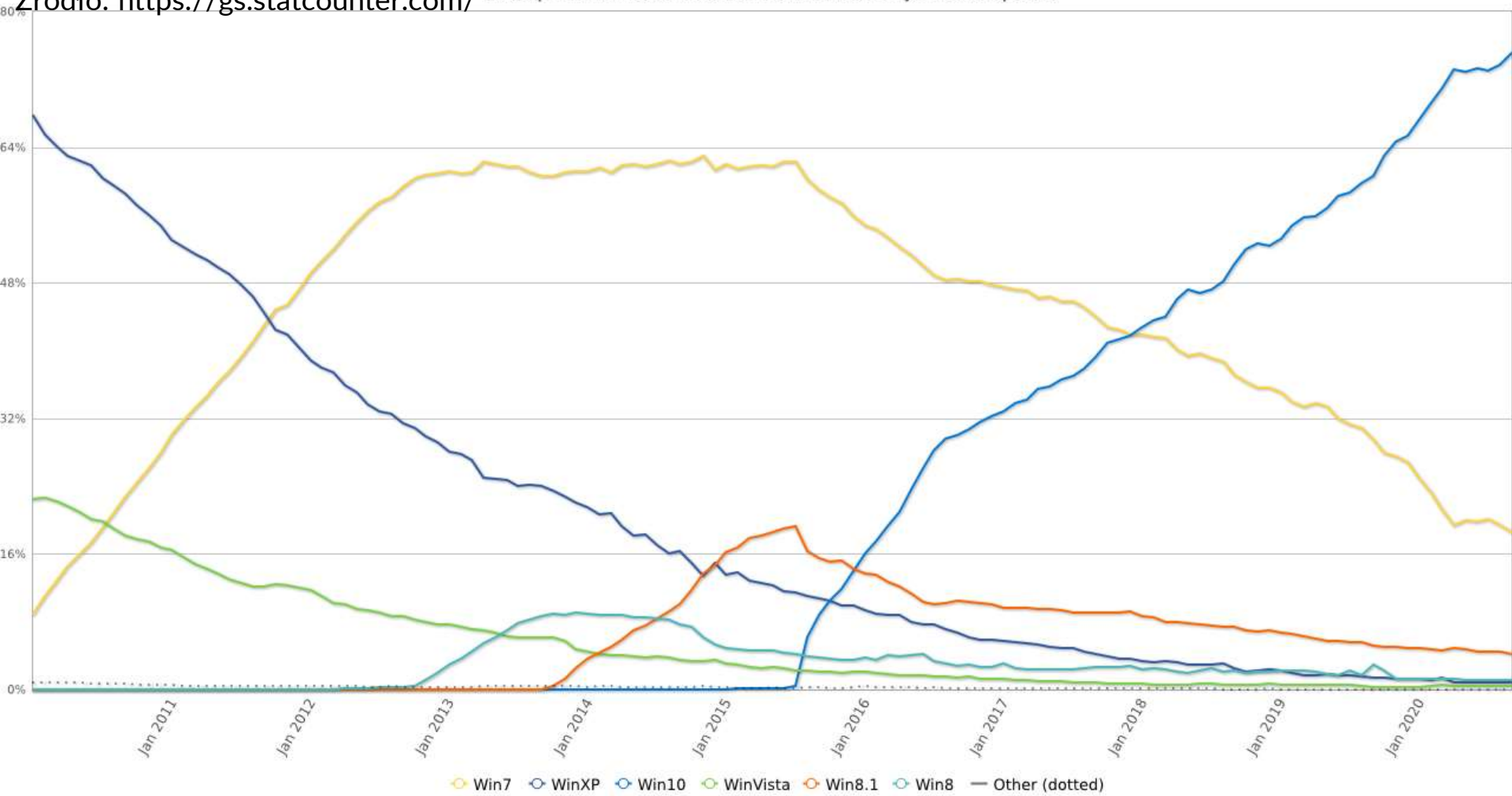


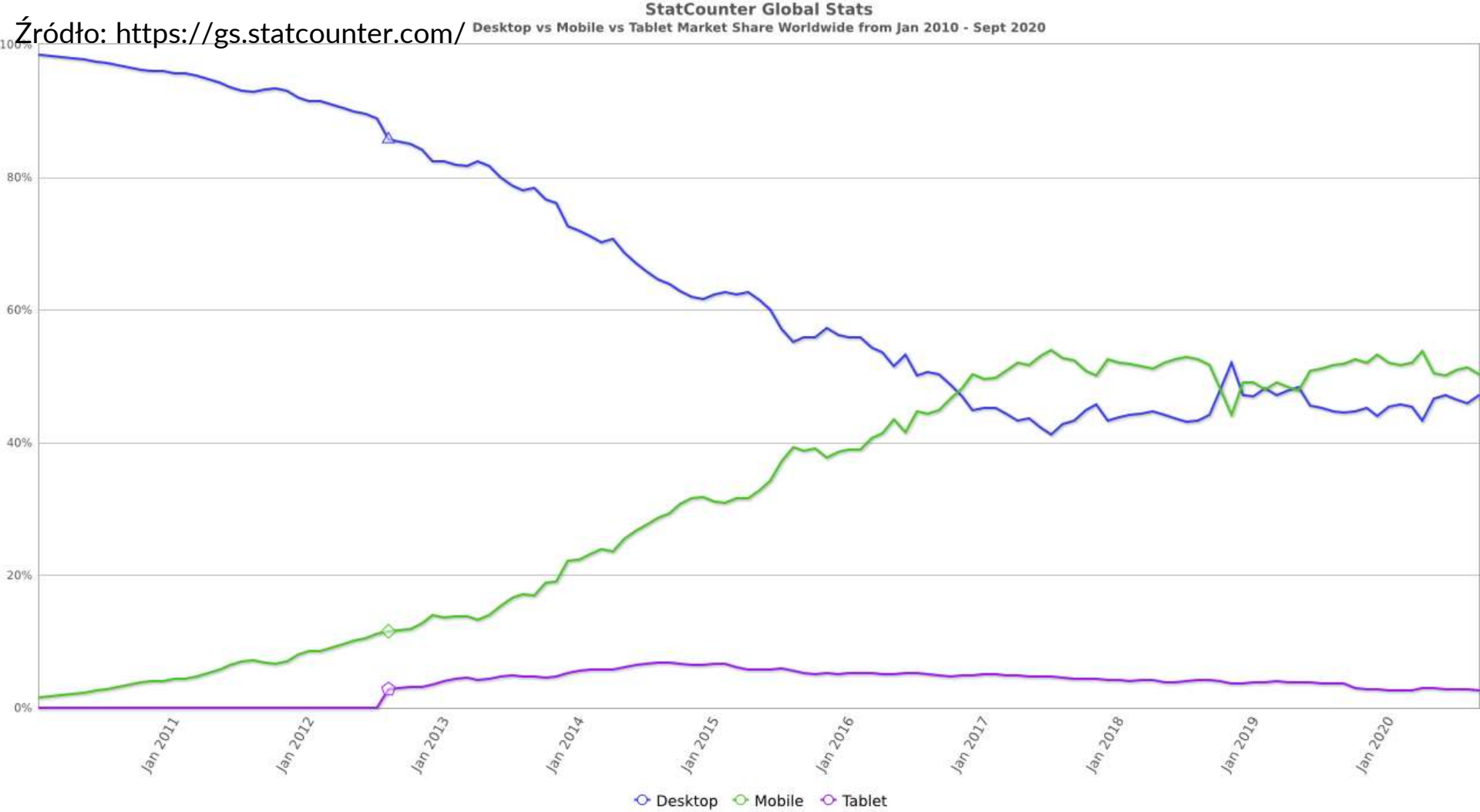
## StatCounter Global Stats

Operating System Market Share Worldwide from Jan 2010 - Sept 2020



**StatCounter Global Stats**  
Desktop Windows Version Market Share Worldwide from Jan 2010 - Sept 2020





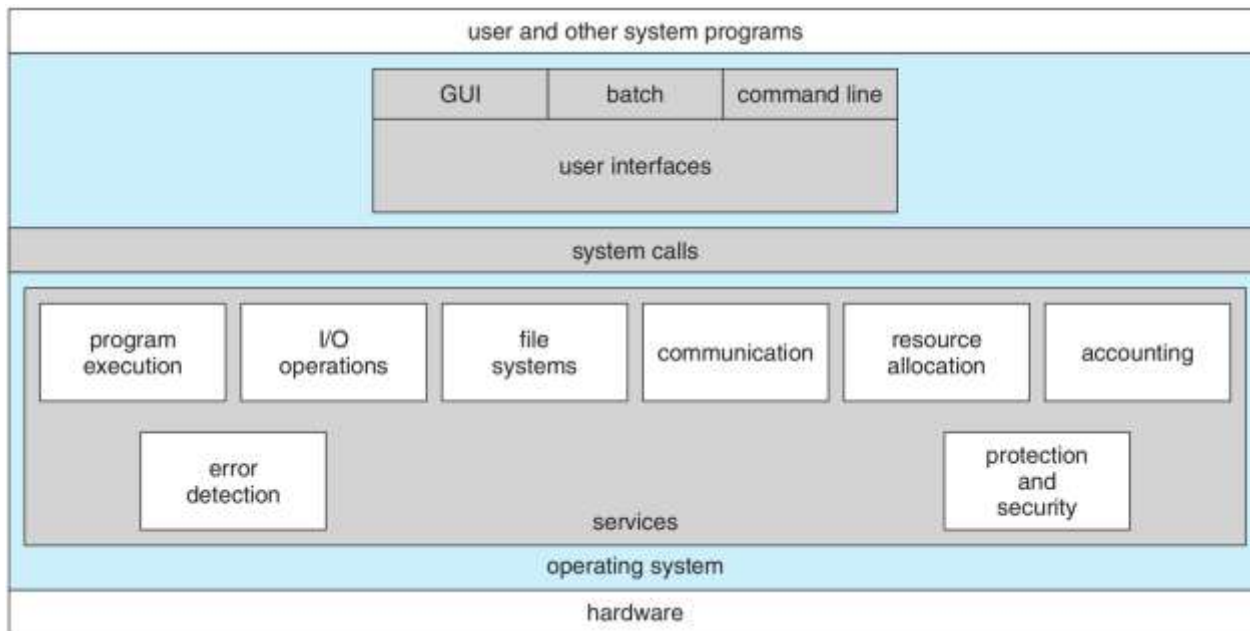


# Pytania...

- Jakie zagrożenia niesie za sobą wielodostęp ?
- Co jest główną zaletą wieloprogramowości ?
- Czy współcześnie może mieć zastosowanie system bez dysku twardego ?
- Jakie uzasadnienie istnienia mają systemy rozproszone ?
- Jaka jest największa trudność w tworzeniu systemu czasu rzeczywistego ?
- Jakie znaczenie ma tryb jądra i tryb użytkownika dla bezpieczeństwa systemu ?



# Struktura systemu operacyjnego





# Interfejs użytkownika

- Graficzny interfejs użytkownika (GUI - graphical user interface) to najpopularniejszy wśród użytkowników biurowych interfejs człowiek-komputer.
- Interpreter poleceń (CLI - command-line interface) jest interfejsem użytkownika (UI) przyjmującym komendy tekstowe oraz funkcje.
- Interfejs *batch* to zbiór komend i dyrektyw kontrolujących te komendy zapisanych w pliku, który po uruchomieniu kolejno uruchamia komendy z tego pliku.
- Interpreter poleceń == *shell*: sh (Bourn shell), bash (Bourne-Again shell), csh (C shell), zsh (Z shell), ksh (Korn shell), itp.
- Interpretacja poleceń: polecenie systemowe (np. cd) lub program (np. rm).



# Podstawowe polecenia bash

- Dokumentacja na temat poleceń dostępna jest (wychodzimy klawiszem 'q'):

`$ man <nazwa polecenia lub programu>`

- **Polecenia:** pwd, cd, ls, cat, cp, mv, mkdir, rm, touch, locate, find, grep, df, du, head, tail, diff, tar, chmod, chown, jobs, kill, ping, wget, history, man, echo, zip, unzip, hostname, useradd, userdel, curl, df, diff, echo, exit, finger, free, grep, groups, less, passwd, ping, shutdown, ssh, reboot, sudo, top, uname, w, whoami
- **Instrukcje:** instrukcjami skryptowymi: if, for, while, until, etc.
- **Edytory tekstu:** vim, pico, etc.

# Poziomy języków programowania

Level 5.

Java/Bytecode

Level 4.

C/C++ Source

Level 3.

ASM Source

Level 2.

Object File

Other Object Files ("Libraries")

Level 1.

Executable File ("Machine Code")





**Skrzyczne**



# Operating systems

## Processes

Krzysztof Rzecki, PhD, DSc

Based on: Abraham Silberschatz, *Operating system concepts*



# From program to process

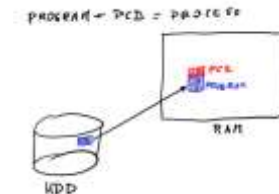
Program code writing:

Source code -> Compiling -> Linking -> Machine code

Running the program:

Machine code, program -> Reading to the main memory -> Adding PCB -> Process

+ Program Counter -> running process





# Process

- The very first operating systems: there is the only one program and it has access to all resources
- Contemporary operating systems: management of running programs -> processes
- The process:
  - code loaded into memory and run
  - unit of work in a time-sharing system
- The management includes: control and separation
- Effect: the system contains collections of processes:
  - operating system processes
  - user processes
- Potentially, all processes are launched simultaneously
- The processor(s) switch between processes, which increases the efficiency of the computer system





# Process creating and terminating

## Creating:

- New batch job
- Interactive login
- Service creation by OS
- Splitting (spawning) an existing process

## Terminating:

- Proper ending
- No enough memory
- Protection breach (of memory)
- Operator or SO intervention

## The process informs about completion:

- HALT instruction
- User action (e.g. log off)
- Failure or error
- Termination of the parent process



# Process creation - interface

Program in path:

```
$ xclock
```

User program:

```
$ ./program
```

Program in background:

```
$ xclock &
```

Program termination:

```
ctrl + c
```

Tasks list:

```
$ jobs
```

Move to the background:

```
$ bg %1
```

Stop the process:

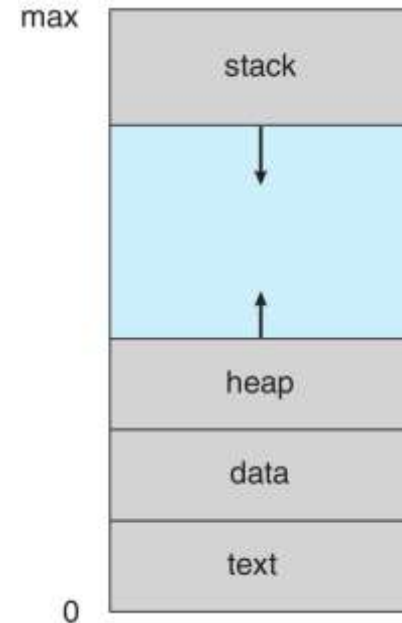
```
ctrl + z
```

Move to the foreground:

```
$ fg %1
```

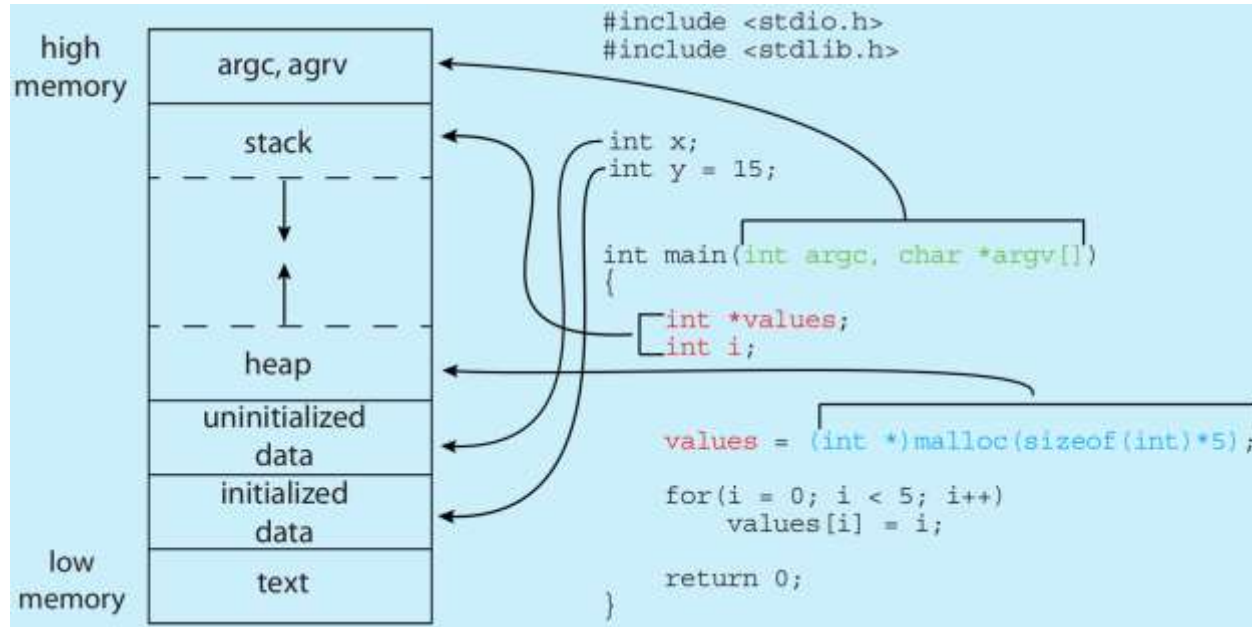
# Process concept

- Batch system: *jobs*, Shared system: *user programs* or *tasks* -> actually: *process*
- Process:
  - Program code, *text section*
  - Program counter == Instruction counter == Indicator of the instruction being executed
  - Process stack (procedure parameters, return addresses, temporary variables)
  - Data section - i.e. global variables
  - Heap - i.e. dynamically allocated memory
- Program is passive - executable file contains list of instructions
- Process is active - program counter points to the next instruction

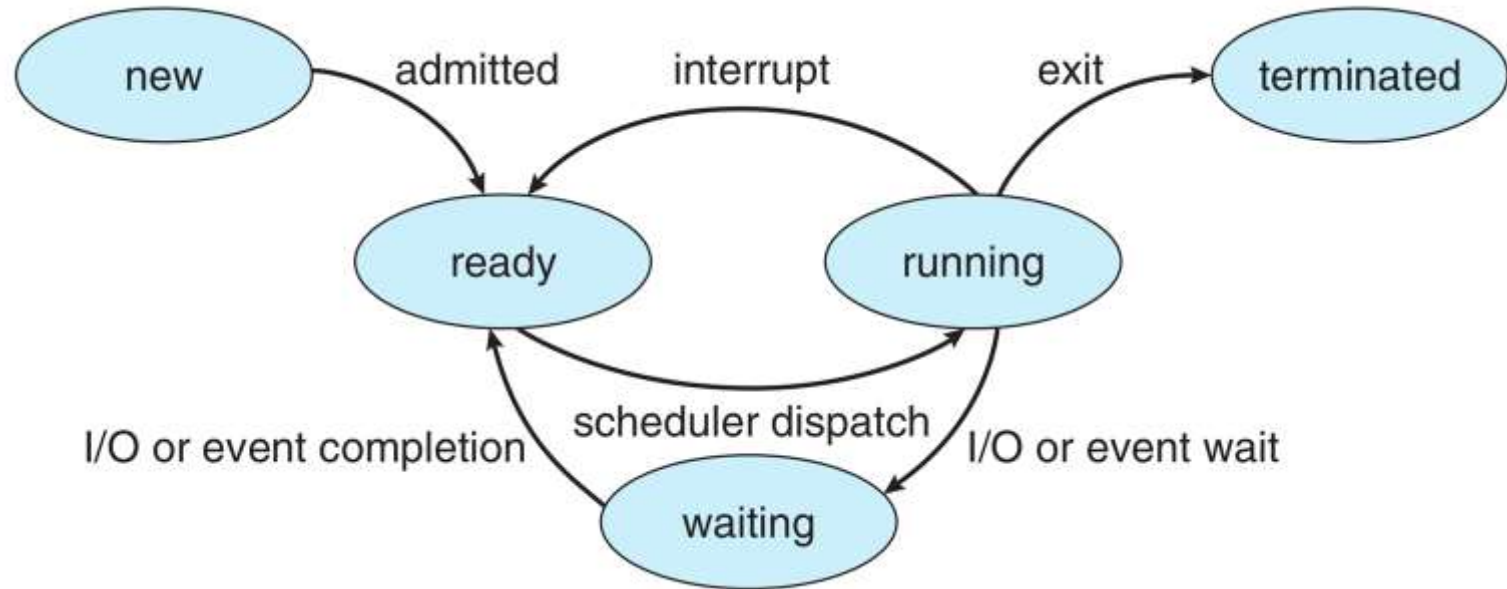


Source: A. Silberschatz, *Operating Systems Concepts Essentials*

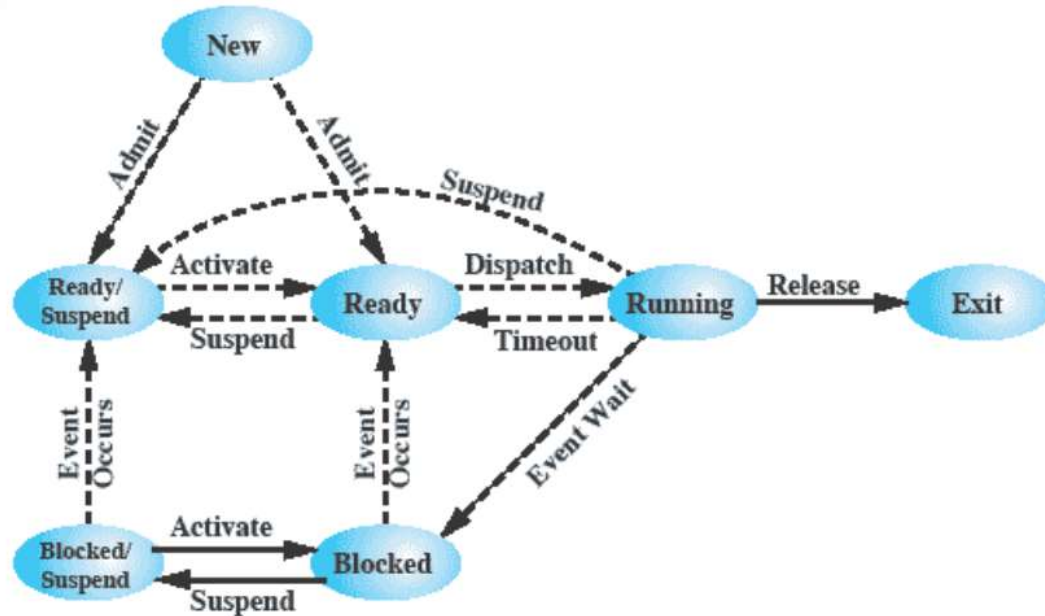
# Program in C



## Process states (five state model)



## Two states of suspend (\*)





## Reasons for suspending the process (\*)

- **Swapping** - the operating system must free a certain amount of (main) memory to run a process that is ready to execute.
- **Another OS problem** - errors.
- **Interactive user request**
- **Timing** - periodic calling of a process (e.g. monitoring) and it can be suspended until the next call.
- **Parent process request** - SIGSTOP / SIGTSTP / SIGCONT

# Process control block

PCB is an area of memory containing various information associated with the particular process.







