

# Systemy operacyjne

## Wykład #3

- ochrona w SO
- interpretacja poleceń w SO
- API i ABI
- komunikacja międzyprocesowa
- struktura warstwowa
- wirtualizacja

# ochrona w systemie operacyjnym

## Ochrona

**mechanizm monitorowania, kontrolowania i nadzorowania dostępu programów, procesów i użytkowników do zasobów systemu.**

**Wykorzystuje autoryzację i autentykację**

# ochrona w systemie operacyjnym

## **Autoryzacja:**

**stwierdzenie, że dany proces/użytkownik ma prawo wykorzystania danego zasobu**

## **Autentykacja:**

**stwierdzenie, że dany proces/użytkownik jest tym, za którego się podaje**

# Ochrona w systemie operacyjnym

**pewnik:**

**każdy system wielozadaniowy system operacyjny musi zawierać mechanizmy ochrony przed wzajemnym niepożądanym oddziaływaniem procesów**

# ochrona w systemie operacyjnym

## Zadania podsystemu ochrony:

- autoryzacja (użytkowników i procesów)
- autentykacja (użytkowników i procesów)
- uprawnienia (użytkowników i procesów)
- mechanizmy wykrywania naruszeń
- mechanizmy wymuszania legalności
- rejestracja działań

# interpretacja poleceń

- mechanizm odbierania i wykonywania poleceń użytkownika
- kontrola składni poleceń
- kontrola semantyki poleceń
- sygnalizacja błędów
- specjalizowany język (np. JCL w systemie OS/360)
- może integralną częścią systemu lub wydzielonym komponentem (np. shell w systemach U\*x)

# oblicze graficzne

- powłoka GUI (Graphical User Interface)
- zastąpienie operacji składniowych manualnymi (np. kliknięcia zamiast wpisywanych poleceń)
- wytworzenie mechanizmów graficznych dla procesów i aplikacji
- dwa warianty:
  - pełna integracja z jądrem systemu (np. MS Windows)
  - wydzielony, elastyczny składnik (np. *X Window + window manager* w systemach U\*x)

# Usługi systemu operacyjnego

- udokumentowany i opisany zestaw czynności, jakie system operacyjny może wykonać na rzecz pracującego procesu
- API – *Application Program Interface* – specyfikacja sposobu, w który aplikacja (proces) wymusza na systemie operacyjnym żądane akcje
- ABI - *Application Binary Interface* – jak, ale nie na poziomie kodu źródłowego, a na poziomie danych binarnych procesu
- Windows API
- Standard Posix
- dostępne dla programistów jako
  - funkcje biblioteczne (w językach wysokiego poziomu)
  - klasy, obiekty, moduły, wrappery



# Usługi systemu operacyjnego

- rodzaje usług:
  - uruchamianie i kończenie procesów (np. `exec()`, `fork()`, `kill()`)
  - przydział i zwalnianie pamięci (np. `alloc()`, `free()`)
  - operacje we/wy (np. `open()`, `close()`, `read()`, `write()`)
  - manipulowanie plikami (np. `unlink()`, `stat()`, `chmod()`)
  - komunikacja międzyprocesowa (np. `signal()`)
  - pobieranie informacji (np. `date()`, `time()`, `clock()`)
  - wykrywanie błędów i wydobywanie się z błędów (np. `errno()`)
  - komunikacja sieciowa (np. `connect()`, `send()`, `recv()`)

