1 Zarządzanie pamięcią w języku C/C++

1.1 Wprowadzenie

Zarządzanie pamięcią w językach C i C++ jest kluczowym aspektem programowania, ponieważ oba języki pozwalają na bezpośrednią kontrolę nad alokacją i dealokacją pamięci. Niewłaściwe zarządzanie pamięcią może prowadzić do wycieków pamięci, błędów segmentacji i nieprzewidywalnych zachowań programu.

1.2 Rodzaje pamięci w C/C++

W językach C i C++ można wyróżnić kilka obszarów pamięci:

- **Pamięć statyczna** – zmienne globalne oraz zmienne zadeklarowane jako `static` są przechowywane w tej pamięci i istnieją przez cały czas działania programu.
- **Stos (stack)** – przechowuje zmienne lokalne oraz adresy powrotne funkcji. Pamięć na stosie jest automatycznie zwalniana po zakończeniu funkcji.
- **Sterta (heap)** – obszar pamięci przeznaczony do dynamicznej alokacji. Zarządzanie pamięcią na stercie jest odpowiedzialnością programisty.

1.3 Alokacja i dealokacja pamięci w języku C

W języku C dynamiczne zarządzanie pamięcią odbywa się za pomocą funkcji bibliotecznych z nagłówka `stdlib.h`:

- `malloc(size_t size)` – alokuje określoną ilość bajtów i zwraca wskaźnik do pierwszego bajtu tej pamięci. Nie inicjalizuje pamięci.
- `calloc(size_t num, size_t size)` – alokuje pamięć dla tablicy elementów i zeruje przydzieloną pamięć.
- `realloc(void* ptr, size_t new_size)` – zmienia rozmiar wcześniej zaalokowanego bloku pamięci.
- `free(void* ptr)` – zwalnia zaalokowaną pamięć.

Przykład użycia funkcji `malloc` i `free`:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr = (int*) malloc(10 * sizeof(int)); // Alokacja tablicy 10 elementów
    if (ptr == NULL) {
        printf("Błąd alokacji pamięci\n");
        return 1;
    }
    free(ptr); // Zwolnienie pamięci
    return 0;
}
```

1.4 Alokacja i dealokacja pamięci w języku C++

W języku C++ dynamiczne zarządzanie pamięcią odbywa się za pomocą operatorów:

- `new` – alokuje pamięć dla pojedynczego obiektu lub tablicy.
- `delete` – zwalnia pamięć przydzieloną pojedynczemu obiektowi.
- `delete[]` – zwalnia pamięć przydzieloną tablicy.

Przykład użycia `new` i `delete`:

```
#include <iostream>

int main() {
    int *ptr = new int(10); // Alokacja pamięci dla pojedynczej liczby
    delete ptr; // Zwolnienie pamięci

    int *arr = new int[10]; // Alokacja pamięci dla tablicy
    delete[] arr; // Zwolnienie pamięci tablicy
    return 0;
}
```

1.5 Problemy związane z zarządzaniem pamięcią

- **Wycieki pamięci** – występują, gdy zaalokowana pamięć nie zostaje zwolniona.
- **Dereferencja pustego wskaźnika** – próba użycia wskaźnika o wartości `NULL` powoduje błąd wykonania.
- **Uszkodzenie pamięci** – zapisanie wartości poza przydzielonym obszarem może prowadzić do nieprzewidywalnych błędów.
- **Podwójne zwalnianie pamięci** – zwolnienie tej samej pamięci więcej niż raz może prowadzić do błędów.

1.6 Mechanizmy zarządzania pamięcią w nowoczesnym C++

Nowoczesny C++ (C++11 i nowsze) wprowadza inteligentne wskaźniki, które ułatwiają zarządzanie pamięcią:

- `std::unique_ptr` – zarządza pojedynczym obiektem i automatycznie zwalnia pamięć po zakończeniu jego żywotności.
- `std::shared_ptr` – zarządza współdzieloną pamięcią i automatycznie zwalnia ją, gdy nie ma już żadnych referencji.
- `std::weak_ptr` – słaby wskaźnik używany w celu uniknięcia cykli odniesień.

Przykład użycia `std::unique_ptr`:

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> ptr = std::make_unique<int>(10);
    std::cout << *ptr << std::endl; // Wyświetla 10
    return 0; // Pamięć zostaje automatycznie zwolniona
}
```

1.7 Podsumowanie

Zarządzanie pamięcią w językach C i C++ jest kluczowym aspektem programowania. Wymaga ono ostrożności i dbałości o poprawne zwalnianie pamięci. Nowoczesne mechanizmy, takie jak inteligentne wskaźniki w C++, znacząco ułatwiają pracę z pamięcią, redukując ryzyko błędów.

2 Funkcja Eulera, relacja kongruencji oraz redukcja modulo działań arytmetycznych

2.1 Funkcja Eulera

Funkcja Eulera, oznaczana jako $\varphi(n)$, jest funkcją liczby naturalnej n, określającą liczbę liczb względnie pierwszych z n (tzn. takich, które nie mają wspólnego dzielnika z n poza 1).

2.1.1 Definicja

Funkcja Eulera $\varphi(n)$ jest zdefiniowana jako:

$$\varphi(n) = |\{k \in \mathbb{N} \mid 1 \leq k \leq n,\ \gcd(k, n) = 1\}|$$

gdzie $\gcd(k, n)$ oznacza największy wspólny dzielnik liczb k i n.

2.1.2 Wzór na funkcję Eulera

Jeżeli liczba n ma rozkład na czynniki pierwsze postaci:

$$n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$$

to funkcja Eulera wyraża się wzorem:

$$\varphi(n) = n \cdot \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right).$$

2.1.3 Przykłady

- $\varphi(6)$: $6 = 2 \cdot 3$, więc:

  $$\varphi(6) = 6 \left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right) = 6 \cdot \frac{1}{2} \cdot \frac{2}{3} = 2.$$

  Liczby względnie pierwsze z 6 to $1, 5$, więc wynik zgadza się z definicją.

- $\varphi(12)$: $12 = 2^2 \cdot 3$, więc:

  $$\varphi(12) = 12 \left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right) = 12 \cdot \frac{1}{2} \cdot \frac{2}{3} = 4.$$

  Liczby względnie pierwsze z 12 to $1, 5, 7, 11$.

2.2 Relacja kongruencji

Relacja kongruencji w arytmetyce modularnej opisuje sytuację, w której dwie liczby mają tę samą resztę z dzielenia przez daną liczbę (moduł).

2.2.1 Definicja

Mówimy, że liczby a i b są kongruentne modulo m, jeżeli:

$$a \equiv b \pmod{m} \quad \Leftrightarrow \quad m \text{ dzieli } (a - b),\ \text{tj. } m|(a - b).$$

2.2.2 Własności kongruencji

- Jeżeli $a \equiv b \pmod{m}$ i $c \equiv d \pmod{m}$, to:
  - $a + c \equiv b + d \pmod{m}$ (kongruencja zachowuje dodawanie),
  - $a - c \equiv b - d \pmod{m}$ (kongruencja zachowuje odejmowanie),
  - $a \cdot c \equiv b \cdot d \pmod{m}$ (kongruencja zachowuje mnożenie).

2.2.3 Przykłady

- $17 \equiv 5 \pmod{6}$, ponieważ $17 - 5 = 12$ jest podzielne przez 6.
- $26 \equiv 2 \pmod{8}$, ponieważ $26 - 2 = 24$ jest podzielne przez 8.

2.3 Redukcja modulo w działaniach arytmetycznych

Redukcja modulo oznacza zastąpienie liczby przez jej resztę z dzielenia przez ustalony moduł m.

2.3.1 Definicja

Dla liczby a i modułu m, liczba r spełniająca:

$$a \equiv r \pmod{m}, \quad \text{gdzie } 0 \leq r < m$$

nazywana jest resztą z dzielenia a przez m.

2.3.2 Przykłady

- $15 \mod 4 = 3$, ponieważ $15 = 4 \cdot 3 + 3$, więc reszta to 3.
- $23 \mod 7 = 2$, ponieważ $23 = 7 \cdot 3 + 2$, więc reszta to 2.

2.3.3 Własności redukcji modulo

- $(a + b) \mod m = [(a \mod m) + (b \mod m)] \mod m$.
- $(a - b) \mod m = [(a \mod m) - (b \mod m)] \mod m$.
- $(a \cdot b) \mod m = [(a \mod m) \cdot (b \mod m)] \mod m$.

2.3.4 Przykład działań arytmetycznych modulo

Niech $a = 17$, $b = 23$, $m = 5$:

- Dodawanie:
$$(17 + 23) \mod 5 = 40 \mod 5 = 0.$$
- Mnożenie:
$$(17 \cdot 23) \mod 5 = 391 \mod 5 = 1.$$

2.4 Podsumowanie

- Funkcja Eulera $\varphi(n)$ określa liczbę liczb względnie pierwszych z n.
- Kongruencja $a \equiv b \pmod{m}$ oznacza, że liczby a i b mają tę samą resztę z dzielenia przez m.
- Redukcja modulo pozwala upraszczać działania arytmetyczne w systemie modularnym.

3 Strategia projektowania „dziel i zwyciężaj”

3.1 Zasada realizacji strategii

Strategia „dziel i zwyciężaj” (ang. *divide and conquer*) jest jednym z fundamentalnych paradygmatów algorytmicznych. Opiera się na rekurencyjnym podziale problemu na mniejsze podproblemy, ich niezależnym rozwiązaniu, a następnie scaleniu wyników w celu uzyskania ostatecznego rozwiązania.

Typowy algorytm oparty na strategii „dziel i zwyciężaj” składa się z trzech głównych etapów:

1. **Podział (Divide)** – problem dzielony jest na kilka mniejszych podproblemów o podobnej strukturze.
2. **Rozwiązanie (Conquer)** – podproblemy są rozwiązywane rekurencyjnie, aż do osiągnięcia przypadku bazowego.
3. **Scalanie (Combine)** – rozwiązania podproblemów są łączone w celu uzyskania rozwiązania pierwotnego problemu.

Strategia ta znajduje szerokie zastosowanie w sortowaniu, wyszukiwaniu, obliczeniach numerycznych oraz problemach geometrycznych i grafowych.

3.2 Przykłady zastosowania strategii

3.2.1 Sortowanie szybkie (QuickSort)

Algorytm QuickSort stosuje strategię „dziel i zwyciężaj” do sortowania tablicy poprzez rekurencyjny podział na mniejsze fragmenty.

Opis działania:

1. Wybierany jest element zwany **pivotem** (np. pierwszy, ostatni, środkowy lub losowy element tablicy).
2. Tablica jest dzielona na dwie części:
   - elementy mniejsze od pivota,
   - elementy większe od pivota.
3. QuickSort jest wywoływany rekurencyjnie na obu częściach tablicy.
4. Po zakończeniu rekurencji otrzymujemy posortowaną tablicę.

Złożoność QuickSort:

- **Najlepszy przypadek:** $O(n \log n)$, gdy pivot zawsze dzieli tablicę na dwie równe części.
- **Średni przypadek:** $O(n \log n)$, gdy pivot zazwyczaj dobrze dzieli tablicę.
- **Najgorszy przypadek:** $O(n^2)$, gdy pivot zawsze wybierany jest najgorzej (np. najmniejszy lub największy element).

3.2.2 Sortowanie przez scalanie (MergeSort)

MergeSort to stabilny algorytm sortowania, który również stosuje podejście „dziel i zwyciężaj".

Opis działania:

1. Tablica jest dzielona na dwie równe części.
2. Obie części są sortowane rekurencyjnie.
3. Posortowane tablice są scalane w jedną uporządkowaną tablicę.

Złożoność MergeSort:

- W każdym poziomie rekurencji algorytm wykonuje $O(n)$ operacji scalania.
- Liczba poziomów rekurencji wynosi $O(\log n)$.
- Łączna złożoność: $O(n \log n)$ w każdym przypadku.

3.2.3 Minimalne drzewo rozpinające (MST) – Algorytm Kruskala

Algorytm Kruskala znajduje minimalne drzewo rozpinające (ang. *Minimum Spanning Tree, MST*) dla spójnego grafu ważonego.

Opis działania:

1. Wszystkie krawędzie grafu są sortowane według wag w porządku niemalejącym (sortowanie można wykonać metodą MergeSort lub QuickSort).
2. Krawędzie są dodawane do MST w kolejności rosnącej wagi, o ile nie tworzą cyklu.
3. Struktura zbiorów rozłącznych (ang. *Union-Find*) pozwala efektywnie zarządzać dołączaniem wierzchołków.

Złożoność algorytmu Kruskala:

- Sortowanie krawędzi: $O(E \log E)$.
- Przetwarzanie zbiorów rozłącznych: $O(E\alpha(V))$, gdzie α to powoli rosnąca funkcja iterowanego logarytmu.
- Łączna złożoność: $O(E \log E)$.

3.2.4 Inne przykłady algorytmów „dziel i zwyciężaj"

- **Algorytm Karacuby** – szybkie mnożenie dużych liczb ($O(n^{1.585})$).
- **Najbliższa para punktów** – znajdowanie dwóch najbliższych punktów w płaszczyźnie ($O(n \log n)$).
- **Szybkie potęgowanie** – algorytm obliczający $a^b \mod m$ w czasie $O(\log b)$.
- **FFT (Fast Fourier Transform)** – szybka transformata Fouriera, stosowana w analizie sygnałów i kryptografii ($O(n \log n)$).

3.3 Szacowanie asymptotycznej złożoności uzyskanych algorytmów

Asymptotyczna złożoność obliczeniowa określa, jak czas działania algorytmu zmienia się wraz ze wzrostem rozmiaru wejścia n. Jest kluczowym aspektem analizy algorytmów i pozwala porównywać ich efektywność.

3.3.1 Szacowanie złożoności rekurencyjnych algorytmów „dziel i zwyciężaj"

Wiele algorytmów opartych na tej strategii można przedstawić w postaci rekurencyjnej zależności czasowej. Istnieją różne metody jej rozwiązywania, w tym:

- Metoda podstawiania (*substitution method*).
- Metoda drzew rekurencji (*recursion tree method*).
- Twierdzenie o rekurencji Mistrza (*Master Theorem*).