# Process Control Block

- Process state (next slide)
- Process number (next slide)
- Program counter (next slide)
- CPU registers - including: accumulators, index registers, stack pointers, general purpose registers, condition registers, etc.
- CPU-scheduling information - including: process priority, pointer to queues, etc.
- Memory-management information - including: contents of boundary registers, page tables or segment tables, etc.
- Accounting information - amount of CPU and real time used, time limits, account numbers, task or process number, etc.
- Information about the input/output status - I/O status information - including: list of I/O devices assigned to the process, list of open files, etc.

process state
process number
program counter
registers
memory limits
list of open files
...

# Process Control Block

Linux kernel:

```
$ /usr/src/
```

```
$ linux-headers-[kernel v.]/
```

```
$ include/linux/sched.h
```

Defined in:

```
task_struct
```

A number of records !



User mode:

```
$ /proc/[PID]
```

```
$ ps
```

# PCB - process state

Process state

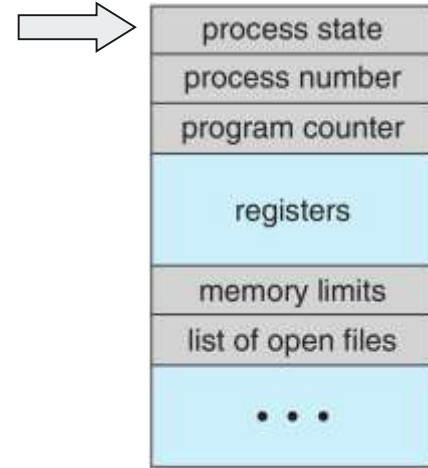
```
$ ps -p [PID] -o pid,status,comm
```

Or:

```
$ cat /proc/[PID]/status | grep Stat
```

Available states:

- R=running,
- S=sleeping in an interruptible wait,
- D=waiting in uninterruptible disk sleep,
- Z=zombie,
- T=traced or stopped (on a signal),
- W=paging





# PCB - process number

PID - Process identifier

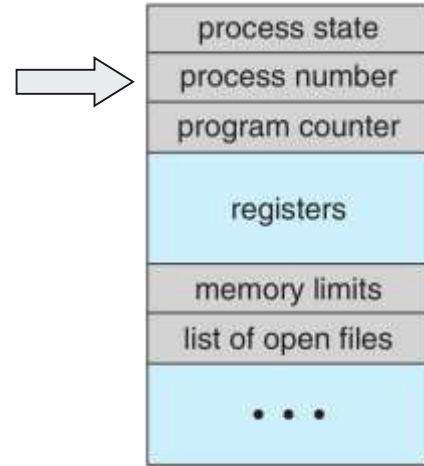
```
$ pidof <program name>
```

or:

```
$ ps aux | grep <program name>
```

The PID number of the process will be returned.

The PID is an integer number unique in the operating system space.





## PCB - process number

Get my own PID:

```
$ pid_t getpid(void);          // unistd.h
```

Get my parent PID:

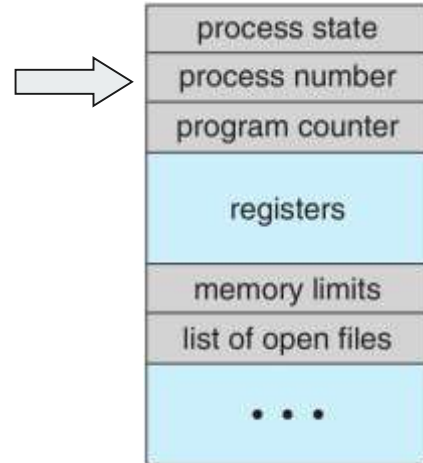
```
$ pid_t getppid(void);        // unistd.h
```

Create the new process and get its PID:

```
$ pid_t fork(void);           // unistd.h
```

Functions where the argument is PID:

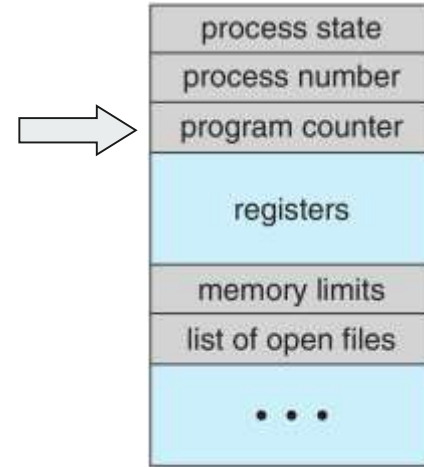
kill, wait, nice



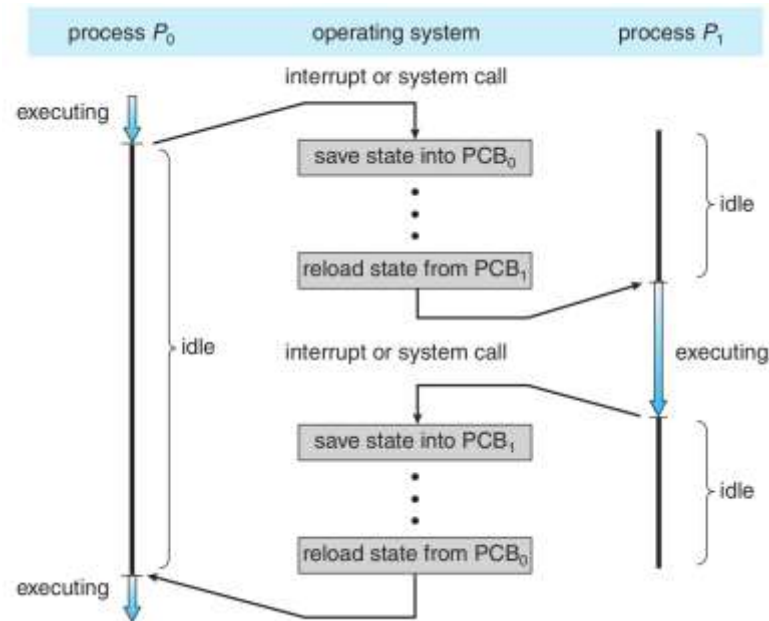


# PCB - program counter

Program counter - address of the next instruction to be executed.

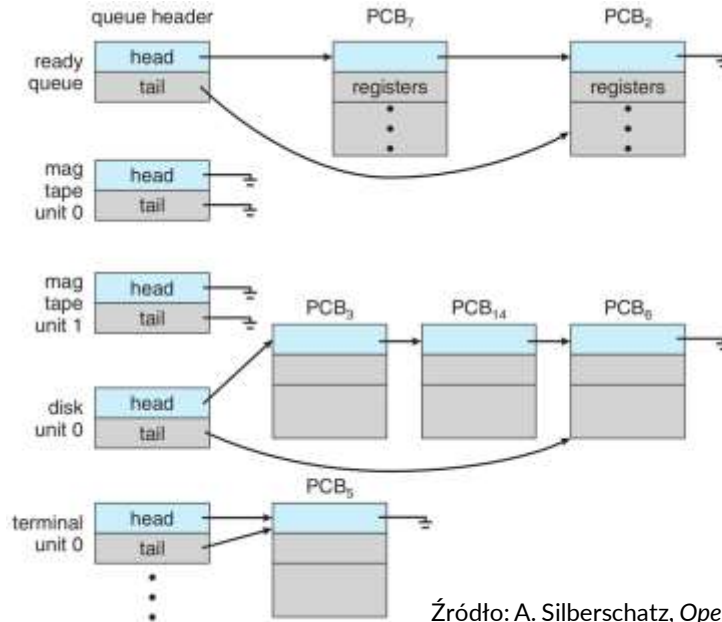


# Process control block - process switching



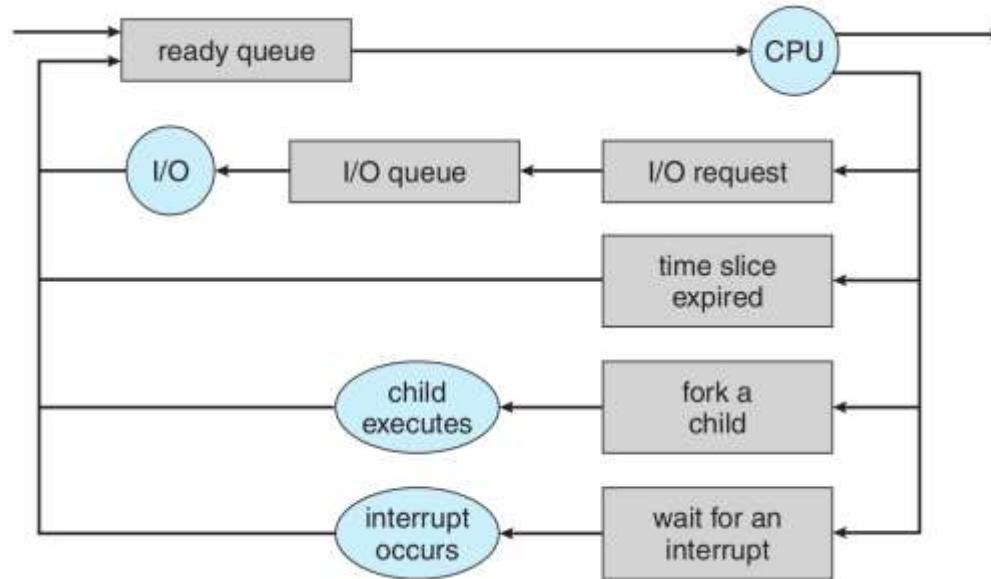
Source: A. Silberschatz, *Operating Systems Concepts Essentials*

# I/O queues





# Process queue



# Process operations - process creation (1)

- Process creation:
  - A given process (parent) can create many new processes (child)
  - Each of the newly created processes can create another one, and processes tree is being created
  - Process identification using PID in the most of the systems is done

```
$ pstree -p | head -n 10
```

```
systemd(1)---ModemManager(824)---{ModemManager}(854)
|      `--{ModemManager}(857)
|      -NetworkManager(711)---{NetworkManager}(819)
|      `--{NetworkManager}(821)
|      -accounts-daemon(10363)---{accounts-daemon}(10366)
|      `--{accounts-daemon}(10372)
|      -acpid(10526)
|      -agetty(876)
```

```
$ ps -ax | head -n 10
```

PID	TTY	STAT	TIME	COMMAND
1	?	Ss	0:01	/sbin/init splash
2	?	S	0:00	[kthreadd]
3	?	Ik	0:00	[rcu_gp]
4	?	Ik	0:00	[rcu_par_gp]
6	?	Ik	0:00	[kworker/0:0H-kblockd]
9	?	Ik	0:00	[mm_percpu_wq]
10	?	S	0:00	[ksoftirqd/0]
11	?	I	0:01	[rcu_sched]
12	?	S	0:00	[migration/0]

Question: why does the difference in: systemd vs. init occurs?



# Parent PID determination

```
$ ps -p `pidof <program name>` -o ppid,pid,status,comm
```

or:

```
$ cat /proc/`pidof <program name>`/status | grep PPid
```

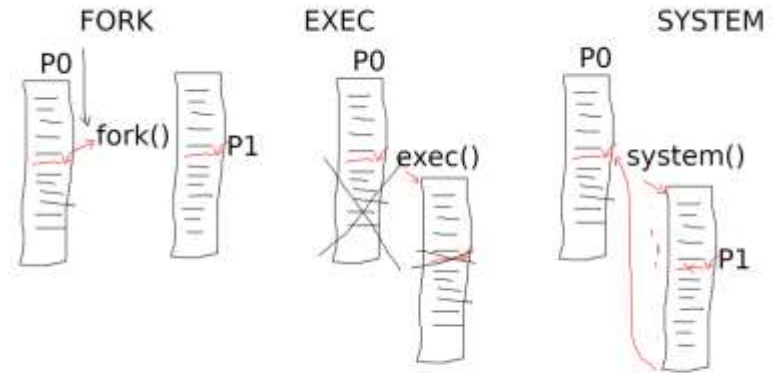
## Process operations - process creation (2)

Two cases may occur during child process creation:

- The parent process continues to execute simultaneously with the child process (fork)
- The parent process waits until any or all of its child processes terminate (fork or system)

There are also two possibilities for addressing memory for a new process:

- The child process is a duplicate of the parent process, i.e. it has the same program code as the parent (fork).
- The child process is the newly loaded program (exec and system).



## Process operations - process creation (3)

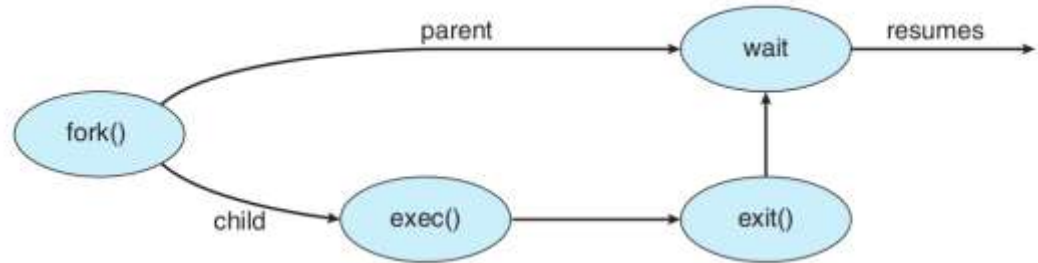
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```





# Running scripts and the child process

Mode without source:

```
$ ./script.sh
```

Mode with source:

```
$ . ./script.sh
```

```
$ source ./script.sh
```



# IPC - Inter-process communication

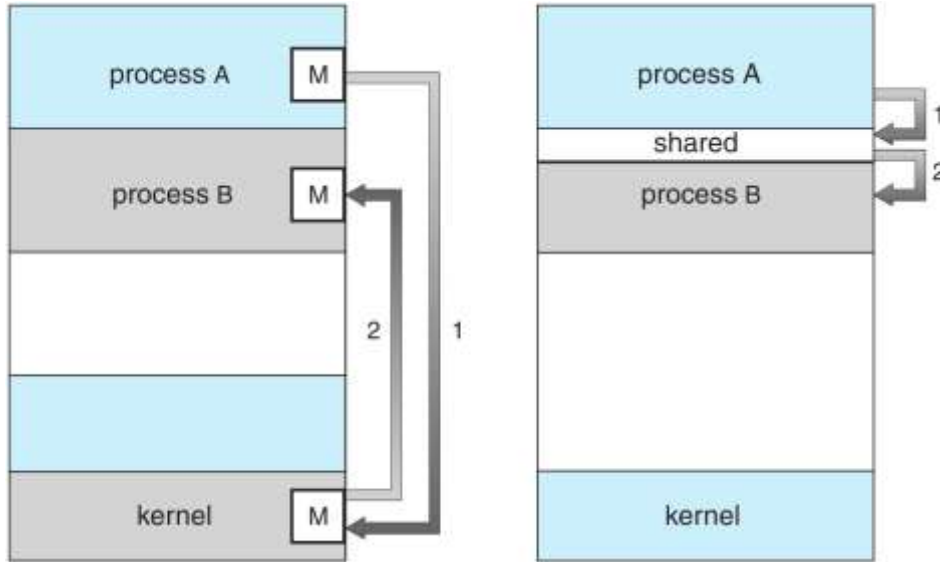
Process coexistence:

- Independent process - other processes do not affect it and it does not affect other processes.
- Cooperative process - can influence other processes or other processes can influence it.

The importance of inter-process communication:

- Information sharing (data, message exchange)
- Speeding up calculations (can they all be parallelized?)
- System modularity
- Convenience

# Inter-process communication models (1)



Message passing  
Shared memory





## Inter-process communication models (2)

- Message passing:
  - Useful for exchanging small amounts of data (no need to prevent conflicts)
  - Easier to implement
- Shared memory:
  - The fastest possible communication
  - The only intervention required from the kernel is to create this memory



# Producent and consument

- Producent - consument model
  - WWW server and HTML browser
  - TEX compiler and PDF viewer
  - etc.



# Shared memory - buffering (1)

Data exchange buffers:

- Unbounded buffer - no size limit:
  - The producer never waits, the consumer waits when the buffer is empty
- Bounded buffer - specific buffer size:
  - The producer waits when the buffer is full, the consumer waits when the buffer is empty

## Shared memory - buffering (2)

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- FIFO queue
- in - the next free cell in buffer
- out - the first free cell in buffer
- $in == out$  - the buffer is empty
- $((in+1) \% BUFFER\_SIZE) == out$  - the buffer is full





## Shared memory - buffering (3)

Producent

Consument

```
item nextProduced;

while (true) {
    /* produce an item in nextProduced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
item nextConsumed;

while (true) {
    while (in == out)
        ; // do nothing

    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* consume the item in nextConsumed */
}
```



# Shared memory

- Based on the example, write a chat program in which two processes exchange messages (numbers or text).