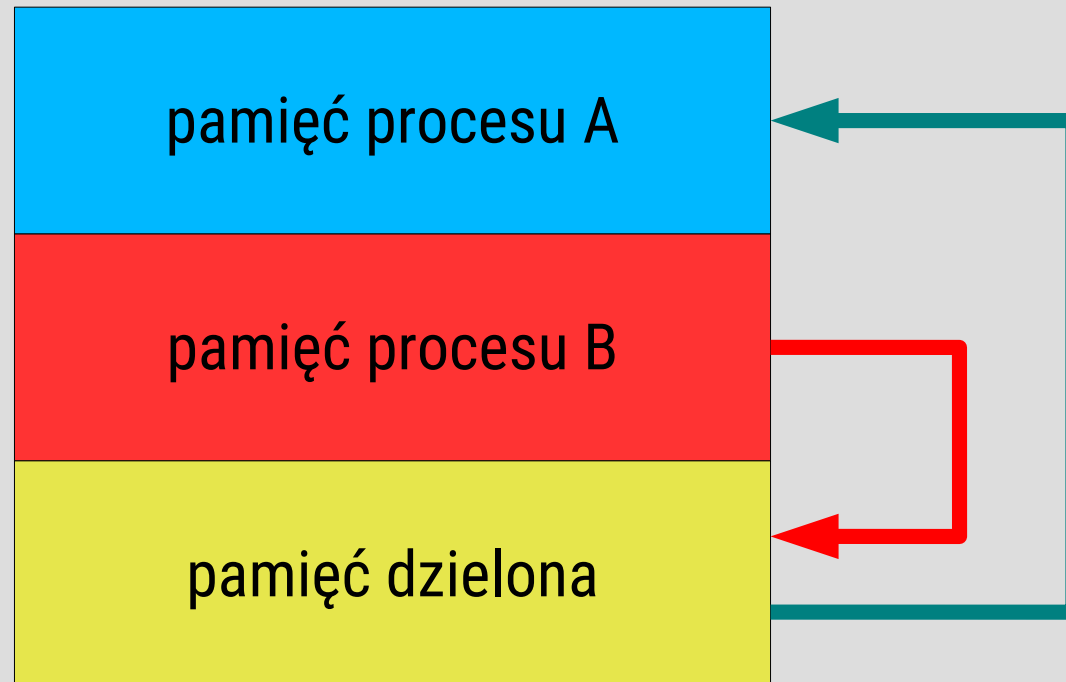
# Usługi systemu operacyjnego

- najczęściej dostępne poprzez wykonanie specjalnego rozkazu uprzywilejowanego
- metody przekazywania parametrów:
  - umieszczanie parametrów w rejestrach procesora
  - umieszczanie parametrów w pamięci i przekazanie adresu
  - umieszczanie parametrów na stosie procesora
    - styl „pascal” (np. API Windows)
    - styl „cdecl”

# komunikacja międzyprocesowa

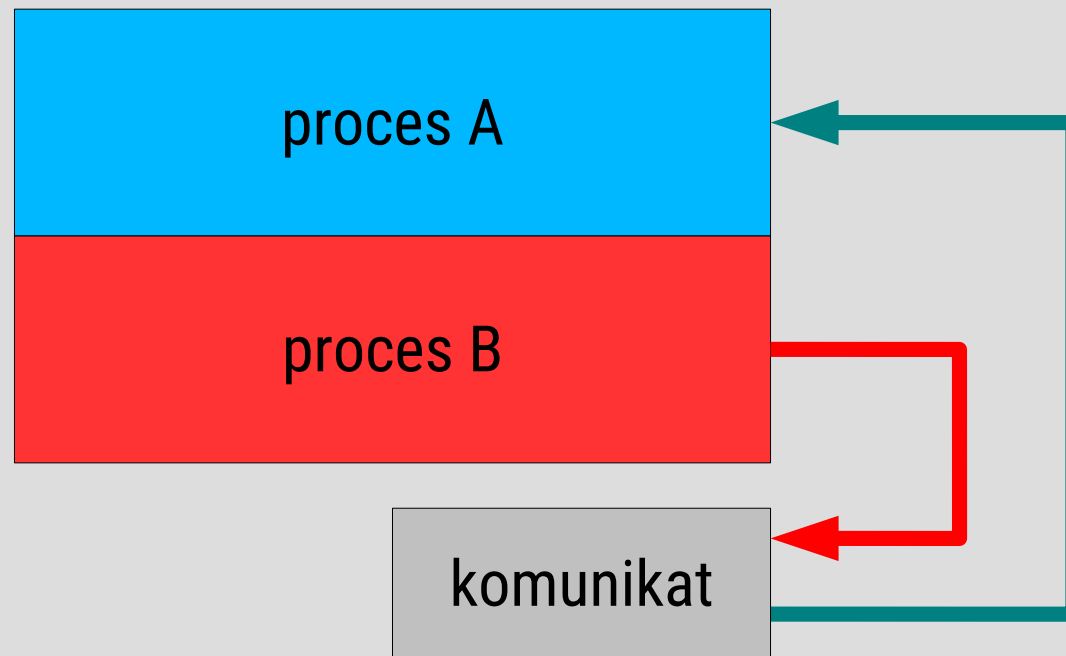
- pamięć dzielona (*shared memory*)
- przekazywanie komunikatów (*message passing*)
- przekazywanie sygnałów (*signals*)
- potoki (*pipelines*)