Twierdzenie Mistrza Dla rekurencji postaci:

$$T(n) = aT(n/b) + O(n^d),$$

gdzie:

- a – liczba podproblemów,
- b – czynnik zmniejszania rozmiaru podproblemu,
- $O(n^d)$ – koszt scalania wyników,

możemy określić asymptotyczną złożoność na podstawie porównania d i $\log_b a$:

1. Jeśli $d > \log_b a$, to $T(n) = O(n^d)$.
2. Jeśli $d = \log_b a$, to $T(n) = O(n^d \log n)$.
3. Jeśli $d < \log_b a$, to $T(n) = O(n^{\log_b a})$.

3.3.2 Złożoność wybranych algorytmów

- **QuickSort:**

$$T(n) = 2T(n/2) + O(n) \Rightarrow O(n \log n) \quad \text{(średni przypadek)}$$

- **MergeSort:**

$$T(n) = 2T(n/2) + O(n) \Rightarrow O(n \log n)$$

- **Algorytm Kruskala:**

$$O(E \log E) \quad \text{(sortowanie krawędzi)}$$

- **Algorytm Karacuby:**

$$T(n) = 3T(n/2) + O(n) \Rightarrow O(n^{\log_2 3}) \approx O(n^{1.585})$$

Podsumowując, strategia „dziel i zwyciężaj" prowadzi do efektywnych algorytmów o złożoności rzędu $O(n \log n)$ dla sortowania oraz $O(E \log E)$ dla problemów grafowych.

3.4 Podsumowanie

- Strategia „dziel i zwyciężaj” polega na podziale problemu na mniejsze podproblemy, ich rozwiązaniu i scaleniu wyników.
- Klasyczne algorytmy oparte na tej strategii to QuickSort ($O(n \log n)$ w średnim przypadku), MergeSort ($O(n \log n)$), algorytm Kruskala ($O(E \log E)$) oraz algorytm Karacuby ($O(n^{1.585})$).
- Algorytmy tego typu często osiągają optymalne lub suboptymalne złożoności w porównaniu do algorytmów iteracyjnych.

4 Podstawowe techniki i strategie projektowania algorytmów

4.1 Wprowadzenie

Projektowanie algorytmów to proces tworzenia efektywnych metod rozwiązania problemów obliczeniowych. W informatyce wyróżnia się kilka podstawowych strategii, które pozwalają konstruować algorytmy o dobrej wydajności.

Najważniejsze techniki projektowania algorytmów to:

- **Zachłanność (greedy algorithms)**
- **Dziel i zwyciężaj (divide and conquer)**
- **Programowanie dynamiczne (dynamic programming)**
- **Algorytmy zachłanne aproksymacyjne**
- **Przeszukiwanie z nawrotami (backtracking)**
- **Metoda programowania liniowego**
- **Heurystyki i algorytmy probabilistyczne**

4.2 Zachłanność (Greedy Algorithms)

Strategia zachłanna polega na podejmowaniu w każdej chwili lokalnie optymalnej decyzji, mając nadzieję, że doprowadzi ona do globalnie optymalnego rozwiązania.

4.2.1 Cechy algorytmów zachłannych

Algorytmy zachłanne działają poprawnie, gdy spełnione są następujące warunki:

- **Właściwość optymalnej podstruktury** – optymalne rozwiązanie problemu zawiera optymalne rozwiązania jego podproblemów.
- **Brak efektu przyszłościowego** – decyzja podjęta na danym etapie nie wpływa na przyszłe decyzje.

4.2.2 Przykłady algorytmów zachłannych

- **Algorytm Kruskala** – znajdowanie minimalnego drzewa rozpinającego (MST).
- **Algorytm Dijkstry** – wyznaczanie najkrótszej ścieżki w grafie.
- **Problem plecakowy (wersja heurystyczna)** – wybór elementów maksymalizujących zysk przy ograniczonej pojemności plecaka.

4.3 Dziel i zwyciężaj (Divide and Conquer)

Technika „dziel i zwyciężaj” polega na podziale problemu na mniejsze podproblemy, rozwiązaniu ich rekurencyjnie i scaleniu wyników.

4.3.1 Przykłady algorytmów opartych na tej strategii

- **MergeSort** – sortowanie przez scalanie ($O(n \log n)$).
- **QuickSort** – sortowanie szybkie ($O(n \log n)$).
- **Algorytm Karacuby** – szybkie mnożenie dużych liczb ($O(n^{1.585})$).

4.4 Programowanie dynamiczne (Dynamic Programming)

Programowanie dynamiczne jest techniką polegającą na zapamiętywaniu wyników obliczeń podproblemów, aby uniknąć ich powtarzania.

4.4.1 Cechy programowania dynamicznego

- **Optymalna podstruktura** – rozwiązanie problemu można skonstruować z optymalnych rozwiązań podproblemów.
- **Nakładanie się podproblemów** – te same podproblemy pojawiają się wielokrotnie.

4.4.2 Przykłady algorytmów dynamicznych

- **Problem plecakowy (wersja dokładna)** – znajdowanie maksymalnej wartości przedmiotów mieszczących się w plecaku.
- **Cięcie pręta** – maksymalizacja wartości pręta po jego podziale.
- **Najdłuższy wspólny podciąg (LCS)** – znajdowanie najdłuższego wspólnego podciągu dwóch ciągów znaków.
- **Fibonacci** – rekurencyjne wyliczanie liczb Fibonacciego z pamięcią podręczną ($O(n)$ zamiast $O(2^n)$).

4.5 Przeszukiwanie z nawrotami (Backtracking)

Technika ta polega na systematycznym eksplorowaniu wszystkich możliwych rozwiązań poprzez cofanie się, gdy napotka się ślepą uliczkę.

4.5.1 Przykłady algorytmów wykorzystujących backtracking

- **Problem N-hetmanów** – umieszczenie N hetmanów na szachownicy tak, aby się nie atakowały.
- **Sudoku** – znajdowanie poprawnego rozwiązania łamigłówki sudoku.
- **Problem Hamiltona** – znalezienie cyklu Hamiltona w grafie.

4.6 Metoda programowania liniowego

Programowanie liniowe pozwala na optymalizację funkcji celu przy ograniczeniach wyrażonych nierównościami liniowymi. Do jego rozwiązania stosuje się m.in. algorytm sympleksowy.

4.6.1 Przykłady zastosowań

- Optymalizacja kosztów produkcji.
- Maksymalizacja zysków w ograniczonych zasobach.

4.7 Heurystyki i algorytmy probabilistyczne

Niektóre problemy są trudne do rozwiązania w czasie wielomianowym. W takich przypadkach stosuje się podejścia heurystyczne lub probabilistyczne, które nie gwarantują optymalnego rozwiązania, ale znajdują przybliżone wyniki w akceptowalnym czasie.

4.7.1 Przykłady algorytmów heurystycznych i probabilistycznych

- **Algorytmy genetyczne** – inspirowane teorią ewolucji, stosowane w optymalizacji.
- **Symulowane wyżarzanie (Simulated Annealing)** – metoda optymalizacji, inspirowana procesem hartowania metali.
- **Monte Carlo** – algorytmy wykorzystujące losowe próbkowanie do przybliżonych obliczeń.

4.8 Podsumowanie

Każda z przedstawionych strategii ma swoje zastosowania w różnych klasach problemów:

- **Algorytmy zachłanne** są szybkie, ale nie zawsze znajdują optymalne rozwiązanie.
- **Dziel i zwyciężaj** pozwala na rekurencyjne podejście do problemów.
- **Programowanie dynamiczne** pozwala unikać zbędnych obliczeń poprzez zapamiętywanie wyników.
- **Backtracking** jest stosowany w problemach wymagających eksploracji przestrzeni rozwiązań.
- **Heurystyki** są użyteczne w problemach, dla których nie istnieją szybkie algorytmy dokładne.

Wybór odpowiedniej strategii zależy od natury problemu oraz wymaganej efektywności rozwiązania.

5 Różnice pomiędzy układami kombinacyjnymi i sekwencyjnymi oraz znaczenie takiego podziału w procesie projektowania

5.1 Wprowadzenie

Układy logiczne dzielimy na:

- **Układy kombinacyjne** – wyjście zależy wyłącznie od aktualnych wartości wejściowych.
- **Układy sekwencyjne** – wyjście zależy zarówno od aktualnych wartości wejściowych, jak i od poprzednich stanów systemu.

Podział ten jest kluczowy w projektowaniu systemów cyfrowych, ponieważ pozwala określić sposób przetwarzania informacji i sterowania.

5.2 Układy kombinacyjne

5.2.1 Definicja

Układy kombinacyjne to układy logiczne, w których wyjście zależy wyłącznie od wartości wejść w danej chwili, bez pamięci stanów poprzednich.

5.2.2 Charakterystyka

- Brak elementów pamiętających – brak rejestrowania poprzednich wartości sygnałów.
- Wyjście zmienia się natychmiast po zmianie wejścia (z niewielkim opóźnieniem wynikającym z propagacji sygnału).
- Opis układu można wyrazić za pomocą funkcji logicznych zmiennych wejściowych.

5.2.3 Przykłady układów kombinacyjnych

- Bramki logiczne (AND, OR, NOT, XOR).
- Sumatory (dodające liczby binarne).
- Multipleksery i demultipleksery.
- Enkodery i dekodery.
- Układy realizujące tablice prawdy.

5.3 Układy sekwencyjne

5.3.1 Definicja

Układy sekwencyjne to układy logiczne, w których wyjście zależy zarówno od aktualnych wartości wejściowych, jak i od wcześniejszego stanu układu.

5.3.2 Charakterystyka

- Posiadają elementy pamiętające (np. przerzutniki).
- Reagują na zmiany wejść, ale ich stan zależy także od wcześniejszych wartości sygnałów.
- Do ich opisu stosuje się tabele stanów i diagramy przejść.

5.3.3 Przykłady układów sekwencyjnych

- Liczniki (np. licznik binarny, Johnsona, pierścieniowy).
- Rejestry przesuwnych.
- Automaty skończone (np. układy sterujące).
- Pamięci cyfrowe (np. SRAM, DRAM).

5.4 Różnice między układami kombinacyjnymi i sekwencyjnymi

Cecha	Układy kombinacyjne	Układy sekwencyjne
Zależność wyjścia	Tylko od wejść	Od wejść i poprzednich stanów
Elementy pamięci	Brak	Przerzutniki, rejestry
Czas odpowiedzi	Natychmiastowy	Wymaga taktowania
Opis matematyczny	Funkcje boolowskie	Tabele i diagramy stanów
Przykłady	Bramki, sumatory	Liczniki, rejestry

Tabela 1: Porównanie układów kombinacyjnych i sekwencyjnych

5.5 Znaczenie podziału w procesie projektowania

Podział na układy kombinacyjne i sekwencyjne jest kluczowy w projektowaniu systemów cyfrowych. Pozwala on na:

- **Optymalizację** – wybór odpowiedniej struktury minimalizuje zużycie zasobów sprzętowych.
- **Podział funkcjonalny** – rozdzielenie logiki sterującej (sekwencyjnej) i przetwarzania danych (kombinacyjnego).
- **Projektowanie układów synchronicznych** – układy sekwencyjne wymagają synchronizacji zegara, co jest istotne dla działania procesorów i pamięci.
- **Elastyczność w implementacji** – układy kombinacyjne można realizować w układach FPGA i ASIC, a układy sekwencyjne są podstawą mikroprocesorów.

5.6 Podsumowanie

- Układy kombinacyjne realizują funkcje logiczne bez pamięci stanu, a układy sekwencyjne wykorzystują pamięć do przechowywania poprzednich stanów.
- Układy sekwencyjne są bardziej złożone, ale niezbędne w sterowaniu i automatyce.
- Podział ten jest kluczowy w projektowaniu systemów cyfrowych i pozwala efektywnie organizować architekturę układów elektronicznych.

6 Podstawowe cechy języka programowania, kwalifikujące do zaliczenia do grupy języków zorientowanych obiektowo

6.1 Wprowadzenie

Programowanie obiektowe (ang. *Object-Oriented Programming, OOP*) to paradygmat programowania, który organizuje kod wokół obiektów – jednostek łączących dane oraz metody operujące na tych danych. Język programowania można uznać za obiektowy, jeśli spełnia pewne kluczowe cechy i zasady.

6.2 Podstawowe cechy języków obiektowych

Język programowania jest klasyfikowany jako obiektowy, jeśli wspiera następujące koncepcje:

6.2.1 1. Abstrakcja (*Abstraction*)

Abstrakcja polega na ukrywaniu szczegółów implementacji i eksponowaniu tylko istotnych właściwości obiektów. Dzięki temu użytkownik korzysta z interfejsu klasy, nie znając jej wewnętrznych mechanizmów.

Przykład w C++:

```
class Samochod {
private:
    string marka;
public:
    void ustawMarke(string m) { marka = m; }
    string pobierzMarke() { return marka; }
};
```

6.2.2 2. Hermetyzacja (*Encapsulation*)

Hermetyzacja oznacza ograniczenie dostępu do wewnętrznych danych obiektu. Osiąga się to poprzez modyfikatory dostępu (`private`, `protected`, `public`) i ukrycie zmiennych wewnątrz klasy.

Zalety hermetyzacji:

- Ochrona przed nieautoryzowanym dostępem do danych.
- Zapewnienie integralności danych poprzez kontrolowany dostęp.

6.2.3 3. Dziedziczenie (*Inheritance*)

Dziedziczenie pozwala na tworzenie nowych klas na podstawie już istniejących. Klasa pochodna (*subclass*) dziedziczy cechy i metody klasy bazowej (*superclass*), co pozwala na ponowne użycie kodu.

Przykład w C++:

```
class Pojazd {
public:
    void info() { cout << "To jest pojazd"; }
};

class Samochod : public Pojazd {
};
```

Tutaj `Samochod` dziedziczy metodę `info()` z klasy `Pojazd`.

