Algorytmy i struktury danych

Algorytmy sortowania

Sortowanie to jedna z podstawowych operacji w informatyce, polegająca na uporządkowaniu elementów (np. liczb, znaków, obiektów) według określonego porządku – najczęściej rosnąco lub malejąco. Istnieje wiele algorytmów sortowania, które różnią się pod względem złożoności czasowej, przestrzennej oraz sposobu działania. W tej sekcji omówiono najważniejsze z nich.

Sortowanie bąbelkowe (Bubble Sort)

Jeden z najprostszych algorytmów sortowania. Działa poprzez wielokrotne przechodzenie przez listę, porównywanie sąsiadujących elementów i zamienianie ich miejscami, jeśli są w złej kolejności.

Złożoność czasowa:

- Najgorszy przypadek: $O(n^2)$
- Średni przypadek: $O(n^2)$
- Najlepszy przypadek (już posortowane): $O(n)$

Zalety: Prosty do zaimplementowania.
Wady: Nieefektywny dla dużych zbiorów danych.

Sortowanie przez wstawianie (Insertion Sort)

Algorytm działa podobnie do sortowania kart w ręku – elementy są kolejno wstawiane na odpowiednie miejsce w już posortowanej części tablicy.

Złożoność czasowa:

- Najgorszy przypadek: $O(n^2)$
- Najlepszy przypadek: $O(n)$

Zastosowania: Dobre dla małych lub wstępnie posortowanych zbiorów.

Sortowanie przez scalanie (Merge Sort)

Algorytm dzieli dane na mniejsze części, sortuje je rekurencyjnie i scala w posortowaną całość. Przykład algorytmu typu *dziel i zwyciężaj*.

Złożoność czasowa: $O(n \log n)$ w każdym przypadku.
Zalety: Stabilny, gwarantowana złożoność.
Wady: Wymaga dodatkowej pamięci.

Sortowanie szybkie (Quick Sort)

Również algorytm typu *dziel i zwyciężaj*. Wybiera tzw. pivot i dzieli dane na elementy mniejsze i większe, po czym sortuje je rekurencyjnie.
Złożoność czasowa:

- Najlepszy/średni przypadek: $O(n \log n)$
- Najgorszy przypadek: $O(n^2)$

Zalety: Bardzo szybki w praktyce, mała liczba operacji.
Wady: Nie jest stabilny, może mieć zły przypadek bez odpowiedniego wyboru pivota.

Sortowanie kubełkowe (Bucket Sort)

Dzieli dane na kilka przedziałów (kubełków), sortuje każdy kubełek osobno (np. Insertion Sortem), a następnie scala je w całość.

Złożoność czasowa: $O(n + k)$ w optymalnych warunkach, gdzie k to liczba kubełków.
Zastosowania: Działa dobrze dla danych z równomiernym rozkładem.

Algorytm	Najlepszy	Średni	Najgorszy	Stabilność
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Tak
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Tak
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Tak
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Nie
Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	Tak

Table 1: Porównanie popularnych algorytmów sortowania

Porównanie algorytmów

Dynamiczne struktury danych

Dynamiczne struktury danych to takie, które mogą zmieniać swój rozmiar w trakcie działania programu – w przeciwieństwie do struktur statycznych, takich jak tablice o stałej długości. Dzięki temu umożliwiają bardziej elastyczne zarządzanie pamięcią oraz dostosowywanie się do zmiennej ilości danych.

Listy jednokierunkowe

Lista jednokierunkowa składa się z węzłów (ang. nodes), z których każdy zawiera dane oraz wskaźnik na kolejny element. Operacje takie jak wstawianie i usuwanie są proste i efektywne, szczególnie na początku listy.
Zalety:

- Elastyczny rozmiar
- Łatwe wstawianie i usuwanie elementów

Wady:

- Brak bezpośredniego dostępu do elementów (trzeba przejść listę od początku)
- Trudniejsze zarządzanie w porównaniu z tablicą

Listy dwukierunkowe

Każdy węzeł zawiera wskaźnik do następnego i poprzedniego elementu, co umożliwia poruszanie się w obu kierunkach.
Zastosowanie: Kolejki, edytory tekstu (np. historia zmian), systemy zarządzania pamięcią.

Stosy (LIFO)

Struktura typu Last In, First Out. Ostatni element dodany do stosu jest pierwszym, który zostanie usunięty. Operacje: `push()` (dodanie) i `pop()` (usunięcie).
Zastosowania: Rekurencja, odwracanie ciągów, analiza nawiasów.

Kolejki (FIFO)

Struktura typu First In, First Out. Elementy są przetwarzane w kolejności, w jakiej zostały dodane. Podstawowe operacje: `enqueue()` (dodanie) i `dequeue()` (usunięcie).
Warianty:

- Kolejki priorytetowe – elementy mają przypisany priorytet
- Kolejki cykliczne – do zarządzania buforami

Drzewa

Hierarchiczna struktura danych. Każdy element (węzeł) ma zero lub więcej dzieci. Najczęściej spotykane typy drzew:

- **Drzewa binarne (BST)** – każdy węzeł ma maksymalnie dwóch potomków.
- **Drzewa AVL** – samobalansujące drzewa binarne zapewniające złożoność $O(\log n)$ dla wyszukiwania, wstawiania i usuwania.

- **Drzewa B** – stosowane w systemach baz danych do efektywnego zarządzania dużą ilością danych.

Grafy

Ogólna struktura danych składająca się z wierzchołków i krawędzi. Może reprezentować zarówno relacje skierowane, jak i nieskierowane.
Zastosowania: Sieci komputerowe, grafy społeczne, algorytmy trasowania, modelowanie relacji.

Hash tables

Struktura pozwalająca na bardzo szybki dostęp do danych przez klucz. Klucze są przekształcane za pomocą funkcji mieszającej (hashującej) na indeksy tablicy.
Zalety:

- Bardzo szybki dostęp (średnio $O(1)$)
- Efektywne dla dużych zbiorów danych

Wady:

- Możliwość kolizji – konieczność ich obsługi (np. przez łańcuchowanie lub adresowanie otwarte)
- Wydajność zależna od jakości funkcji hashującej

Klasy złożoności algorytmów

Złożoność algorytmu określa, jak bardzo rosną zasoby wymagane do jego wykonania (czas, pamięć) w zależności od rozmiaru danych wejściowych. Klasy złożoności służą do grupowania problemów algorytmicznych według trudności ich rozwiązania.

Złożoność obliczeniowa

Najczęściej analizuje się złożoność czasową, czyli liczbę operacji wykonywanych przez algorytm w zależności od rozmiaru wejścia n. Do zapisu używa się notacji **O-wielkie** ($O(n)$), która określa górne ograniczenie liczby operacji.
Przykładowe klasy złożoności czasowej:

- $O(1)$ – złożoność stała
- $O(\log n)$ – logarytmiczna (np. wyszukiwanie binarne)
- $O(n)$ – liniowa
- $O(n \log n)$ – np. szybkie sortowanie, merge sort
- $O(n^2)$, $O(n^3)$ – wielomianowa
- $O(2^n)$, $O(n!)$ – wykładnicza i silniowa (dla problemów NP-trudnych)

Notacje asymptotyczne

- $\mathbf{O(f(n))}$ – górne ograniczenie (najgorszy przypadek)
- $\Omega(\mathbf{f(n)})[\mathbf{Omega}]$ – dolne ograniczenie (najlepszy przypadek)
- $\Theta(\mathbf{f(n)})[\mathbf{Theta}]$ – ścisłe ograniczenie (średni przypadek)

Klasy złożoności problemów

Klasy złożoności to grupy problemów decyzyjnych o podobnej trudności obliczeniowej.

- **P (Polynomial time)** – problemy, które można rozwiązać algorytmem w czasie wielomianowym.
- **NP (Nondeterministic Polynomial time)** – problemy, których rozwiązanie można zweryfikować w czasie wielomianowym.
- **NP-complete** – najtrudniejsze problemy z klasy NP; jeśli dla jednego z nich znajdzie się algorytm działający w czasie wielomianowym, to dla wszystkich innych też.
- **NP-hard** – problemy co najmniej tak trudne jak problemy NP, ale niekoniecznie należące do NP (np. problemy optymalizacyjne).

Przykłady problemów

- **P:** sortowanie (Merge Sort), znajdowanie największego elementu, BFS, DFS.
- **NP:** problem spełnialności (SAT), problem komiwojażera (TSP), problem kolorowania grafu.
- **NP-trudne:** wersje optymalizacyjne TSP i SAT, problem rozkładania zajęć na planie lekcji.

Znaczenie klas złożoności

Zrozumienie klas złożoności jest kluczowe dla projektowania efektywnych algorytmów oraz określania, czy dany problem można rozwiązać w rozsądnym czasie dla dużych danych wejściowych. Pozwala to również odróżnić problemy, dla których warto szukać algorytmu, od tych, gdzie lepiej używać aproksymacji lub heurystyk.

Programowanie

Języki programowania

Języki programowania to formalne systemy służące do tworzenia instrukcji, które mogą być wykonane przez komputer. Różne języki oferują różne paradygmaty, składnie i przeznaczenie, dlatego dobór odpowiedniego języka zależy od rodzaju projektu, wydajności, dostępnych bibliotek czy wygody programisty.