# pamięć dzielona



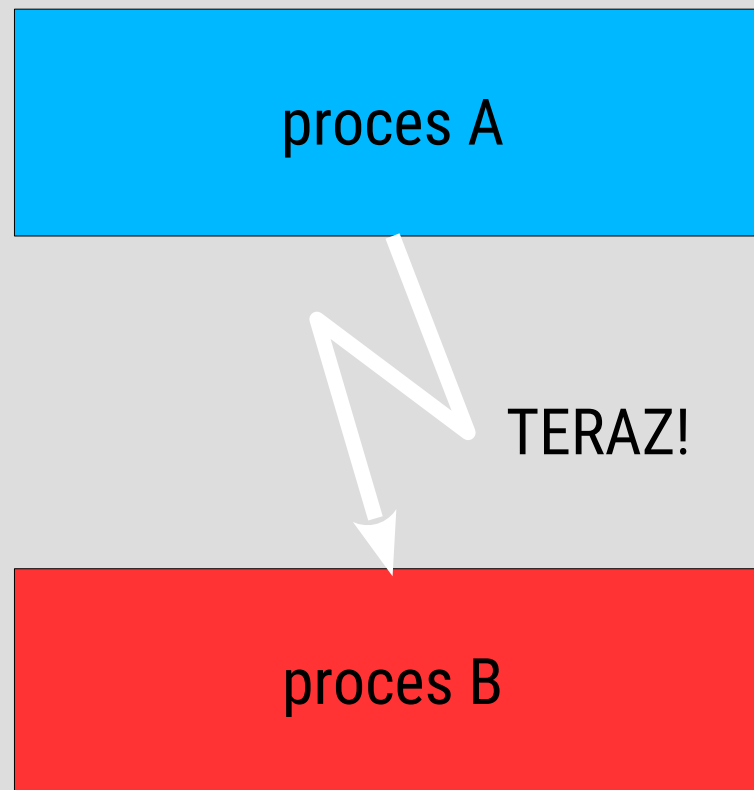
metoda szybka i wydajna, jednak wymaga dodatkowych mechanizmów synchronizacji