6.2.4 4. Polimorfizm (*Polymorphism*)

Polimorfizm umożliwia definiowanie wielu wersji tej samej metody, działających na różnych typach danych. Wyróżnia się:

- **Polimorfizm statyczny** (przeciążanie funkcji i operatorów).
- **Polimorfizm dynamiczny** (metody wirtualne i nadpisywanie metod).

Przykład polimorfizmu dynamicznego w C++:

```
class Pojazd {
public:
    virtual void dzwiek() { cout << "Ogólny dźwięk pojazdu"; }
};

class Samochod : public Pojazd {
public:
    void dzwiek() override { cout << "Dźwięk silnika samochodu"; }
};
```

6.3 Dodatkowe cechy języków obiektowych

Oprócz czterech podstawowych filarów OOP, języki obiektowe często wspierają również inne mechanizmy:

6.3.1 1. Klasy i obiekty

Klasy są szablonami do tworzenia obiektów, a obiekty są instancjami klas.

Przykład w C++:

```
class Osoba {
public:
    string imie;
    int wiek;
};

int main() {
    Osoba o1;
    o1.imie = "Jan";
    o1.wiek = 30;
}
```

6.3.2 2. Konstruktor i destruktor

Konstruktor to specjalna metoda wywoływana podczas tworzenia obiektu, a destruktor – podczas usuwania obiektu.

```
class Samochod {
public:
    Samochod() { cout << "Tworzę samochód"; }
    ~Samochod() { cout << "Usuwam samochód"; }
};
```

6.3.3 3. Interfejsy i klasy abstrakcyjne

Interfejsy i klasy abstrakcyjne umożliwiają definiowanie wspólnych metod dla różnych klas.

Przykład w C++:

```
class Figura {
public:
    virtual void rysuj() = 0; // Klasa abstrakcyjna
};
```

6.4 Znaczenie obiektowego podejścia w projektowaniu oprogramowania

Programowanie obiektowe ma kluczowe znaczenie w projektowaniu oprogramowania:

- **Reużywalność kodu** – dzięki dziedziczeniu możliwe jest ponowne użycie już istniejącego kodu.
- **Łatwiejsza konserwacja** – hermetyzacja zapewnia lepszą kontrolę nad modyfikacjami kodu.
- **Lepsza organizacja kodu** – klasy i obiekty pozwalają na logiczne grupowanie danych i metod.
- **Łatwiejsza skalowalność** – programowanie obiektowe umożliwia łatwe dodawanie nowych funkcjonalności.

6.5 Przykłady języków zorientowanych obiektowo

- **C++** – język wieloparadygmatowy, umożliwiający zarówno programowanie strukturalne, jak i obiektowe.
- **Java** – język w pełni obiektowy, gdzie każda klasa dziedziczy po klasie nadrzędnej `Object`.
- **Python** – dynamiczny język z pełnym wsparciem dla programowania obiektowego.
- **C#** – język silnie związany z platformą .NET, wspierający zarówno OOP, jak i programowanie funkcyjne.

6.6 Podsumowanie

- Języki obiektowe charakteryzują się abstrakcją, hermetyzacją, dziedziczeniem i polimorfizmem.
- Obiektowe podejście ułatwia organizację kodu, reużywalność i konserwację.
- Do najpopularniejszych języków OOP należą C++, Java, Python i C#.

7 Rodzaje komunikatów niewerbalnych

7.1 Wprowadzenie

Komunikacja niewerbalna obejmuje wszelkie formy przekazywania informacji, które nie wykorzystują słów. Obejmuje gesty, mimikę, postawę ciała, kontakt wzrokowy, ton głosu oraz inne sposoby wyrażania emocji i intencji.

7.2 Podstawowe rodzaje komunikacji niewerbalnej

7.2.1 1. Mowa ciała (*Kinezjetyka*)

Mowa ciała to ruchy i postawy ciała, które przekazują emocje oraz nastawienie rozmówcy. Do najważniejszych elementów należą:

- **Gesty** – np. kiwanie głową, machanie ręką, skrzyżowanie rąk na klatce piersiowej.
- **Mimika twarzy** – ruchy mięśni twarzy, które odzwierciedlają emocje (np. uśmiech, zmarszczenie brwi).
- **Postawa ciała** – sposób, w jaki stoimy lub siedzimy, może wyrażać otwartość, zamknięcie, pewność siebie lub uległość.

7.2.2 2. Kontakt wzrokowy

Oczy odgrywają kluczową rolę w komunikacji niewerbalnej. Kontakt wzrokowy może oznaczać:

- Zainteresowanie rozmową (utrzymanie kontaktu wzrokowego).
- Niepewność lub uległość (unikanie kontaktu wzrokowego).
- Dominację lub agresję (intensywne spojrzenie).

7.2.3 3. Proksemika (dystans interpersonalny)

Proksemika to badanie przestrzeni między rozmówcami. Wyróżnia się cztery strefy dystansu:

- **Strefa intymna** (0–45 cm) – bliskie relacje, rodzina, partnerzy.
- **Strefa osobista** (45–120 cm) – rozmowy z przyjaciółmi i znajomymi.
- **Strefa społeczna** (120–360 cm) – kontakty zawodowe.
- **Strefa publiczna** (powyżej 360 cm) – wykłady, przemówienia.

7.2.4 4. Parajęzyk (cechy wokalne)

Parajęzyk to sposób, w jaki mówimy, niezależnie od treści wypowiedzi. Obejmuje:

- **Ton głosu** – może wyrażać emocje, np. radość, smutek, złość.
- **Tempo mówienia** – szybkie mówienie może wskazywać na zdenerwowanie, wolne na powagę.
- **Natężenie głosu** – głośność wypowiedzi wpływa na odbiór emocji.

7.2.5 5. Dotyk (*Haptyka*)

Dotyk jest istotnym elementem komunikacji, szczególnie w relacjach międzyludzkich. Może oznaczać:

- Sympatię (uścisk dłoni, poklepanie po plecach).
- Dominację (silny uścisk dłoni).
- Komfort lub pocieszenie (przytulenie).

7.2.6 6. Wygląd zewnętrzny

Ubiór, fryzura, biżuteria i ogólny wygląd wpływają na sposób postrzegania danej osoby. Może on sugerować status społeczny, profesjonalizm, kreatywność lub przynależność do określonej grupy.

7.2.7 7. Chronemika (zarządzanie czasem)

Sposób, w jaki ludzie zarządzają czasem, może być formą komunikacji niewerbalnej:

- Punktualność świadczy o szacunku do rozmówcy.
- Spóźnienia mogą być odbierane jako brak organizacji lub lekceważenie.

7.2.8 8. Artefakty (przedmioty)

Przedmioty, które nosimy lub używamy, mogą przekazywać informacje o statusie społecznym, zainteresowaniach lub osobowości. Przykłady:

- Markowe ubrania mogą sugerować prestiż.
- Kolorystyka i styl biżuterii mogą podkreślać indywidualność.

7.3 Podsumowanie

Komunikacja niewerbalna odgrywa kluczową rolę w interakcjach międzyludzkich. Jej główne elementy to:

- **Mowa ciała** (gesty, mimika, postawa ciała).
- **Kontakt wzrokowy** (utrzymanie lub unikanie spojrzenia).
- **Proksemika** (dystans interpersonalny).
- **Parajęzyk** (ton głosu, tempo mówienia).
- **Dotyk** (haptyka – uściski dłoni, przytulenia).
- **Wygląd zewnętrzny** (ubiór, fryzura, akcesoria).
- **Chronemika** (sposób zarządzania czasem).
- **Artefakty** (przedmioty symbolizujące status i osobowość).

Rozumienie komunikacji niewerbalnej pozwala na lepszą interpretację intencji rozmówców i skuteczniejsze przekazywanie informacji w różnych kontekstach społecznych i zawodowych.

8 Metody rozwiązywania układów równań liniowych stosowane do bardzo dużych układów równań (wraz z uzasadnieniem)

8.1 Wprowadzenie

Układy równań liniowych pojawiają się w wielu dziedzinach nauki i techniki, takich jak analiza danych, fizyka, inżynieria czy sztuczna inteligencja. W przypadku bardzo dużych układów, standardowe metody analityczne okazują się niewydajne ze względu na wysoką złożoność obliczeniową i wymagania pamięciowe.

Wyróżnia się dwie główne klasy metod:

- **Metody bezpośrednie** – dają dokładne rozwiązanie w skończonej liczbie kroków, ale mogą być kosztowne obliczeniowo.
- **Metody iteracyjne** – pozwalają uzyskać rozwiązanie przybliżone w sposób efektywny, szczególnie dla układów rzadkich.

8.2 Metody bezpośrednie

8.2.1 1. Metoda eliminacji Gaussa

Metoda ta polega na przekształceniu macierzy współczynników do postaci trójkątnej, a następnie rozwiązaniu układu równań poprzez podstawianie wsteczne.

Zalety:

- Dokładne rozwiązanie w skończonej liczbie operacji.
- Może być efektywna dla małych i średnich układów.

Wady:

- Dla dużych układów $O(n^3)$ operacji sprawia, że metoda staje się niepraktyczna.
- Może być niestabilna numerycznie dla źle uwarunkowanych macierzy.

8.2.2 2. Metoda faktoryzacji LU

Metoda ta polega na rozkładzie macierzy A na iloczyn macierzy dolnotrójkątnej L i górnotrójkątnej U, tj. $A = LU$, co umożliwia szybkie rozwiązanie układu.

Zalety:

- Skuteczniejsza od eliminacji Gaussa w przypadku wielokrotnego rozwiązywania układów z tą samą macierzą A.

Wady:

- Koszt obliczeniowy porównywalny z eliminacją Gaussa ($O(n^3)$).
- Wymaga pełnej macierzy współczynników, co może być problematyczne dla układów rzadkich.

8.2.3 3. Metoda faktoryzacji Cholesky'ego

Jest to specjalny przypadek faktoryzacji LU, stosowany dla macierzy symetrycznych i dodatnio określonych. Polega na dekompozycji $A = LL^T$.

Zalety:

- Szybsza niż ogólna faktoryzacja LU (koszt $O(n^3/3)$).

Wady:

- Ograniczone zastosowanie tylko do macierzy symetrycznych dodatnio określonych.

8.3 Metody iteracyjne

Dla bardzo dużych układów równań, szczególnie gdy macierz jest rzadka, korzystniejsze okazują się metody iteracyjne, które pozwalają uzyskać przybliżone rozwiązanie w krótszym czasie.

8.3.1 1. Metoda Jacobiego

Metoda Jacobiego polega na iteracyjnym poprawianiu przybliżeń, bazując na wartości z poprzedniego kroku:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}}{a_{ii}}.$$

Zalety:

- Nadaje się do obliczeń równoległych.
- Prosta w implementacji.

Wady:

- Wolna zbieżność.
- Działa tylko dla macierzy o dominującej przekątnej.

8.3.2 2. Metoda Gaussa-Seidla

Jest modyfikacją metody Jacobiego, w której nowe wartości $x_i^{(k+1)}$ są wykorzystywane natychmiast w dalszych obliczeniach.

Zalety:

- Szybsza zbieżność niż metoda Jacobiego.

Wady:

- Nie zawsze gwarantuje zbieżność.

8.3.3 3. Metoda gradientu sprzężonego

Jest jedną z najskuteczniejszych metod iteracyjnych do rzadkich układów równań liniowych. Opiera się na minimalizacji funkcji kwadratowej:

$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}.$$

Zalety:

- Złożoność $O(n)$ w wielu przypadkach, co jest dużą poprawą w stosunku do metod bezpośrednich.
- Skuteczna dla dużych, rzadkich macierzy.

Wady:

- Wymaga macierzy symetrycznej i dodatnio określonej.

8.3.4 4. Metody wielosiatkowe

Metody wielosiatkowe (*Multigrid methods*) rozwiązują problem na różnych poziomach siatki, co przyspiesza zbieżność.

Zalety:

- Bardzo szybka zbieżność (często $O(n)$).

Wady:

- Trudność w implementacji i dostosowaniu do konkretnego problemu.

8.4 Podsumowanie

Wybór metody zależy od charakterystyki układu równań:

- Metody bezpośrednie (Gaussa, LU, Cholesky'ego) są dokładne, ale kosztowne obliczeniowo ($O(n^3)$).
- Metody iteracyjne (Jacobiego, Gaussa-Seidla, gradientu sprzężonego) są efektywne dla dużych, rzadkich układów.
- Metody wielosiatkowe są jednymi z najszybszych, ale trudniejsze w implementacji.

W praktyce dla bardzo dużych układów równań liniowych najczęściej stosuje się metody iteracyjne, zwłaszcza metodę gradientu sprzężonego i metody wielosiatkowe, ze względu na ich dobrą skalowalność i efektywność pamięciową.

9 Charakterystyka wybranej metody poszukiwania minimum funkcji jednej zmiennej

9.1 Wprowadzenie

Poszukiwanie minimum funkcji jednej zmiennej jest kluczowym zagadnieniem w optymalizacji. Istnieje wiele metod rozwiązania tego problemu, które można podzielić na:

- **Metody analityczne** – bazujące na pochodnych funkcji.
- **Metody numeryczne** – bazujące na iteracyjnym przeszukiwaniu przedziału.

Wśród metod numerycznych jedną z najczęściej stosowanych jest **metoda złotego podziału**, którą omówimy poniżej.

9.2 Metoda złotego podziału

Metoda złotego podziału (ang. *Golden Section Search*) jest algorytmem optymalizacyjnym do znajdowania minimum funkcji jednej zmiennej bez konieczności obliczania pochodnych. Opiera się na iteracyjnym dzieleniu przedziału poszukiwań w stosunku złotej liczby.

9.2.1 Zasada działania

Metoda działa według następujących kroków:

1. Wybieramy początkowy przedział poszukiwań $[a, b]$.
2. Obliczamy dwa punkty wewnętrzne:

$$x_1 = b - \tau(b - a), \quad x_2 = a + \tau(b - a),$$

   gdzie $\tau = \frac{\sqrt{5}-1}{2} \approx 0.618$ to współczynnik złotej proporcji.
3. Obliczamy wartości funkcji w punktach $f(x_1)$ i $f(x_2)$.
4. Eliminujemy część przedziału, w której minimum na pewno się nie znajduje:
   - Jeśli $f(x_1) > f(x_2)$, to nowe $a = x_1$.
   - Jeśli $f(x_1) < f(x_2)$, to nowe $b = x_2$.
5. Powtarzamy procedurę do momentu, aż długość przedziału będzie mniejsza niż zadana tolerancja ε.