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the identifier for the shared memory segment */
    int segment_id;
    /* a pointer to the shared memory segment */
    char *shared_memory;
    /* the size (in bytes) of the shared memory segment */
    const int size = 4096;

    /* allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* write a message to the shared memory segment */
    sprintf(shared_memory, "Hi there!");

    /* now print out the string from shared memory */
    printf("%s\n", shared_memory);

    /* now detach the shared memory segment */
    shmdt(shared_memory);

    /* now remove the shared memory segment */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}
```

Source: A. Silberschatz, *Operating Systems Concepts Essentials*



# Synchronization

- Message passing can be blocking or non-blocking:
  - Blocking sending
  - Non-blocking sending
  - Blocking receiving
  - Non-blocking receiving



# Buffering

- Zero capacity - the queue has a length of zero
- Limited length (bounded capacity) - the queue has a fixed length
- Unbounded capacity - the queue has unlimited length





**Turbacz**



A scenic view of a rocky coastline. In the foreground, there are large, light-colored rocks. The water is clear and turquoise, revealing a rocky seabed with green algae. A rope hangs vertically from a cliff on the right side of the frame. The background shows more of the rocky cliff and the sea.

# Systemy Operacyjne

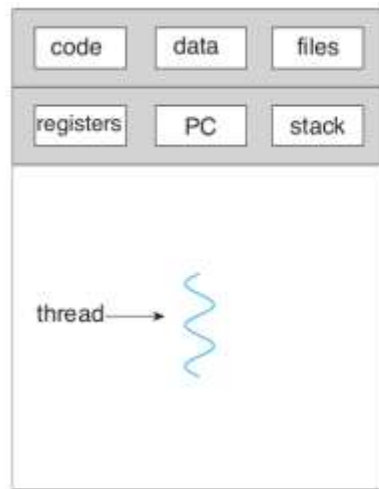
Wątki

Dr hab. inż. Krzysztof Rzecki, prof. AGH

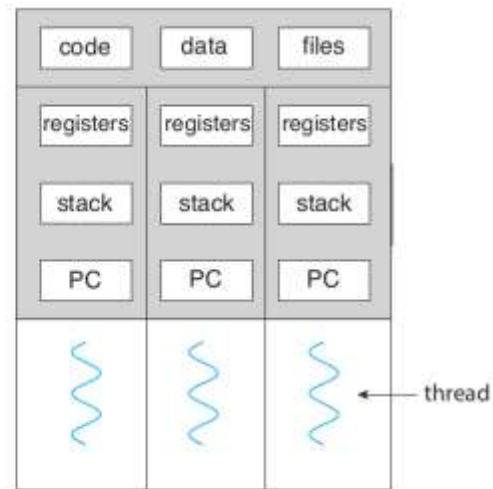
Na podstawie: Abraham Silberschatz, *Koncepcje systemów operacyjnych*

# Wątek

- Wątek to podstawowa jednostka wykorzystania procesora.
- Wątek zawiera: identyfikator (numer), licznik programu, rejestry oraz stos.
- Wątek współdzieli z innymi wątkami należącymi do tego samego procesu: sekcję kodu, sekcję danych oraz inne zasoby systemu operacyjnego (np. otwarte pliki, sygnały).



single-threaded process



multithreaded process

Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*



# Proces vs. Wątek

## Proces

- Proces osadzony jest w dwóch charakterystykach:
  - Właściciela zasobów, w tym przestrzeni adresowej zawierającej obraz procesu.
  - Planowania i wykonywania, która przeplata się razem z innymi procesami.
- Te dwie wyżej wymienione charakterystyki są niezależnie traktowane w systemie operacyjnym.

## Wątek

- Jednostką wykonywanego zadania jest *wątek*, inaczej lekki proces (ang. *lightweight process*).
- Właściciel określony jest przez proces (zadanie), do którego wątek należy.

# Proces

## Wątek

- Proces to wykonywanie programu.
- *Heavy weight process.*
- Czaso- i zasobochłonne: tworzenie, terminacja, przełączanie kontekstu.
- Komunikacja: pamięć dzielona lub wymiana komunikatów jako mechanizmy specjalne.
- Procesy są izolowane.
- Przełączanie procesów odbywa się przez funkcje systemowe.
- Dla jądra dwa procesy to dwa procesy.
- Zablokowanie jednego procesu nie wpływa na fakt zablokowania innego procesu.

## VS.

- Wątek to część danego procesu.
- *Lightweight process.*
- Szybsze i zużywające mniej zasobów na tworzenie, terminację i przełączanie kontekstu.
- Komunikacja: bezpośrednio współdzielone wszystkie zasoby danego procesu.
- Wątki współdzielą.
- Przełączanie wątków odbywa się bez wywoływania przerw do jądra.
- Dla jądra dwa wątki to jeden proces.
- Zablokowanie procesu, to zablokowanie jego wszystkich wątków.



# Proces

## Wątek

- Zablockowanie się procesu macierzystego uniemożliwia tworzenie procesów potomnych.
- Proces ma własny PCB, stos oraz przestrzeń adresową.
- Zmiany w procesie macierzystym nie mają wpływu na procesy potomne.

## vs.

- Zablockowanie pierwszego wątku nie wpływa na działanie pozostałych wątków procesu.
- Ma rodzica PCB, własny TCB, stos oraz współdzieloną przestrzeń adresową.
- Zmiany w procesie wpływają na zmiany w wątkach tego procesu.

# Obserwowanie wątków w systemie Linux

```
$ ps -T -o pid,tid,comm -p `pidof insync`
```

PID	TID	COMMAND
4224	4224	insync
4224	4260	QXcbEventQueue
4224	4280	QDBusConnection
4224	4281	gmain
4224	4282	gdbus
4224	4284	insync
4224	4312	insync



Proces



Wątki

Nazwa wątku, nazwa procesu

```
$ pstree -p `pidof insync`
```

```
insync(4224) --QtWebEngineProc(4336) ---QtWebEngineProc(4338) ---QtWebEngineProc(4357) --{QtWebEngineProc}(4358)
```



Proces

```
|
```

```
|
```

```
...
```

```
|-{insync}(4260)
```

```
|-{insync}(4280)
```

```
|-{insync}(4281)
```



Wątki



Procesy potomne



## PID vs. TID

- PID = *process identifier*
- TID = *thread identifier*
- Jeśli proces ma tylko jeden wątek, to PID == TID
- Jeśli proces ma wiele wątków, to pierwszy z nich ma unikalny TID w zakresie tego procesu
- Jądro systemu nie rozróżnia szczególnie wątku od procesu
- Dla jądra wątki to procesy, które współdzielą pewne zasoby
- Kiedy tworzymy nowy proces za pomocą `fork()`, to otrzymuje on nowy PID i TID (PID==TID)
- Kiedy tworzymy nowy wątek to otrzymuje on PID taki jak procesu oraz nowy TID.
- Alias: LWP = *Light-Weight Process*





# Obserwowanie wątków kernela w Linux

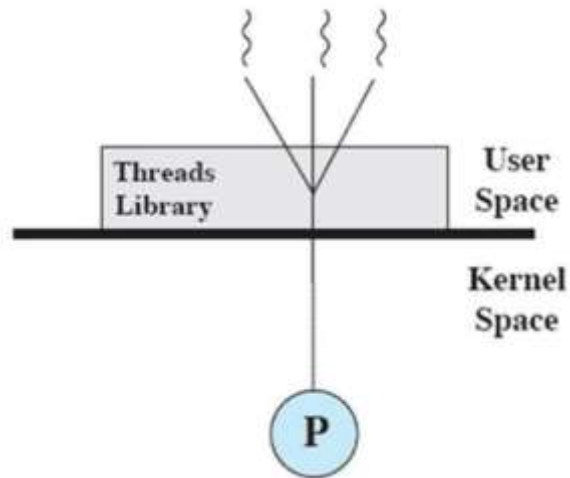
```
$ pstree -p 2
kthreadd(2) +-UVM Tools Event(977)
              |-UVM deferred re(976)
              |-UVM global queue(975)
              |-acpi_thermal_pm(134)
              |-ata_sff(117)
              |-blkcg_punt_bio(115)
              |-charger_manager(164)
              |-cpuhp/0(14)
```

Dlaczego mają odmienny PID ?

Dlatego, że to jest TID.

# Wątki na poziomie użytkownika

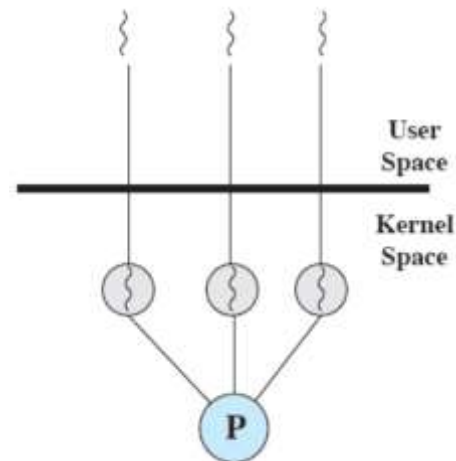
- Zarządzanie wątkami odbywa się na poziomie aplikacji.
- Jądro nie ma wiedzy na temat wątków.



Źródło: TODO

# Wątki na poziomie jądra

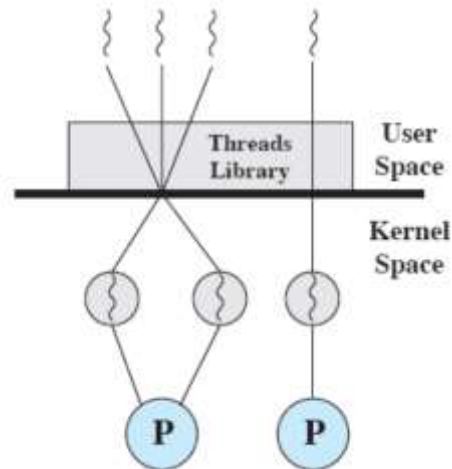
- Jądro zarządza kontekstem dla procesu oraz wątków. Nie ma zarządzania wątkami na poziomie aplikacji.
- Zalety:
  - Kernel może jednocześnie planować realizację wielu wątków z jednego procesu na wielu procesorach/rdzeniach.
  - Jeśli jeden wątek w procesie jest zablokowany, jądro może planować inny wątek tego samego procesu.
  - Funkcjonalność jądra może być wielowątkowa.
- Wada: przekazanie kontroli między wątkami w obrębie procesu wymaga kernel-mode.



Źródło: TODO

# Rozwiązanie łączone

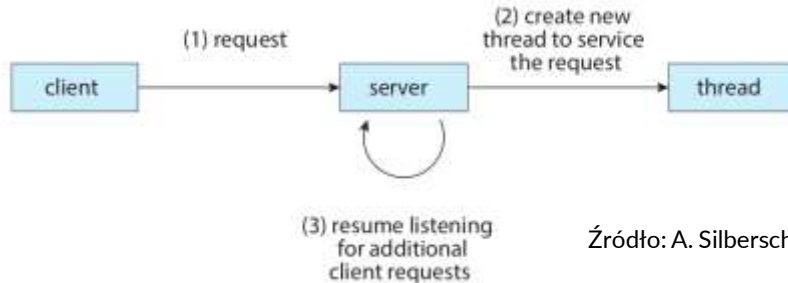
- Tworzenie wątku odbywa się w przestrzeni użytkownika.
- Planowanie (*scheduling*) oraz synchronizacja wątków odbywa się w jądrze



Źródło: TODO

# Wątki - zastosowania

- Program serwera obsługujący żądania (ang. *requests*) programów klienckich:
  - Serwer stron WWW i przeglądarka internetowa.
  - Serwer poczty elektronicznej (ang. *mail transfer agent*) i program pocztowy.
  - Sprawdzanie pisowni w edytorze tekstu.
- Prowadzenie obliczeń macierzowych:
  - Wykonywanie operacji na tych samych danych.

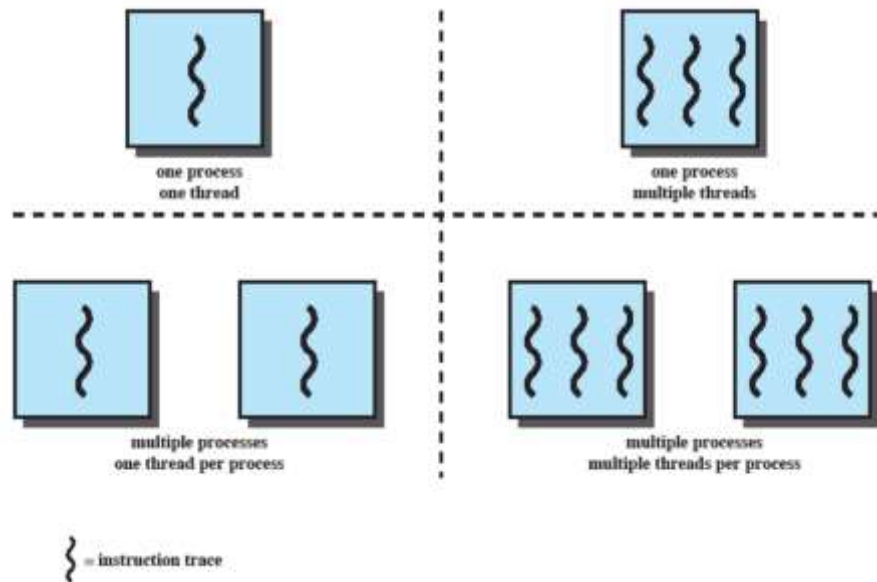


Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

**Uwaga!** Tworzenie wątku jest mniej obciążające niż tworzenie nowego procesu.

# Wielowątkowość

- Zdolność systemu operacyjnego do wspierania wielu ścieżek wykonywania w obrębie jednego procesu.
- MS-DOS - single user process with single thread.
- Some UNIX - multiple user processes with single thread per process.
- Java run-time env. - single process with multiple threads.
- Windows, Solaris, modern UNIX, Linux, etc. - multiple processes with multiple threads per process.



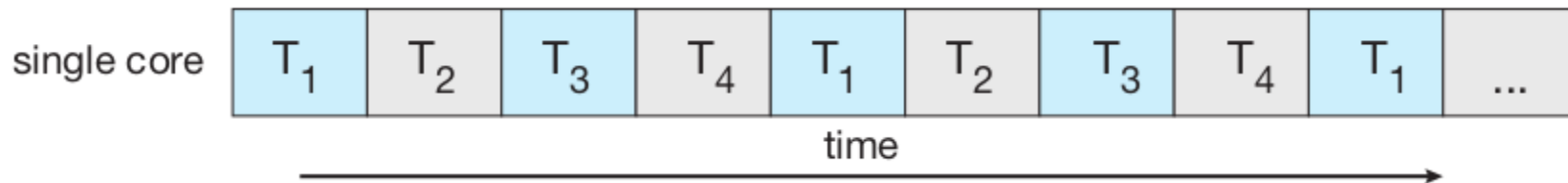
Źródło: TODO



# Zalety oprogramowania wielowątkowego

- **Responsywność** - jeśli część aplikacji jest zablokowana, inna jej część może wykonywać operacje, a cała aplikacja sprawia wrażenie ciągłego działania. Zastosowanie: w aplikacji, kiedy jedna wywołana operacja wykonywana jest w tle, interfejs użytkownika pozostaje responsywny.
- **Współdzielenie zasobów** - w przypadku procesów współdzielenie zasobów odbywa się tylko poprzez pamięć współdzieloną, albo przesyłanie komunikatów. Wątki współdzielą zasoby wprost. Współdzielenie kodu i danych umożliwia wątkom działać w tej samej przestrzeni adresowej.
- **Ekonomia** - alokowanie pamięci i zasobów przy tworzeniu procesu jest bardziej kosztowne, niż w przypadku wątków. Przełączanie kontekstu jest także szybsze w przypadku wątków.
- **Skalowalność** - aplikacje wielowątkowe mogą działać w architekturze wielordzeniowej. Aplikacja jednowątkowa może być wykonana tylko na jednym rdzeniu procesora.

# Współbieżność i równoległość

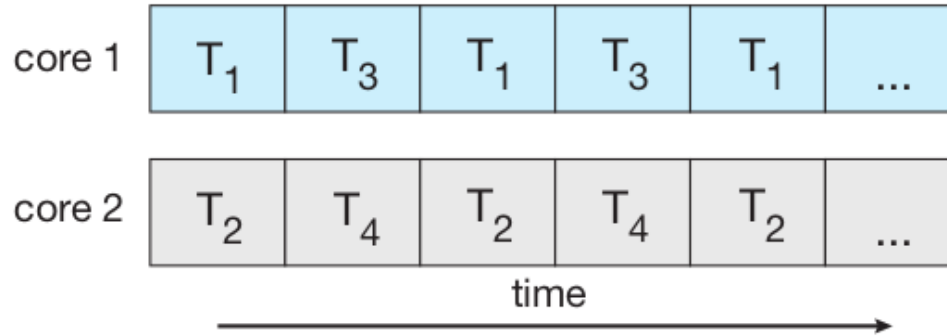


Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

**Współbieżność** (ang. *concurrency*) - umożliwia więcej niż jednemu zadaniu być wykonywanym. Do realizacji współbieżności nie jest wymagany system wielordzeniowy.



# Współbieżność i równoległość



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

**Równoległość** (ang. *parallelism*) - umożliwia więcej niż jednemu zadaniu być wykonywanym **JEDNOCZEŚNIE**. Do realizacji równoległości jest wymagany system wielordzeniowy.

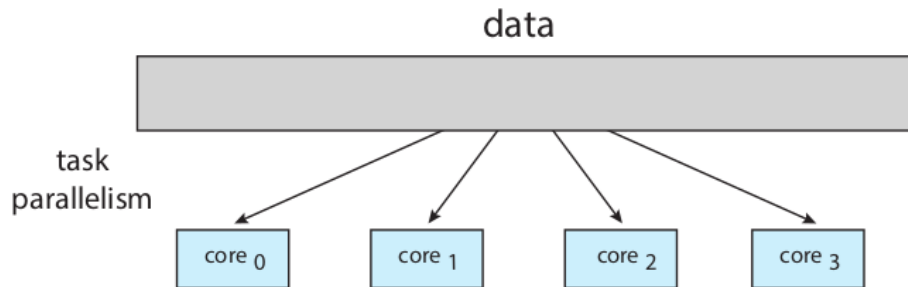
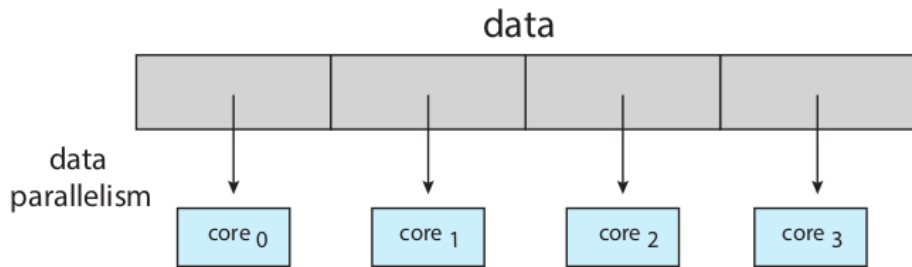
Pytanie: czy może zaistnieć współbieżność bez równoległości ?



# Wyzwania programowe

- Identyfikacja zadań, w szczególności na zadania niezależne między sobą.
- Balansowanie zadaniami celem zrównoważenia obciążenia.
- Dzielenie danych między wydzielone zadania.
- Zależność danych występująca w szczególności przy następstwie obliczeń.
- Testowanie i debugowanie są zdecydowanie trudniejsze niż w programach jednowątkowych.

# Typy równoległości



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

**Dane** dzielone są między procesami/rdzeniami wykonującymi **tego samego typu** operacje.

Przykład: sumowanie zakresów komórek.

**Zadania** dzielone są między procesami/rdzeniami, a każdy wątek realizuje **unikalną** operację.

Przykład: jednoczesne wyznaczanie min i max.

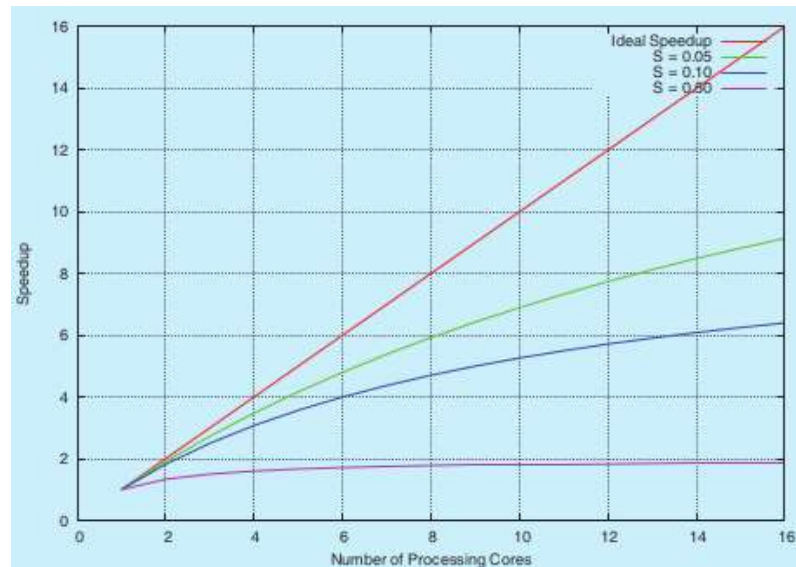
# Prawo Amdahl'a

Wyraża potencjalny wzrost wydajności obliczeń przez dodanie kolejnych rdzeni obliczeniowych do obsługi aplikacji, która ma dwa komponenty: podlegający i niepodlegający zrównolegleniu.

S - procentowy udział niepodlegającego zrównolegleniu kodu.

N - liczba rdzeni przypisanych do zadania.

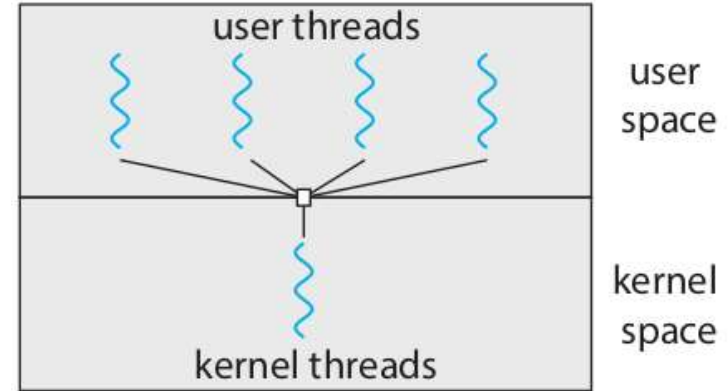
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

# Model wielowątkowy - Many-to-One

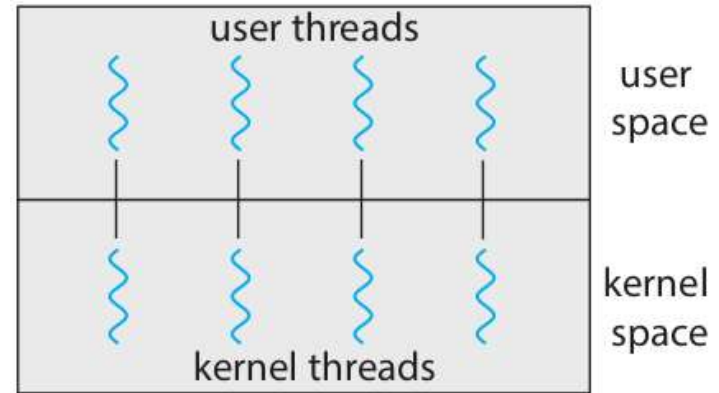
- Zarządzanie wątkami wykonywane jest w przestrzeni użytkownika (przez bibliotekę).
- Zaleta: wydajność.
- Wada 1: zablokowanie całego procesu, jeśli któryś wątek wykona blokujące wywołanie systemowe.
- Wada 2: wątki nie zostaną uruchomione równolegle w systemie wielordzeniowym.
- To tzw. zielone wątki (ang. *green threads*) - można je uruchomić w środowisku nie wspierającym wielowątkowości.
- Przykłady: Solaris, wczesne wersje Java.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

# Model wielowątkowy - One-to-One

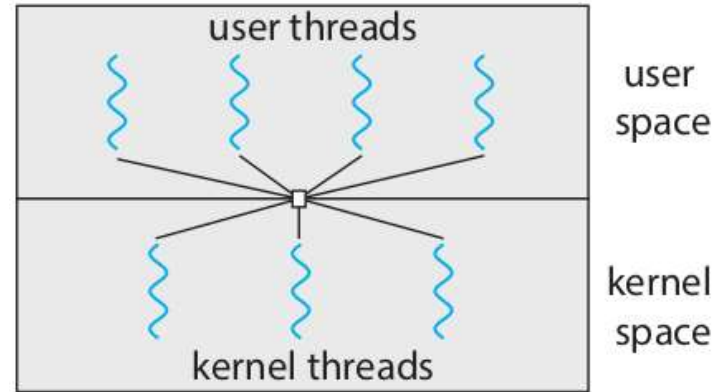
- Model wprowadza niezależną współbieżność, tzn. dany wątek może być realizowany także wtedy, gdy inny wywoła blokującą funkcję systemową.
- Model wprowadza także równoległość.
- Wada: każdy wątek użytkownika tworzy wątek w jądrze, a duża ich liczba może obniżać wydajność systemu.
- Przykłady: Linux, Windows.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

## Model wielowątkowy - Many-to-Many

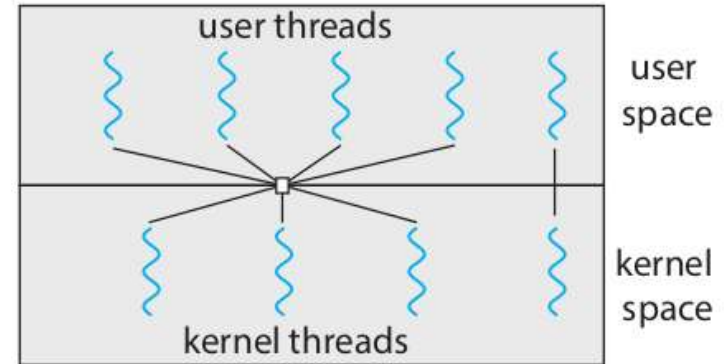
- Model multiplexuje wiele wątków w przestrzeni użytkownika z równą lub mniejszą liczbą wątków w przestrzeni jądra.
- Liczba wątków w przestrzeni jądra może być specyficzna względem aplikacji lub sprzętu.
- Model ten jest pozbawiony wad modeli Many-to-One i One-to-One.
- Trudny w implementacji.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

# Model wielowątkowy - Two-level

Jak na rysunku obok.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*





# Biblioteki programistyczne dla wątków

- Biblioteki dostarczają API (ang. *Application Programming Interface*) do tworzenia i zarządzania wątkami.
- Podejście 1: Wątki w przestrzeni użytkownika, bez wsparcia ze strony jądra. To oznacza, że wynikiem wywołania funkcji bibliotecznej jest wywołanie funkcji lokalnej, nie systemowej.
- Podejście 2: Wątki w przestrzeni jądra ze wsparciem systemu operacyjnego. To oznacza, że kod i dane biblioteki istnieją w przestrzeni jądra, a wywołanie funkcji jest wywołaniem systemowym.



## Biblioteki programistyczne dla wątków (2)

- POSIX Pthreads - przestrzeń użytkownika lub przestrzeń jądra.
  - Windows thread library - przestrzeń jądra.
  - Java thread API - bezpośrednio w programach Java, tak jak dana implementacja JVM zależna od hostującego systemu operacyjnego.
- 
- POSIX i Windows - dane zadeklarowane globalnie są współdzielone między wątkami procesu.
  - Java - nie ma danych globalnych, dostęp do współdzielonych danych musi zostać nadany.



# Strategie tworzenia wątków

**Strategia asynchroniczna** - wątek tworzy wątek potomny i następnie kontynuuje swoje działanie. Oba wątki działają współbieżnie i niezależnie.

Zastosowanie:

- Serwery wielowątkowe.
- Responsywny interfejs użytkownika.

**Strategia synchroniczna** - wątek tworzy wątki potomne i przechodzi w stan oczekiwania na zakończenie wykonywania ich zadań. O ile wątki potomne działają współbieżnie, to wątek macierzysty po prostu czeka.

Zastosowanie:

- Obliczenia z przestaniem zadań częściowych i oczekiwaniem na wyniki.



# Pula wątków

Powody:

- Tworzenie wątków zajmuje pewien czas (mniejszy niż procesów potomnych), a może mogą być wykorzystane ponownie.
- Brak kontroli liczby powstających wątków może doprowadzić do przeciążenia zasobów (procesor, pamięć) systemu.

Rozwiązanie: **pula wątków** (ang. *thread pool*)

- Utworzenie zadanej liczby wątków.
- Umieszczanie wątków w puli wątków.
- Wątki oczekują na przydzielenie zadania.
- Serwer otrzymuje żądanie.
- Serwer przekazuje żądanie do puli wątków.
- Jeśli w puli jest wolny wątek, przejmuje on żądanie i zajmuje się jego obsługą.
- Jeśli brak jest wolnych wątków w puli, zadanie jest kolejkowane.
- Po zakończeniu obsługi danego żądania wątek wraca do puli i oczekuje na nowe.
- Pula wątków najlepiej działa, gdy zadania obsługiwane są asynchronicznie.



# Rozmiar puli wątków

Rozmiar puli wątków może być zależny od:

- Liczby rdzeni procesora.
- Ilości fizycznej pamięci RAM.
- Może być też dynamicznie zmieniany w zależności od aktualnie działających wątków (obserwując ich obciążenie).



# Wywołania systemowe: `fork()` oraz `exec()`

**Problem:** czy po wywołaniu przez wątek funkcji systemowej `fork()` proces potomny duplikuje wszystkie wątki, czy nowy proces jest jedno-wątkowy ?

**Odpowiedź:** sprawdzić i odpowiedź przedstawić na forum UPEL.

**Działanie:** wywołanie `exec()` spowoduje zastąpienie całego procesu i wszystkich wątków.

**Przypadek I:** jeśli `exec()` wywołany jest zaraz po `fork()`, wówczas duplikowanie wszystkich wątków nie jest potrzebne, bo i tak proces zostanie zastąpiony w funkcji `exec()`.

**Przypadek II:** jeśli `exec()` wywołany jest później, dany `fork()` powinien zduplikować wszystkie wątki.



# Obsługa sygnałów

Procedura obsługi sygnałów:

- Sygnał jest generowany przez zdarzenie.
- Sygnał jest dostarczany do procesu.
- Proces musi obsłużyć sygnał.

**Sygnał synchroniczny:** dzielenie przez 0, nielegalny dostęp do pamięci.

**Sygnał asynchroniczny:** wciśnięcie np. [ ctrl+c ].

**Zagadka:** który wątek otrzyma sygnał ?

Obsługa sygnału:

- **domyślna obsługa sygnału** - jeśli brak zdefiniowanej obsługi sygnału, zajmuje się nią jądro systemu operacyjnego (sygnał może być zignorowany lub zakończyć działanie programu),
- **zdefiniowana przez użytkownika obsługa sygnału.**



# Sygnał a program wielowątkowy

## Gdzie dostarczyć sygnał ?

- Do wątku, do którego sygnał pasuje.
- Do każdego wątku w procesie.
- Do wybranych wątków w procesie.
- Wybrać jeden wątek do przechwytywania wszystkich sygnałów danego procesu.

## Wysyłanie sygnału do procesu:

```
kill(pid_t pid, int signal)
```

Przechwyci go pierwszy nieblokujący wątek.

## Wysyłanie sygnału do wybranego wątku:

```
pthread_kill(pthread_t tid, int signal)
```





**Dubrownik /**





# Systemy Operacyjne

**Komunikacja międzyprocesowa**

Dr hab. inż. Krzysztof Rzecki, prof. AGH

Na podstawie: Abraham Silberschatz, *Koncepcje systemów operacyjnych*

Na podstawie: Richard Stevens, *Unix Network Programming - IPC*



# Podstawy komunikacji międzyprocesowej

Procesy uruchomione jednocześnie mogą być:

- Niezależne (ang. *independent*) - nie współdzielą danych z żadnym innym wykonywanym procesem w systemie operacyjnym.
- Współpracujące (ang. *cooperating*) - może wpływać lub można na niego wpływać przez inne procesy wykonywane w systemie.

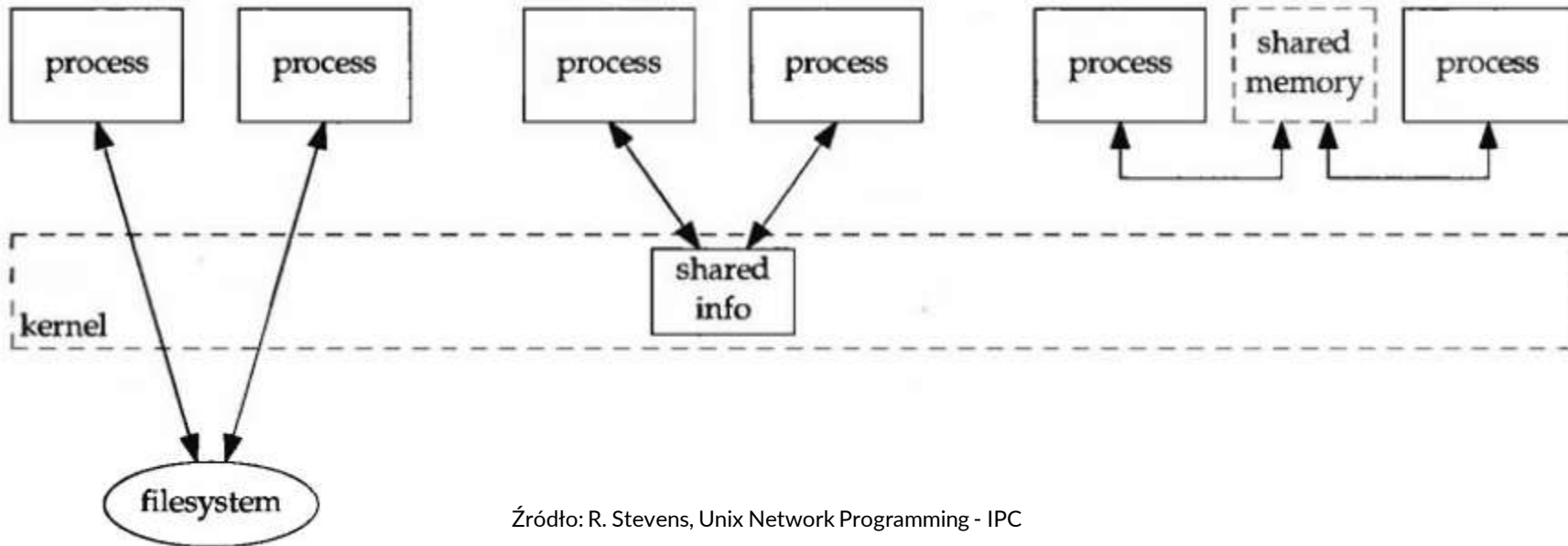
Czy zatem proces odczytujący dane z dysku jest procesem współpracującym, czy niezależnym ?



# Zastosowania komunikacji międzyprocesowej

- Współdzielenie informacji
- Przyspieszenie obliczeń
- Modularność oprogramowania

# Trzy metody dzielenia informacji



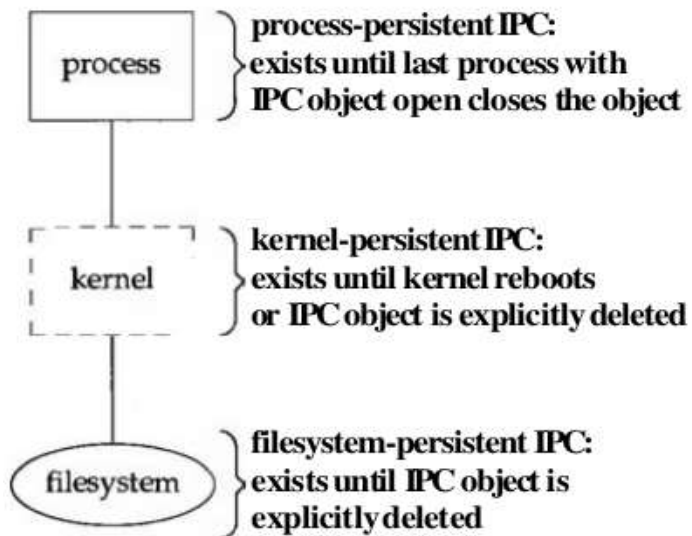
Źródło: R. Stevens, Unix Network Programming - IPC



# Implementacje IPC

- Wymiana przez pliki
- Pamięć dzielona
- Sygnały
- Potoki nazwane lub nienazwane
- Semafor
- Kolejki
- Gniazda dziedziny UNIX
- Gniazda udp/tcp
- RPC

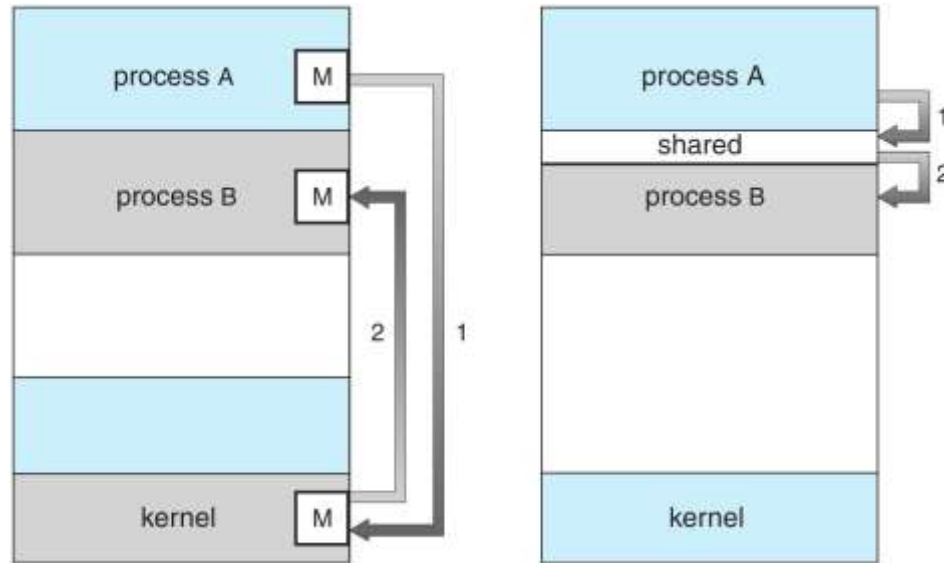
# Trwałość obiektów IPC



Type of IPC	Persistence
Pipe FIFO	process process
Pcsix mutex Posix condition variable Posix read-write lock fcntl record locking	process process process process
Posix message queue Pcsix named semaphore Pcsix memory-based semaphore Posix shared memory	kernel kernel process kernel
System V message queue System V semaphore System V shared memory	kernel kernel kernel
TCP socket UDP socket Unix domain socket	process process process

Źródło: R. Stevens, Unix Network Programming - IPC

# Modele komunikacji międzyprocesowej (było!)

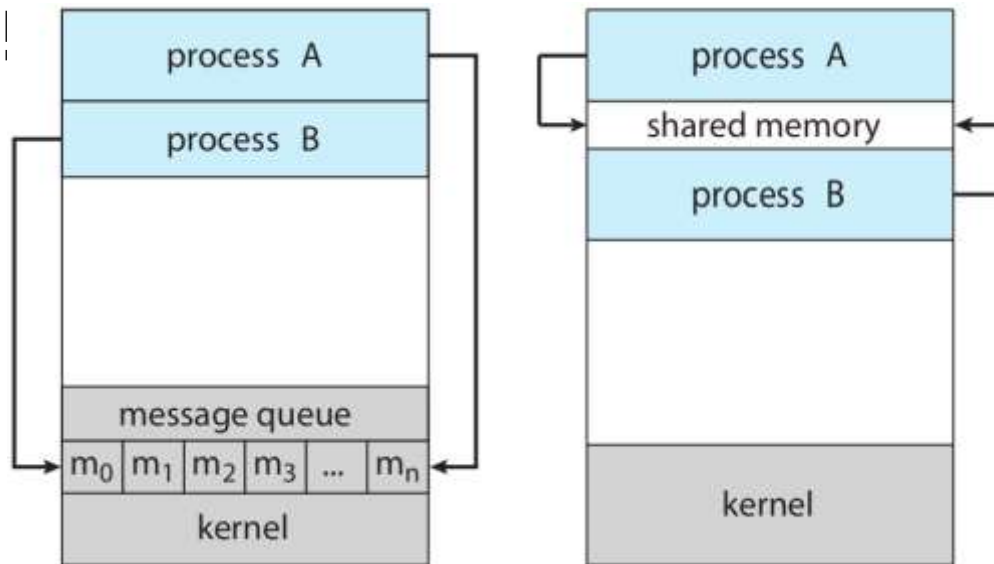


Message passing  
Shared memory



# Modele komunikacji międzyprocesowej

(nowe)



Message passing  
Shared memory



# Message-Passing Systems

Message-Passing Systems (systemy przekazywania wiadomości):

- Komunikujące się procesy mogą rezydować na różnych stacjach
- Komunikujące się procesy mogą rezydować na różnych typach i wersjach systemów operacyjnych
- Infrastruktura systemu przekazywania wiadomości obejmuje co najmniej dwie operacje:
  - `send(message)`
  - `receive(message)`
- Ze względu na wielkość wiadomości rozróżniamy:
  - System bezpośredni, czyli o stałej długości komunikatów (*straight-forward*) - prosta implementacja w systemie operacyjnym, skomplikowane użytkowanie ze względu na fragmentację,
  - System o zmiennej długości komunikatów (*variable-sized*) - skomplikowana implementacja w systemie operacyjnym, proste użytkowanie.



# Communication link

- *Communication link* - logiczne powiązanie między procesami zestawiające kanał komunikacji między nimi. Nie interesuje nas zatem, czy to jest shared memory, szyna sprzętowa, czy sieć.
- Implementacje tego powiązania obejmują zagadnienia:
  - Komunikacji pośredniej i bezpośredniej.
  - Komunikacji synchronicznej i asynchronicznej.
  - Buforowanie automatyczne lub na żądanie.



# Komunikacja bezpośrednia

- **Komunikacja bezpośrednia** (ang. *direct communication*) - każdy proces, który uczestniczy w komunikacji musi bezpośrednio wskazać nadawcę, czy odbiorcę:
  - `send(P, message)` - wyślij wiadomość do procesu P.
  - `receive(Q, message)` - odbierz wiadomość od procesu Q.
- **Własności:**
  - Link jest zestawiany automatycznie, a procesy muszą tylko znać swoją nazwę.
  - Link jest zestawiany dokładnie między dwoma procesami.
  - Między każdą parą komunikujących się procesów zestawiany jest dokładnie jeden link.
  - Występuje symetria w adresacji (P, Q), choć spotykane są warianty asymetryczne (np. P i id).



# Komunikacja pośrednia

- **Komunikacja pośrednia** (ang. *indirect communication*) - procesy komunikują się przez skrzynki (ang. *mailbox*) lub porty identyfikowane np. przez liczby całkowite:
  - `send(A, message)` - wyślij wiadomość do skrzynki A.
  - `receive(A, message)` - odbierz wiadomość ze skrzynki A.
- **Własności:**
  - Link jest zestawiany między parą współdzielących skrzynkę procesów.
  - Link jest może zostać zestawiony przez większą liczbę procesów.
  - Między każdą parą komunikujących się procesów mogą istnieć różne linki komunikacyjne.
- **System operacyjny musi obsłużyć następujące mechanizmy:**
  - Utworzenie skrzynki.
  - Wyśłanie i odebranie wiadomości do i ze skrzynki.
  - Usunięcie skrzynki.

(Rysunek na tablicy!)



# Synchronizacja

- Komunikacja synchroniczna, blokująca:
  - Blokujące wysyłanie - proces wysyłający jest zablokowany na wysłaniu, aż wysłana wiadomość zostanie odebrana przez proces odbierający lub skrzynkę.
  - Blokujący odbiór - odbiorca zostaje zablokowany do czasu otrzymania wiadomości.
- Komunikacja asynchroniczna, nieblokująca
  - Nieblokujące wysyłanie - proces wysyłający wysyła wiadomość i nie jest blokowany na metodzie wysyłającej.
  - Nieblokujący odbiór - odbiorca otrzymuje gotową wiadomość lub informację o braku jej dostępności.

W przypadku blokującego nadawcy i odbiorcy mamy do czynienia z ang. *Rendezvous*.

(Rysunek na tablicy!)



# Buforowanie

- Pojemność zerowa (ang. *zero capacity*) - maksymalna długość kolejki wynosi zero, czyli link komunikacyjny nie może mieć żadnej wiadomości oczekującej w sobie, co oznacza, że nadawca musi zostać zablokowany do czasu odbioru wiadomości przez odbiorcę.
- Ograniczona pojemność (ang. *bounded capacity*) - kolejka ma określoną, skończoną długość  $n$ , czyli co najwyżej  $n$  wiadomości może zostać umieszczonych w kolejce. Jeśli kolejka nie jest pełna, można dołożyć wiadomość (nadawca zostaje zablokowany), jeśli kolejka jest pusta, nie można pobrać żadnej wiadomości (odbiorca zostaje zablokowany).
- Nieograniczona pojemność (ang. *unbounded capacity*) - długość kolejki jest potencjalnie nieskończona (nadawca nigdy nie jest blokowany).

(Rysunek na tablicy!)



# POSIX Shared Memory

- Tworzenie dowiązania do wspólnej przestrzeni w pamięci:
  - `int fd = shm_open(name, O_CREAT | O_RDWR, 0666);` // pisanie i czytanie
  - `int fd = shm_open(name, O_RDONLY, 0666);` // tylko czytanie
- Ustawianie wielkości pamięci dzielonej:
  - `ftruncate(fd, 4096);`
- Utworzenie miejsca w pamięci:
  - `ptr = (char *) mmap (0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);`
- Pisanie do pamięci dzielonej:
  - `sprintf(ptr, "%s", message);` // umieszczenie komunikatu w miejscu 'ptr'
  - `ptr += strlen(message);` // przesunięcie wskazania w pamięci
- Czytanie z pamięci dzielonej:
  - `printf("%s", (char *)ptr);`
- Usunięcie obiektu współdzielonego:
  - `shm_unlink(name);`





## POSIX Shared Memory

Description	mq_open	sem_open	shm_open
read-only	O_RDONLY		O_RDONLY
write-only	O_WRONLY		
read-write	O_RDWR		O_RDWR
create if it does not already exist	O_CREAT	O_CREAT	O_CREAT
exclusive create	O_EXCL	O_EXCL	O_EXCL
nonblocking mode	O_NONBLOCK		
truncate if it <b>already</b> exists			O_TRUNC



## System V IPC

	Message queues	Semaphores	Shared memory
Header	<code>&lt;sys/msg.h&gt;</code>	<code>&lt;sys/sem.h&gt;</code>	<code>&lt;sys/shm.h&gt;</code>
Function to create or open	<code>msgget</code>	<code>semget</code>	<code>shmget</code>
Function for control operations	<code>msgctl</code>	<code>semctl</code>	<code>shmctl</code>
Functions for IPC operations	<code>msgsnd</code> <code>msgrcv</code>	<code>semop</code>	<code>shmat</code> <code>shmdt</code>



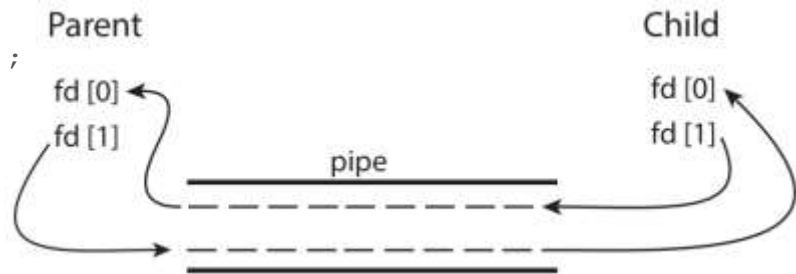
# Potok (ang. *pipe*)

- Jedne z pierwszych metod IPC, jedna z najprostszych form komunikacji.
- Przy projektowaniu potoków należy uwzględnić cztery kwestie:
  - Czy potok pozwala na dwukierunkową (ang. *bidirectional*), czy jest to jednokierunkowa (ang. *unidirectional*) komunikacja ?
  - Jeśli możliwa jest komunikacja dwukierunkowa, to czy jest ona half duplex (dane przesyłane są tylko w jednym kierunku w danym momencie), czy full duplex (dane przesyłane są w obu kierunkach w danym momencie) ?
  - Czy jest relacja między komunikującymi się procesami (np. rodzic - dziecko) ?
  - Czy potoki mogą komunikować się poprzez sieć, czy tylko między procesami na tej samej maszynie ?

# Potok zwykły (ang. *ordinary pipe*)

- Utworzenie potoku:
  - `int fd[2];`
  - `pipe(int fd[])`
- Pisanie do potoku (deskryptor `fd[1]`):
  - `write(fd[WRITE END], write msg, strlen(write msg)+1);`
- Czytanie z potoku (deskryptor `fd[0]`):
  - `read(fd[READ END], read msg, BUFFER SIZE);`
- Potok w praktyce:
  - `ls | less`
  - `cat file.txt | wc -l`

( Przedyskutować efekt użycia `fork()` )



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*



# Gniazda i komunikacja sieciowa

Osobny wykład.





# Systemy Operacyjne

**Synchronizacja procesów**

Dr hab. inż. Krzysztof Rzecki, prof. AGH

Na podstawie: Abraham Silberschatz, *Koncepcje systemów operacyjnych*



# Kooperacja procesów

Sposoby kooperacji:

- Bezpośrednie współdzielenie przestrzeni adresacji (zarówno kod, jak i dane)
- Współdzielenie danych przez system plików lub komunikaty

Skutki kooperacji:

- Utrata spójności danych
- Wzajemne blokowanie



Zobacz: wykład pt. "Procesy"

Zobacz: wykład pt. "Komunikacja międzyprocesowa"

## Producent - konsument - pamięć dzielona

- **Producent** to proces produkujący informację, którą konsumuje **konsument**
- Przykład: kompilator - assembler, assembler - loader, klient - serwer, etc.
- Dwa typy buforów:
  - Nieskończony - konsument czeka, gdy bufor jest pusty; producent zawsze może umieszczać dane,
  - Skończony - konsument czeka, gdy bufor jest pusty; producent czeka, gdy bufor jest pełny.
- `in` - następna wolna pozycja w buforze
- `out` - pierwsza pełna pozycja w buforze
- `in == out` - bufor jest pusty
- `((in + 1) % BUFFER_SIZE) == out` - bufor jest pełny

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



# Producent - konsument