# przekazywanie komunikatów



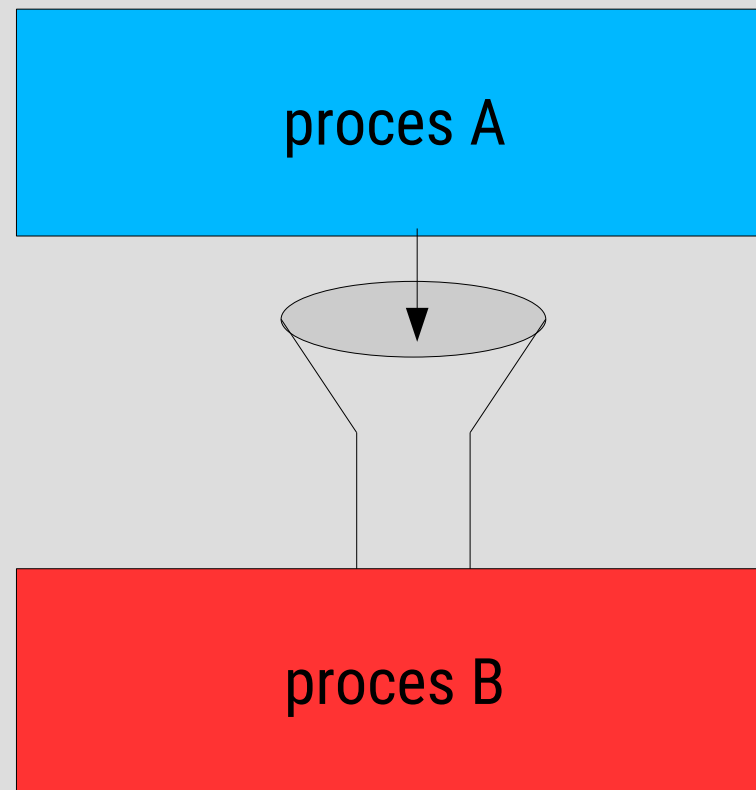
nadaje się do przekazywania niedużych porcji danych

# przekazywanie sygnałów



nadaje się do przekazywania niewielkiego zbioru prostych informacji

# potoki



nadaje się do przekazywania dużych strumieni danych

# funkcje dodatkowe

- rozliczanie (*accounting* )
- wykrywanie incydentów



# programy (narzędzia) systemowe

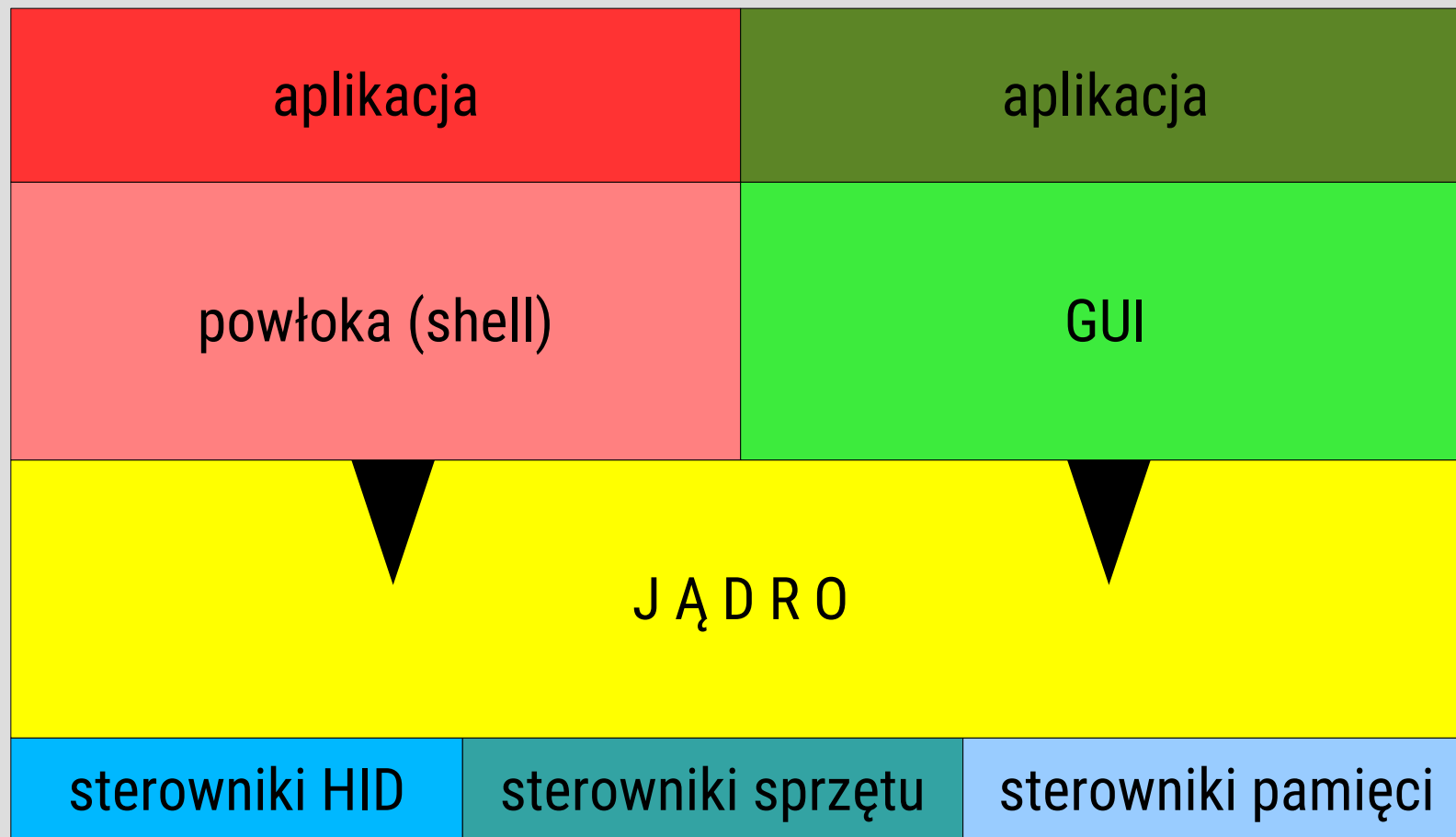
- tworzenie środowiska usprawniającego pracę użytkownika i administratora
  - manipulowanie plikami
  - informowanie o stanie i zasobach systemu
  - tworzenie i modyfikowanie plików (edytory)
  - narzędzia deweloperskie (kompilatory, biblioteki, ...)
  - interpretery języków skryptowych
  - ładowanie i wykonywanie programów
  - komunikacja
  - aplikacje (przeglądarki, programy pocztowe, ...)