Podział języków programowania

- **Języki niskiego poziomu** – bliskie językowi maszynowemu (np. asembler), pozwalają na pełną kontrolę nad sprzętem.
- **Języki wysokiego poziomu** – bardziej abstrakcyjne, zbliżone do języka naturalnego, łatwiejsze w pisaniu i utrzymaniu (np. Python, Java, C++).
- **Języki interpretowane vs kompilowane:**
  - *Interpretowane* – kod wykonywany linia po linii (np. Python, JavaScript).
  - *Kompilowane* – kod źródłowy tłumaczony do kodu maszynowego przed uruchomieniem (np. C, C++).

Paradygmaty programowania

- **Proceduralny** – program jako zbiór funkcji i instrukcji wykonywanych krok po kroku (np. C, Pascal).
- **Obiektowy** – programy budowane wokół obiektów i klas (np. Java, C++, Python).
- **Funkcyjny** – skupia się na wyrażeniach matematycznych i funkcjach bez efektów ubocznych (np. Haskell, Scala).
- **Deklaratywny** – opisuje, *co* program ma zrobić, nie *jak* (np. SQL, Prolog).

Przykłady języków i ich zastosowania

- **C/C++** – wydajne, niskopoziomowe języki, używane m.in. w systemach wbudowanych, grach, sterownikach.
- **Python** – prosty i uniwersalny język do prototypowania, analizy danych, AI, automatyzacji.
- **Java** – popularny język obiektowy, wykorzystywany w aplikacjach webowych, mobilnych (Android), systemach backendowych.
- **JavaScript** – kluczowy język front-endu w aplikacjach webowych (HTML + CSS + JS), ale także w back-endzie (Node.js).
- **C#** – rozwijany przez Microsoft, używany w aplikacjach desktopowych, grach (Unity), systemach enterprise.
- **Go (Golang)** – wydajny język do systemów rozproszonych, sieciowych, mikroserwisów.
- **Rust** – nowoczesny język z naciskiem na bezpieczeństwo pamięci i wydajność.

Przykłady użycia

- **Python:** skrypty, automatyzacja, machine learning, aplikacje webowe (Django, Flask).
- **JavaScript:** dynamiczne strony internetowe, interfejsy użytkownika.
- **C++:** gry komputerowe (Unreal Engine), sterowniki, silniki fizyki.
- **Java:** aplikacje korporacyjne, aplikacje mobilne (Android).
- **SQL:** operacje na bazach danych.

Wybór języka

Wybór języka zależy od wielu czynników:

- Typ aplikacji (np. web, embedded, AI)
- Wydajność i zasobożerność
- Ekosystem i dostępność bibliotek
- Krzywa uczenia się
- Wymagania zespołu/projektu

Środowiska programistyczne

Środowiska programistyczne (IDE – *Integrated Development Environments*) to zintegrowane aplikacje służące do tworzenia, testowania i debugowania oprogramowania. Łączą one edytor kodu źródłowego, narzędzia do kompilacji, debugowania, a często również zarządzania wersjami oraz automatyzacji budowy projektu.

Cechy środowisk programistycznych

- **Edytor kodu źródłowego** – z kolorowaniem składni, autouzupełnianiem, nawigacją po kodzie.
- **Kompilator / interpreter** – umożliwia tłumaczenie kodu na język maszynowy lub wykonywanie go w czasie rzeczywistym.
- **Debugger** – narzędzie do wykrywania błędów i analizowania działania programu krok po kroku.
- **Zarządzanie projektem** – integracja z systemami budowania (np. Maven, Gradle, Make) oraz kontrolą wersji (np. Git).
- **Wtyczki i rozszerzenia** – możliwość rozbudowy funkcjonalności IDE.

Popularne środowiska programistyczne

- **Visual Studio Code** – lekkie, rozbudowane edytorowe środowisko od Microsoftu, z dużym ekosystemem wtyczek, obsługą wielu języków (m.in. JavaScript, Python, C++).
- **IntelliJ IDEA** – zaawansowane IDE dla Javy i innych języków JVM (Kotlin, Scala), cenione za rozbudowane funkcje refaktoryzacji i wsparcie dla dużych projektów.
- **PyCharm** – środowisko specjalizowane dla Pythona, zawiera wsparcie dla Django, nauki danych, testowania jednostkowego.
- **Eclipse** – rozbudowane, wieloplatformowe środowisko, często wykorzystywane w projektach Java i C/C++, wspiera pluginy i rozszerzenia.
- **NetBeans** – zintegrowane środowisko dla Javy, C/C++, PHP, wspierane przez Apache.
- **Xcode** – IDE Apple do tworzenia aplikacji na macOS, iOS, iPadOS, watchOS i tvOS (Swift, Objective-C).
- **Android Studio** – oparte na IntelliJ, przeznaczone do tworzenia aplikacji na Androida, zawiera emulator, analizator kodu, narzędzia graficzne.
- **CLion** – środowisko od JetBrains do języków C i C++, z wbudowanym debugerem i analizą kodu.
- **Atom / Sublime Text** – lekkie edytory kodu z możliwością rozbudowy o funkcje IDE.

Kryteria wyboru środowiska

- **Język programowania** – niektóre IDE są dostosowane do konkretnych języków.
- **Typ projektu** – różne środowiska są zoptymalizowane pod aplikacje mobilne, webowe, embedded itp.
- **Integracje i wsparcie dla technologii** – np. frameworki webowe, systemy kontroli wersji, bazy danych.
- **Wydajność i zasobożerność** – ważne w dużych projektach lub przy słabszych komputerach.
- **Preferencje użytkownika** – wygoda interfejsu, dostępność skrótów, personalizacja.

IDE vs edytory kodu

- **IDE** oferują kompleksowy zestaw narzędzi w jednej aplikacji, często kosztem większego zużycia zasobów.
- **Edytory kodu** (jak VS Code, Sublime) są lekkie i szybkie, ale mogą wymagać konfiguracji i instalacji rozszerzeń, by osiągnąć funkcjonalność IDE.

Cechy programowania obiektowego

Programowanie obiektowe (OOP – *Object-Oriented Programming*) to paradygmat programowania, w którym głównymi elementami programu są obiekty. Obiekt to struktura zawierająca zarówno dane, jak i funkcje operujące na tych danych. OOP jest oparte na kilku fundamentalnych cechach, które umożliwiają tworzenie elastycznych, łatwych do rozbudowy i utrzymania programów.

Abstrakcja

Abstrakcja polega na ukrywaniu szczegółów implementacyjnych i udostępnianiu jedynie istotnych informacji na temat obiektu. Dzięki temu programista może skupić się na tym, *co* obiekt robi, zamiast na tym, *jak* to robi.
Przykład: Obiekt `Samochod` może posiadać metody takie jak `uruchomSilnik()` lub `zatrzymaj()`, ale szczegóły implementacji działania silnika są ukryte.

Enkapsulacja

Enkapsulacja polega na ukrywaniu danych obiektu i zapewnieniu dostępu do nich tylko poprzez określone metody. Ochrona danych pozwala na kontrolowanie, w jaki sposób są one modyfikowane, co zapobiega nieautoryzowanemu dostępowi.
Przykład: Zmienna `saldo` w obiekcie `KontoBankowe` jest ukryta, a jej wartość może być modyfikowana tylko poprzez metody takie jak `depozyt()` lub `wyplata()`.

Dziedziczenie

Dziedziczenie pozwala na tworzenie nowych klas na podstawie istniejących klas (nadrzędnych), co umożliwia wielokrotne użycie kodu. Klasa podrzędna (dziedzicząca) może rozszerzać lub modyfikować zachowanie klasy nadrzędnej.
Przykład: Klasa `Pojazd` może mieć wspólne właściwości (np. `kolory`, `prędkość`), a klasy `Samochod` i `Motocykl` mogą dziedziczyć te właściwości, dodając swoje specyficzne metody (np. `otwórzBagażnik()` w `Samochod`).

Polimorfizm

Polimorfizm umożliwia obiektom różnych klas reagowanie na te same polecenia w różny sposób. Dzięki temu można traktować obiekty różnych klas w sposób jednolity, bez potrzeby dokładnego sprawdzania, jakiego typu są obiekty.
Przykład: Metoda `wyświetlInformacje()` może być zaimplementowana w różnych klasach (np. `Samochod`, `Motocykl`), ale jej wywołanie na obiekcie tych klas spowoduje inne działanie w zależności od typu obiektu.

Kapsułkowanie

Kapsułkowanie jest często używane zamiennie z enkapsulacją i oznacza proces organizowania kodu w jednostki, które są samodzielne i posiadają zdefiniowane interfejsy. Pozwala to na zarządzanie złożonością systemu poprzez podzielenie go na mniejsze, łatwiejsze do kontrolowania części.

Reużywalność kodu

Programowanie obiektowe sprzyja tworzeniu kodu, który można wielokrotnie wykorzystywać w różnych częściach programu lub nawet w różnych projektach. Dzięki dziedziczeniu i polimorfizmowi możliwe jest ponowne wykorzystanie klas bez konieczności ponownego pisania tych samych fragmentów kodu.

Przykłady w praktyce

- **Klasa i obiekt:** Klasa to szablon, obiekt to konkretna instancja tej klasy. Na przykład, klasa `Pracownik` może zawierać atrybuty takie jak `imie`, `nazwisko`, `stanowisko`, a obiekty tej klasy będą przechowywać konkretne dane.
- **Dziedziczenie i rozszerzenia:** Klasa `Figura` może być klasą bazową, a klasy `Kolo` i `Prostokat` dziedziczą jej metody, ale dodają własne (np. `obliczPole()`).
- **Polimorfizm:** Metoda `rysuj()` w klasie `Figura` może zostać nadpisana w klasach dziedziczących, aby dostarczyć specyficzną implementację dla każdej figury.