```
while (true) {  
    /* produce an item in nextProduced */  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

**Producent**

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in nextConsumed */  
}
```

**Konsument**

# Producent - konsument

```
while (true) {  
    /* produce an item in nextProduced */  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Producent

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in nextConsumed */  
}
```

Konsument



**Warunek wyścigu** (ang. race condition) to sytuacja, w której dwa lub więcej procesów wykonuje operację na zasobach dzielonych (odczyt lub zapis), a ostateczny wynik tej operacji jest zależny kolejności tego dostępu.

# Warunek wyścigu

Niskopoziomowy `count++`

```
register1 = count  
register1 = register1 + 1  
count = register1
```

Niskopoziomowy `count--`

```
register2 = count  
register2 = register2 - 1  
count = register2
```

Niech `count == 5`

$T_0$ :	producer	execute	$register_1 = count$	$\{register_1 = 5\}$
$T_1$ :	producer	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
$T_2$ :	consumer	execute	$register_2 = count$	$\{register_2 = 5\}$
$T_3$ :	consumer	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
$T_4$ :	producer	execute	$count = register_1$	$\{count = 6\}$
$T_5$ :	consumer	execute	$count = register_2$	$\{count = 4\}$



# Sekcja krytyczna

**Sekcja krytyczna** (ang. *critical section*) to segment kodu, w którym proces może zmieniać wartości zmiennych, aktualizować tabele, pisać do pliku, etc. Podstawową własnością sekcji krytycznej jest to, że w tym samym czasie żaden inny proces nie może realizować swojej sekcji krytycznej (obejmującej te same zasoby).

- **Sekcja wejścia** (ang. *entry section*) - segment kodu, w którym zgłaszane jest żądanie dostępu do zasobu celem realizacji wzajemnego wykluczenia.
- **Sekcja krytyczna** (ang. *critical section*)
- **Sekcja wyjścia** (ang. *exit section*) - segment kodu, w którym zgłaszane jest zwolnienie zasobu.
- **Sekcja pozostałego kodu** (ang. *remainder section*) - nie związana z obsługą współdzielenia zasobów część pozostała część kodu.



# Sekcje kodu

W ramce oznaczone zostały sekcje sterujące przebywaniem w sekcji krytycznej.

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```



# Wymagania dot. rozwiązania sekcji krytycznej

Rozwiązanie problemu sekcji krytycznej musi spełniać następujące wymagania:

- **Wzajemne wykluczenie** (ang. *mutual exclusion*) oznacza, że jeśli jeden proces wykonuje swoją sekcję krytyczną, to żaden inny proces nie może wykonać swojej sekcji krytycznej.
- **Postęp** (ang. *progress*) oznacza, że jeśli żaden proces nie jest w sekcji krytycznej i jakiś proces chciałby wejść do swojej sekcji krytycznej, to tylko procesy nierealizujące swojej sekcji kodu pozostałego mogą brać udział w decydowaniu, który z nich wejdzie do swojej sekcji krytycznej.
- **Skończony czas oczekiwania** (ang. *bounded waiting*) oznacza, że czas oczekiwania na wejście do sekcji krytycznej dla każdego procesu powinien być ograniczony.



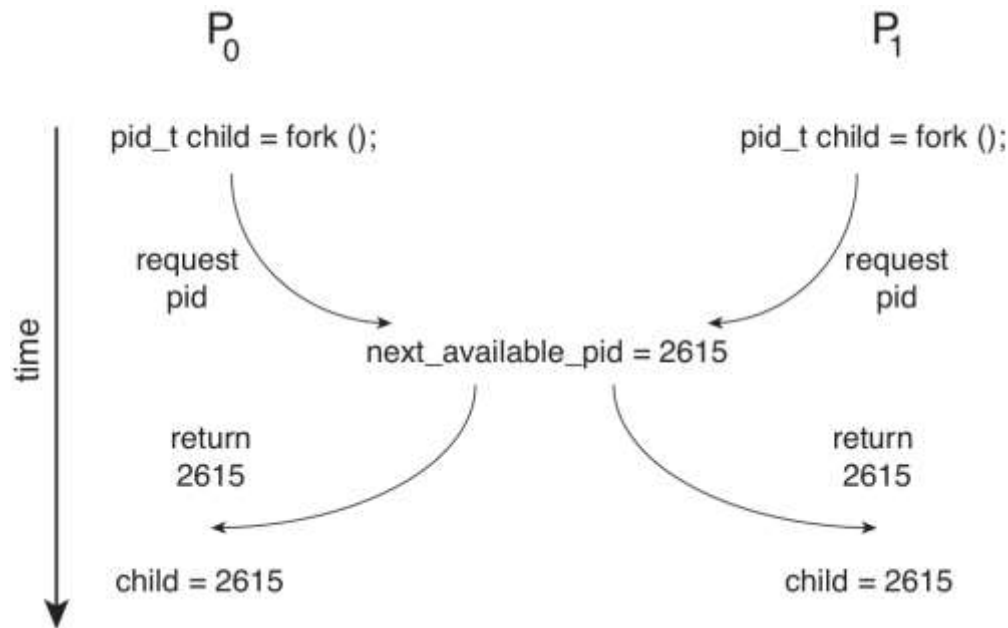
# Zakleszczenie, ang. *deadlock*

Zakleszczeniem jest sytuacja, kiedy dwa procesy wzajemnie czekają na zwolnienie zasobów.

$P_0$	$P_1$
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>



## Sekcja krytyczna - przydzielanie PID po `fork()`






# Wywłaszczenie jądra (ang. *Kernel preemption*)

**Wywłaszczenie** - technika, w której planista (algorytm szeregujący zadania, ang. *dispatcher*) może wstrzymać aktualnie wykonywane zadanie, aby umożliwić wykonywanie innemu zadaniu. Zawieszenie (np. zapętlenie) zadania nie powoduje zawieszenia całego systemu.

**Wywłaszczenie jądra** - jądro pozwala na wywłaszczenie własnego kodu, co oznacza, że w wykonywanie jego kodu może zostać przerwane na czas wykonywania przez procesor innego zadania.

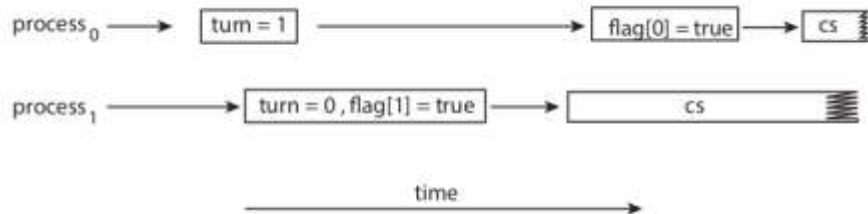
**Wywłaszczanie przerw** - przerwania są zwykle niewywłaszczalne, dlatego powinny być krótkie.

Cecha	Jądro wywłaszczające	Jądro niewywłaszczające
Definicja 	Pozwala na usuwanie i podmianę procesu wykonywanego w trybie kernela, a w efekcie wykonywane jest zadanie o najwyższym priorytecie.	Pozwala na wywłaszczenie procesu wykonywanego w trybie kernela, co oznacza konieczność czekania na jego zakończenie.
Warunek wyścigu	Występuje - wiele procesów jest aktywnych	Nie występuje - jeden proces jest aktywny
Responsywność	Większa i deterministyczna responsywność	Mniejsza i niedeterministyczna responsywność
Implementacja	Skomplikowany projekt i implementacja	Mniej skomplikowany projekt i implementacja
Bezpieczeństwo	Większa stabilność pracy i użyteczność	Mniejsza stabilność pracy i użyteczność
Semaforey	Nie wymaga użycia semaforów	Dane współdzielone wymagają semaforów
Programowanie RT	Większa użyteczność w programowaniu RT	Mniejsza użyteczność w programowaniu RT
Wywłaszczanie	Jest	Brak
Przykłady	Linux od 2.6, IRIX, Solaris, NetBSD od v5 Mikrokernele: Windows NT, Vista, 7 i 10	Windows XP, Windows 2000, Linux do 2.4

Wywłaszczający - ang. *preemptive*

# Algorytm Peterson'a

- Dwa procesy  $P_0$  oraz  $P_1$
- Dwa procesy współdzielą:  
int turn;  
boolean flag[2];
- Zmienna turn wskazuje, którego procesu jest kolej na wejście do sekcji krytycznej.
- Tablica flag wskazuje, czy proces jest gotowy na wejście do sekcji krytycznej.



do {

```
flag[i] = TRUE;  
turn = j;  
while (flag[j] && turn == j);
```

critical section

```
flag[i] = FALSE;
```

remainder section

} while (TRUE);



# Synchronizacja sprzętowa

Mechanizmy:

- Blokowanie przerwań
- Test and set lock - TSL
- Swap
- TLS + czas oczekiwania

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```



## Test and set lock - TSL

- Wymagane wsparcie procesora do realizacji instrukcji atomowych
- Realizacja sekwencyjna instrukcji atomowych (także w przypadku SMP)
- Instrukcja atomowa: TestAndSet()

```
boolean TestAndSet(boolean *target) {  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}  
  
do {  
    while (TestAndSet(&lock))  
        ; // do nothing  
  
    // critical section  
  
    lock = FALSE;  
  
    // remainder section  
} while (TRUE);
```



# Swap

- Wymagane wsparcie procesora do realizacji instrukcji atomowych
- Realizacja sekwencyjna instrukcji atomowych (także w przypadku SMP)
- Instrukcja atomowa: Swap()
- Inicjalizacja globalnych zmiennych:

```
boolean waiting[n];  
boolean lock;
```

```
void Swap(boolean *a, boolean *b) {  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}  
  
do {  
    key = TRUE;  
    while (key == TRUE)  
        Swap(&lock, &key);  
  
    // critical section  
  
    lock = FALSE;  
  
    // remainder section  
} while (TRUE);
```



# TLS + czas oczekiwania

Test and set lock oraz Swap:

- Spełniają wymaganie wzajemnego wykluczenia, ale
- Nie spełniają wymagania dot. skończonego czasu oczekiwania
- Obok: algorytm spełniający wszystkie wymagania dot. sekcji krytycznej

```
do {  
    waiting[i] = TRUE;  
    key = TRUE;  
    while (waiting[i] && key)  
        key = TestAndSet(&lock);  
    waiting[i] = FALSE;  
  
    // critical section  
  
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
  
    if (j == i)  
        lock = FALSE;  
    else  
        waiting[j] = FALSE;  
  
    // remainder section  
} while (TRUE);
```





# Semafor

- Semafor  $S$  to zmienna całkowita
- Semafor można modyfikować tylko w:
  - `wait()`
  - `signal()`
- Kiedy jeden proces modyfikuje semafor, żaden inny nie może tego robić
- Testowanie warunku  $S \leq 0$  oraz inkrementacja  $S--$  musi wykonać się bez przerwania

```
wait(S) {  
    while S <= 0  
        ; // no-op  
    S--;  
}  
  
signal(S) {  
    S++;  
}
```



## Typy semaforów

- Semafor binarny (ang. *binary semaphore*)  
= **mutex lock** od: *mutual exclusion*, czyli wzajemne wykluczenie  
Przy zastosowaniu do sekcji krytycznej:
  - procesy współdzielą semafor, mutex=1,
  - każdy proces działa jak na listingu obok.
- Semafor zliczający (ang. *counting semaphore*)  
Zastosowanie do sekcji krytycznej, kiedy dany zasób ma wiele instancji.

```
wait(S) {  
    while S <= 0  
        ; // no-op  
    S--;  
}
```

```
signal(S) {  
    S++;  
}
```

```
do {  
    wait(mutex);  
  
    // critical section  
  
    signal(mutex);  
  
    // remainder section  
} while (TRUE);
```



# Problemy synchronizacyjne

- Problem ograniczonego bufora
- Problem czytelników i pisarzy
- Problem ucztujących filozofów



# Problem ograniczonego bufora

Założenia:

- Pula buforów, rozmiar puli wynosi  $n$
- Każdy bufor może zawierać jeden obiekt
- Zmienna mutex jest semaforem b. do puli
- Na początku  $\text{mutex} = 1$
- Semaforey empty i full to liczność pustych/pełnych buforów
- Na początku  $\text{empty} = n, \text{full} = 0$

```
do {  
    . . .  
    // produce an item in nextp  
    . . .  
    wait(empty);  
    wait(mutex);  
    . . .  
    // add nextp to buffer  
    . . .  
    signal(mutex);  
    signal(full);  
} while (TRUE);
```

Producent

```
do {  
    wait(full);  
    wait(mutex);  
    . . .  
    // remove an item from buffer to nextc  
    . . .  
    signal(mutex);  
    signal(empty);  
    . . .  
    // consume the item in nextc  
    . . .  
} while (TRUE);
```

Konsument



# Problem czytelników i pisarzy - definicja

## Założenia:

- Istnieje współdzielona baza danych
- Dwa typy procesów: piszące i czytające
- Procesy czytające mogą w dowolnej liczbie osiągać dostęp do bazy
- Jeśli jeden proces piszący ma dostęp, w tym czasie żaden inny proces (ani piszący, ani czytający) nie może mieć dostępu

## Warianty:

- I. Żaden proces czytający nie czeka na dostęp, chyba, że proces piszący go uzyskał. Ryzyko zagłodzenia pisarzy.
- II. Jeśli pisarz oczekuje na dostęp, żaden czytelnik nie może rozpocząć czytania. Może dojść do zagłodzenia czytelników.

# Problem czytelników i pisarzy - rozwiązanie I

- Czytelnicy współdzielą:

```
semaphore mutex, wrt;           // init: 1
int readcount;
                                // init: 0
```

- wrt jest wspólny także dla pisarzy
- wrt jest mureksem obsługującym pisarzy
- wrt jest także dla pierwszego i ostatniego czytelnika w sekcji krytycznej,
- mutex obsługuje zmienną readcount
- readcount - liczba aktualnie czytających

```
do {
    wait(wrt);
    . . .
    // writing is performed
    . . .
    signal(wrt);
} while (TRUE);
```

Pisarz

```
do {
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(wrt);
    signal(mutex);
    . . .
    // reading is performed
    . . .
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
} while (TRUE);
```

Czytelnik

# Problem uczących filozofów - definicja



- Rozważmy pięciu filozofów siedzących przy stole jak na obrazku obok
- Na środku stołu jest miska ryżu, wokół pięć talerzy, po jednym dla każdego filozofa
- Pomiedzy talerzami jest pięć sztućców
- Od czasu do czasu dany filozof chce zjeść
- Aby zjeść muszą być wolne dwa sztućce
- Jedząc filozof ma wyłączność na 2 sztućce
- Po skończeniu jedzenia zwalnia sztućce



# Problem ucztujących filozofów - rozwiązanie

- Filozof próbuje wziąć sztućce wywołując:  
wait()
- Filozof odkłada sztućce wywołując:  
signal()
- Filozofowie współdzielą sztućce:

```
semaphore chopstick[5];          // init:
```

1

- Rozwiązania:
  1. Max 4-ch filozofów przy stole
  2. Można podnieść sztućce tylko wtedy, jeśli oba są wolne (podnieść w sekcji krytycznej)
  3. Parzyści filozofowie podnoszą najpierw lewy, potem prawy sztuciec, a nieparzyści odwrotnie: najpierw prawy, potem lewy