# programy (narzędzia) systemowe

## SPOSTRZEŻENIE

Przeciętny użytkownik postrzega system operacyjny poprzez perspektywę **narzędzi** systemowych, a nie **usług**.

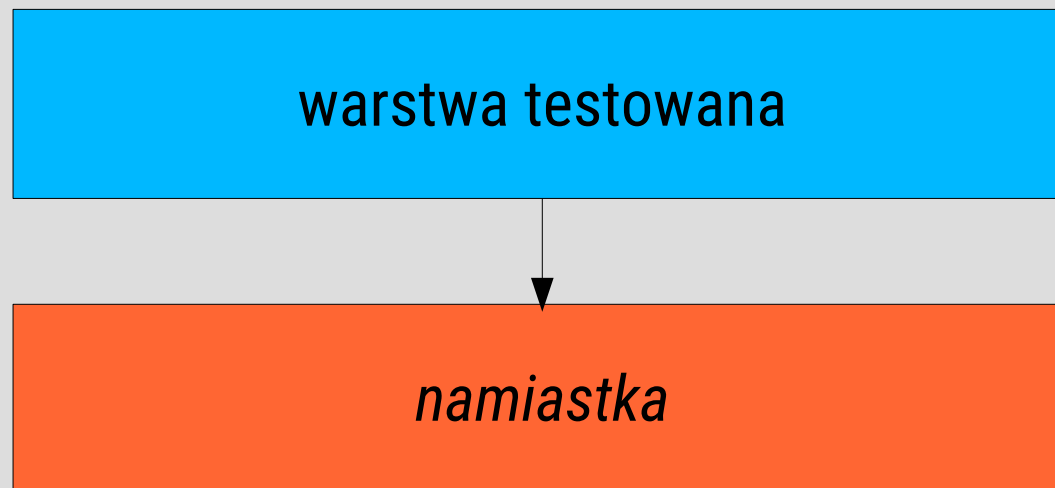
# struktura typowego systemu



# struktura warstwowa

- system projektuje się jako układ warstw (ang. *layers*)
- warstwa najniższa (0) - sprzęt
- warstwa najwyższa (N) – aplikacja
- założenie: warstwa  $i$  używa tylko usług dostarczanych przez warstwę  $i-1$
- modularność
- ukrywanie operacji i danych
- abstrakcje sprzętu i funkcjonalności
- łatwe testowanie i uruchamianie systemu (**namiastki** i **sterowniki**)
- kłopotliwa definicja abstrakcji warstw
- kosztowne narzuty