9.2.2 Zalety metody złotego podziału

- **Nie wymaga obliczania pochodnych**, co jest istotne dla funkcji nieregularnych lub kosztownych obliczeniowo.
- **Gwarantuje zbieżność** do minimum w skończonej liczbie kroków.
- **Efektywność** – każde skrócenie przedziału wymaga jedynie jednej dodatkowej oceny funkcji.

9.2.3 Wady metody złotego podziału

- **Nie jest superszybka** – zbieżność jest liniowa, co oznacza, że potrzeba wielu iteracji dla wysokiej dokładności.
- **Działa tylko dla jednej zmiennej** – nie nadaje się do optymalizacji funkcji wielu zmiennych.
- **Wymaga znajomości przedziału początkowego** – nie zawsze jest łatwo określić dobry zakres poszukiwań.

9.3 Podsumowanie

Metoda złotego podziału jest skutecznym narzędziem do znajdowania minimum funkcji jednej zmiennej, gdy nie można wykorzystać metod bazujących na pochodnych. Dzięki eliminacji zbędnych przedziałów poszukiwań umożliwia efektywne zbliżanie się do minimum, choć jej zbieżność nie jest tak szybka jak w metodach gradientowych.

10 Procesy i wątki: definicje, cechy wspólne i różnice, metody tworzenia procesów i wątków w różnych systemach operacyjnych

10.1 Definicje

10.1.1 Proces

Proces to program w trakcie wykonywania, który posiada własną przestrzeń adresową oraz zasoby przydzielone przez system operacyjny.

Cechy procesu:

- Każdy proces ma własną przestrzeń adresową w pamięci.
- Posiada przynajmniej jeden wątek wykonania.
- Komunikacja między procesami (*Inter-Process Communication, IPC*) wymaga mechanizmów systemowych (np. kolejki komunikatów, potoki, pamięć współdzielona).

10.1.2 Wątek

Wątek (ang. *thread*) to najmniejsza jednostka wykonania w obrębie procesu. Wątki współdzielą przestrzeń adresową i zasoby procesu.

Cechy wątku:

- Wszystkie wątki w procesie działają w tej samej przestrzeni adresowej.
- Wątki mogą współdzielić dane globalne i zasoby.
- Komunikacja między wątkami jest szybka, gdyż odbywa się poprzez pamięć współdzieloną.

10.2 Cechy wspólne procesów i wątków

- Zarówno procesy, jak i wątki są jednostkami wykonywania programu.
- Mogą być tworzone dynamicznie w trakcie działania systemu.
- Współbieżność – systemy wielozadaniowe pozwalają na równoczesne wykonywanie wielu procesów i wątków.

10.3 Różnice między procesami a wątkami

10.4 Metody tworzenia procesów i wątków w różnych systemach operacyjnych

10.4.1 Tworzenie procesów

1. Unix/Linux: Najczęściej stosowaną metodą tworzenia nowego procesu jest funkcja `fork()`.

Cecha	Proces	Wątek
Przestrzeń adresowa	Oddzielna dla każdego procesu	Współdzielona między wątkami w procesie
Zasoby	Każdy proces posiada własne zasoby	Wątki współdzielą zasoby procesu
Przełączanie kontekstu	Kosztowne, wymaga przełączenia pamięci	Szybkie, ponieważ przestrzeń adresowa jest wspólna
Komunikacja	Wymaga mechanizmów IPC	Łatwa i szybka dzięki pamięci współdzielonej
Zależność	Procesy są niezależne	Wątki mogą na siebie oddziaływać

Tabela 2: Porównanie procesów i wątków

```
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Proces potomny\n");
    } else {
        printf("Proces rodzicielski\n");
    }
    return 0;
}
```

Innym podejściem jest `exec()`, które zastępuje obraz procesu nowym programem.

2. Windows: W systemie Windows nowy proces tworzy się za pomocą `CreateProcess()`.

```
#include <windows.h>

int main() {
    STARTUPINFO si = { sizeof(si) };
    PROCESS_INFORMATION pi;

    CreateProcess(NULL, "notepad.exe", NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);

    WaitForSingleObject(pi.hProcess, INFINITE);
    return 0;
}
```

10.4.2 Tworzenie wątków

1. Unix/Linux: Wątki w systemach uniksowych tworzy się za pomocą biblioteki POSIX Threads (pthread):

```
#include <pthread.h>
#include <stdio.h>

void* funkcja_watku(void* arg) {
    printf("Wątek uruchomiony\n");
    return NULL;
}

int main() {
    pthread_t watek;
    pthread_create(&watek, NULL, funkcja_watku, NULL);
    pthread_join(watek, NULL);
    return 0;
}
```

2. Windows: W systemie Windows wątki tworzy się za pomocą funkcji `CreateThread()`:

```
#include <windows.h>

DWORD WINAPI funkcja_watku(LPVOID lpParam) {
    printf("Wątek uruchomiony\n");
    return 0;
}

int main() {
    HANDLE watek = CreateThread(NULL, 0, funkcja_watku, NULL, 0, NULL);
    WaitForSingleObject(watek, INFINITE);
    return 0;
}
```

10.5 Podsumowanie

- Procesy to niezależne jednostki wykonawcze, posiadające własne zasoby i przestrzeń adresową.
- Wątki działają w obrębie procesu i współdzielą jego pamięć oraz zasoby.
- Tworzenie nowych procesów w systemach Unix/Linux odbywa się głównie poprzez `fork()`, a w Windows poprzez `CreateProcess()`.
- Tworzenie wątków w Unix/Linux odbywa się za pomocą biblioteki POSIX Threads (`pthread`), a w Windows przy użyciu `CreateThread()`.
- Wątki są bardziej efektywne pod względem wydajności niż procesy, ale wymagają synchronizacji, aby uniknąć konfliktów dostępu do zasobów.

11 Tablice mieszające

11.1 Wprowadzenie

Tablice mieszające (ang. *hash tables*) to struktury danych, które umożliwiają szybkie wyszukiwanie, wstawianie i usuwanie elementów. Wykorzystują funkcję mieszającą (*hash function*), która przekształca klucz na indeks w tablicy, umożliwiając efektywne przechowywanie i dostęp do danych.

11.2 Zasada działania

Tablica mieszająca działa w oparciu o funkcję mieszającą $h(k)$, która przekształca klucz k w indeks tablicy:

$$h(k) = k \mod m$$

gdzie m to rozmiar tablicy mieszającej.

Przykład: Jeśli $m = 10$ i klucz $k = 27$, to:

$$h(27) = 27 \mod 10 = 7.$$

Oznacza to, że element o kluczu 27 zostanie zapisany w komórce 7.

11.3 Funkcje mieszające

Dobra funkcja mieszająca powinna:

- generować równomierny rozkład wartości,
- być szybka do obliczenia,
- minimalizować liczbę kolizji.

Przykładowe funkcje mieszające:

- Funkcja resztowa: $h(k) = k \mod m$.
- Mnożeniowa: $h(k) = \lfloor m(kA \mod 1) \rfloor$, gdzie A to stała (np. $A \approx 0.618$).
- Kryptograficzne funkcje mieszające: MD5, SHA-256 (stosowane w systemach bezpieczeństwa).

11.4 Kolizje i ich rozwiązywanie

Kolizja występuje, gdy dwa różne klucze generują ten sam indeks w tablicy mieszającej. Można je rozwiązać na kilka sposobów:

11.4.1 1. Łańcuchowanie (*Chaining*)

Każda komórka tablicy zawiera listę elementów o tym samym indeksie mieszania.

```
class HashTable {
    vector<list<int>> table;
public:
    void insert(int key) {
        int index = key % table.size();
        table[index].push_back(key);
    }
};
```

11.4.2 2. Otwarta adresacja (*Open Addressing*)

Gdy kolizja wystąpi, szukamy innego miejsca w tablicy.

Przykładowe strategie rozwiązywania kolizji:

- **Liniowe próbkowanie**: $h(k, i) = (h(k) + i) \mod m$.
- **Kwadratowe próbkowanie**: $h(k, i) = (h(k) + i^2) \mod m$.
- **Podwójne mieszanie**: $h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m$.

11.5 Złożoność czasowa

- **Średni przypadek**: $O(1)$ dla wstawiania, wyszukiwania i usuwania.
- **Najgorszy przypadek**: $O(n)$, gdy wszystkie klucze trafiają do jednej komórki (np. przy złej funkcji mieszającej).

11.6 Zastosowania tablic mieszających

- Implementacja słowników i map (`std::unordered_map` w C++).
- Buforowanie danych (np. DNS cache).
- Algorytmy kryptograficzne i struktury danych do sprawdzania duplikatów.

11.7 Podsumowanie

- Tablice mieszające to wydajna struktura danych umożliwiająca szybkie operacje.
- Odpowiednia funkcja mieszająca i strategia obsługi kolizji są kluczowe dla wydajności.
- Są szeroko stosowane w bazach danych, kompilatorach i systemach operacyjnych.

12 Hierarchia pamięci, z uwzględnieniem pamięci podręcznej oraz pamięci wirtualnej

12.1 Wprowadzenie

Hierarchia pamięci to struktura organizacyjna systemu komputerowego, która ma na celu optymalne zarządzanie danymi i ich dostępnością. Pamięci o wysokiej szybkości są kosztowne i mają ograniczoną pojemność, natomiast pamięci o dużej pojemności są wolniejsze. Dlatego stosuje się wielopoziomową hierarchię, w której pamięć szybsza działa jako bufor dla pamięci wolniejszej.

12.2 Poziomy hierarchii pamięci

Hierarchia pamięci składa się z kilku poziomów, uporządkowanych według rosnącego czasu dostępu i malejącej szybkości:

- **Rejestry procesora** – najszybsza pamięć, bezpośrednio dostępna przez CPU.
- **Pamięć podręczna (cache)** – szybka pamięć buforowa, redukująca liczbę dostępów do pamięci RAM.
- **Pamięć operacyjna (RAM)** – główna pamięć komputera, przechowująca dane i kod wykonywanego programu.
- **Pamięć wirtualna** – mechanizm symulujący dodatkową pamięć RAM na dysku twardym.
- **Pamięć masowa** – dyski SSD, HDD, nośniki optyczne przechowujące dane trwałe.
- **Pamięć zewnętrzna** – nośniki wymienne, serwery sieciowe, chmura obliczeniowa.

12.3 Pamięć podręczna (cache)

Pamięć podręczna jest małą, ale bardzo szybką pamięcią umieszczoną blisko jednostki centralnej (CPU). Przechowuje często używane dane, co znacząco przyspiesza działanie systemu.

12.3.1 Poziomy pamięci cache

- **L1** – najszybsza i najmniejsza (kilka-kilkadziesiąt KB), bezpośrednio połączona z rdzeniem procesora.
- **L2** – większa (setki KB – kilka MB), współdzielona przez kilka rdzeni.
- **L3** – najwolniejsza z cache procesora, ale największa (kilka MB – kilkadziesiąt MB), wspólna dla wszystkich rdzeni.

12.3.2 Zasada działania pamięci podręcznej

Dane w pamięci podręcznej są przechowywane zgodnie z zasadą lokalności:

- **Lokalność czasowa** – jeśli dany blok pamięci został użyty, istnieje duże prawdopodobieństwo, że wkrótce zostanie ponownie użyty.
- **Lokalność przestrzenna** – jeśli odczytano pewien obszar pamięci, sąsiednie adresy prawdopodobnie będą wkrótce potrzebne.

12.3.3 Strategie zarządzania pamięcią podręczną

- **Mapowanie bezpośrednie** – każdemu blokowi pamięci RAM odpowiada dokładnie jedna linia cache.
- **Mapowanie skojarzeniowe** – każdy blok pamięci może być przechowywany w dowolnym miejscu cache.
- **Mapowanie skojarzeniowe zestawowe** – blok pamięci może być umieszczony w jednym z kilku określonych miejsc.

12.3.4 Wskaźnik trafień (hit rate)

Efektywność pamięci podręcznej określa się za pomocą wskaźnika trafień:

$$Hit\ rate = \frac{\text{Liczba trafień}}{\text{Liczba wszystkich żądań}}$$

Im wyższy wskaźnik trafień, tym lepsza efektywność pamięci podręcznej.

12.4 Pamięć wirtualna

Pamięć wirtualna to technika pozwalająca na zwiększenie dostępnej pamięci operacyjnej poprzez wykorzystanie przestrzeni na dysku twardym jako dodatkowej pamięci.

12.4.1 Zasada działania

Gdy ilość dostępnej pamięci RAM jest niewystarczająca, system operacyjny przenosi rzadko używane strony pamięci na dysk twardy (pliki stronicowania lub wymiany). Proces ten nosi nazwę **stronicowania**.

12.4.2 Zalety pamięci wirtualnej

- Pozwala uruchamiać programy wymagające więcej pamięci, niż jest fizycznie dostępne.
- Izoluje procesy, zapobiegając ich wzajemnemu nadpisywaniu pamięci.

12.4.3 Wady pamięci wirtualnej

- **Wolniejsza niż RAM** – dostęp do danych na dysku jest znacznie wolniejszy niż w pamięci operacyjnej.
- **Zjawisko thrashingu** – częste przenoszenie stron między RAM a dyskiem może prowadzić do spadku wydajności.

12.4.4 Mechanizmy zarządzania pamięcią wirtualną

- **Stronicowanie** – podział pamięci na strony o stałej wielkości, które mogą być przenoszone między RAM a dyskiem.
- **Segmentacja** – dzielenie pamięci na segmenty odpowiadające logicznym częściom programu.

12.5 Podsumowanie

- Hierarchia pamięci obejmuje różne poziomy, od szybkich rejestrów po wolne nośniki zewnętrzne.
- Pamięć podręczna redukuje liczbę operacji dostępu do RAM, poprawiając wydajność procesora.
- Pamięć wirtualna umożliwia symulację większej pamięci RAM, ale może powodować spadek wydajności z powodu operacji wymiany stron.
- Efektywność pamięci zależy od wskaźnika trafień w pamięci podręcznej i optymalnego zarządzania pamięcią wirtualną.