```
do {  
    wait(chopstick[i]);  
    wait(chopstick[(i+1) % 5]);  
    . . .  
    // eat  
    . . .  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    . . .  
    // think  
    . . .  
} while (TRUE);
```





**Kudłacze**



# Systemy Operacyjne

**Pamięć główna**

Dr hab. inż. Krzysztof Rzecki, prof. AGH

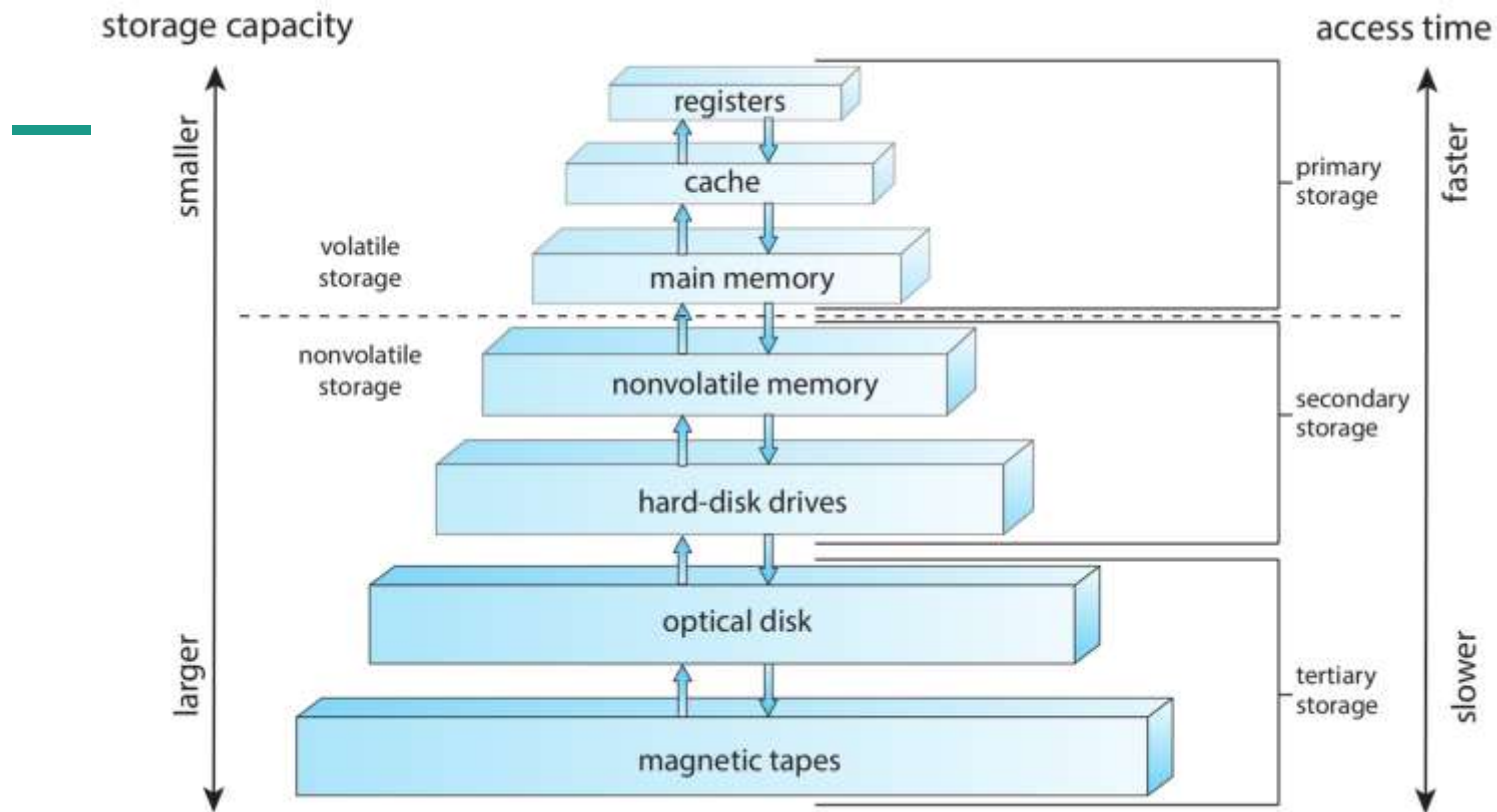
Na podstawie: Abraham Silberschatz, *Koncepcje systemów operacyjnych*



# Współdzielenie zasobów

- CPU: współdzielenie w czasie
- MEM: współdzielenie w ilości
- HDD: współdzielenie w dostępie
- NET: współdzielenie w czasie

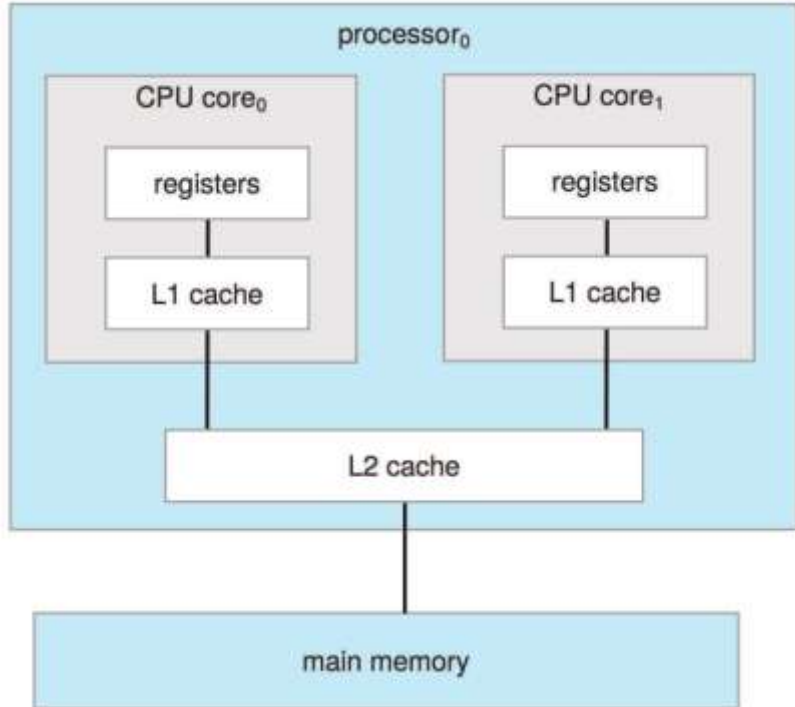




Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

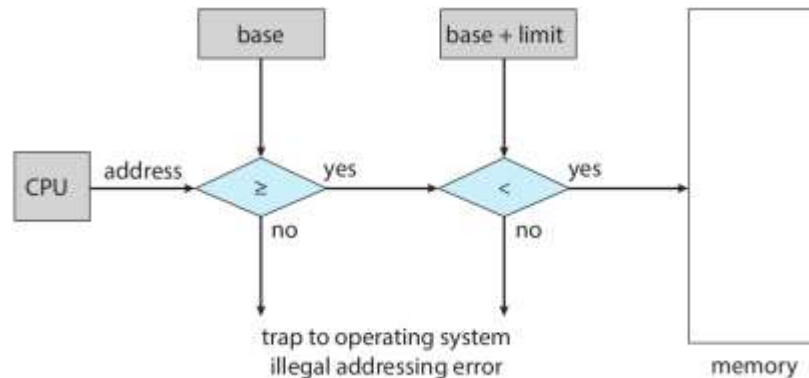
# Procesor + pamięć

- Procesor ładuje instrukcje tylko z **pamięci głównej**, więc każdy program musi być do niej najpierw załadowany.
- Pamięć główna (ang. *main memory*, RAM - ang. *random-access memory*) wykonana jest w technologii półprzewodnikowej zwanej DRAM - ang. *dynamic random-access memory*.
- Nie wszystko mieści się w pamięci RAM oraz pamięć ta jest ulotna, stąd wymagana jest pamięć dodatkowa, tj. ang. *secondary storage* (ang. *hard-disk drives* - HDDs lub ang. *nonvolatile memory* - NVM).



# Podstawy adresowania

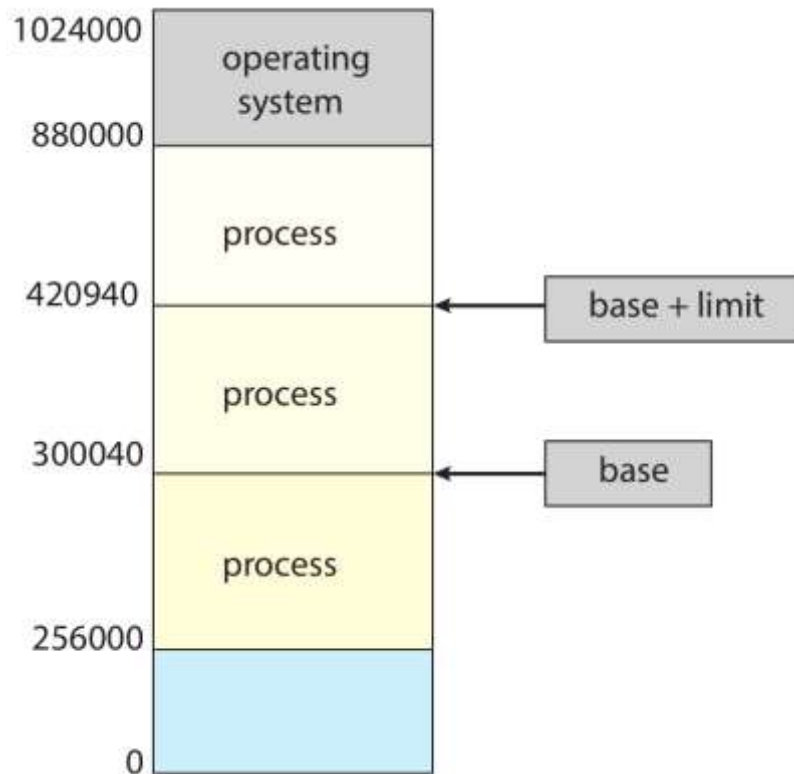
- Pamięć główna oraz rejestry wbudowane w CPU to jedyna pamięć, którą CPU może adresować bezpośrednio.
- Jeśli dana, na której mają być wykonywane operacje znajduje się na którejkolwiek z pozostałych pamięci, musi zostać najpierw skopiowana w obszar o bezpośrednim dostępie.
- Adresowanie rejestrów, w odróżnieniu od adresowania pamięci głównej, zazwyczaj odbywa się w jednym takcie procesora.
- Aby w trakcie uzyskiwania dostępu do pamięci głównej procesor nie marnował cykli, wykorzystywany jest cache.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

# Izolacja pamięci

- Izolacja pamięci dla każdego procesu gwarantuje indywidualną przestrzeń.
- Pamięć przydzieloną procesowi wyznaczają dwa rejestry: baza i limit.
- Proces użytkownika (*user mode*) może zapisywać i odczytywać tylko pamięć w przydzielonym zakresie.
- *Dlaczego ?*
- System operacyjny (*kernel mode*) może swobodnie operować po całej pamięci, w szczególności zmieniać bazę i limit.
- *Jakie to ma zastosowanie ?*



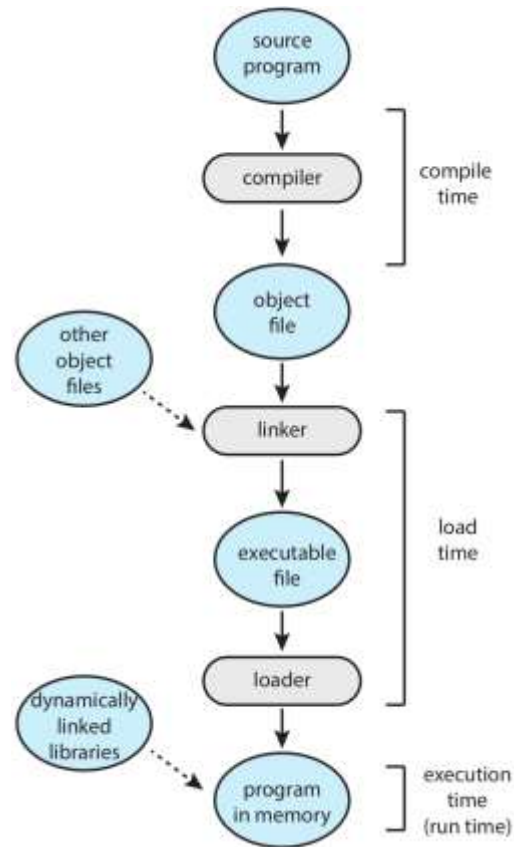
Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

# Wiązanie adresu

- Adres w kodzie programu może być symboliczny, np. nazwa zmiennej.
- Kompilator wiąże w/w adres symboliczny do adresu relokowalnego (względem danego modułu).
- Linker lub loader zamienia adres relokowalny w bezwzględny.

Wiązanie może wystąpić na każdym z etapów:

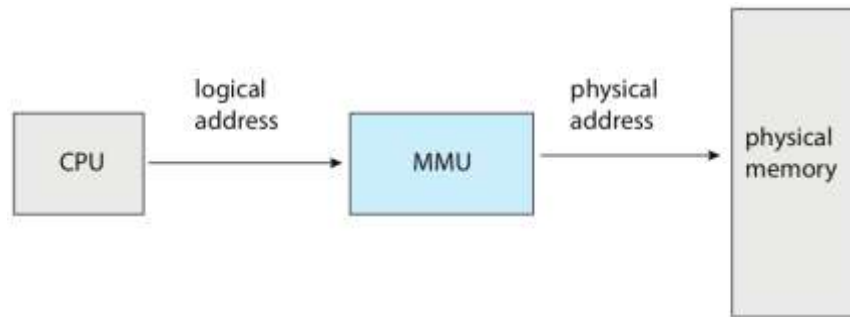
- Kompilacja (stare systemy),
- Ładowanie (MS-DOS),
- Wykonywanie - najczęściej (obecnie).





# Logiczna i fizyczna przestrzeń adresowa

- Adres logiczny - adres widziany przez proces i dla niego dostępny.
- Adres fizyczny - adres na szynie adresowej.
- Translacja adresów zajmuje się MMU - ang. *Memory management unit*.
- Wiązanie adresów w czasie:
  - kompilacji -> adres logiczny == fizyczny,
  - ładowania -> adres logiczny == fizyczny,
  - wykonywania -> adres logiczny != fizyczny.
- Adres wirtualny = adres logiczny w sytuacji wiązania w czasie wykonywania.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Ćw.: w C/C++ statyczna/dynamiczna rezerwacja pamięci, odczyt adresu zmiennej, kilkukrotne uruchomienie programu.



# Dynamiczne ładowanie funkcjonalności

- Po stronie programisty leży konstrukcja programu z dynamicznie ładowanymi funkcjonalnościami.
- Funkcjonalności ładowane są dopiero w momencie ich wywołania.
- Kiedy któraś funkcjonalność wywołuje inną, a ta nie jest załadowana do pamięci, następuje jej załadowanie.
- Zaletą jest ładowanie funkcjonalności tylko wtedy, kiedy są potrzebne.



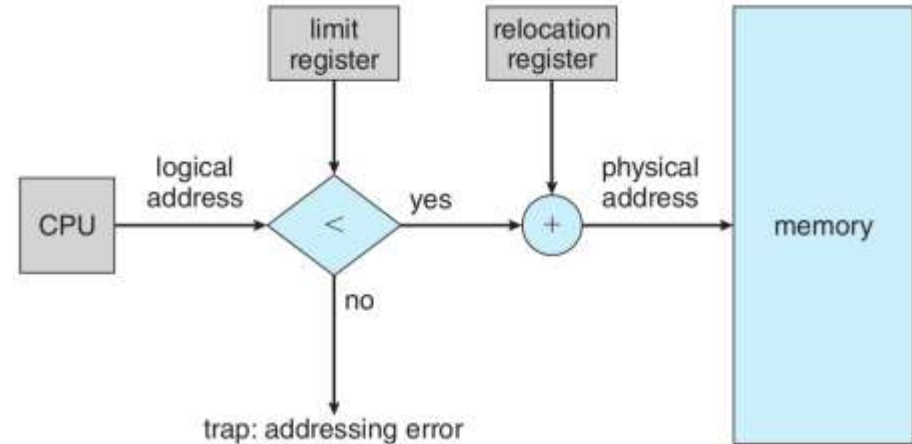
# Linkowanie dynamiczne i statyczne

- DLL - ang. *Dynamic-Link Library* - biblioteka systemowa linkowana do programu w momencie jego uruchomienia (wymaga wsparcia ze strony systemu operacyjnego).
- Zalety DLL:
  - nie trzeba kopiować/implementować istniejących już modułów programu (oszczędność pamięci),
  - biblioteka może zostać jednokrotnie załadowana do pamięci, a wiele procesów może z niej korzystać,
  - biblioteki mogą być aktualizowane jednokrotnie, jakkolwiek każdy program może używać wskazanej wersji.
- Linkowanie statyczne - włączanie biblioteki systemowej do obrazu binarnego programu.

Pytanie C/C++: jak linkować statycznie/dynamicznie ?

# Ochrona pamięci

- Relocation register (rejestr bazowy, rejestr relokacji) - początkowy adres fizyczny obszaru przeznaczanego dla procesu.
- Limit register - zawiera zakres adresów logicznych.
- Każdy proces wskazany przez scheduler CPU do uruchomienia sprawdzany jest względem w/w rejestrów.
- Dzięki temu można chronić system operacyjny i inne programy przed modyfikacją przez ten program.

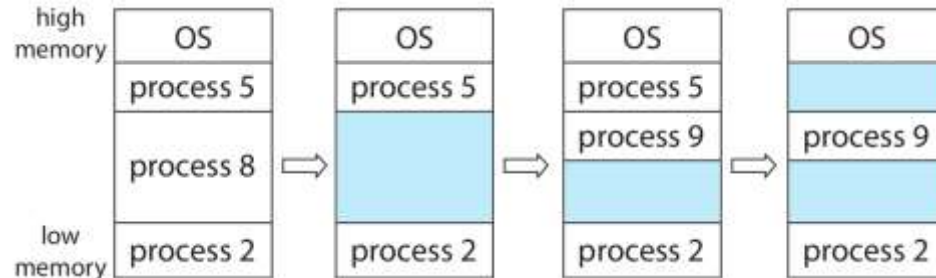


Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Ćw.: odczyt wskaźnika spoza przydziel. pamięci.

# Alokacja pamięci

- Metoda *variable-partition* - przypisanie procesu do zakresu pamięci i pamiętanie przez SO, które części pamięci są zajęte, a które wolne.
- Jeśli nie ma możliwości umieścić proces w pamięci: a/ proces nie jest uruchamiany, b/ proces oczekuje w kolejce.
- Zarządzanie wolną przestrzenią (ang. *hole*) obejmuje jej dzielenie i łączenie i nazywane jest ang. *dynamic storage allocation problem*:
  - First fit - pierwszy pasujący
  - Best fit - najlepiej pasujący
  - Worst fit - największy wolny



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

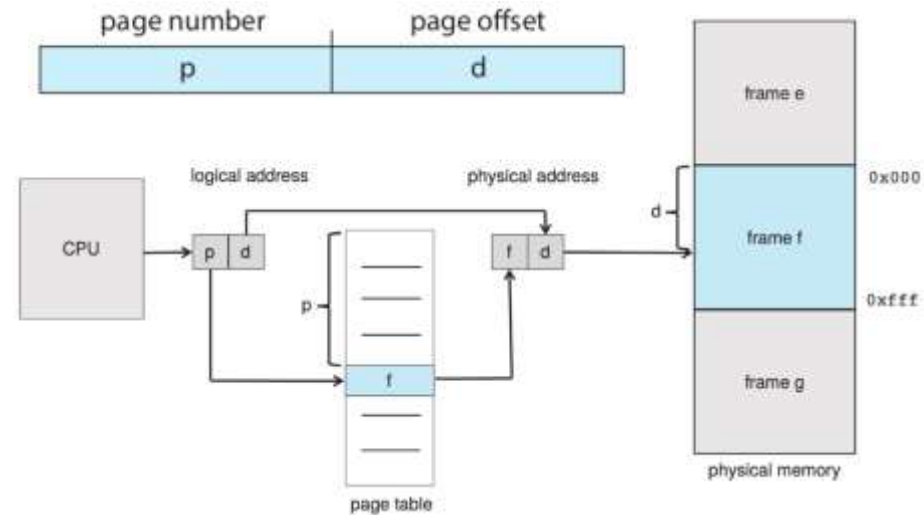


# Fragmentacja

- W przypadku strategii first-fit i best-fit szybko dochodzi do tzw. fragmentacji zewnętrznej.