Korzyści z programowania obiektowego

Programowanie obiektowe ułatwia tworzenie złożonych aplikacji, ponieważ pozwala na modularne podejście do kodu. Obiekty są autonomiczne, co ułatwia zarządzanie nimi i umożliwia ich łatwą wymianę lub modyfikację w miarę rozwoju projektu. OOP sprzyja również testowalności i utrzymaniu aplikacji w dłuższym okresie czasu, ponieważ zmiany w jednej części systemu mają minimalny wpływ na inne części.

Systemy operacyjne

Funkcje systemów operacyjnych

System operacyjny (OS) to oprogramowanie, które zarządza zasobami komputerowymi oraz zapewnia usługi niezbędne do uruchomienia aplikacji. Jego zadaniem jest kontrolowanie i koordynowanie użycia sprzętu komputerowego przez różne aplikacje i procesy. Funkcje systemu operacyjnego można podzielić na kilka głównych kategorii, które są kluczowe dla efektywnego działania systemu komputerowego.

Zarządzanie procesami

System operacyjny odpowiedzialny jest za tworzenie, planowanie, zarządzanie i kończenie procesów. Proces to program w trakcie wykonywania, który posiada swój stan, dane oraz przestrzeń pamięci. Kluczowe zadania związane z zarządzaniem procesami to:

- **Tworzenie procesów** – system operacyjny tworzy nowe procesy w odpowiedzi na działania użytkownika lub aplikacji.
- **Zarządzanie stanami procesów** – procesy mogą przechodzić przez różne stany, takie jak gotowy, uruchomiony, oczekujący.
- **Planowanie procesów** – OS decyduje, który proces będzie wykonywany w danym momencie, zarządzając tym samym dostępem do procesora.
- **Synchronizacja procesów** – zapewnia koordynację między współbieżnymi procesami, aby zapobiec konfliktom przy dostępie do wspólnych zasobów.
- **Zakończenie procesów** – system operacyjny odpowiedzialny jest za prawidłowe zakończenie procesów po ich wykonaniu.

Zarządzanie pamięcią

Zarządzanie pamięcią w systemie operacyjnym obejmuje alokację, śledzenie i zwalnianie pamięci, a także zapewnienie, że procesy nie ingerują w przestrzeń pamięci innych procesów. Do głównych zadań systemu operacyjnego w zakresie zarządzania pamięcią należy:

- **Alokacja pamięci** – system operacyjny przypisuje odpowiednią ilość pamięci dla procesów, dbając o jej efektywne wykorzystanie.
- **Pamięć wirtualna** – umożliwia korzystanie z większej ilości pamięci, niż fizycznie dostępna, przez wykorzystanie pamięci na dyskach twardych jako rozszerzenia RAM.
- **Zarządzanie stronami pamięci** – system dzieli pamięć na małe jednostki zwane stronami i zarządza ich przenoszeniem między pamięcią główną a pamięcią wirtualną.
- **Zwalnianie pamięci** – system operacyjny odpowiada za zwolnienie pamięci zajmowanej przez procesy po ich zakończeniu.

Zarządzanie urządzeniami wejścia/wyjścia

System operacyjny kontroluje dostęp do urządzeń wejścia/wyjścia (I/O), takich jak klawiatury, myszy, drukarki czy dyski. Funkcje zarządzania urządzeniami I/O obejmują:

- **Interfejs do urządzeń I/O** – OS udostępnia interfejsy umożliwiające aplikacjom komunikowanie się z urządzeniami zewnętrznymi.
- **Sterowniki urządzeń** – system operacyjny używa sterowników, które umożliwiają komunikację między urządzeniem a systemem.
- **Buforowanie I/O** – dane przesyłane między urządzeniami a systemem są buforowane, aby poprawić wydajność operacji I/O.
- **Zarządzanie dostępem do urządzeń** – OS zapewnia, że urządzenia są używane przez jeden proces w danym momencie, zapobiegając kolizjom.

Zarządzanie plikami

System operacyjny zapewnia mechanizmy zarządzania plikami, umożliwiając tworzenie, modyfikowanie, usuwanie i organizowanie plików w systemie. Do głównych funkcji związanych z zarządzaniem plikami należą:

- **System plików** – system operacyjny organizuje dane na dyskach w postaci plików i katalogów, umożliwiając ich przechowywanie i łatwe odnajdywanie.
- **Operacje na plikach** – system operacyjny obsługuje operacje takie jak odczyt, zapis, kopiowanie, usuwanie oraz zmiana nazw plików.
- **Uprawnienia do plików** – OS kontroluje, które procesy i użytkownicy mają dostęp do danych w plikach oraz jakie operacje mogą na nich wykonać (np. odczyt, zapis, wykonanie).
- **Zarządzanie przestrzenią dyskową** – system operacyjny monitoruje dostępność miejsca na dyskach oraz zarządza alokacją przestrzeni dla plików.

Zarządzanie bezpieczeństwem

Bezpieczeństwo systemu operacyjnego jest kluczowe dla ochrony danych i zapobiegania nieautoryzowanemu dostępowi. Do funkcji bezpieczeństwa należą:

- **Uwierzytelnianie użytkowników** – system operacyjny weryfikuje tożsamość użytkowników, np. poprzez logowanie za pomocą haseł lub biometrii.
- **Kontrola dostępu** – system operacyjny kontroluje, którzy użytkownicy i procesy mogą uzyskać dostęp do danych, urządzeń czy usług.
- **Zarządzanie uprawnieniami** – OS przypisuje odpowiednie uprawnienia do plików i zasobów, zapewniając, że tylko upoważnieni użytkownicy mają dostęp do wrażliwych danych.
- **Szyfrowanie danych** – system operacyjny może implementować mechanizmy szyfrowania, aby chronić dane przed nieautoryzowanym dostępem.

Zarządzanie siecią

System operacyjny zarządza komunikacją między komputerami w sieci, umożliwiając wymianę danych oraz współdzielenie zasobów. Do funkcji zarządzania siecią należą:

- **Obsługa protokołów sieciowych** – OS obsługuje protokoły takie jak TCP/IP, które umożliwiają wymianę danych między komputerami w sieci.
- **Zarządzanie połączeniami** – system operacyjny zapewnia tworzenie i kontrolowanie połączeń sieciowych, takich jak połączenia TCP.
- **Zarządzanie adresami IP** – OS przydziela i zarządza adresami IP dla komputerów w sieci.
- **Ochrona sieci** – system operacyjny monitoruje ruch sieciowy i stosuje zabezpieczenia, takie jak zapory ogniowe (firewalle), aby chronić system przed atakami.

Zarządzanie pamięcią

Zarządzanie pamięcią jest jednym z kluczowych zadań systemu operacyjnego, które polega na efektywnym alokowaniu, monitorowaniu i zwalnianiu pamięci dla uruchomionych procesów. Celem jest zapewnienie, że procesy mogą korzystać z pamięci w sposób bezpieczny i efektywny, unikając zarówno niepotrzebnego marnotrawstwa zasobów, jak i kolizji między procesami.

Rodzaje pamięci

W systemach operacyjnych pamięć jest zazwyczaj podzielona na różne rodzaje, które pełnią różne funkcje:

- **Pamięć operacyjna (RAM)** – pamięć o szybkim dostępie, w której przechowywane są aktywne procesy i dane. Jest to pamięć ulotna, co oznacza, że dane zostaną utracone po wyłączeniu komputera.

- **Pamięć wirtualna** – technologia, która umożliwia systemowi operacyjnemu korzystanie z przestrzeni dyskowej jako rozszerzenia pamięci RAM. Dzięki pamięci wirtualnej, procesy mogą pracować w większej przestrzeni pamięci, niż jest fizycznie dostępna.
- **Pamięć cache** – pamięć o bardzo szybkim dostępie, która jest używana do przechowywania danych i instrukcji, które są często wykorzystywane przez procesor. Cache jest używana, by przyspieszyć dostęp do danych, które normalnie musiałyby być pobierane z RAM.
- **Pamięć ROM** – pamięć tylko do odczytu, w której przechowywane są dane stałe, takie jak BIOS lub firmware.

Alokacja pamięci

Alokacja pamięci polega na przydzielaniu obszarów pamięci dla procesów. Istnieją różne techniki zarządzania pamięcią, które umożliwiają jej efektywne wykorzystanie:

- **Alokacja statyczna** – pamięć jest przydzielana procesowi na stałe podczas jego tworzenia. W tym przypadku proces ma określoną wielkość pamięci, która nie zmienia się w czasie jego życia.
- **Alokacja dynamiczna** – pamięć jest przydzielana procesowi w miarę potrzeb. Proces może żądać dodatkowej pamięci lub zwolnić już niepotrzebną przestrzeń w trakcie swojego działania.
- **Podział pamięci** – technika polegająca na podzieleniu dostępnej pamięci na segmenty, z których każdy jest przydzielany innemu procesowi. Może to być podział na stałe fragmenty (partycje) lub dynamiczny.