13 Pojęcie i kategorie wynalazku

13.1 Definicja wynalazku

Wynalazek to nowe i użyteczne rozwiązanie techniczne dotyczące produktu lub sposobu wytwarzania, które można zastosować w przemyśle. Wynalazki są chronione prawnie poprzez patenty, które zapewniają wyłączne prawo do korzystania z wynalazku przez określony czas.

Zgodnie z **Ustawą – Prawo własności przemysłowej** (Dz.U. 2001 nr 49 poz. 508), wynalazkiem jest rozwiązanie techniczne spełniające następujące warunki:

- **Nowość** – wynalazek nie może być częścią stanu techniki.
- **Poziom wynalazczy** – nie może być oczywisty dla osoby znającej stan techniki.
- **Przemysłowa stosowalność** – wynalazek musi mieć zastosowanie w przemyśle, rolnictwie, leśnictwie lub innych dziedzinach gospodarki.

13.2 Kategorie wynalazków

Wynalazki można podzielić na kilka kategorii w zależności od ich charakteru i zastosowania.

13.2.1 1. Wynalazki produktowe

Obejmują nowe urządzenia, substancje chemiczne, materiały i kompozycje. Wynalazki produktowe mogą obejmować:

- Nowe materiały (np. stop metalu, polimer).
- Leki i substancje farmaceutyczne.
- Urządzenia techniczne (np. nowy typ silnika, układu elektronicznego).

13.2.2 2. Wynalazki procesowe

Dotyczą nowych sposobów wytwarzania lub przetwarzania materiałów i produktów. Przykłady:

- Nowa metoda syntezy chemicznej.
- Usprawniony proces produkcji mikroprocesorów.
- Technika zwiększająca efektywność energetyczną.

13.2.3 3. Wynalazki dotyczące zastosowania

Obejmują nowe sposoby użycia znanych substancji, urządzeń lub metod w nowym kontekście. Przykłady:

- Nowe zastosowanie znanego leku w leczeniu innej choroby.
- Wykorzystanie istniejącej technologii w nowej branży.

13.2.4 4. Wynalazki dotyczące oprogramowania

Dotyczą algorytmów i metod obliczeniowych, jeśli mają zastosowanie techniczne. Przykłady:

- Nowe metody kompresji danych.
- Algorytmy kryptograficzne.
- Sposób sterowania urządzeniem przez oprogramowanie.

13.2.5 5. Wynalazki biotechnologiczne

Obejmują odkrycia w dziedzinie biologii, medycyny i inżynierii genetycznej. Przykłady:

- Modyfikacje genetyczne roślin.
- Nowe szczepy bakterii wykorzystywane w przemyśle farmaceutycznym.
- Metody inżynierii tkankowej.

13.3 Ograniczenia w zakresie patentowania

Nie wszystkie rozwiązania mogą zostać opatentowane. Zgodnie z przepisami prawa patentowego nie podlegają ochronie:

- Odkrycia naukowe i teorie matematyczne.
- Metody działalności gospodarczej, finansowej i organizacyjnej.
- Programy komputerowe jako takie (bez zastosowania technicznego).
- Wynalazki sprzeczne z porządkiem publicznym lub moralnością.

13.4 Podsumowanie

- Wynalazek to rozwiązanie techniczne spełniające kryteria nowości, poziomu wynalazczego i przemysłowej stosowalności.
- Można wyróżnić wynalazki produktowe, procesowe, biotechnologiczne, zastosowania oraz oprogramowania.
- Nie wszystkie rozwiązania mogą być opatentowane – prawo wyklucza np. odkrycia naukowe i metody organizacyjne.

14 Charakterystyka wybranego modelu oświetlenia oraz jego komponentów

14.1 Wprowadzenie

Modele oświetlenia w grafice komputerowej symulują sposób, w jaki światło oddziałuje z obiektami, aby uzyskać realistyczny wygląd sceny. Oświetlenie wpływa na percepcję kształtu, głębi i materiałów obiektów. Istnieje wiele modeli oświetlenia, jednak jednym z najczęściej stosowanych w grafice trójwymiarowej jest **model Phonga**.

14.2 Model oświetlenia Phonga

Model Phonga jest empirycznym modelem oświetlenia, który opisuje sposób, w jaki światło odbija się od powierzchni. Składa się z trzech głównych komponentów:

$$I = I_a + I_d + I_s \tag{1}$$

gdzie:

- I_a – składnik oświetlenia otoczenia (ambient),
- I_d – składnik oświetlenia rozproszonego (diffuse),
- I_s – składnik oświetlenia odbitego (specular).

14.2.1 1. Składnik oświetlenia otoczenia (*Ambient Light*)

Jest to globalny składnik światła, który symuluje efekt światła odbitego od wielu powierzchni. Modeluje oświetlenie w obszarach, do których bezpośrednie światło nie dociera.

Wzór:

$$I_a = k_a I_L \tag{2}$$

gdzie:

- k_a – współczynnik odbicia światła otoczenia,
- I_L – natężenie światła otoczenia.

14.2.2 2. Składnik oświetlenia rozproszonego (*Diffuse Light*)

Modeluje rozpraszanie światła na powierzchni obiektu. Natężenie światła zależy od kąta padania światła na powierzchnię i opisuje efekt, w którym powierzchnie ustawione prostopadle do kierunku światła są jaśniejsze.

Wzór:

$$I_d = k_d I_L \max(0, \vec{N} \cdot \vec{L}) \tag{3}$$

gdzie:

- k_d – współczynnik odbicia światła rozproszonego,
- $\vec{N}$ – wektor normalny do powierzchni,
- $\vec{L}$ – wektor kierunku światła.

14.2.3 3. Składnik oświetlenia odbitego (*Specular Light*)

Symuluje efekt połysku i refleksów świetlnych na powierzchniach błyszczących. Wartość ta zależy od kąta między wektorem odbicia światła a kierunkiem patrzenia.

Wzór:

$$I_s = k_s I_L \max(0, \vec{R} \cdot \vec{V})^n \tag{4}$$

gdzie:

- k_s – współczynnik odbicia światła zwierciadlanego,
- $\vec{R}$ – wektor odbitego światła,
- $\vec{V}$ – wektor kierunku obserwacji,
- n – współczynnik połysku (im wyższy, tym mniejsze i bardziej skoncentrowane refleksy).

14.3 Rodzaje źródeł światła

W modelu Phonga mogą występować różne typy źródeł światła:

- **Światło punktowe** – emituje światło we wszystkich kierunkach z jednego punktu.
- **Światło kierunkowe** – symuluje światło pochodzące z bardzo odległego źródła (np. słońca), promienie światła są równoległe.
- **Światło reflektorowe** – podobne do światła punktowego, ale ograniczone do stożka.

14.4 Zalety modelu Phonga

- Prosta i szybka implementacja w porównaniu do bardziej zaawansowanych metod.
- Dobrze odwzorowuje realistyczne oświetlenie, w tym połysk powierzchni.
- Może być stosowany w renderingu czasu rzeczywistego.

14.5 Wady modelu Phonga

- Nie uwzględnia efektów globalnego oświetlenia, takich jak odbicia czy załamania światła.
- Zakłada jednolite odbicie światła dla całej powierzchni, co może powodować brak realizmu.

14.6 Podsumowanie

Model Phonga jest jednym z najczęściej stosowanych modeli oświetlenia w grafice komputerowej. Składa się z trzech komponentów: oświetlenia otoczenia, rozproszonego i odbitego. Mimo swoich ograniczeń, jego prostota i efektywność sprawiają, że jest szeroko stosowany w grafice 3D, szczególnie w aplikacjach działających w czasie rzeczywistym.

15 Metody zwielokrotnienia kanałów transmisyjnych oraz media transmisyjne

15.1 Wprowadzenie

Zwielokrotnienie kanałów transmisyjnych to technika umożliwiająca jednoczesne przesyłanie wielu sygnałów przez jedno medium transmisyjne. Pozwala to na efektywne wykorzystanie dostępnej przepustowości oraz zwiększenie liczby użytkowników korzystających z tej samej infrastruktury sieciowej.

15.2 Metody zwielokrotnienia kanałów transmisyjnych

15.2.1 1. Zwielokrotnienie częstotliwościowe (FDM – Frequency Division Multiplexing)

FDM polega na podziale pasma częstotliwościowego medium transmisyjnego na wiele niepokrywających się zakresów, z których każdy jest wykorzystywany do przesyłania innego sygnału.

Zastosowania:

- Radio i telewizja analogowa (np. kanały radiowe na różnych częstotliwościach).
- Systemy telefonii analogowej.
- Sieci światłowodowe (technologia WDM).

Zalety:

- Umożliwia jednoczesną transmisję wielu sygnałów.
- Brak konieczności synchronizacji czasowej.

Wady:

- Może prowadzić do zakłóceń międzykanałowych (interferencja sąsiednich pasm).
- Wymaga zastosowania filtrów pasmowych.

15.2.2 2. Zwielokrotnienie czasowe (TDM – Time Division Multiplexing)

TDM przydziela każdemu sygnałowi dostęp do kanału transmisyjnego na określony przedział czasu (slot czasowy).

Zastosowania:

- Telefonia cyfrowa (np. systemy ISDN, E1/T1).
- Sieci komputerowe (np. technologia SONET/SDH).

Zalety:

- Wysoka efektywność wykorzystania pasma.
- Brak interferencji między sygnałami.

Wady:

- Wymaga precyzyjnej synchronizacji.
- Opóźnienia w transmisji pakietów.

15.2.3 3. Zwielokrotnienie kodowe (CDM – Code Division Multiplexing)

CDM polega na przypisaniu każdemu użytkownikowi unikalnego kodu, dzięki czemu wiele sygnałów może być przesyłanych równocześnie w tym samym paśmie częstotliwościowym.

Zastosowania:

- Sieci telefonii komórkowej (np. UMTS, CDMA).
- Systemy satelitarne.

Zalety:

- Wysoka odporność na zakłócenia.
- Możliwość dynamicznego przydzielania zasobów.

Wady:

- Skomplikowane techniki kodowania i dekodowania.
- Większe zapotrzebowanie na moc obliczeniową.

15.2.4 4. Zwielokrotnienie przestrzenne (SDM – Space Division Multiplexing)

SDM wykorzystuje fizyczną separację kanałów transmisyjnych, np. poprzez zastosowanie wielu przewodów lub anten.

Zastosowania:

- Światłowody wielordzeniowe.
- Systemy MIMO (Multiple Input Multiple Output) w sieciach 4G i 5G.

Zalety:

- Możliwość znacznego zwiększenia przepustowości.
- Brak interferencji między kanałami.

Wady:

- Wymaga większej liczby urządzeń nadawczo-odbiorczych.

15.2.5 5. Zwielokrotnienie długości fali (WDM – Wavelength Division Multiplexing)

WDM stosowane w światłowodach pozwala na przesyłanie wielu sygnałów optycznych o różnych długościach fali w tym samym medium transmisyjnym.

Zastosowania:

- Sieci światłowodowe dalekiego zasięgu (np. DWDM – Dense WDM).

Zalety:

- Ogromna przepustowość transmisji.
- Niski poziom zakłóceń.

Wady:

- Wysoki koszt infrastruktury.

15.3 Media transmisyjne

Media transmisyjne dzielą się na przewodowe i bezprzewodowe.

15.3.1 1. Media przewodowe

- **Przewody miedziane** – stosowane w sieciach telefonicznych i Ethernet (skrętka UTP/STP).
- **Światłowody** – wykorzystujące fale świetlne do transmisji, zapewniające bardzo dużą przepustowość.
- **Kable koncentryczne** – stosowane w telewizji kablowej i starszych sieciach Ethernet.

15.3.2 2. Media bezprzewodowe

- **Fale radiowe** – stosowane w sieciach Wi-Fi, Bluetooth i telefonii komórkowej.
- **Podczerwień** – wykorzystywana w pilotach zdalnego sterowania.
- **Fale mikrofalowe** – stosowane w komunikacji satelitarnej i punkt-punkt.

15.4 Podsumowanie

- Istnieje wiele metod zwielokrotnienia kanałów transmisyjnych, w tym FDM, TDM, CDM, SDM i WDM.
- Każda z metod ma swoje zalety i ograniczenia, zależne od zastosowania.
- Media transmisyjne mogą być przewodowe (miedź, światłowody) lub bezprzewodowe (fale radiowe, mikrofale).

16 Główne modele procesu wytwarzania oprogramowania

16.1 Wprowadzenie

Model procesu wytwarzania oprogramowania określa metodykę organizacji prac nad projektem informatycznym. Jego celem jest zapewnienie wysokiej jakości produktu, efektywnego zarządzania zasobami oraz minimalizacji ryzyka błędów i opóźnień.

16.2 Główne modele wytwarzania oprogramowania

16.2.1 1. Model kaskadowy (*Waterfall*)

Model kaskadowy to najstarszy i najbardziej klasyczny model wytwarzania oprogramowania, w którym proces przebiega liniowo, etap po etapie.

Etapy modelu kaskadowego:

1. Analiza wymagań.
2. Projektowanie systemu.
3. Implementacja.
4. Testowanie.
5. Wdrożenie.
6. Utrzymanie.

Zalety:

- Jasna struktura i dobrze zdefiniowane etapy.
- Łatwość zarządzania projektem.
- Dobra dokumentacja techniczna.

Wady:

- Brak elastyczności – trudność wprowadzania zmian na późniejszych etapach.
- Wysokie ryzyko wykrycia błędów dopiero na etapie testowania.
- Długi czas realizacji przed dostarczeniem działającego produktu.

16.2.2 2. Model V

Jest rozszerzeniem modelu kaskadowego, w którym do każdego etapu rozwoju przypisany jest odpowiedni etap testowania.

Etapy modelu V:

- **Faza projektowania:** analiza wymagań, projekt systemu, projektowanie modułów.