- Najgorszy przypadek fragmentacji: między każdą dwójką procesów jest wolna przestrzeń.
- Oprócz doboru strategii znaczenie ma też położenie nowego procesu (na początku czy na końcu wolnego miejsca?).
- Statystycznie, po pewnym czasie strategii first-fit aż 50% bloków zmarnowanych będzie na fragmentację.
- W przypadku podzielenia pamięci na bloki przydzielona pamięć dla danego procesu może być większa niż proces wymaga - różnica między wielkością przydzieloną a wymaganą to fragmentacja wewnętrzna.
- Rozwiązanie: kompaktowanie (Java: garbage collection) - możliwe tylko w czasie działania.
- Rozwiązanie: stronicowanie (następny slajd).

# Stronicowanie, ang. *paging*

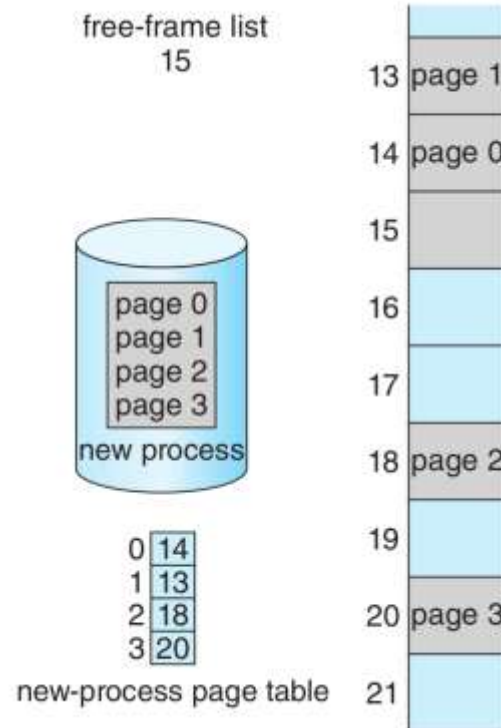
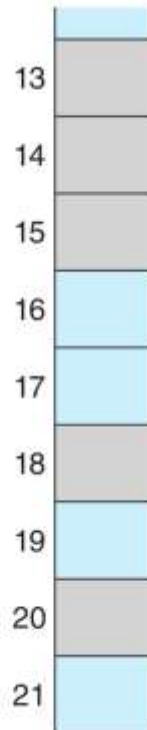
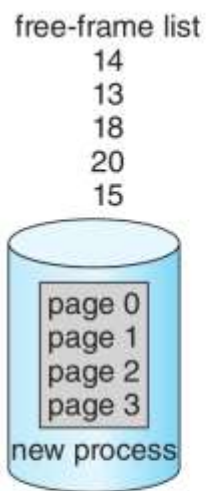
- Stronicowanie - logiczna przestrzeń adresowa jest nieciągła.
- Pamięć fizyczna dzielona jest na bloki, ang. *frames*.
- Pamięć logiczna dzielona jest na bloki tej samej wielkości, ang. *pages*.
- Kiedy proces ma być wykonywany, jego strony ładowane są do ramek dostępnej pamięci.
- Każdy proces ma swoją tablicę stron.
- Rozmiar strony zależny jest od sprzętu i wynosi od 4KB do 1GB i ze względu na adresację jest potęgą 2 (getconf PAGESIZE).



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

# Alokacja pamięci

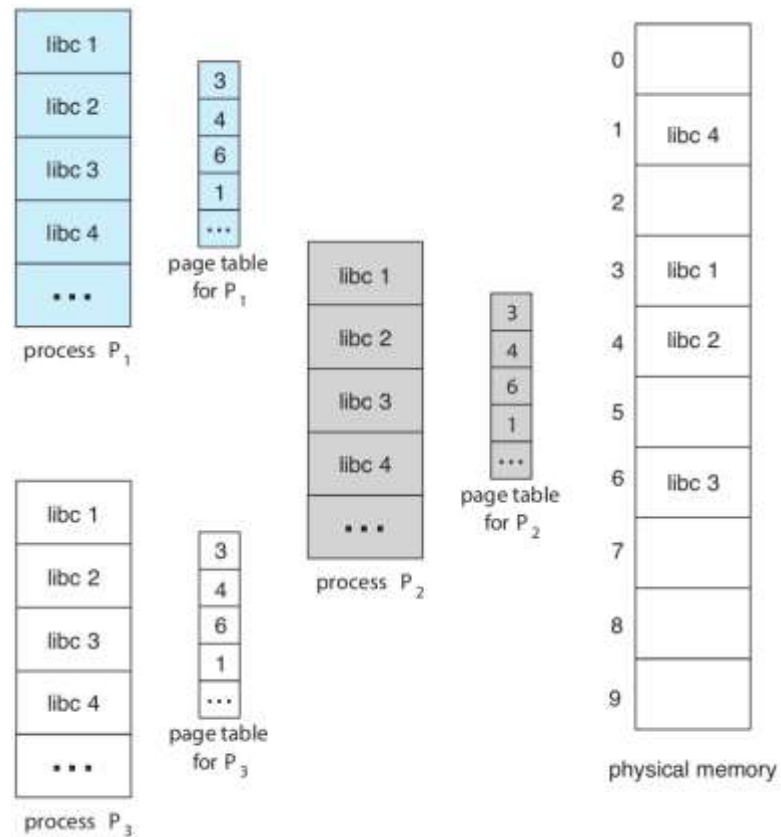
- Programista jest zupełnie odseparowany od widoku na pamięć fizyczną.
- Program może być rozrzucony po obszarach pamięci.
- Tablicą stronicowania oraz tablicą ramek zarządza system operacyjny.



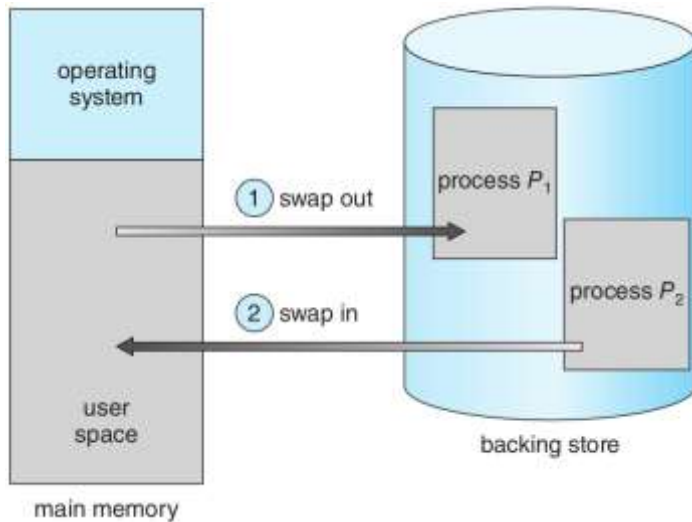


# Współdzielenie

Przykład kodu wtórnego: biblioteka `libc`

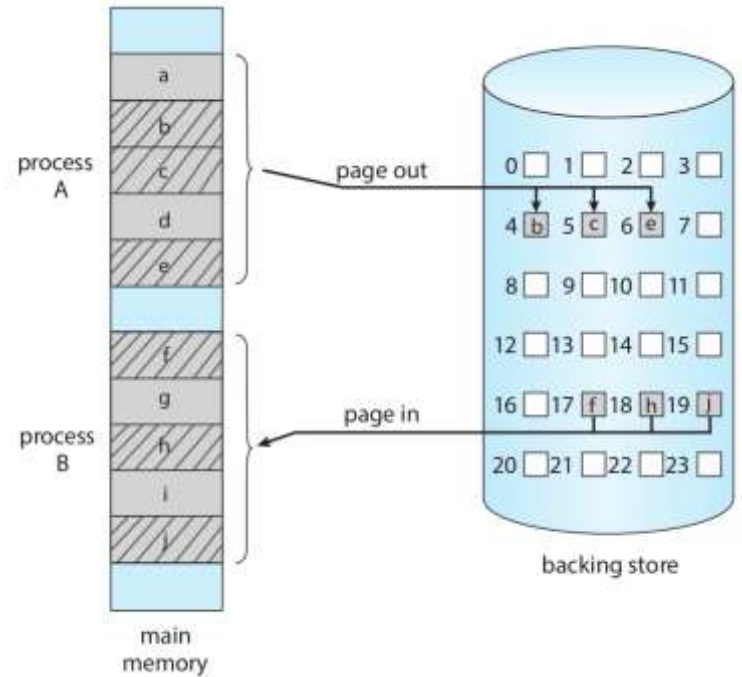


# Swap



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Kiedyś



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Obecnie



# Zadanie

Zakładając logiczną przestrzeń adresową składającą się z 64 stron o rozmiarze 1024 słów zmapowaną na fizyczną przestrzeń adresową o rozmiarze 32 ramek:

- Jaki rozmiar ma adres logiczny ?
- Jaki rozmiar ma adres fizyczny ?



**Skrzyczne**





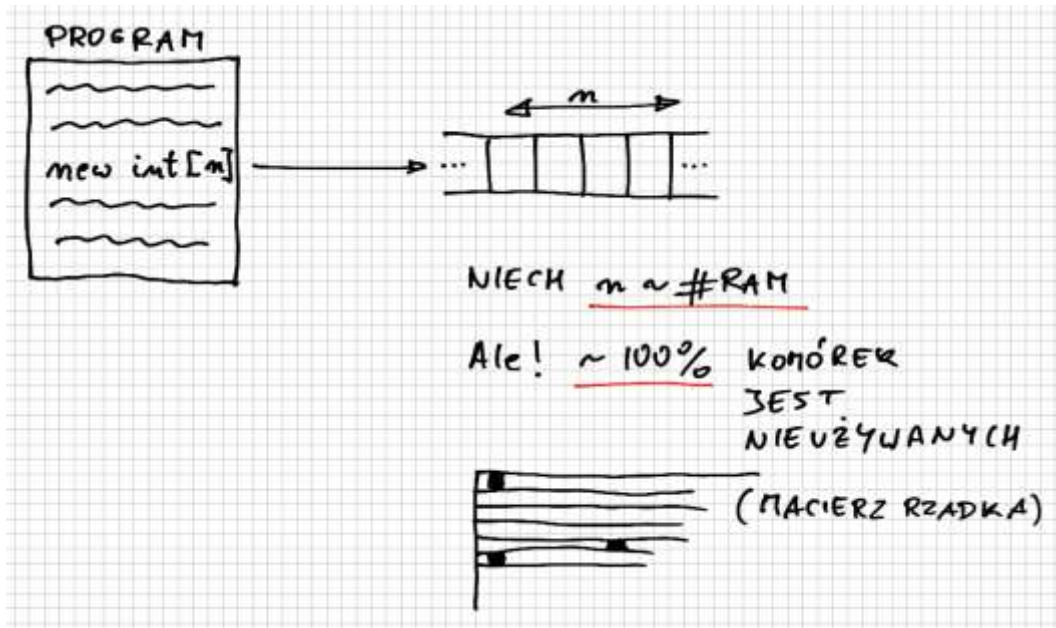
# Systemy Operacyjne

Pamięć wirtualna

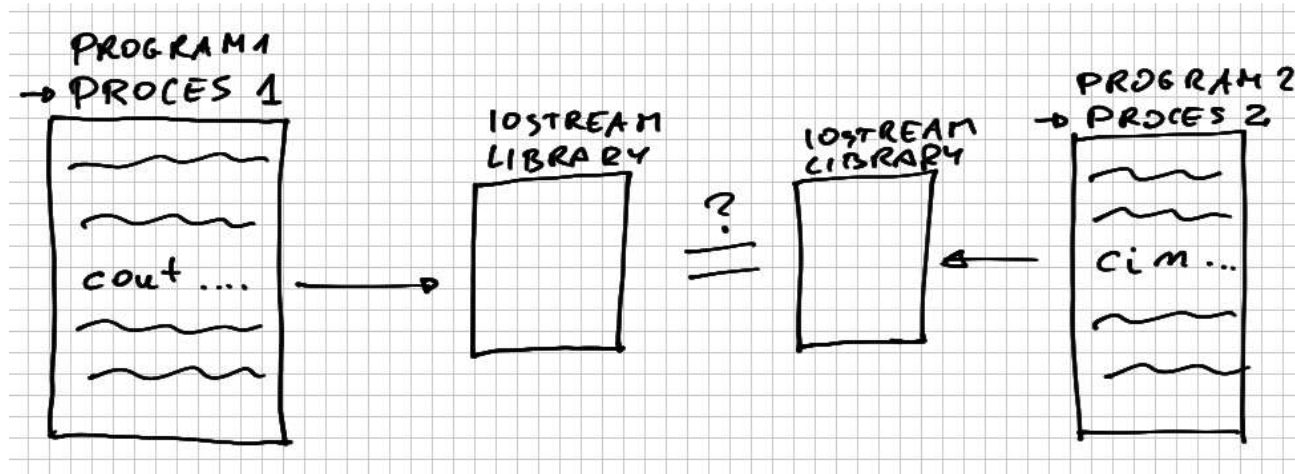
Dr hab. inż. Krzysztof Rzecki, prof. AGH

Na podstawie: Abraham Silberschatz, *Koncepcje systemów operacyjnych*

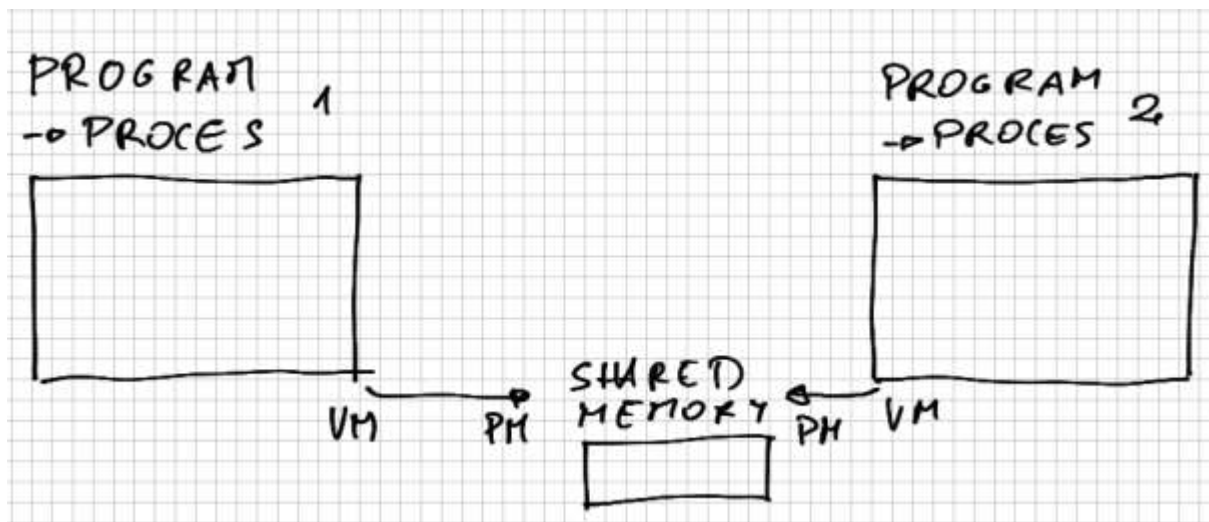
## Przypadek 1



## Przypadek 2

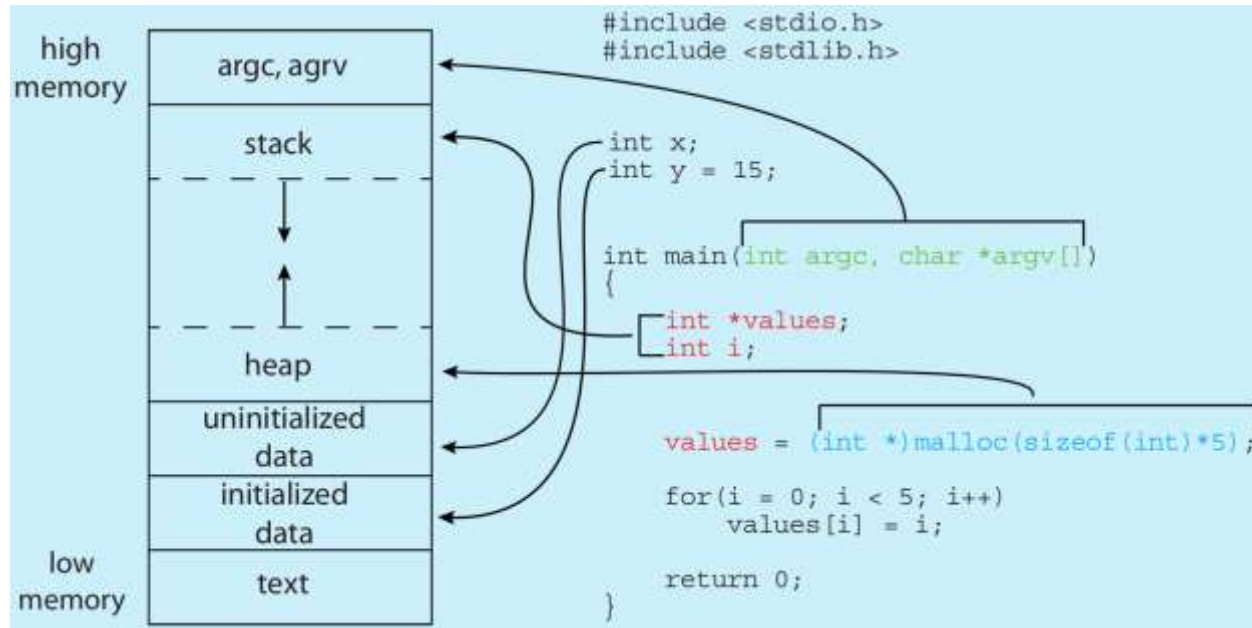


## Przypadek 3





# Program w C - przypomnienie



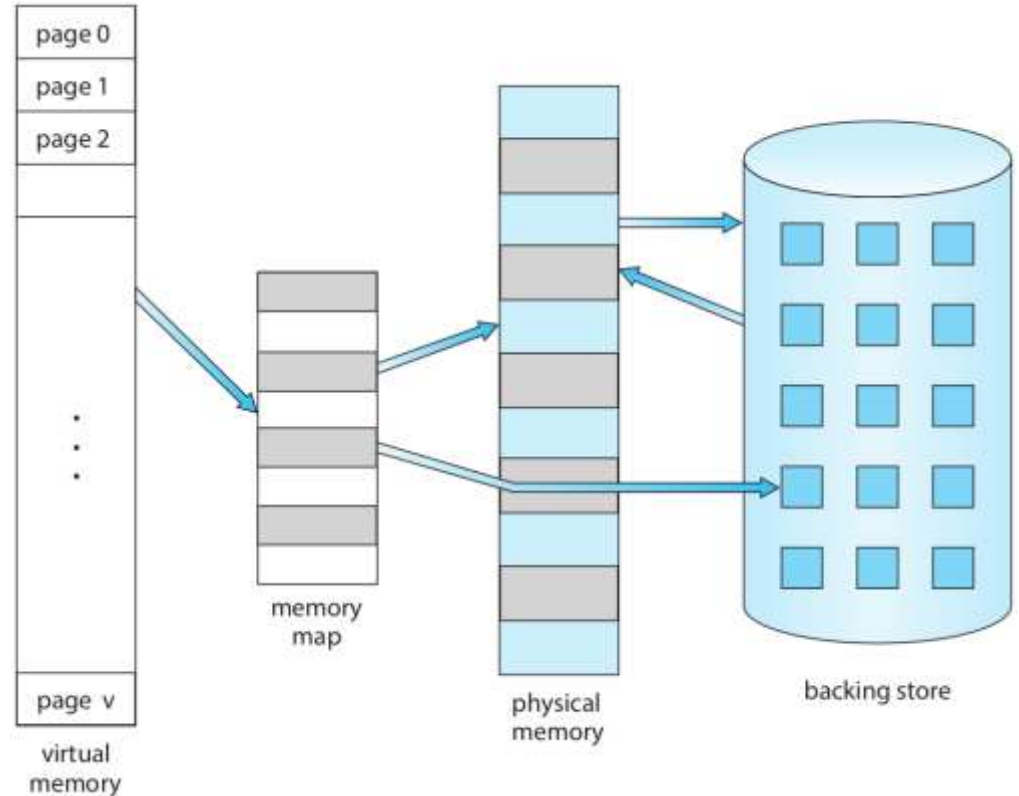


# Pamięć wirtualna

- Pamięć wirtualna to technika pozwalająca na wykonywanie procesów, które nie są całkowicie w pamięci (m.in. np. z powodu ich rozmiaru).
- Pamięć wirtualna jest abstraktem pamięci głównej jako bardzo dużej macierzy, oddzielając pamięć logiczną przeznaczoną dla programisty od pamięci fizycznej.
- Pamięć wirtualna pozwala procesom na współdzielenie plików, bibliotek oraz implementację pamięci dzielonej.
- Implementacja pamięci wirtualnej nie jest prosta, a nieostrożne korzystanie z niej wpływa znacząco na wydajność procesów.

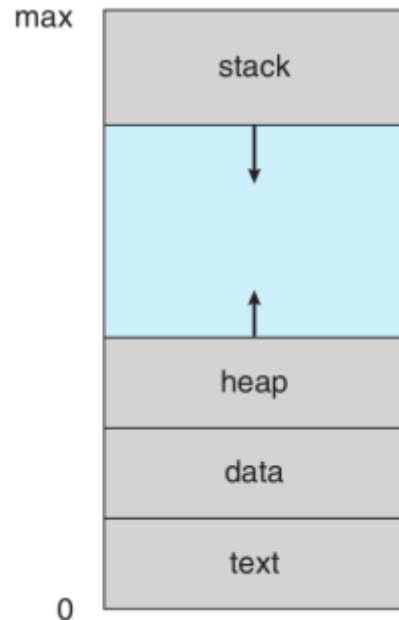
# Pamięć wirtualna

- Pamięć wirtualna oddzielona jest od pamięci fizycznej.
- Dzięki temu możliwe jest stosowanie olbrzymiej pamięci wirtualnej przy niewielkiej pamięci fizycznej.
- Programista nie musi zajmować się obsługą i ilością pamięci fizycznej.
- Pamięć zapasowa (ang. *backing store*, ang. *swap space*) - przestrzeń poza pamięcią główną.



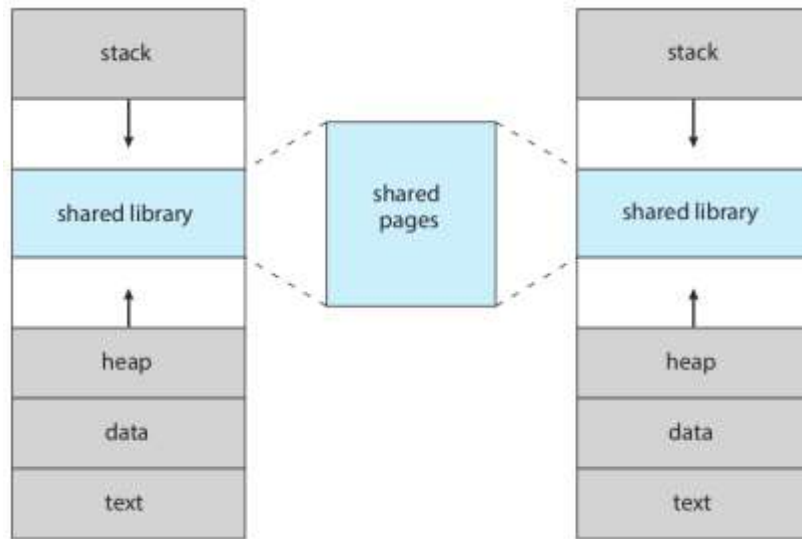
# Wirtualna przestrzeń adresowa

- Stos - ang. *Stack* - miejsce, w którym dane umieszczane są w sposób uporządkowany, w tym miejscu odkładane są dane dotyczące wywołań funkcji, zmiennych (statyczne w tym globalne, automatyczne w tym lokalne). Miejscem tym zarządza program.
- Sberta - ang. *Heap* - miejsce, w którym dane umieszczane są swobodnie wg zapotrzebowania, zwykle przez wywołanie malloc/new, są to zmienne (dynamiczne). Miejscem tym zarządza programista.



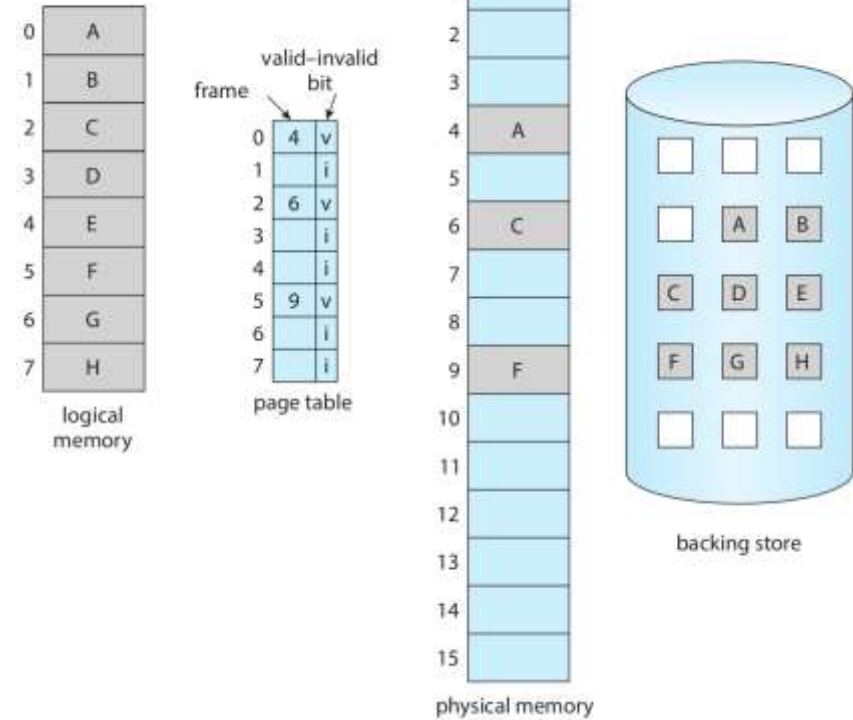
# Współdzielenie przestrzeni pamięci fizycznej

- Adresy wirtualne bibliotek systemowych mogą być mapowane z różnych procesów (w trybie tylko do odczytu) na wspólną część w przestrzeni adresów fizycznych.
- Procesy mogą współdzielić obszar pamięci (ang. *shared memory*) celem wymiany komunikatów i ją też muszą adresować w przestrzeni adresów wirtualnych.



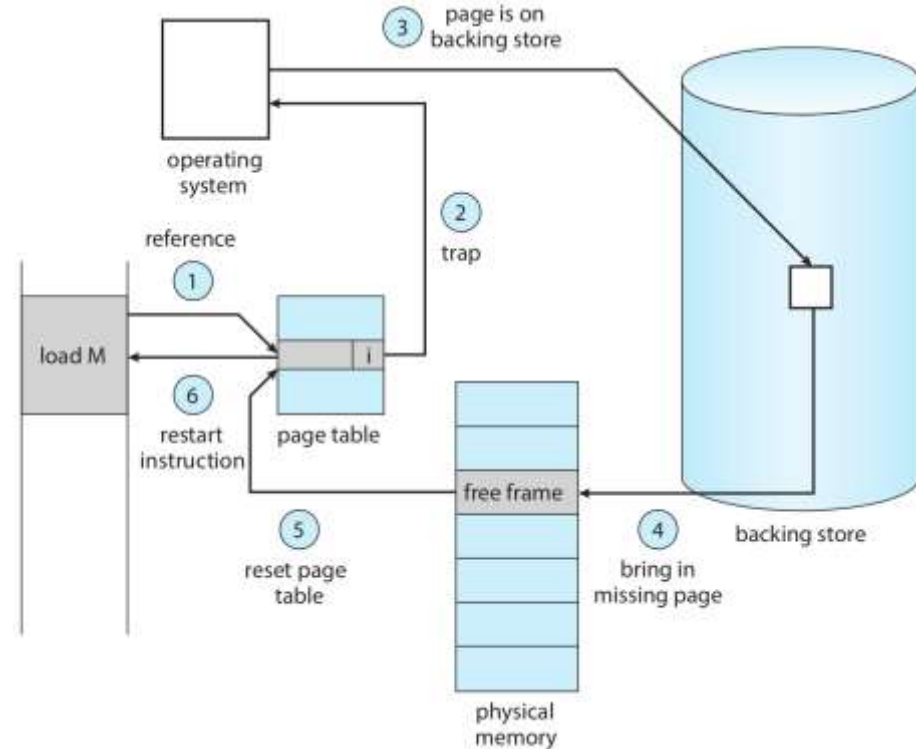
# Stronicowanie na żądanie

- Strona jest wprowadzana do pamięci wtedy, gdy jest potrzebna.
- Tablica stron zawiera dodatkowo bity poprawności odwołania:
  - 1 - strona znajduje się w pamięci głównej (ang. *valid*)
  - 0 - strona znajduje się poza pamięcią główną (ang. *invalid*)
- Odwołanie do strony z bitem == 0:
  - Błąd braku strony (ang. *page fault*)
  - Realizacja procedury z następnego slajdu.



## Obsługa błędu brak strony

1. Sprawdzenie tablicy stron (valid vs. invalid).
2. Jeśli referencja ma stan *invalid*, to:
  - a. Brak strony w ogóle -> terminowanie procesu.
  - b. Brak strony w pamięci głównej, ale jest w pamięci zapasowej, przejdź do (3).
3. Zgłoszenie do SO zapotrzebowania na wczytanie strony z pamięci zapasowej.
4. Odnajdowanie na liście wolnej ramki i wczytanie strony do ramki.
5. Aktualizacja tablicy stron.
6. Restart instrukcji, która wywołała błąd.



## Lista wolnych ramek (ang. *Free-Frame List*)

- W momencie wystąpienia błędu strony, system operacyjny musi dostarczyć daną stronę z pamięci zapasowej do pamięci głównej.
- W tym celu stosowana jest lista wolnych ramek:



- System operacyjny zwykle stosuje technikę ang. *zero-fill-on-demand*, która zeruje ramki przed użyciem (względny bezpieczeństwa).
- W momencie startu systemu, cała dostępna pamięć umieszczana jest na liście wolnych ramek.





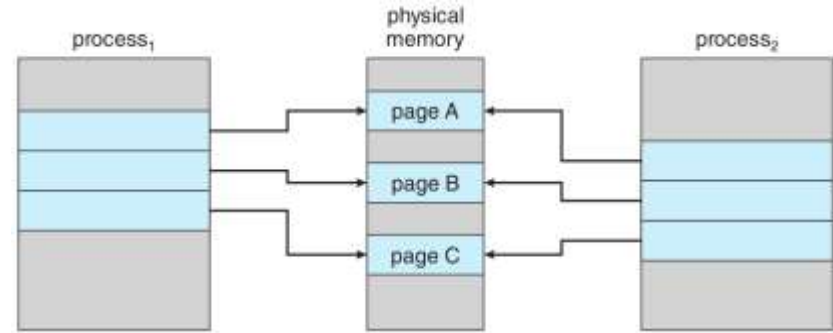
## Przebieg stronicowania na żądanie

1. Odwołanie/przerwanie do systemu operacyjnego.