Zarządzanie pamięcią wirtualną

Pamięć wirtualna pozwala na wykorzystanie przestrzeni dyskowej jako rozszerzenia pamięci RAM, co daje większą elastyczność w zarządzaniu zasobami systemu. System operacyjny dzieli pamięć wirtualną na jednostki zwane stronami, które mogą być przenoszone między pamięcią RAM a przestrzenią na dysku twardym, znaną jako *pliki wymiany* (ang. swap files).

- **Stronicowanie** – pamięć wirtualna jest dzielona na małe jednostki o stałej wielkości zwane stronami. Strony są ładowane do pamięci RAM w miarę potrzeby, a te, które nie są aktualnie używane, mogą być zapisane na dysku. Stronicowanie pozwala na efektywne wykorzystanie dostępnej pamięci i zmniejszenie wpływu braku pamięci RAM.
- **Segmentacja** – technika, w której pamięć wirtualna jest dzielona na segmenty o zmiennej wielkości, zależnie od potrzeb procesów. Każdy segment może reprezentować różne rodzaje danych, takie jak kod programu, dane czy stos.
- **Swapping** – proces przenoszenia stron lub całych segmentów między pamięcią RAM a przestrzenią wymiany na dysku. Zjawisko to może prowadzić do tzw. *thrashingu*, czyli sytuacji, w której system traci wydajność przez częste operacje przenoszenia danych.

Zarządzanie przestrzenią adresową

Każdy proces w systemie operacyjnym ma przypisaną swoją przestrzeń adresową, która pozwala mu na dostęp do pamięci w sposób izolowany od innych procesów. Izolacja ta zapobiega nieautoryzowanemu dostępowi do danych innych procesów.

- **Pamięć fizyczna** – odnosi się do rzeczywistej pamięci RAM zainstalowanej w systemie.
- **Pamięć wirtualna** – jest abstrakcją pamięci, którą procesy widzą, a która może obejmować zarówno pamięć RAM, jak i przestrzeń wymiany na dysku.
- **Tablice stron** – służą do tłumaczenia wirtualnych adresów pamięci na fizyczne. Kiedy proces żąda dostępu do pamięci, system operacyjny sprawdza tablicę stron, aby określić, gdzie znajduje się odpowiednia strona w pamięci.
- **Tłumaczenie adresów** – proces, w którym system operacyjny mapuje adresy wirtualne na adresy fizyczne. Proces ten odbywa się w jednostce zarządzającej pamięcią, zwanej MMU (Memory Management Unit).

Zarządzanie fragmentacją pamięci

Fragmentacja pamięci to problem polegający na tym, że pomimo dostępności wolnej przestrzeni w pamięci, może być ona nieużyteczna z powodu rozproszenia wolnych bloków. Istnieją dwa rodzaje fragmentacji:

- **Fragmentacja wewnętrzna** – występuje, gdy w przydzielonym obszarze pamięci pozostaje wolne miejsce, które nie może zostać wykorzystane, ponieważ jest za małe na wymagany blok danych.
- **Fragmentacja zewnętrzna** – występuje, gdy pamięć jest podzielona na wiele małych, wolnych bloków, które łącznie stanowią wystarczającą ilość pamięci, ale żaden z tych bloków nie jest wystarczająco duży, aby pomieścić nowy proces.

Aby rozwiązać problem fragmentacji, systemy operacyjne stosują różne techniki, takie jak:

- **Kompaktowanie pamięci** – proces, który przenosi dane w pamięci w taki sposób, aby utworzyć jeden duży, wolny blok pamięci, który może być użyty przez nowy proces.
- **Alokacja z defragmentacją** – zmiana sposobu przydzielania pamięci, aby zminimalizować fragmentację.

Zarządzanie pamięcią współbieżnych procesów

System operacyjny zapewnia mechanizmy, które pozwalają wielu procesom na współdzielenie pamięci, zachowując jednocześnie izolację ich danych. Ważne mechanizmy zarządzania pamięcią w przypadku współbieżnych procesów to:

- **Segmentacja pamięci** – umożliwia procesom dostęp do wspólnych zasobów, takich jak biblioteki czy zmienne globalne, zachowując izolację w innych częściach pamięci.
- **Pamięć współdzielona** – pozwala na dzielenie się pamięcią między różnymi procesami, co jest szczególnie przydatne w aplikacjach wymagających dużej wydajności.
- **Mechanizmy synchronizacji** – pozwalają na bezpieczny dostęp do współdzielonej pamięci, zapobiegając problemom takim jak warunki wyścigu.

Algorytmy szeregowania zadań

Szeregowanie zadań jest kluczowym aspektem zarządzania procesami w systemach operacyjnych, mającym na celu optymalne przypisanie dostępnych zasobów (w szczególności procesora) do wykonywanych procesów. Algorytmy szeregowania decydują, który proces w danym momencie uzyska dostęp do procesora, co ma bezpośredni wpływ na efektywność systemu oraz czas reakcji aplikacji.

Cele algorytmu szeregowania

Celem algorytmu szeregowania jest:

- **Minimalizacja czasu oczekiwania** – skrócenie czasu, jaki procesy muszą czekać w kolejce przed rozpoczęciem wykonania.
- **Minimalizacja czasu obrotu** – czas obrotu to czas od momentu przyjścia procesu do momentu jego zakończenia, w tym czas oczekiwania oraz czas wykonania.
- **Zrównoważenie obciążenia procesora** – zapewnienie, że procesor jest jak najlepiej wykorzystany, bez nadmiernych opóźnień lub bezczynności.
- **Sprawiedliwość** – zapewnienie, że wszystkie procesy otrzymują sprawiedliwy dostęp do procesora, unikając tzw. „głodowania” (starvation), kiedy niektóre procesy nie mogą uzyskać dostępu do procesora przez długi czas.

Rodzaje algorytmów szeregowania

- **Szeregowanie FIFO (First In, First Out)** – jest jednym z najprostszych algorytmów, który działa na zasadzie „pierwszy wchodzi, pierwszy wychodzi". Procesy są wykonywane w kolejności ich przyjścia do systemu. Mimo swojej prostoty, algorytm FIFO może prowadzić do dużych czasów oczekiwania, zwłaszcza gdy procesy różnią się czasem wykonania.
- **Szeregowanie z priorytetami** – w tym algorytmie każdy proces ma przypisany priorytet, a procesy o wyższym priorytecie są wykonywane przed procesami o niższym priorytecie. Priorytet może być statyczny (ustalony na początku) lub dynamiczny (może się zmieniać w trakcie działania procesu). Problemem tego algorytmu jest możliwość „głodowania" procesów o niskim priorytecie, które nigdy nie otrzymują dostępu do procesora.
- **Szeregowanie krótkiego procesu (SJF - Shortest Job First)** – w tym algorytmie proces, który ma najkrótszy czas wykonania, jest wykonywany jako pierwszy. SJF jest efektywny w minimalizowaniu średniego czasu oczekiwania, ale wymaga znajomości czasu wykonywania procesów, co w praktyce bywa trudne do przewidzenia. Może również prowadzić do problemu głodowania długich procesów.
- **Szeregowanie z preempcją (Preemptive Scheduling)** – w tym algorytmie procesy mogą być przerwane, jeśli pojawi się proces o wyższym priorytecie lub jeśli proces przekroczy określony czas wykonania (np. w algorytmie Round Robin). Preempcja pozwala na bardziej sprawiedliwe przydzielanie procesora, ale może wiązać się z kosztami związanymi z przełączaniem kontekstu, co zmniejsza wydajność.
- **Round Robin (RR)** – jeden z najczęściej stosowanych algorytmów w systemach wielozadaniowych. Każdy proces otrzymuje czas na wykonanie (tzw. kwant czasu) i po jego upływie jest przerywany i przenoszony na koniec kolejki procesów. Dzięki temu procesy są równomiernie obsługiwane, ale dla procesów, które wymagają długiego czasu na wykonanie, może to prowadzić do dużych opóźnień.
- **Algorytm najkrótszego pozostałego czasu (SRT - Shortest Remaining Time)** – jest to wersja preemptywna algorytmu SJF. Proces, który ma najmniejszy pozostały czas wykonania, jest wykonywany w pierwszej kolejności. W przeciwieństwie do SJF, ten algorytm może przerwać procesy już działające, jeśli znajdzie się proces o krótszym czasie wykonania. Algorytm ten może również prowadzić do głodowania długich procesów.
- **Algorytm loterii** – procesy są przydzielane losowe "losy", a procesy, które mają więcej losów, mają większą szansę na otrzymanie procesora. Algorytm ten pozwala na wprowadzenie pewnego elementu losowości i może być bardziej elastyczny w zrównoważeniu obciążenia systemu.

Wydajność algorytmów szeregowania

Ocena wydajności algorytmu szeregowania zadań może odbywać się na podstawie kilku kryteriów:

- **Czas oczekiwania** – średni czas, jaki procesy spędzają w kolejce przed rozpoczęciem wykonania.
- **Czas obrotu** – całkowity czas od momentu przyjścia procesu do zakończenia jego wykonania, w tym czas oczekiwania oraz czas wykonania.
- **Wydajność procesora** – procent czasu, w którym procesor jest aktywnie używany przez procesy, w przeciwieństwie do czasu, gdy jest bezczynny.
- **Fairness (sprawiedliwość)** – miara, w jakiej algorytm przydziela czas procesora wszystkim procesom, bez faworyzowania żadnego z nich.