- **Faza implementacji:** kodowanie.
- **Faza testowania:** testy jednostkowe, integracyjne, systemowe i akceptacyjne.

Zalety:

- Testowanie jest zintegrowane z każdym etapem rozwoju.
- Wczesne wykrywanie błędów.

Wady:

- Nadal jest modelem sztywnym, utrudniającym wprowadzanie zmian.
- Wymaga dużej ilości dokumentacji.

16.2.3 3. Model iteracyjny

W modelu iteracyjnym rozwój oprogramowania odbywa się w cyklach (iteracjach), w których dodawane są kolejne funkcjonalności.

Zalety:

- Możliwość wczesnego dostarczenia działających fragmentów systemu.
- Lepsza adaptacja do zmieniających się wymagań.

Wady:

- Potrzeba częstej komunikacji z klientem.
- Wymaga dobrej organizacji pracy.

16.2.4 4. Model przyrostowy (*Incremental*)

W modelu przyrostowym system jest budowany stopniowo poprzez dodawanie kolejnych modułów.

Zalety:

- Klient otrzymuje działające wersje systemu na wczesnych etapach.
- Mniejsze ryzyko błędów projektowych.

Wady:

- Może prowadzić do problemów z integracją modułów.
- Wymaga dobrego zarządzania wersjami.

16.2.5 5. Model spiralny

Łączy elementy modelu kaskadowego i iteracyjnego, koncentrując się na minimalizacji ryzyka poprzez wielokrotne cykle planowania, projektowania, implementacji i testowania.

Zalety:

- Dobre zarządzanie ryzykiem.
- Elastyczność w dostosowywaniu wymagań.

Wady:

- Złożoność zarządzania procesem.
- Wysoki koszt realizacji.

16.2.6 6. Metodyki zwinne (*Agile*)

Zwinne podejścia do wytwarzania oprogramowania, takie jak **Scrum** czy **Kanban**, opierają się na iteracyjnym podejściu, ścisłej współpracy z klientem i szybkim dostarczaniu wartości.

Zalety:

- Duża elastyczność w dostosowywaniu projektu do zmieniających się wymagań.
- Wczesne dostarczanie wartościowych funkcjonalności.

Wady:

- Wymaga intensywnej komunikacji i zaangażowania zespołu.
- Trudności w planowaniu długoterminowym.

16.3 Podsumowanie

- Model kaskadowy i model V są dobrze ustrukturyzowane, ale mało elastyczne.
- Modele iteracyjne i przyrostowe umożliwiają stopniowe wdrażanie funkcjonalności.
- Model spiralny zapewnia dobrą kontrolę ryzyka.
- Metodyki Agile umożliwiają szybkie reagowanie na zmiany i ścisłą współpracę z klientem.

17 Mechanizmy indeksowania w relacyjnych bazach danych

17.1 Wprowadzenie

Indeksowanie w relacyjnych bazach danych to technika optymalizacyjna, która przyspiesza wyszukiwanie, sortowanie oraz filtrowanie danych. Indeksy są strukturami danych, które umożliwiają szybki dostęp do wierszy tabeli bez konieczności przeszukiwania całej tabeli.

17.2 Rodzaje indeksów

17.2.1 1. Indeks klastrowany (Clustered Index)

W indeksie klastrowanym dane w tabeli są fizycznie przechowywane w kolejności określonej przez indeks. Każda tabela może mieć tylko jeden indeks klastrowany.

Zalety:

- Przyspiesza operacje wyszukiwania i sortowania według klucza indeksu.
- Wydajniejszy dla zapytań, które zwracają zakresy danych.

Wady:

- Wolniejsza operacja wstawiania i aktualizacji, ponieważ może wymagać reorganizacji danych.
- Może zajmować więcej miejsca na dysku.

Przykład w SQL:

```
CREATE CLUSTERED INDEX idx_klastrowany ON Pracownicy (Nazwisko);
```

17.2.2 2. Indeks nieklastrowany (Non-clustered Index)

Indeks nieklastrowany przechowuje wskaźniki do rzeczywistych danych, nie zmieniając ich fizycznego rozmieszczenia.

Zalety:

- Można utworzyć wiele indeksów nieklastrowanych dla jednej tabeli.
- Przyspiesza wyszukiwanie według wartości, które nie są kluczami głównymi.

Wady:

- Może spowolnić operacje `INSERT`, `UPDATE` i `DELETE`.
- Każdy indeks dodatkowo zużywa przestrzeń dyskową.

Przykład w SQL:

```
CREATE INDEX idx_nieklastrowany ON Pracownicy (Stanowisko);
```

17.2.3 3. Indeks wielokolumnowy (Composite Index)

Jest to indeks tworzony na więcej niż jednej kolumnie, co przyspiesza wyszukiwanie połączeń między danymi.

Zalety:

- Efektywność w zapytaniach, które filtrują dane według kilku kolumn.

Wady:

- Zapytania muszą używać pierwszej kolumny indeksu, aby indeks był efektywny.

Przykład w SQL:

```
CREATE INDEX idx_wielokolumnowy ON Pracownicy (Nazwisko, Imie);
```

17.2.4 4. Indeks unikalny (Unique Index)

Indeks, który zapewnia unikalność wartości w danej kolumnie.

Zalety:

- Zapobiega duplikacji danych.

Przykład w SQL:

```
CREATE UNIQUE INDEX idx_unikalny ON Pracownicy (Email);
```

17.2.5 5. Indeks pełnotekstowy (Full-text Index)

Stosowany do wyszukiwania w dużych zbiorach tekstowych.

Zastosowanie:

- Wyszukiwanie pełnotekstowe w bazach danych (np. w dokumentach).

Przykład w SQL Server:

```
CREATE FULLTEXT INDEX ON Dokumenty (Tresc);
```

17.3 Podsumowanie

- Indeksy przyspieszają wyszukiwanie danych, ale mogą zwiększyć czas operacji modyfikacji.
- Istnieją indeksy klastrowane, nieklastrowane, wielokolumnowe, unikalne i pełnotekstowe.
- Wybór odpowiedniego indeksu zależy od charakterystyki danych i częstości wykonywania operacji.

18 Możliwości i ograniczenia transakcji w relacyjnych bazach danych

18.1 Wprowadzenie

Transakcja w relacyjnej bazie danych to zbiór operacji wykonywanych jako jedna, niepodzielna jednostka. Każda transakcja musi spełniać zasady ACID (Atomicity, Consistency, Isolation, Durability), aby zapewnić spójność i niezawodność systemu bazodanowego.

18.2 Możliwości transakcji w relacyjnych bazach danych

18.2.1 1. Spójność danych (Consistency)

Transakcje zapewniają, że baza danych pozostaje w stanie spójnym przed i po wykonaniu transakcji. Jeśli operacja narusza integralność danych, system cofa zmiany.

Przykład: Jeśli przelewamy 100 zł z konta A na konto B, suma środków na obu kontach musi pozostać taka sama.

```
BEGIN TRANSACTION;
UPDATE Konto SET saldo = saldo - 100 WHERE id = 1;
UPDATE Konto SET saldo = saldo + 100 WHERE id = 2;
COMMIT;
```

18.2.2 2. Odporność na błędy (Atomicity)

Transakcje są niepodzielne – jeśli któraś operacja nie powiedzie się, cała transakcja zostaje anulowana (*rollback*).

Przykład: Jeśli nastąpi awaria systemu po pierwszej operacji, ale przed drugą, transakcja zostanie wycofana, aby uniknąć niespójności.

```
BEGIN TRANSACTION;
UPDATE Konto SET saldo = saldo - 100 WHERE id = 1;
IF ERROR THEN ROLLBACK;
UPDATE Konto SET saldo = saldo + 100 WHERE id = 2;
COMMIT;
```

18.2.3 3. Izolacja transakcji (Isolation)

Zapewnia, że jednoczesne transakcje nie wpływają na siebie nawzajem. System może używać różnych poziomów izolacji:

- **Read Uncommitted** – transakcje mogą odczytywać dane niezatwierdzone przez inne transakcje.
- **Read Committed** – transakcje odczytują tylko zatwierdzone zmiany.
- **Repeatable Read** – transakcja widzi te same dane przy każdym odczycie.
- **Serializable** – najwyższy poziom izolacji, blokuje równoczesne transakcje.