2. Zapisz stan rejestrów oraz procesu.
3. Stwierdź, że przerwanie wywołane było przez błąd strony.
4. Sprawdź, czy odwołanie jest właściwe i określ położenie strony w pamięci zapasowej.
5. Wywołaj odczyt z pamięci zapasowej do wolnej ramki:
  - a. Czekaj w kolejce, aż żądanie odczytu zostanie obsłużone.
  - b. Odczekaj czas działania urządzenia.
  - c. Rozpocznij transfer strony do wolnej ramki.
6. Podczas oczekiwania, przekaz CPU innemu procesowi.
7. Przechwyć przerwanie z podsystemu I/O (zakończenie wczytywania).
8. Zapisz rejestry oraz stan innego procesu (jeśli krok 6 jest wykonany).
9. Określ, że przerwanie było z pamięci zapasowej.
10. Zaktualizuj tablicę stron, aby wskazać, że określona strona jest teraz w pamięci.
11. Czekaj, aż CPU zostanie przypisany znów temu procesowi.
12. Wznów rejestry, stan procesu oraz nową tablicę stron i wznów działanie procesu.

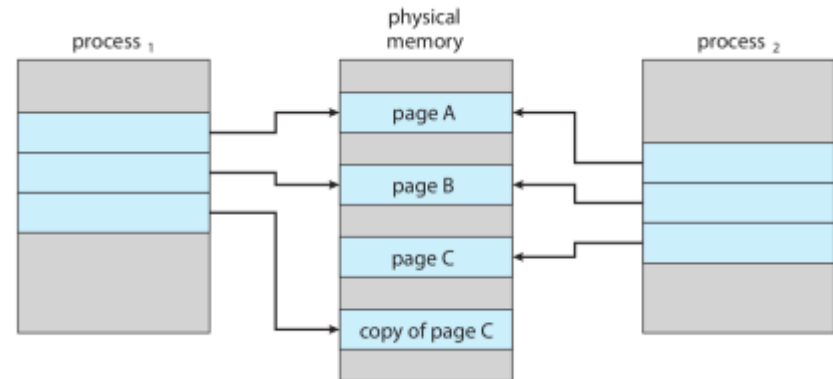
Czasochłonne!

# Copy-on-Write

- COW w technice `fork()` pozwala na współdzielenie tych samych stron w pamięci.
- Tylko ta strona, która jest modyfikowana wymaga kopiowania.
- Często po `fork()` występuje `exec()` i wtedy okazuje się, że kopiowanie w ogóle nie jest potrzebne.

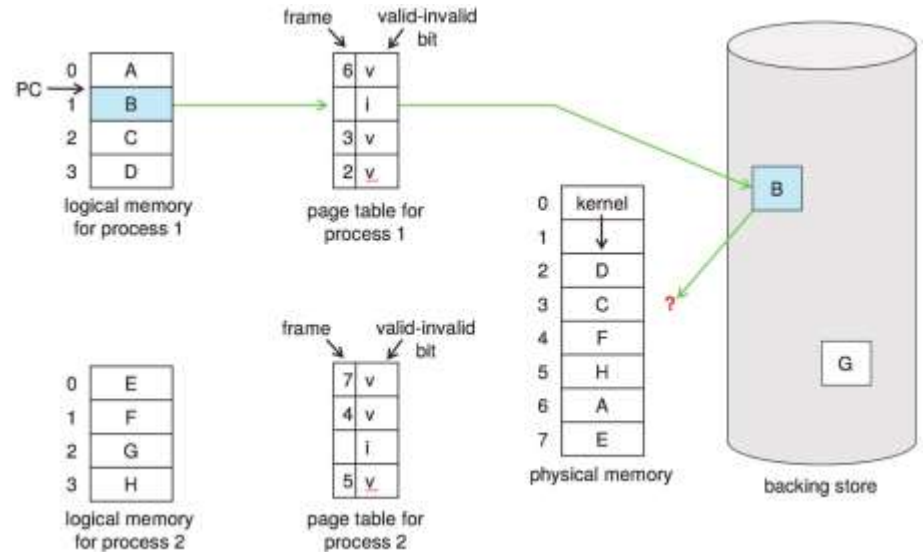


Modyfikacja strony C



## Over-allocation

- W trakcie wykonywania procesu, występuje błąd strony.
- System operacyjny odnajduje stronę w pamięci zapasowej, ale stwierdza, że nie ma wolnych ramek - cała pamięć jest zajęta.
- Jednym z rozwiązań jest terminowanie procesu... ale to nie jest rozwiązanie.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*



# Zastępowanie stron

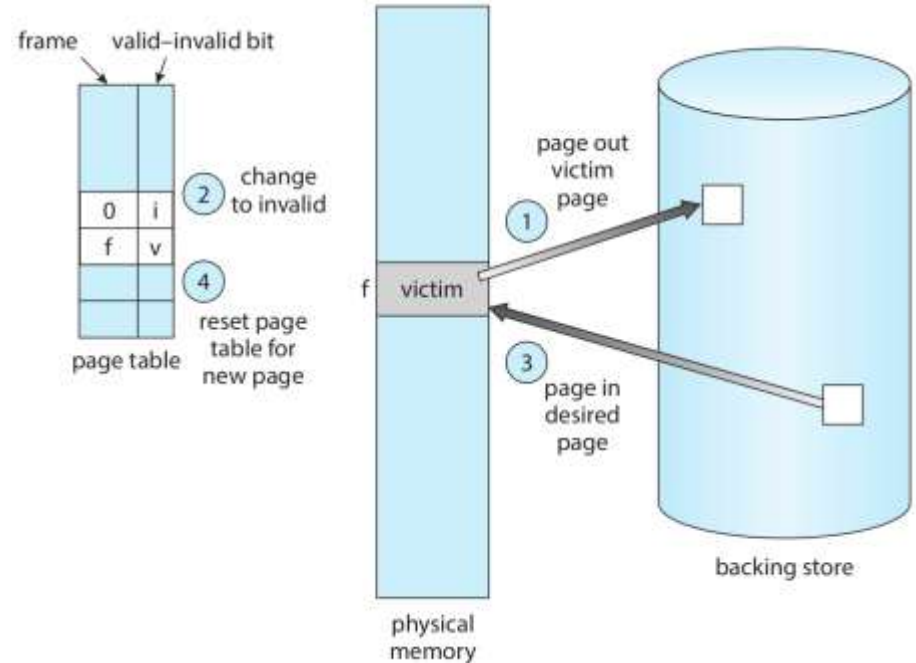
- Co kiedy wolne ramki się skończą ?
- Co w sytuacji, kiedy nie wszystkie strony są aktualnie w użyciu ?
- Jak zastępować strony nieużywane ?
- Czy ładować wszystkie strony, czy tylko przewidziane do użycia ?
- Co w sytuacji, kiedy odwołania do stron są w kilku miejscach programu ?
- Czy pozwalać na uruchomienie programów, dla których nie ma dostępnych ramek ?
- Jak wybierać ramki do zastąpienia ?
- Jak informować procesy o zastąpieniu ramki ?

Uwaga: adresacja logiczna = strony, adresacja fizyczna = ramki.

# Zastępowanie stron

1. Znajdź stronę w pamięci zapasowej.
2. Szukaj wolnej ramki:
  - Jeśli jest, użyj jej.
  - Jeśli brak, użyj algorytmu wyboru ramki podlegającej wymianie (ang. *victim frame*).
  - Zapisz jej zawartość do pamięci zapasowej, zaktualizuj tablice ramek i stron.
3. Wczytaj oczekiwaną stronę do zwolnionej ramki. Zaktualizuj tablice ramek i stron.
4. Kontynuuj działanie procesu.

Celem optymalizacji stosuje się bit modyfikacji, tzn. brudny bit (ang. *dirty bit*). Np. strony z kodem programu zasadniczo są read-only.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*



# Algorytmy zastępowania stron

- Algorytm FIFO:
  - najstarsza z umieszczonych w pamięci stron (głowa) jest zastępowana, nowe strony umieszczane są na końcu kolejki (ogon),
  - wada: z jednej strony strona zastępowana może być czymś, co było dawno temu umieszczone w pamięci i jest już nieużywane, ale może to być też często używana zmienna istniejąca od początku procesu.
- Optymalne zastępowanie stron:
  - zastąp stronę, która nie będzie używana przez najdłuższy czas.
  - wada: trzeba znać przyszłość ;-)
- Algorytm LRU - ang. *least recently used*
  - zastąp stronę, która najdłużej nie była używana,
  - implementacje: liczniki (timestamp ostatniego użycia), stos (przekładanie na wierzch użytej strony),
  - dalsze modyfikacje i optymalizacje.



# Alokacja ramek

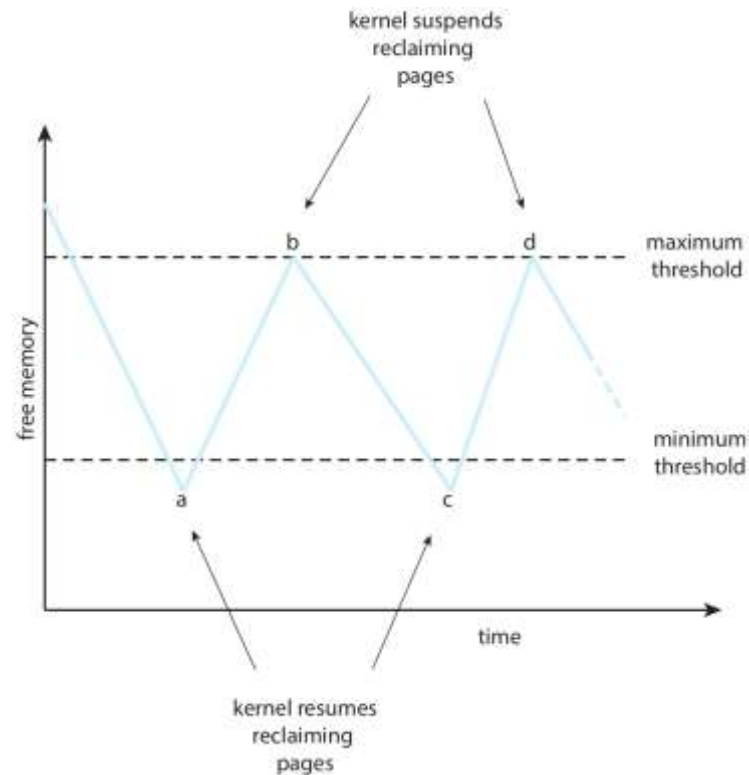
- **Przykład:** system ma 128 ramek pamięci fizycznej, OS zajmuje 35, a 93 ramki są dla procesów.
- W przypadku czystego stronicowania na żądanie (proces w całości ładowany jest do pamięci zapasowej), proces użytkownika wywoła 93 razy błąd strony, a przy żądaniu 94-tej strony zadziała algorytm zastępowania stron. Po zakończeniu procesu zwolnią się wszystkie ramki.
- **Strategie:**
- Minimalna liczba ramek - określona jest minimalna liczba ramek, które muszą zostać zaalokowane. Powód: im mniej zaalokowanych ramek, tym bardziej spada wydajność wykonywanych procesów.
- Równa alokacja - ramki po równo podzielone są między procesy, np. 93 ramki podzielić między 5 procesów oznacza, że każdy z nich otrzyma 18 ramek. Pozostałe 3 ramki trafią do puli wolnych.
- Proporcjonalna alokacja - ramki przydzielane są proporcjonalnie do wielkości procesów.

## Globalna i lokalna alokacja

- Alokacja globalna / zastępowanie globalne - realizacja procesu wymaga zastępowania ramek pośród wszystkich ramek, także tych zaalokowanych do innego procesu.
- Alokacja lokalna / zastępowanie lokalne - realizacja procesu wymaga zastępowania ramek tylko pośród zaalokowanych do procesu.

Alokacja globalna ma szczególne zastosowanie w przypadku priorytetyzowania procesów.

Rysunek obok: strategia utrzymania wolnej pamięci w alokacji globalnej.







## Major and minor page faults

- Major page fault - występuje wtedy, gdy strona jest wywoływana, a nie znajduje się w pamięci. Obsługa tego błędu wymaga odczytania wskazanej strony z pamięci zapasowej do wolnej ramki i aktualizacji tablicy stron. Stronicowanie na żądanie zwykle generuje znaczną liczbę tych błędów.
- Minor page fault - występuje wtedy, gdy proces nie ma logicznego mapowania do strony, ale ta strona jest w pamięci. Błąd ten występuje w jednym z przypadków:
  - Proces odwołuje się do biblioteki dzielonej, która jest w pamięci, ale proces nie ma do niej mapowania. W tym przypadku wystarczy zaktualizować tablicę stron.
  - Proces utracił stronę, która trafiła na listę wolnych ramek, ale nie została jeszcze wyzerowana. W tym przypadku strona jest ponownie przypisana do procesu i usunięta z listy wolnych ramek.

Na laboratorium:

```
ps -eo min_flt, maj_flt, cmd
```