Problemy związane z szeregowanie zadań

Pomimo wielu zalet, algorytmy szeregowania mogą napotkać na różne problemy w zależności od zastosowanego podejścia:

- **Głodowanie** – może wystąpić, gdy procesy o niskim priorytecie lub długim czasie wykonania nie otrzymują dostępu do procesora przez długi czas. Problem ten jest szczególnie widoczny w algorytmach szeregowania opartych na priorytetach.
- **Przełączanie kontekstu** – w algorytmach takich jak Round Robin, gdzie procesy są często przerywane, może wystąpić kosztowne przełączanie kontekstu, które obniża wydajność systemu.
- **Złożoność implementacyjna** – niektóre algorytmy, takie jak SJF czy SRT, mogą wymagać znajomości przewidywanego czasu wykonania procesów, co może być trudne do określenia w systemie rzeczywistym.

Optymalizacja algorytmów szeregowania

Aby uzyskać lepszą wydajność systemu, mogą być stosowane różne techniki optymalizacji algorytmów szeregowania:

- **Dostosowanie priorytetów** – w algorytmach opartych na priorytetach, priorytety procesów mogą być dynamicznie dostosowywane na podstawie ich zachowania lub potrzeb.
- **Multilevel Queue Scheduling** – podejście, które używa wielu kolejek procesów, każda z innym algorytmem szeregowania. Procesy są klasyfikowane do odpowiednich kolejek w zależności od ich rodzaju (np. procesy interaktywne, procesy wsadowe).
- **Proporcjonalne szeregowanie** – w tym podejściu algorytm przydziela procesor proporcjonalnie do zasobów, jakie procesy potrzebują.

Sieci komputerowe

Adresowanie

Adresowanie w sieciach komputerowych umożliwia jednoznaczną identyfikację urządzeń i kierowanie pakietów danych w sieci.

Adres IP

Adres IP (Internet Protocol) to unikalny identyfikator przypisany urządzeniu w sieci.

- **IPv4** – składa się z 32 bitów, zapisanych w postaci czterech oktetów oddzielonych kropkami, np. `192.168.1.1`. Daje około 4,3 miliarda unikalnych adresów.
- **IPv6** – składa się ze 128 bitów, zapisanych szesnastkowo w ośmiu blokach, np. `2001:0db8:85a3:0000:0000:8a2e:0370:7334`. Umożliwia niemal nieograniczoną liczbę adresów.

Podział adresów IP

- **Adresy publiczne** – globalnie unikalne, routowalne w Internecie.
- **Adresy prywatne** – używane wewnątrz sieci lokalnych; np.:
  - `10.0.0.0 – 10.255.255.255`
  - `172.16.0.0 – 172.31.255.255`
  - `192.168.0.0 – 192.168.255.255`
- **Adresy specjalne** – np. `127.0.0.1` (loopback), `0.0.0.0`, adresy multicastowe.

Maska podsieci

Maska podsieci (ang. subnet mask) określa, która część adresu IP identyfikuje sieć, a która urządzenie w tej sieci. Przykład: `255.255.255.0`.

CIDR – Classless Inter-Domain Routing

CIDR to metoda reprezentacji zakresów adresów IP i ich podziału. Zamiast tradycyjnych klas A/B/C używa się notacji z ukośnikiem, np. `192.168.1.0/24`.

Adresy MAC

Adres MAC (Media Access Control) to fizyczny adres karty sieciowej, unikalny dla każdego urządzenia. Zapis szesnastkowy, np. `00:1A:2B:3C:4D:5E`.

Adresowanie w warstwach OSI

- Warstwa 2 (łącza danych): adres MAC.
- Warstwa 3 (sieci): adres IP.

Tłumaczenie adresów

- **ARP (Address Resolution Protocol)** – tłumaczy adresy IP na adresy MAC.
- **DNS (Domain Name System)** – tłumaczy nazwy domenowe (np. `example.com`) na adresy IP.
- **NAT (Network Address Translation)** – tłumaczy prywatne adresy IP na publiczne i odwrotnie, umożliwiając dostęp do Internetu z wielu urządzeń przez jeden adres publiczny.

Topologie

Topologia sieci odnosi się do sposobu fizycznego lub logicznego rozmieszczenia elementów sieciowych (takich jak komputery, routery, przełączniki) oraz ich połączeń.

Topologie fizyczne i logiczne

- **Topologia fizyczna** – określa rzeczywiste rozmieszczenie kabli i urządzeń.
- **Topologia logiczna** – określa sposób przepływu danych między urządzeniami, niezależnie od fizycznego rozmieszczenia.

Rodzaje topologii sieciowych

- **Topologia magistrali (bus)** Wszystkie urządzenia są podłączone do jednej wspólnej magistrali transmisyjnej. Dane są przesyłane wzdłuż jednego kabla.

  Zalety: mały koszt, łatwa instalacja.
  Wady: awaria kabla unieruchamia całą sieć, kolizje danych.

- **Topologia pierścienia (ring)** Każde urządzenie jest połączone z dwoma sąsiadami, tworząc zamknięty pierścień. Dane krążą w jednym (lub obu) kierunkach.

  Zalety: brak kolizji przy odpowiednim protokole.
  Wady: awaria jednego węzła może przerwać całą transmisję (chyba że zastosuje się mechanizmy redundancji).

- **Topologia gwiazdy (star)** Wszystkie urządzenia są podłączone do centralnego punktu – najczęściej switcha lub huba.

  Zalety: łatwe zarządzanie i rozbudowa, awaria jednego urządzenia nie wpływa na działanie sieci.
  Wady: awaria centralnego punktu powoduje zatrzymanie całej sieci.

- **Topologia siatki (mesh)** Każde urządzenie jest bezpośrednio połączone z wieloma innymi urządzeniami.

  Zalety: wysoka niezawodność i odporność na awarie.
  Wady: kosztowna i skomplikowana w budowie.

- **Topologia drzewa (tree)** Hierarchiczna struktura łącząca wiele topologii gwiazdy w jeden system.

  Zalety: dobre skalowanie, łatwe zarządzanie dużymi sieciami.
  Wady: zależność od głównych węzłów – ich awaria wpływa na duże części sieci.

- **Topologia hybrydowa** Kombinacja różnych topologii, stosowana w nowoczesnych, złożonych sieciach.

  Zalety: elastyczność i możliwość dostosowania do potrzeb.
  Wady: wyższe koszty wdrożenia i zarządzania.

Wybór topologii

Dobór odpowiedniej topologii zależy od:

- skali sieci,
- budżetu,
- wymaganej niezawodności,
- łatwości konserwacji i rozbudowy.

Systemy zarządzania bazami danych

Modele baz danych

Modele baz danych definiują strukturę i sposób przechowywania danych. Najważniejsze modele to:

- **Model hierarchiczny** – dane są uporządkowane w strukturę drzewa. Każdy węzeł ma jednego rodzica i wielu potomków. Przykład: IBM IMS.
- **Model sieciowy** – dane reprezentowane jako graf, gdzie rekordy mogą mieć wiele powiązań. Umożliwia bardziej złożone relacje niż model hierarchiczny.
- **Model relacyjny** – dane przechowywane są w tabelach (relacjach), a ich relacje definiowane są za pomocą kluczy. Najczęściej stosowany model. Przykład: MySQL, PostgreSQL.
- **Model obiektowy** – dane jako obiekty, zgodne z paradygmatem programowania obiektowego. Przykład: db4o.
- **Model dokumentowy / NoSQL** – przechowuje dane w postaci dokumentów (np. JSON). Używany w systemach Big Data. Przykład: MongoDB.

Hurtownie danych

Hurtownia danych (ang. Data Warehouse) to specjalistyczna baza danych zoptymalizowana pod kątem analizy danych.

- **Charakterystyka**:
  - Integruje dane z różnych źródeł.
  - Dane są zdenormalizowane i przechowywane w sposób historyczny.
  - Optymalizowana pod kątem zapytań analitycznych (OLAP – Online Analytical Processing).
- **Architektura hurtowni danych**:
  - **Warstwa źródłowa** – systemy transakcyjne, pliki zewnętrzne.
  - **ETL (Extract, Transform, Load)** – proces przetwarzania i ładowania danych.
  - **Baza danych hurtowni** – centralne repozytorium danych.
  - **Warstwa prezentacji** – raportowanie, analiza, dashboardy.

Big Data

Big Data odnosi się do zbiorów danych o ogromnym wolumenie, zmienności i różnorodności, których przetwarzanie wymaga specjalnych narzędzi i technologii.

Cechy Big Data (5V)

- **Volume (Wolumen)** – ilość danych liczona w terabajtach, petabajtach lub więcej.
- **Velocity (Szybkość)** – dane są generowane i analizowane w czasie rzeczywistym.
- **Variety (Różnorodność)** – dane mają różne formaty: tekst, wideo, obrazy, dane sensoryczne.
- **Veracity (Wiarygodność)** – jakość i dokładność danych.
- **Value (Wartość)** – korzyści biznesowe, jakie można uzyskać z analizy danych.

Technologie Big Data

- **Hadoop** – framework do rozproszonego przechowywania i przetwarzania danych.
- **Spark** – platforma do szybkiego przetwarzania danych in-memory.
- **Kafka** – system przesyłania strumieniowego danych.
- **NoSQL** – bazy danych nienastawione na relacje, np. MongoDB, Cassandra.