# struktura warstwowa namiastka



# struktura warstwowa sterownik



# Jądro systemu - koncepcje

- mikrojądro
- jądro monolityczne
- jądro z modułami
- inne koncepcje

# mikrojądro (*microkernel*)

- jądro systemu maksymalnie zredukowane
- mały zbiór funkcji o charakterze elementarnym
- operacje bardziej złożone przeniesione do trybu użytkownika (*user space*)
- komunikacja z jądrem najczęściej poprzez przekazywanie komunikatów
- jądro = warstwa abstrakcji sprzętu (HAL – *hardware abstraction layer*)
- zalety:
  - łatwość rozszerzania i dodawania nowych funkcji
  - łatwość przenoszenia na nowe platformy
  - większa stabilność systemu
  - większe bezpieczeństwo



# jądro monolityczne (*monolithic kernel*)

- jądro systemu zawiera wszystkie – również bardzo abstrakcyjne – funkcje systemu
- wszystkie operacje wykonywane w trybie jądra
- wady
  - mała podatność na modernizacje
  - słaba przenośność
- zalety:
  - szybkość działania
  - wydajność
  - oszczędność w wykorzystaniu pamięci i zasobów

# jądro z modułami

## *(loadable modules kernel)*

- jądro podobne do mikrojądra (podstawowy zbiór usług, najczęściej niezbędnych do wystartowania systemu)
- dynamiczne dołączanie modułów (wykonywanych w trybie jądra) na żądanie lub automatyczne
- usuwanie niepotrzebnych modułów w czasie pracy systemu
- większa elastyczność niż w jądrach monolitycznych
- lepsza wydajność niż w mikrojądrach
- typowe klasy modułów:
  - szyfrowanie, kompresja i kryptografia
  - sterowniki urządzeń fizycznych
  - usługi sieciowe
  - różne systemy plików (inne niż natywne)

# jądro GNU HURD

*(Hird of Unix-Replacing Daemons)*

*(Hurd of Interfaces Representing Depth)*

- zasadniczo oparte o koncepcję mikrojądra
- w ścisłym technicznym znaczeniu HURD nie jest jądrem (zwartym fragmentem kodu), a zespołem współdziałających serwerów (demonów)
- pracę serwerów koordynuje jądro GNU Mach
- Mach jest bardziej rozbudowany niż typowe mikrojądro, ale też znacznie zredukowany w stosunku do jądra monolitycznego

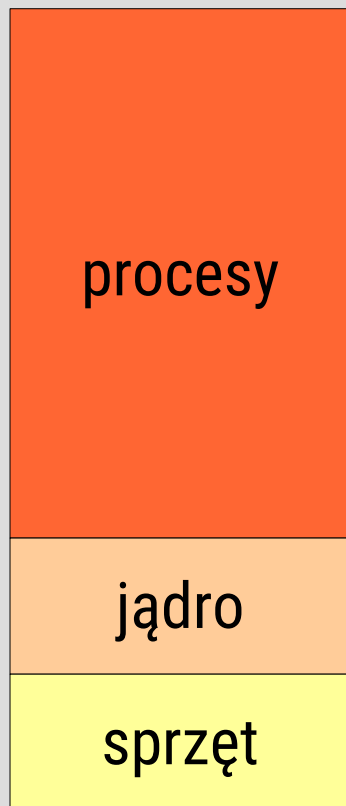
# wirtualizacja

- maszyna wirtualna (*virtual machine*) - logiczna konkluzja podejścia warstwowego
- traktuje sprzęt i funkcje systemowe jako należące do tej samej warstwy
- maszyna wirtualna tworzy interfejs identyczny z podstawowym sprzętem (wirtualna kopia komputera)
- SO tworzy złudzenie wielu procesów pracujących na swych własnych procesorach z własną (wirtualną) pamięcią i własnym sprzętem