Przetwarzanie w chmurze

Cloud Computing

Cloud Computing (przetwarzanie w chmurze) to model dostarczania zasobów obliczeniowych, takich jak serwery, pamięć masowa, bazy danych, sieci czy oprogramowanie, przez Internet („chmurę") na żądanie i za opłatą uzależnioną od użycia.

Cechy przetwarzania w chmurze

- **Elastyczność** – możliwość skalowania zasobów w górę lub w dół zależnie od potrzeb.
- **Model płatności za użycie** – opłaty tylko za wykorzystane zasoby.
- **Dostępność** – dostęp do usług z dowolnego miejsca z Internetem.
- **Automatyzacja** – dynamiczne zarządzanie i konfiguracja zasobów.
- **Wielodostępność (multi-tenancy)** – wielu użytkowników korzysta z tych samych fizycznych zasobów.

Modele wdrożenia chmury

- **Chmura publiczna** – zasoby oferowane przez zewnętrznego dostawcę (np. AWS, Microsoft Azure, Google Cloud).
- **Chmura prywatna** – dedykowana infrastruktura działająca w ramach organizacji.
- **Chmura hybrydowa** – połączenie chmury publicznej i prywatnej.
- **Chmura wielochmurowa (multi-cloud)** – użycie wielu dostawców chmurowych w jednej organizacji.

SaaS vs PaaS vs IaaS

Przetwarzanie w chmurze oferowane jest na różnych poziomach abstrakcji, określanych jako modele usług:

SaaS – Software as a Service

- Gotowe aplikacje udostępniane użytkownikom przez Internet.
- Przykłady: Google Workspace, Microsoft 365, Dropbox.
- Użytkownik nie zarządza infrastrukturą, tylko korzysta z aplikacji.

PaaS – Platform as a Service

- Środowisko programistyczne i uruchomieniowe dla aplikacji.
- Przykłady: Heroku, Google App Engine, Azure App Service.
- Programista skupia się na kodzie, a dostawca zarządza serwerami i infrastrukturą.

IaaS – Infrastructure as a Service

- Udostępnianie podstawowych zasobów obliczeniowych – maszyn wirtualnych, sieci, dysków.
- Przykłady: Amazon EC2, Microsoft Azure Virtual Machines.
- Użytkownik ma pełną kontrolę nad systemem operacyjnym, siecią i przechowywaniem danych.

Porównanie modeli usług

Zarządzane przez dostawcę	SaaS	PaaS	IaaS
Aplikacje	✓		
Dane	✓	✓	
System operacyjny	✓	✓	
Maszyny wirtualne	✓	✓	✓
Sieć i fizyczna infrastruktura	✓	✓	✓

Inżynieria oprogramowania

Inżynieria oprogramowania to dziedzina informatyki zajmująca się systematycznym podejściem do tworzenia, rozwoju i utrzymania oprogramowania. Łączy wiedzę z zakresu informatyki, zarządzania projektami, jakości i procesów produkcyjnych.

Wymagania funkcjonalne

Wymagania funkcjonalne określają, co system powinien robić – opisują konkretne funkcje i zachowania systemu z punktu widzenia użytkownika.

Przykłady wymagań funkcjonalnych

- System powinien umożliwiać użytkownikowi logowanie się za pomocą loginu i hasła.
- Po dodaniu produktu do koszyka, system powinien wyświetlić jego zawartość.
- System powinien wysyłać e-mail z potwierdzeniem po złożeniu zamówienia.

Zbieranie wymagań

Proces pozyskiwania wymagań obejmuje:

- **Wywiady** z interesariuszami.
- **Analizę dokumentacji**.
- **Warsztaty z użytkownikami**.
- **Prototypowanie** – tworzenie wstępnych wersji aplikacji w celu lepszego zrozumienia oczekiwań.

Specyfikacja wymagań

Dokumentacja wymagań powinna być:

- **Jednoznaczna** – każda funkcja opisana dokładnie.
- **Kompletna** – obejmować wszystkie funkcje.
- **Sprawdzalna** – możliwa do przetestowania.

Notacje w modelowaniu dziedziny biznesowej

Modelowanie dziedziny biznesowej polega na graficznym i tekstowym przedstawieniu procesów, danych i zależności w organizacji.

UML (Unified Modeling Language)

UML to standardowa notacja służąca do modelowania systemów informatycznych. Główne typy diagramów:

- **Diagram przypadków użycia (Use Case)** – przedstawia interakcje użytkowników z systemem.
- **Diagram klas** – pokazuje strukturę systemu w postaci klas i ich relacji.
- **Diagram sekwencji** – prezentuje kolejność komunikatów między obiektami w czasie.
- **Diagram aktywności** – modeluje przepływ działań i decyzji w systemie.

BPMN (Business Process Model and Notation)

BPMN to notacja do modelowania procesów biznesowych:

- **Procesy (Pools i Lanes)** – wyodrębnianie ról i odpowiedzialności.
- **Zdarzenia, zadania, bramki** – do opisywania przepływu procesów.
- **Strzałki (Sequence Flow)** – wskazują kolejność działań.

Korzyści z modelowania

- Lepsze zrozumienie systemu przez interesariuszy.
- Ułatwienie komunikacji między zespołem technicznym i biznesowym.
- Możliwość wcześniejszego wykrywania błędów i nieścisłości.

Internet Rzeczy

Internet Rzeczy (ang. Internet of Things, IoT) to koncepcja, w której urządzenia fizyczne – od prostych sensorów po złożone maszyny – są połączone z Internetem i mogą wymieniać dane między sobą oraz z systemami centralnymi. Celem IoT jest automatyzacja, monitorowanie i analiza procesów w czasie rzeczywistym.

Sieci czujników

Sieci czujników (ang. Wireless Sensor Networks – WSN) są kluczowym elementem architektury IoT. Składają się z rozproszonych urządzeń pomiarowych wyposażonych w sensory, moduły komunikacyjne i źródła zasilania.

Cechy sieci czujników

- **Bezprzewodowa komunikacja** – najczęściej przy użyciu protokołów takich jak ZigBee, LoRa, Bluetooth Low Energy (BLE) lub Wi-Fi.
- **Energooszczędność** – czujniki pracują na baterii, co wymaga minimalizacji zużycia energii.
- **Lokalność przetwarzania** – część danych może być wstępnie przetwarzana lokalnie (edge computing).
- **Skalowalność** – możliwość dodawania nowych węzłów do istniejącej sieci.

Zastosowania

- Monitorowanie środowiska (np. wilgotność, temperatura, jakość powietrza).
- Systemy inteligentnego rolnictwa.
- Przemysłowe systemy predykcyjnego utrzymania ruchu.
- Systemy zarządzania ruchem i parkowaniem w miastach.

Wearable tech

Wearable tech (technologia ubieralna) to urządzenia noszone bezpośrednio na ciele, które monitorują różne parametry biologiczne lub umożliwiają interakcję z otoczeniem.

Przykłady urządzeń wearable

- **Smartwatche** – monitorują puls, kroki, sen, umożliwiają odbieranie powiadomień.
- **Opaski fitness** – skupiają się na aktywności fizycznej i zdrowiu.
- **Inteligentne okulary** – np. Google Glass, oferujące rozszerzoną rzeczywistość.
- **Ubrania z sensorami** – wykrywające parametry fizjologiczne w czasie rzeczywistym.

Wyzwania technologii wearable

- **Zarządzanie energią** – konieczność częstego ładowania.
- **Bezpieczeństwo danych** – wrażliwe informacje zdrowotne muszą być odpowiednio chronione.
- **Kompatybilność i integracja** – współpraca z innymi systemami i aplikacjami.

Zastosowania

- Zdrowie i fitness (personalizacja opieki zdrowotnej).
- Sport (analiza wydolności i techniki).
- Bezpieczeństwo pracy (monitorowanie pracowników w trudnych warunkach).
- Rozrywka i rzeczywistość rozszerzona (AR).

Systemy wbudowane

Systemy wbudowane (ang. Embedded Systems) to specjalizowane systemy komputerowe zaprojektowane do wykonywania określonych zadań w ramach większego systemu technicznego. Charakteryzują się ograniczonymi zasobami, wysoką niezawodnością oraz często pracą w czasie rzeczywistym.

Zastosowanie

Systemy wbudowane znajdują szerokie zastosowanie w wielu dziedzinach życia i przemysłu.

Przykłady zastosowań

- **Motoryzacja** – sterowniki silnika (ECU), systemy ABS, poduszki powietrzne, systemy infotainment.
- **Elektronika użytkowa** – smartfony, telewizory, sprzęt AGD.
- **Automatyka przemysłowa** – sterowniki PLC, systemy SCADA, roboty przemysłowe.
- **Medycyna** – urządzenia monitorujące (np. EKG), pompy insulinowe, aparaty do obrazowania.
- **Telekomunikacja** – routery, przełączniki, modemy.
- **Systemy obronne i kosmiczne** – systemy nawigacyjne, kontrola lotu.

Cechy systemów wbudowanych

- **Dedykowane działanie** – zaprojektowane do jednej, konkretnej funkcji.
- **Ograniczone zasoby** – mała pamięć, niska moc obliczeniowa.
- **Czas rzeczywisty** – odpowiedź w zdefiniowanym czasie.
- **Wysoka niezawodność i trwałość** – praca często w trudnych warunkach środowiskowych.
- **Integracja z elektroniką** – często osadzone bezpośrednio na płytce drukowanej (PCB).

Języki programowania systemów wbudowanych

Ze względu na ograniczenia zasobów i wymogi czasowe, systemy wbudowane programuje się w językach umożliwiających kontrolę nad sprzętem.

Popularne języki

- **C** – najczęściej używany język, umożliwia efektywny dostęp do pamięci i rejestrów.
- **C++** – używany w bardziej złożonych systemach, z możliwością programowania obiektowego.
- **Assembly (Assembler)** – wykorzystywany w krytycznych fragmentach kodu wymagających maksymalnej optymalizacji.
- **Python/MicroPython** – używany w prototypowaniu i edukacji (np. Raspberry Pi, Micro:bit).
- **Rust** – zyskuje popularność dzięki bezpieczeństwu pamięci i wydajności.

Środowiska programistyczne i narzędzia

- **IDE:** MPLAB X, Keil uVision, STM32CubeIDE, PlatformIO.
- **Kompilatory:** GCC (np. arm-none-eabi-gcc), IAR Embedded Workbench.
- **Debuggery:** JTAG, SWD (Serial Wire Debug), UART Debug.

Architektura komputerów

Architektura von Neumanna

Architektura von Neumanna to klasyczny model komputera, zaproponowany przez Johna von Neumanna w 1945 roku. Stanowi ona podstawę działania większości współczesnych komputerów i opisuje sposób organizacji elementów systemu komputerowego.

Główne założenia

Architektura von Neumanna opiera się na kilku fundamentalnych założeniach:

- **Pamięć wspólna** – dane i instrukcje programu są przechowywane w tej samej pamięci.
- **Wykonywanie programu sekwencyjnie** – komputer wykonuje instrukcje jedna po drugiej, chyba że wystąpi skok warunkowy lub bezwarunkowy.
- **Jednostka centralna (CPU)** – składa się z jednostki arytmetyczno-logicznej (ALU) oraz jednostki sterującej, które wykonują obliczenia i sterują przepływem danych.
- **Rejestry** – szybka pamięć wewnętrzna procesora służąca do przechowywania tymczasowych danych i adresów.

Elementy składowe architektury

1. **Jednostka arytmetyczno-logiczna (ALU)** – odpowiada za wykonywanie operacji matematycznych i logicznych.
2. **Jednostka sterująca (CU)** – interpretuje instrukcje programu i steruje działaniem pozostałych elementów systemu.
3. **Pamięć operacyjna (RAM)** – przechowuje dane i instrukcje, które mają zostać przetworzone.
4. **Urządzenia wejścia/wyjścia (I/O)** – umożliwiają komunikację komputera z użytkownikiem oraz zewnętrznymi urządzeniami.
5. **Szyna danych, adresowa i sterująca** – służą do przesyłania danych, adresów i sygnałów sterujących pomiędzy komponentami.

Cykl maszynowy

Cykl maszynowy w architekturze von Neumanna składa się z następujących faz:

1. **Pobranie instrukcji (fetch)** – instrukcja jest pobierana z pamięci do rejestru instrukcji.
2. **Dekodowanie instrukcji (decode)** – jednostka sterująca rozpoznaje instrukcję i przygotowuje odpowiednie sygnały sterujące.
3. **Wykonanie instrukcji (execute)** – ALU wykonuje operację, np. dodawanie liczb lub porównanie.
4. **Zapis wyniku (write back)** – wynik operacji jest zapisywany w odpowiednim rejestrze lub pamięci.

Zalety i wady architektury von Neumanna

Zalety:

- Prosta i logiczna struktura, łatwa do implementacji.
- Możliwość przechowywania i modyfikacji programu w pamięci.
- Uniwersalność – ta sama maszyna może wykonywać różne programy.

Wady:

- **Wąskie gardło von Neumanna** – ograniczenie wynikające z tego, że dane i instrukcje są przesyłane tym samym kanałem, co może powodować opóźnienia.
- Brak równoległości – tradycyjna architektura von Neumanna nie wspiera natywnie przetwarzania równoległego.

Współczesne zastosowania i modyfikacje

Chociaż nowoczesne komputery wprowadzają liczne modyfikacje (np. pamięć podręczna, przetwarzanie potokowe, wielordzeniowość), podstawowe założenia architektury von Neumanna nadal obowiązują. Alternatywą jest np. architektura Harvardzka, w której dane i instrukcje są przechowywane w oddzielnych pamięciach, co pozwala na równoczesne ich pobieranie i zwiększa wydajność.

Architektura procesora

Architektura procesora odnosi się do organizacji wewnętrznej i zestawu funkcjonalności jednostki centralnej (CPU). Określa sposób, w jaki procesor wykonuje instrukcje, komunikuje się z pamięcią, oraz realizuje operacje arytmetyczne, logiczne i sterujące. Jest kluczowym czynnikiem wpływającym na wydajność i możliwości komputera.

Podstawowe komponenty procesora

- **Jednostka arytmetyczno-logiczna (ALU)** – wykonuje operacje matematyczne (dodawanie, odejmowanie itp.) oraz logiczne (AND, OR, NOT, XOR).
- **Jednostka sterująca (CU)** – odpowiada za dekodowanie instrukcji i generowanie sygnałów sterujących, które kierują przepływem danych w CPU.
- **Rejestry** – małe, bardzo szybkie komórki pamięci przechowujące dane tymczasowe, wyniki obliczeń oraz adresy.
- **Szyna danych i adresowa** – umożliwiają komunikację pomiędzy CPU a pamięcią oraz urządzeniami wejścia/wyjścia.
- **Licznik rozkazów (PC – Program Counter)** – przechowuje adres bieżącej instrukcji do wykonania.
- **Rejestr instrukcji (IR – Instruction Register)** – przechowuje aktualnie wykonywaną instrukcję.

Rodzaje architektur procesora

- **CISC (Complex Instruction Set Computer)** – architektura o złożonym zestawie instrukcji. Każda instrukcja może wykonywać wiele operacji. Przykład: x86.
- **RISC (Reduced Instruction Set Computer)** – architektura o uproszczonym zestawie instrukcji, w której każda instrukcja wykonuje jedną prostą operację. Przykład: ARM, MIPS.
- **VLIW (Very Long Instruction Word)** – pozwala na wykonywanie wielu operacji w ramach jednej, bardzo długiej instrukcji.
- **Superskalarna architektura** – umożliwia jednoczesne wykonanie wielu instrukcji w jednym cyklu zegara, przy użyciu wielu jednostek wykonawczych.

Techniki zwiększające wydajność procesora

- **Potokowość (Pipelining)** – dzieli wykonanie instrukcji na etapy (np. pobranie, dekodowanie, wykonanie), co pozwala na jednoczesne przetwarzanie wielu instrukcji.
- **Wielowątkowość (Multithreading)** – umożliwia przełączanie między wątkami w obrębie jednego rdzenia, co zwiększa wykorzystanie zasobów procesora.
- **Wielordzeniowość (Multicore)** – procesor zawiera wiele rdzeni wykonawczych, co pozwala na równoczesne wykonywanie wielu zadań.
- **Pamięci podręczne (cache)** – szybka pamięć zintegrowana z procesorem (L1, L2, L3), przechowująca często używane dane, aby zmniejszyć opóźnienia dostępu do pamięci RAM.
- **Out-of-order execution** – instrukcje mogą być wykonywane w innej kolejności niż w kodzie, jeśli umożliwia to szybsze zakończenie obliczeń bez naruszania poprawności.

Rozwój procesorów

Procesory przeszły długą drogę od prostych jednostek wykonujących kilka milionów instrukcji na sekundę (MIPS), do nowoczesnych chipów zawierających miliardy tranzystorów. Współczesne procesory często zawierają:

- wiele rdzeni (CPU cores),
- wbudowane procesory graficzne (GPU),
- zaawansowane mechanizmy zarządzania energią,
- obsługę wirtualizacji.

Przykładowi producenci i architektury

- **Intel** – architektura x86/x86-64 (np. Core i5, i7).
- **AMD** – architektura x86-64 (np. Ryzen, EPYC).
- **ARM Holdings** – architektura ARM stosowana w smartfonach, tabletach i mikrokomputerach (np. Raspberry Pi, Apple M1/M2).
- **Apple** – własna architektura ARM (Apple Silicon).

Bezpieczeństwo systemów informatycznych

Zagrożenia

Systemy informatyczne są narażone na wiele rodzajów zagrożeń, które mogą prowadzić do utraty danych, naruszenia prywatności, czy zakłócenia działania usług.

Rodzaje zagrożeń

- **Malware** – złośliwe oprogramowanie, takie jak wirusy, robaki, trojany, ransomware.
- **Ataki sieciowe** – np. DoS (Denial of Service), DDoS, sniffing, spoofing, man-in-the-middle.
- **Phishing i socjotechnika** – próby wyłudzenia informacji przez podszywanie się pod zaufane źródła.
- **Luki w oprogramowaniu** – błędy i podatności umożliwiające nieautoryzowany dostęp.
- **Nieautoryzowany dostęp fizyczny** – kradzież urządzeń, podsłuchy, manipulacje sprzętowe.

Skutki zagrożeń

- Utrata lub wyciek danych (np. danych osobowych, finansowych).
- Przerwanie ciągłości działania systemów.
- Straty finansowe, reputacyjne i prawne.