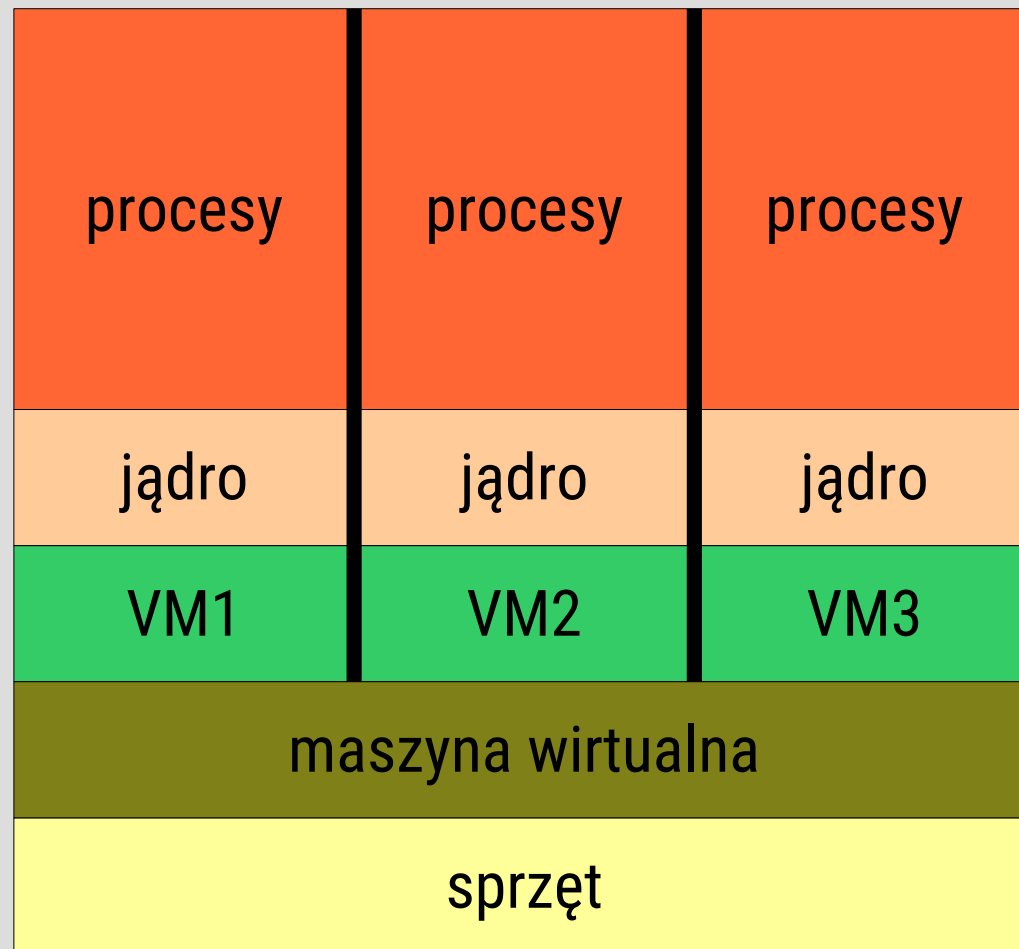
# wirtualizacja

- zasoby fizycznego komputera są dzielone pomiędzy maszyny wirtualne
- odpowiedni *scheduling* CPU daje wrażenie, że każdy użytkownik (np. proces) ma swój własny procesor
- spooling i system plików tworzą wirtualne urządzenia wejścia-wyjścia i wirtualną pamięć pomocniczą

# Środowisko wirtualne



system  
rzeczywisty



system wirtualny

# wirtualizacja - zalety

- pełna ochrona zasobów systemowych - każda maszyna wirtualna jest całkowicie odizolowana od pozostałych
- idealna platforma do badania i rozwoju nowych systemów informatycznych (rozwój systemu dokonuje się na maszynie wirtualnej, zamiast na fizycznej)
- pozwala eksploatować starsze wersje oprogramowania na nowym, niekompatybilnym sprzęcie i systemie operacyjnym