Metody uwierzytelniania i autoryzacji

Aby chronić systemy informatyczne, stosuje się mechanizmy kontroli dostępu – uwierzytelnianie i autoryzację.

Uwierzytelnianie (ang. authentication)

Proces potwierdzania tożsamości użytkownika. Typowe metody:

- **Coś, co wiesz** – hasło, PIN.
- **Coś, co masz** – token, karta inteligentna, urządzenie mobilne.
- **Coś, czym jesteś** – biometryka (odcisk palca, skan twarzy, siatkówka).

Stosuje się również uwierzytelnianie wieloskładnikowe (MFA), które łączy więcej niż jeden z powyższych czynników.

Autoryzacja (ang. authorization)

Proces nadawania użytkownikowi odpowiednich uprawnień po uwierzytelnieniu. Przykłady:

- systemy uprawnień oparte na rolach (RBAC – Role-Based Access Control),
- listy kontroli dostępu (ACL – Access Control List),
- macierze uprawnień.

Zarządzanie tożsamością

Obejmuje tworzenie, aktualizowanie i usuwanie kont użytkowników oraz kontrolę ich uprawnień. Coraz częściej stosuje się:

- **SSO (Single Sign-On)** – jedno logowanie do wielu systemów,
- **Federacyjne systemy tożsamości** – np. logowanie za pomocą kont Google, Facebook,
- **IAM (Identity and Access Management)** – kompleksowe zarządzanie tożsamością i dostępem w organizacjach.

Logika i operacje na zbiorach

Bramki logiczne

Bramki logiczne to podstawowe elementy budujące układy cyfrowe. Przetwarzają one sygnały binarne (0 i 1) zgodnie z regułami algebry Boole'a.

Podstawowe bramki logiczne

- **NOT (negacja)** – odwraca wartość logiczną sygnału: 1 staje się 0, a 0 staje się 1.
- **AND (koniunkcja)** – wynik to 1 tylko wtedy, gdy oba wejścia są równe 1.
- **OR (alternatywa)** – wynik to 1, jeśli przynajmniej jedno wejście jest równe 1.

Bramki złożone

- **NAND** – negacja koniunkcji; odwrotność bramki AND.
- **NOR** – negacja alternatywy; odwrotność bramki OR.
- **XOR (alternatywa rozłączna)** – wynik to 1 tylko wtedy, gdy jedno z wejść jest równe 1, ale nie oba jednocześnie.
- **XNOR** – odwrotność XOR; wynik to 1, gdy oba wejścia mają tę samą wartość.

Zastosowania bramek logicznych

Bramki logiczne wykorzystywane są w konstrukcji:

- układów arytmetycznych (np. sumatorów),
- układów sterujących,
- pamięci komputerowych,
- procesorów i mikroprocesorów.

Rodzaje operacji na zbiorach

Operacje na zbiorach są kluczowe w matematyce dyskretnej i informatyce. Pozwalają na przetwarzanie grup danych i logiczne manipulowanie nimi.

Podstawowe operacje na zbiorach

- **Suma zbiorów (A ∩ B)** – zbiór elementów należących do A lub do B (lub do obu).
- **Przecięcie zbiorów (A ∪ B)** – zbiór elementów wspólnych dla A i B.
- **Różnica zbiorów (A\B)** – zbiór elementów należących do A, ale nie do B.
- **Dopełnienie zbioru (A')** – zbiór wszystkich elementów spoza zbioru A (względem uniwersum).

Prawo de Morgana

Zasady przekształcania wyrażeń logicznych i zbiorów:

$$\neg(A \wedge B) = \neg A \vee \neg B$$

$$\neg(A \vee B) = \neg A \wedge \neg B$$

W kontekście zbiorów:

$$(A \cap B)' = A' \cup B' \quad \text{oraz} \quad (A \cup B)' = A' \cap B'$$

Zastosowanie w informatyce

Operacje na zbiorach wykorzystywane są m.in. w:

- wyszukiwaniu danych i filtracji (np. SQL, zapytania do baz danych),
- budowie języków programowania (wyrażenia regularne, parsery),
- analizie stanów logicznych i modelowaniu przepływu informacji.

Rachunek prawdopodobieństwa

Rozkłady prawdopodobieństwa

Rozkład prawdopodobieństwa opisuje, jakie wartości może przyjąć zmienna losowa oraz jakie jest prawdopodobieństwo ich wystąpienia.

Rozkład dyskretny

Dotyczy zmiennych losowych, które mogą przyjmować skończoną lub przeliczalną liczbę wartości. Przykłady:

- **Rozkład jednostajny dyskretny** – wszystkie możliwe wyniki są jednakowo prawdopodobne, np. rzut symetryczną kostką.
- **Rozkład Bernoulliego** – zmienna przyjmuje wartość 1 z prawdopodobieństwem p, a 0 z prawdopodobieństwem $1 - p$.
- **Rozkład dwumianowy (Bernoulliego)** – liczba sukcesów w n niezależnych próbach Bernoulliego.
- **Rozkład Poissona** – modeluje liczbę zdarzeń zachodzących w ustalonej jednostce czasu przy stałym średnim natężeniu.

Rozkład ciągły

Dotyczy zmiennych losowych, które mogą przyjmować dowolną wartość z pewnego przedziału. Przykłady:

- **Rozkład jednostajny ciągły** – każda wartość z określonego przedziału ma tę samą gęstość prawdopodobieństwa.
- **Rozkład normalny (Gaussa)** – dzwonowaty rozkład o parametrach: średnia μ i odchylenie standardowe σ; bardzo często występuje w naturze.
- **Rozkład wykładniczy** – opisuje czas między kolejnymi zdarzeniami w procesie Poissona.

Zmienne losowe

Zmienna losowa to funkcja przyporządkowująca każdemu możliwemu wynikowi doświadczenia losowego określoną wartość liczbową.

Rodzaje zmiennych losowych

- **Dyskretna zmienna losowa** – np. liczba wyrzuconych oczek, liczba sukcesów.
- **Ciągła zmienna losowa** – np. czas oczekiwania, długość życia baterii.

Charakterystyki zmiennych losowych

- **Wartość oczekiwana** $\mathbb{E}[X]$ – średnia ważona wszystkich możliwych wartości, ważona ich prawdopodobieństwami.
- **Wariancja** $\mathrm{Var}(X)$ – miara rozproszenia wyników wokół wartości oczekiwanej.
- **Odchylenie standardowe** $\sigma = \sqrt{\mathrm{Var}(X)}$ – pierwiastek kwadratowy z wariancji.

Niezależność i zależność zmiennych

Dwie zmienne losowe są niezależne, jeśli wiedza o jednej z nich nie wpływa na drugą. Przykładowo, rzuty dwiema kostkami są niezależne, ale wzrost i waga człowieka są zwykle zależne.

Rozkłady łączone i brzegowe

- **Rozkład łączny** – opisuje prawdopodobieństwo wystąpienia jednocześnie konkretnych wartości dwóch (lub więcej) zmiennych losowych.
- **Rozkład brzegowy** – opisuje rozkład jednej zmiennej niezależnie od drugiej.

Matematyka dyskretna

Grafy

Grafy to struktury matematyczne używane do modelowania relacji między obiektami. Składają się z wierzchołków (ang. vertices) oraz krawędzi (ang. edges), które łączą pary wierzchołków.

Podstawowe pojęcia

- **Graf nieskierowany** – krawędzie nie mają kierunku (relacja symetryczna).
- **Graf skierowany (digraf)** – krawędzie mają określony kierunek (relacja niesymetryczna).
- **Wierzchołek (ang. vertex)** – podstawowy element grafu.
- **Krawędź (ang. edge)** – łączy dwa wierzchołki; może być skierowana lub nie.
- **Stopień wierzchołka** – liczba krawędzi wychodzących (w grafie skierowanym: osobno wejściowy i wyjściowy).
- **Ścieżka** – ciąg kolejnych krawędzi łączących pewien zbiór wierzchołków.
- **Cykl** – ścieżka, która zaczyna się i kończy w tym samym wierzchołku.
- **Spójność** – graf jest spójny, jeśli istnieje ścieżka między każdą parą wierzchołków.

Szczególne typy grafów

- **Drzewo** – graf nieskierowany, acykliczny i spójny; posiada dokładnie $n-1$ krawędzi dla n wierzchołków.
- **Las** – zbiór rozłącznych drzew.
- **Graf pełny (kompletny)** – każdy wierzchołek jest połączony z każdym innym.
- **Graf dwudzielny** – zbiór wierzchołków można podzielić na dwie grupy tak, że każda krawędź łączy wierzchołki z różnych grup.
- **Graf ważony** – krawędzie mają przypisane wagi (np. koszty, odległości).

Reprezentacje grafów

- **Macierz sąsiedztwa** – macierz $n \times n$, gdzie wpis 1 (lub waga) oznacza istnienie krawędzi między wierzchołkami.
- **Lista sąsiedztwa** – dla każdego wierzchołka podajemy listę jego sąsiadów.

Zastosowania grafów

Grafy są szeroko stosowane w informatyce i innych dziedzinach:

- analiza sieci komputerowych i społecznych,
- algorytmy trasowania (np. Dijkstra, BFS, DFS),
- planowanie i harmonogramowanie zadań,
- modelowanie relacji i zależności w bazach danych,
- optymalizacja (np. minimalne drzewa rozpinające, ścieżki Hamiltona, cykle Eulera).