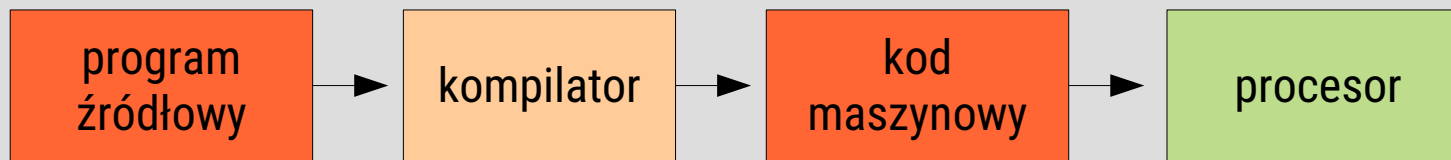
# wirtualizacja - wady

- trudna implementacja – konieczność zrealizowania dokładnej kopii maszyny bazowej
- wzajemna izolacja maszyn wirtualnych uniemożliwia bezpośrednie dzielenie zasobów
- gorsza wydajność (procesor musi obsługiwać wiele maszyn wirtualnych – koszt przełączania kontekstu)

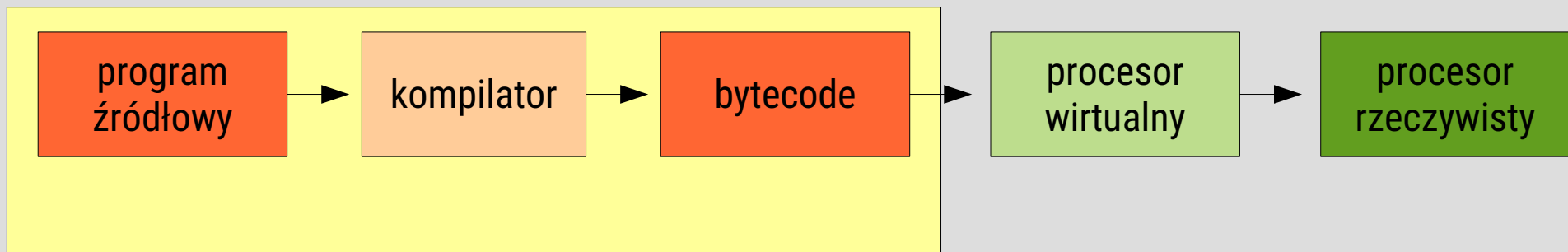


# maszyna wirtualna Javy

## kompilacja tradycyjna



## kompilacja na maszynę wirtualną



# maszyna wirtualna Javy

- kompilator języka Java (dawniej Sun Microsystems, obecnie Oracle) wytwarza niezależny od architektury kod pośredni, tzw. *bytecode*
- *bytecode* wykonywany przez wirtualną maszynę Javy (*Java Virtual Machine – JVM*)
- JVM = specyfikacja abstrakcyjnego komputera (maszyna ze stosem)
- składniki JVM:
  - ładowacz klas (*class loader*)
  - weryfikator klas (*class verifier*)
  - interpreter *bytecode*'u (*runtime interpreter*)

# maszyna wirtualna Javy

- kompilacja w locie (*Just-In-Time* – *JIT*) – przekształcanie bytecode'u w język maszynowy - podniesienie wydajności
- JVM jest (było?) implementowana w przeglądarkach internetowych
- systemy operacyjne oparte na Javie: JavaOS, JX
- telefony komórkowe – często posiadały wbudowane procesory rdzennego kodu Javy
- podejścia konkurencyjne: maszyna Dalvik (Android) + bytecode Dex (Dalvik Executable)
- wada: duża podatność na inżynierię odwrotną (np. Java Decompiler)

# Środowisko CLR dla platformy .NET (MS Windows)