Przykład: Ustawienie izolacji transakcji w SQL Server:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION;
SELECT * FROM Konto WHERE id = 1;
COMMIT;
```

18.2.4 4. Trwałość (Durability)

Po zatwierdzeniu transakcji (`COMMIT`), zmiany są trwale zapisane w bazie, nawet jeśli nastąpi awaria systemu.

Przykład: Po zatwierdzeniu przelewu, saldo konta jest zapisane na dysku i nie można go cofnąć w przypadku awarii.

18.3 Ograniczenia transakcji w relacyjnych bazach danych

18.3.1 1. Problemy współbieżności

Równoczesne transakcje mogą powodować konflikty:

- **Dirty Read** – transakcja odczytuje dane, które mogą zostać wycofane.
- **Non-repeatable Read** – dane mogą zmieniać się między odczytami w jednej transakcji.
- **Phantom Read** – nowe wiersze mogą pojawić się między zapytaniami.

18.3.2 2. Narzut wydajnościowy

Wyższe poziomy izolacji (np. Serializable) mogą prowadzić do blokowania transakcji, co spowalnia działanie systemu.

18.3.3 3. Problemy z długimi transakcjami

Długotrwałe transakcje mogą blokować inne operacje i prowadzić do zatorów w systemie.

18.3.4 4. Możliwość zakleszczeń (Deadlocks)

Jeśli dwie transakcje blokują te same zasoby i czekają na siebie nawzajem, może dojść do zakleszczenia.

Przykład: Transakcja A blokuje tabelę X, a transakcja B blokuje tabelę Y – jeśli A próbuje uzyskać dostęp do Y, a B do X, powstaje zakleszczenie.

18.3.5 5. Brak wsparcia dla rozproszonych transakcji

Niektóre systemy bazodanowe mają ograniczone wsparcie dla transakcji obejmujących wiele baz danych.

18.4 Podsumowanie

- Transakcje zapewniają spójność, atomowość, izolację i trwałość zmian w bazie.
- Mechanizmy izolacji chronią przed błędami współbieżności.

- Ograniczenia obejmują problemy wydajnościowe, zakleszczenia i konflikty transakcji.
- Odpowiednie zarządzanie poziomami izolacji pozwala na optymalizację wydajności systemu.

19 Znane metody oceny jakości modeli klasyfikacyjnych i regresyjnych

19.1 Wprowadzenie

Ocena jakości modeli predykcyjnych jest kluczowym etapem w analizie danych. Wyróżnia się dwie główne kategorie modeli:

- **Modele klasyfikacyjne** – przewidują etykiety klas (np. czy e-mail to spam czy nie).
- **Modele regresyjne** – przewidują wartości liczbowe (np. cena nieruchomości).

Ocena jakości modeli opiera się na różnych miarach, które odzwierciedlają skuteczność przewidywań.

19.2 Metody oceny modeli klasyfikacyjnych

19.2.1 1. Macierz pomyłek (Confusion Matrix)

Jest to tabela przedstawiająca liczbę poprawnych i błędnych klasyfikacji.

	Klasa rzeczywista: Pozytywna	**Klasa rzeczywista: Negatywna**
Przewidziana: Pozytywna	TP (True Positive)	FP (False Positive)
Przewidziana: Negatywna	FN (False Negative)	TN (True Negative)

Tabela 3: Macierz pomyłek

19.2.2 2. Miary jakości klasyfikacji

- **Dokładność (Accuracy)**

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

  Określa procent poprawnie sklasyfikowanych przypadków.

- **Precyzja (Precision)**

$$Precision = \frac{TP}{TP + FP}$$

  Informuje, jaki procent pozytywnie sklasyfikowanych przykładów rzeczywiście jest pozytywny.

- **Czułość (Recall, Sensitivity)**

$$Recall = \frac{TP}{TP + FN}$$

  Określa, jaki procent rzeczywistych pozytywnych przypadków został poprawnie wykryty.

- **Wartość F1 (F1-score)**

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

  Średnia harmoniczna precyzji i czułości.

- **Krzywa ROC i AUC (Area Under Curve)**
  - Krzywa ROC (Receiver Operating Characteristic) przedstawia zależność między czułością a 1-specyficznością.
  - AUC (pole pod krzywą ROC) określa skuteczność klasyfikatora – im większa wartość, tym lepszy model.

19.3 Metody oceny modeli regresyjnych

19.3.1 1. Średni błąd absolutny (Mean Absolute Error, MAE)

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

Określa średnią wartość błędu między rzeczywistymi a przewidywanymi wartościami.

19.3.2 2. Średni błąd kwadratowy (Mean Squared Error, MSE)

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Wartości większe są bardziej karane, przez co bardziej uwzględnia duże błędy.

19.3.3 3. Pierwiastek średniego błędu kwadratowego (Root Mean Squared Error, RMSE)

$$RMSE = \sqrt{MSE}$$

Jest to miara podobna do MSE, ale zachowuje tę samą jednostkę, co wartości przewidywane.

19.3.4 4. Współczynnik determinacji (R-squared, R^2)

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

Określa, jaka część wariancji zmiennej zależnej jest wyjaśniona przez model.

19.4 Podsumowanie

- Modele klasyfikacyjne oceniane są za pomocą dokładności, precyzji, czułości, wartości F1 oraz krzywej ROC.
- Modele regresyjne ocenia się poprzez MAE, MSE, RMSE oraz współczynnik determinacji R^2.
- Wybór odpowiedniej metryki zależy od charakteru problemu oraz konsekwencji błędów w predykcji.

20 Problem przekleństwa wymiarowości

20.1 Wprowadzenie

Przekleństwo wymiarowości (ang. *curse of dimensionality*) to zjawisko, w którym wzrost liczby wymiarów przestrzeni cech prowadzi do istotnych problemów obliczeniowych, analitycznych oraz modelowania danych. Problemy te występują w uczeniu maszynowym, statystyce, analizie danych i eksploracji dużych zbiorów danych.

20.2 Przyczyny przekleństwa wymiarowości

- **Wzrost objętości przestrzeni** – im więcej wymiarów, tym większa przestrzeń do przeszukiwania.
- **Rozrzedzenie danych** – w wysokiej wymiarowości dane stają się rzadkie, co utrudnia efektywne modelowanie.
- **Problemy z odległościami** – wymiary wpływają na metryki odległości, co powoduje, że punkty stają się bardziej równomiernie rozmieszczone.
- **Wzrost złożoności obliczeniowej** – operacje na danych wysokiej wymiarowości wymagają większych zasobów obliczeniowych.

20.3 Konsekwencje przekleństwa wymiarowości

20.3.1 1. Problemy w klasyfikacji i klasteryzacji

Wysoka wymiarowość może prowadzić do sytuacji, w której klasyfikatory i algorytmy klasteryzacji tracą skuteczność, ponieważ różnice między punktami stają się mniej wyraźne.

20.3.2 2. Wzrost liczby danych wymaganych do efektywnego modelowania

Dla wysokiej wymiarowości konieczna jest większa liczba próbek, aby uniknąć przeuczenia (ang. *overfitting*).

20.3.3 3. Problemy w analizie podobieństwa

W metrykach takich jak odległość euklidesowa większość punktów może znajdować się w podobnych odległościach, co utrudnia identyfikację najbliższych sąsiadów.

20.4 Metody radzenia sobie z przekleństwem wymiarowości

20.4.1 1. Redukcja wymiarowości

Redukcja liczby wymiarów może poprawić efektywność algorytmów i ich interpretowalność.

Przykłady metod:

- **Główne składowe (PCA)** – transformacja danych w nową przestrzeń o mniejszej liczbie wymiarów.

- **Analiza składowych niezależnych (ICA)** – rozkład danych na komponenty niezależne statystycznie.
- **t-SNE, UMAP** – metody redukcji wymiarowości do wizualizacji danych.

20.4.2 2. Wybór cech (Feature Selection)

Selekcja najistotniejszych cech pozwala zmniejszyć wymiarowość bez utraty informacji.
Metody:

- **Filtry** – analiza cech na podstawie korelacji, testów statystycznych.
- **Metody osadzone** – np. LASSO, które usuwa mniej istotne cechy podczas uczenia modelu.

20.4.3 3. Użycie algorytmów odpornych na wysoką wymiarowość

Niektóre algorytmy, np. drzewa decyzyjne czy metody oparte na rzadkich reprezentacjach, mogą lepiej działać w wysokiej wymiarowości.

20.5 Podsumowanie

- Przekleństwo wymiarowości utrudnia analizę i modelowanie danych, prowadząc do problemów w klasyfikacji, klasteryzacji i eksploracji danych.
- Wysoka wymiarowość zwiększa wymagania obliczeniowe i prowadzi do rozrzedzenia danych.
- Metody takie jak redukcja wymiarowości (PCA, t-SNE), selekcja cech i dobór odpowiednich algorytmów pomagają radzić sobie z tym problemem.

21 Działanie i zastosowanie naiwnego klasyfikatora Bayesa

21.1 Wprowadzenie

Naiwny klasyfikator Bayesa (ang. *Naïve Bayes Classifier*) to probabilistyczny model klasyfikacji, który opiera się na twierdzeniu Bayesa. Jest szeroko stosowany ze względu na prostotę, efektywność obliczeniową oraz dobrą skuteczność w wielu zastosowaniach.

21.2 Zasada działania

Naiwny klasyfikator Bayesa wykorzystuje twierdzenie Bayesa do obliczenia prawdopodobieństwa przynależności obiektu do danej klasy:

$$P(C_k|X) = \frac{P(X|C_k)P(C_k)}{P(X)}$$

gdzie:

- $P(C_k|X)$ – prawdopodobieństwo, że obiekt o cechach X należy do klasy C_k.
- $P(X|C_k)$ – prawdopodobieństwo uzyskania cech X przy założeniu, że obiekt należy do klasy C_k.
- $P(C_k)$ – prawdopodobieństwo wystąpienia klasy C_k (tzw. *prior*).
- $P(X)$ – prawdopodobieństwo wystąpienia cech X (można je pominąć, ponieważ jest stałe dla wszystkich klas).

Założenie naiwności: cechy są niezależne warunkowo, co oznacza, że prawdopodobieństwo wystąpienia danej cechy nie zależy od innych cech:

$$P(X|C_k) = P(x_1|C_k)P(x_2|C_k)\dots P(x_n|C_k)$$

21.3 Rodzaje naiwnego klasyfikatora Bayesa

- **Gaussian Naïve Bayes** – dla cech ciągłych, zakłada rozkład normalny.
- **Multinomial Naïve Bayes** – dla danych dyskretnych, stosowany np. w klasyfikacji tekstów.
- **Bernoulli Naïve Bayes** – dla cech binarnych (obecność/nieobecność cechy).

21.4 Zastosowania naiwnego klasyfikatora Bayesa

21.4.1 1. Klasyfikacja tekstu i analiza sentymentu

- Filtrowanie spamu (np. klasyfikacja e-maili jako spam/nie-spam).
- Analiza sentymentu (np. klasyfikacja recenzji jako pozytywne/negatywne).
- Kategoryzacja dokumentów (np. klasyfikacja artykułów do kategorii tematycznych).

21.4.2 2. Rozpoznawanie wzorców

- Klasyfikacja obrazów (np. rozpoznawanie cyfr ręcznie pisanych).
- Wykrywanie oszustw (np. analiza anomalii w transakcjach bankowych).

21.4.3 3. Medycyna i diagnostyka

- Klasyfikacja chorób na podstawie objawów.
- Wspomaganie diagnoz medycznych.

21.5 Zalety i wady naiwnego klasyfikatora Bayesa

Zalety:

- Prostota i łatwość implementacji.
- Niska złożoność obliczeniowa ($O(n)$).
- Skuteczność nawet w przypadku małych zbiorów danych.
- Nie wymaga dużej ilości próbek do działania.

Wady:

- Założenie niezależności cech – w rzeczywistych danych cechy często są skorelowane.
- Może mieć problemy w przypadku skrajnych wartości cech (np. niewystępowanie pewnej cechy w zbiorze uczącym).
- Wrażliwość na jakość danych wejściowych.

21.6 Podsumowanie

- Naiwny klasyfikator Bayesa wykorzystuje twierdzenie Bayesa do klasyfikacji danych.
- Zakłada niezależność cech, co upraszcza obliczenia.
- Jest szeroko stosowany w klasyfikacji tekstu, rozpoznawaniu wzorców i diagnostyce medycznej.
- Mimo swojej prostoty, często daje dobre wyniki w praktyce.

22 Porównanie protokołów TCP i UDP

22.1 Wprowadzenie

TCP (Transmission Control Protocol) i UDP (User Datagram Protocol) to dwa główne protokoły transportowe stosowane w sieciach komputerowych. Oba służą do przesyłania danych, jednak różnią się sposobem działania i zastosowaniem.

22.2 Charakterystyka protokołu TCP

TCP to protokół połączeniowy, zapewniający niezawodny transfer danych.

Cechy TCP:

- **Połączeniowy** – wymaga nawiązania sesji między nadawcą a odbiorcą.
- **Gwarantuje dostarczenie danych** – wykorzystuje mechanizmy retransmisji w razie utraty pakietów.
- **Kontrola przepływu** – dostosowuje prędkość transmisji do możliwości odbiorcy.
- **Podział na segmenty** – dane są dzielone na segmenty, numerowane i składane w odpowiedniej kolejności.
- **Wykrywanie błędów** – wykorzystuje sumy kontrolne.

Przykłady zastosowań TCP:

- Przeglądanie stron WWW (HTTP, HTTPS).
- Transfer plików (FTP).
- Poczta elektroniczna (SMTP, IMAP, POP3).

22.3 Charakterystyka protokołu UDP

UDP to protokół bezpołączeniowy, który oferuje szybki, ale mniej niezawodny transfer danych.

Cechy UDP:

- **Bezpołączeniowy** – nie wymaga ustanowienia sesji przed przesłaniem danych.
- **Brak gwarancji dostarczenia** – pakiety mogą ginąć lub docierać w innej kolejności.
- **Brak retransmisji** – utracone pakiety nie są ponownie przesyłane.
- **Niskie opóźnienia** – nadaje się do transmisji wymagających minimalnego czasu reakcji.

Przykłady zastosowań UDP:

- Transmisje strumieniowe audio i wideo (VoIP, IPTV).
- Gry online wymagające szybkiej wymiany danych.
- Protokoły DNS i DHCP.

22.4 Porównanie TCP i UDP

Cecha	TCP	UDP
Typ protokołu	Połączeniowy	Bezpołączeniowy
Gwarancja dostarczenia	Tak (retransmisja)	Nie
Kolejność pakietów	Zachowana	Może być losowa
Kontrola błędów	Tak	Minimalna
Kontrola przepływu	Tak	Nie
Szybkość	Wolniejszy (ze względu na kontrolę)	Szybszy
Zastosowania	HTTP, FTP, e-mail	VoIP, DNS, gry online

Tabela 4: Porównanie protokołów TCP i UDP

22.5 Podsumowanie

- TCP zapewnia niezawodność i kontrolę transmisji, ale kosztem wydajności.
- UDP jest szybszy i lepiej nadaje się do zastosowań wymagających niskich opóźnień.
- Wybór protokołu zależy od specyfiki aplikacji – TCP dla aplikacji wymagających niezawodności, UDP dla aplikacji czasu rzeczywistego.

23 Podstawowe modele kontroli dostępu

23.1 Wprowadzenie

Kontrola dostępu to mechanizm zapewniający bezpieczeństwo systemów informatycznych poprzez ograniczenie dostępu do zasobów. Istnieją różne modele kontroli dostępu, które definiują zasady przyznawania i egzekwowania uprawnień użytkowników.

23.2 Główne modele kontroli dostępu

23.2.1 1. Kontrola dostępu oparta na listach kontroli dostępu (ACL – Access Control List)

ACL to zbiór reguł określających, kto i w jaki sposób może uzyskać dostęp do danego zasobu.

Cechy modelu ACL:

- Każdy zasób ma przypisaną listę użytkowników i ich uprawnień.
- Można definiować szczegółowe reguły dostępu dla poszczególnych użytkowników lub grup.

Przykłady zastosowań:

- Systemy plików (np. NTFS w Windows).
- Zapory sieciowe (firewalle).

23.2.2 2. Dyskrecjonalna kontrola dostępu (DAC – Discretionary Access Control)

DAC pozwala właścicielowi zasobu na dowolne zarządzanie dostępem do niego.

Cechy modelu DAC:

- Uprawnienia mogą być przekazywane innym użytkownikom.
- Umożliwia elastyczne zarządzanie dostępem.

Wady modelu DAC:

- Możliwość nieautoryzowanego przekazywania uprawnień.
- Brak centralnej kontroli nad dostępem.

Przykłady zastosowań:

- Systemy operacyjne Windows i Linux.
- Systemy bazodanowe.

23.2.3 3. Obowiązkowa kontrola dostępu (MAC – Mandatory Access Control)

MAC to model, w którym uprawnienia są przydzielane centralnie przez administratora i użytkownicy nie mogą ich zmieniać.

Cechy modelu MAC:

- Każdy zasób i użytkownik mają przypisane poziomy klasyfikacji (np. tajne, poufne).
- System decyduje, kto może uzyskać dostęp na podstawie polityk bezpieczeństwa.

Przykłady zastosowań:

- Systemy rządowe i wojskowe.
- Bezpieczne systemy operacyjne (np. SELinux).

23.2.4 4. Kontrola dostępu oparta na rolach (RBAC – Role-Based Access Control)

RBAC przyznaje użytkownikom uprawnienia na podstawie przypisanych im ról.

Cechy modelu RBAC:

- Użytkownicy są przypisani do ról, a role posiadają określone uprawnienia.
- Ułatwia zarządzanie uprawnieniami w dużych organizacjach.

Przykłady zastosowań:

- Systemy korporacyjne (np. ERP).
- Systemy baz danych i chmury obliczeniowe.

23.2.5 5. Kontrola dostępu oparta na atrybutach (ABAC – Attribute-Based Access Control)

ABAC przyznaje dostęp na podstawie atrybutów użytkownika, zasobu i kontekstu.

Cechy modelu ABAC:

- Uprawnienia są dynamicznie przyznawane na podstawie atrybutów (np. lokalizacja, czas).
- Zapewnia wysoką elastyczność i bezpieczeństwo.

Przykłady zastosowań:

- Zaawansowane systemy bezpieczeństwa w chmurze.
- Systemy zgodne z regulacjami (np. HIPAA, GDPR).

Model	Cechy	Zastosowanie
ACL	Lista reguł dla zasobu	Systemy plików, firewalle
DAC	Właściciel zasobu zarządza dostępem	Systemy operacyjne, bazy danych
MAC	Centralne zarządzanie dostępem	Systemy rządowe, wojskowe
RBAC	Uprawnienia nadawane według ról	Organizacje, systemy ERP
ABAC	Dynamiczne uprawnienia na podstawie atrybutów	Systemy chmurowe, zgodność z regulacjami

Tabela 5: Porównanie modeli kontroli dostępu

23.3 Porównanie modeli kontroli dostępu

23.4 Podsumowanie

- Kontrola dostępu jest kluczowym elementem bezpieczeństwa systemów informatycznych.
- Modele ACL i DAC są stosowane w systemach operacyjnych i bazach danych.
- MAC zapewnia najwyższy poziom bezpieczeństwa w systemach rządowych i wojskowych.
- RBAC i ABAC są szeroko stosowane w dużych organizacjach i systemach chmurowych.

24 Rodzaje symulacji komputerowych; ich charakterystyka i przykłady

24.1 Wprowadzenie

Symulacja komputerowa to proces modelowania rzeczywistych systemów za pomocą programów komputerowych w celu analizy ich zachowania w różnych warunkach. Symulacje znajdują zastosowanie w nauce, inżynierii, ekonomii oraz medycynie.

24.2 Rodzaje symulacji komputerowych

24.2.1 1. Symulacje deterministyczne

Symulacje deterministyczne to modele, w których dla tych samych warunków początkowych wyniki są zawsze identyczne.

Charakterystyka:

- Brak losowych elementów – przebieg symulacji jest w pełni przewidywalny.
- Modelowanie procesów fizycznych i technicznych.

Przykłady:

- Symulacje dynamiki pojazdów (np. analiza ruchu mechanicznego).
- Obliczenia przepływu ciepła i mechaniki płynów.
- Modele ruchu planet w Układzie Słonecznym.

24.2.2 2. Symulacje stochastyczne

Symulacje stochastyczne wykorzystują losowość i prawdopodobieństwo do modelowania rzeczywistości.

Charakterystyka:

- Wyniki różnią się dla tych samych parametrów wejściowych.
- Używane w modelach rzeczywistych, gdzie występuje niepewność.

Przykłady:

- Modelowanie giełdy papierów wartościowych.
- Symulacje epidemiologiczne (np. rozprzestrzenianie chorób).
- Metoda Monte Carlo w analizie ryzyka.

24.2.3 3. Symulacje ciągłe

Modele ciągłe opisują systemy za pomocą równań różniczkowych.

Charakterystyka:

- Ciągła zmiana wartości zmiennych w czasie.
- Stosowane w naukach przyrodniczych i inżynierii.

Przykłady:

- Modelowanie klimatu i zmian pogodowych.
- Dynamika populacji w ekosystemach.
- Symulacje obwodów elektrycznych.

24.2.4 4. Symulacje dyskretne

Modele dyskretne działają na zdarzeniach zachodzących w określonych momentach czasu.

Charakterystyka:

- System zmienia się w określonych punktach czasu.
- Wykorzystywane w modelowaniu systemów kolejkowych, produkcyjnych i logistycznych.

Przykłady:

- Modelowanie przepływu ruchu na skrzyżowaniach.
- Symulacja działania systemów produkcyjnych.
- Analiza wydajności sieci komputerowych.

24.2.5 5. Symulacje hybrydowe

Łączą elementy symulacji ciągłej i dyskretnej.

Przykłady:

- Modelowanie systemów medycznych (np. układu krwionośnego z impulsami pracy serca).
- Symulacje systemów cyber-fizycznych (np. robotyka autonomiczna).

24.2.6 6. Symulacje w czasie rzeczywistym

Symulacje, w których przetwarzanie danych odbywa się na bieżąco.

Przykłady:

- Trenażery lotnicze.
- Symulacje jazdy samochodem (np. systemy ADAS).
- Gry komputerowe i rzeczywistość wirtualna.

24.3 Podsumowanie

- Symulacje komputerowe pozwalają modelować rzeczywiste procesy i systemy.
- Wyróżniamy symulacje deterministyczne, stochastyczne, ciągłe, dyskretne, hybrydowe i w czasie rzeczywistym.
- Wybór modelu symulacji zależy od charakterystyki badanego systemu i jego zastosowania.

25 Podstawowe rodzaje licencji na oprogramowanie komputerowe w kontekście etycznej strony przestrzegania praw autorskich

25.1 Wprowadzenie

Licencje na oprogramowanie określają warunki, na jakich użytkownicy mogą korzystać z programów komputerowych. Przestrzeganie praw autorskich i licencyjnych jest istotnym aspektem etyki w informatyce, ponieważ wpływa na ochronę twórczości programistów i uczciwe korzystanie z oprogramowania.

25.2 Podstawowe rodzaje licencji

25.2.1 1. Licencje własnościowe (proprietary)

Licencje własnościowe przyznają użytkownikowi ograniczone prawa do korzystania z oprogramowania, a kod źródłowy pozostaje zamknięty.

Charakterystyka:

- Użytkownik nie ma dostępu do kodu źródłowego.
- Zabronione jest modyfikowanie i rozpowszechnianie oprogramowania.
- Producent oprogramowania ma pełną kontrolę nad jego dystrybucją i rozwojem.

Przykłady:

- Microsoft Windows, Microsoft Office.
- Adobe Photoshop.

Etyczne aspekty:

- Nielegalne kopiowanie i udostępnianie oprogramowania narusza prawa autorskie.
- Korzystanie z pirackiego oprogramowania jest nieetyczne i często niezgodne z prawem.

25.2.2 2. Licencje wolnego i otwartego oprogramowania (FOSS – Free and Open Source Software)

Licencje FOSS pozwalają użytkownikom na swobodne korzystanie, modyfikowanie i rozpowszechnianie oprogramowania.

Charakterystyka:

- Kod źródłowy jest dostępny dla użytkowników.
- Możliwość modyfikowania i rozwoju oprogramowania przez społeczność.
- Niektóre licencje wymagają zachowania otwartości kodu w pochodnych projektach.

Przykłady:

- Linux (licencja GPL).
- Mozilla Firefox (licencja MPL).
- LibreOffice (licencja LGPL).

Etyczne aspekty:

- Promuje dzielenie się wiedzą i współpracę.
- Zachęca do uczciwego korzystania z technologii i innowacji.

25.2.3 3. Licencje darmowe (Freeware)

Oprogramowanie udostępniane bezpłatnie, ale zazwyczaj z ograniczeniami dotyczącymi modyfikacji lub rozpowszechniania.

Przykłady:

- Skype.
- Adobe Acrobat Reader.

Etyczne aspekty:

- Nie oznacza wolności modyfikacji – użytkownicy muszą przestrzegać warunków licencji.
- Korzystanie z darmowego oprogramowania zamiast pirackiego jest etycznym wyborem.

25.2.4 4. Licencje na oprogramowanie współdzielone (Shareware)

Użytkownik może korzystać z oprogramowania przez określony czas, po czym powinien wykupić licencję.

Przykłady:

- WinRAR.
- Total Commander.

Etyczne aspekty:

- Korzystanie z wersji próbnych jest zgodne z etyką, ale obchodzenie ograniczeń czasowych jest nieetyczne.

25.2.5 5. Licencje publiczne (Public Domain)

Oprogramowanie, które nie jest objęte prawami autorskimi – może być dowolnie używane, modyfikowane i rozpowszechniane.

Przykłady:

- SQLite.
- Niektóre starsze wersje oprogramowania.

Etyczne aspekty:

- Pełna swoboda użytkowania i modyfikacji.
- Dbanie o prawidłowe przypisywanie autorstwa jest etycznym obowiązkiem użytkowników.

25.2.6 6. Licencje Creative Commons

Stosowane głównie do treści cyfrowych, ale także w oprogramowaniu.

Rodzaje:

- CC BY – dozwolone dowolne użycie pod warunkiem podania autora.
- CC BY-SA – wymaga udostępniania pochodnych prac na tej samej licencji.
- CC BY-NC – zakazuje komercyjnego wykorzystania.

Przykłady:

- Dokumentacja projektów open-source.
- Materiały edukacyjne.

25.3 Podsumowanie

- Licencje definiują zasady korzystania z oprogramowania i chronią prawa autorskie.
- Wybór licencji wpływa na dostępność i rozwój oprogramowania.
- Etyczne korzystanie z oprogramowania obejmuje przestrzeganie licencji i unikanie piractwa.

26 Budowa indeksowego systemu plików na przykładzie EXT4

26.1 Wprowadzenie

EXT4 (Fourth Extended Filesystem) to jeden z najczęściej używanych systemów plików w systemach Linux. Jest następcą EXT3 i wprowadza liczne usprawnienia, takie jak większa wydajność, lepsza obsługa dużych plików i zwiększona niezawodność. EXT4 wykorzystuje indeksowaną strukturę plików, co pozwala na efektywne przechowywanie i wyszukiwanie danych.

26.2 Podział partycji na struktury w EXT4

Każda partycja sformatowana w systemie plików EXT4 jest podzielona na kilka podstawowych struktur:

- **Superblok** – przechowuje informacje o systemie plików.
- **Bloki grup (Block Groups)** – system dzieli partycję na grupy bloków dla lepszej organizacji.
- **i-węzły (inodes)** – zawierają metadane plików i katalogów.
- **Bitmapy bloków i i-węzłów** – zarządzają przydziałem bloków i i-węzłów.
- **Dane użytkownika** – przechowywane w blokach danych.

26.3 Struktura systemu plików EXT4

- **Superblok** – zawiera kluczowe informacje o systemie plików, takie jak:
  - Rozmiar systemu plików.
  - Liczba bloków i i-węzłów.
  - Znacznik czasu ostatniego montowania.
  - Flagi systemowe i opcje montowania.
- **Grupy bloków** – EXT4 dzieli przestrzeń dyskową na grupy bloków, co poprawia wydajność dostępu do danych. Każda grupa zawiera:
  - Superblok (opcjonalnie w każdej grupie).
  - Bitmapę bloków – określa, które bloki są wolne/zajęte.
  - Bitmapę i-węzłów – przechowuje informacje o dostępnych i-węzłach.
  - Tablicę i-węzłów – przechowuje struktury i-węzłów.
  - Dane użytkownika – rzeczywista zawartość plików.
- **i-węzły (inodes)** – każdemu plikowi/katalogowi odpowiada jeden i-węzeł, który zawiera:
  - Identyfikator właściciela i grupy.

  - Uprawnienia dostępu.
  - Znaczniki czasu (utworzenia, modyfikacji, ostatniego dostępu).
  - Wskaźniki do bloków danych przechowujących zawartość pliku.

- **Struktura katalogów** – katalogi są specjalnymi plikami zawierającymi listę nazw plików i odpowiadających im i-węzłów.
- **Bloki danych** – przechowują rzeczywiste treści plików.

26.4 Sposób przechowywania informacji o plikach i katalogach

26.4.1 1. System indeksowania plików

EXT4 wykorzystuje strukturę i-węzłów do przechowywania metadanych plików. Każdy i-węzeł zawiera wskaźniki do bloków danych:

- Wskaźniki bezpośrednie – adresują pierwsze kilka bloków pliku.
- Wskaźnik pośredni – wskazuje na blok zawierający adresy kolejnych bloków.
- Wskaźnik podwójnie pośredni – wskazuje na blok, który zawiera wskaźniki do bloków pośrednich.
- Wskaźnik potrójnie pośredni – umożliwia obsługę bardzo dużych plików.

26.4.2 2. Extents – optymalizacja przydzielania bloków

EXT4 wprowadza mechanizm *extents*, który zastępuje tradycyjne listy bloków. *Extents* to ciągłe fragmenty przestrzeni dyskowej przypisane do pliku, co:

- Zmniejsza fragmentację.
- Przyspiesza operacje odczytu i zapisu.
- Poprawia wydajność obsługi dużych plików.

26.4.3 3. Jurnalowanie (journaling)

EXT4 wykorzystuje **dziennik** (*journal*) do rejestrowania operacji przed ich zapisaniem na dysku, co zapobiega utracie danych w przypadku awarii.

Tryby pracy journalingu:

- **Journal** – pełne zapisywanie operacji, największa niezawodność, ale najwolniejsza metoda.
- **Ordered** (domyślny) – zapis metadanych do dziennika, a następnie danych na dysk.
- **Writeback** – metadane mogą być zapisywane przed danymi, co zwiększa ryzyko utraty danych.

26.5 Podsumowanie

- EXT4 to nowoczesny system plików wykorzystywany w systemach Linux.
- Struktura EXT4 obejmuje superblok, grupy bloków, tablice i-węzłów oraz bloki danych.
- i-węzły przechowują metadane plików i wskazują na ich rzeczywiste dane.
- Mechanizm *extents* redukuje fragmentację i zwiększa wydajność.
- Journaling chroni przed utratą danych w przypadku awarii systemu.

27 Komunikacja międzyprocesowa z wykorzystaniem pamięci współdzielonej, semaforów i gniazd

27.1 Wprowadzenie

Komunikacja międzyprocesowa (IPC – *Inter-Process Communication*) to mechanizmy umożliwiające wymianę danych i synchronizację między procesami działającymi w tym samym systemie operacyjnym lub w różnych systemach. Istnieje wiele metod IPC, w tym pamięć współdzielona, semafory i gniazda.

27.2 Pamięć współdzielona (Shared Memory)

Pamięć współdzielona to mechanizm IPC, który pozwala różnym procesom na bezpośredni dostęp do wspólnego obszaru pamięci.

Charakterystyka:

- Umożliwia szybkie przesyłanie danych między procesami.
- Procesy muszą synchronizować dostęp do pamięci, aby unikać konfliktów.

Zalety:

- Bardzo szybka wymiana danych.
- Minimalna narzutowa komunikacja.

Wady:

- Brak mechanizmu synchronizacji – wymaga dodatkowych narzędzi (np. semaforów).
- Może prowadzić do problemów z bezpieczeństwem danych.

Przykład w języku C:

```
int shm_id = shmget(IPC_PRIVATE, 1024, IPC_CREAT | 0666);
char *shm_ptr = (char*) shmat(shm_id, NULL, 0);
```

27.3 Semafory

Semafory to mechanizm synchronizacji procesów, który pozwala na kontrolę dostępu do zasobów współdzielonych.

Rodzaje semaforów:

- **Semafory binarne** – działają jak blokada (0/1).
- **Semafory liczbowe** – pozwalają ograniczyć liczbę jednoczesnych dostępów do zasobu.

Zastosowanie:

- Synchronizacja dostępu do pamięci współdzielonej.
- Unikanie warunków wyścigu.

Przykład użycia semaforów w C:

```
sem_t sem;
sem_init(&sem, 0, 1);
sem_wait(&sem);
// Sekcja krytyczna
sem_post(&sem);
```

27.4 Gniazda (Sockets)

Gniazda umożliwiają komunikację między procesami działającymi na tym samym lub różnych komputerach poprzez sieć.

Rodzaje gniazd:

- **Gniazda domeny UNIX** – służą do komunikacji międzyprocesowej w jednym systemie operacyjnym.
- **Gniazda sieciowe** – umożliwiają komunikację przez sieć (TCP/UDP).

Zastosowanie:

- Komunikacja klient-serwer (np. HTTP, FTP).
- Wymiana danych między aplikacjami rozproszonymi.

Przykład w języku C (gniazdo TCP):

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
bind(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr));
listen(sockfd, 5);
int new_sock = accept(sockfd, (struct sockaddr*)&client_addr, &addrlen);
```

27.5 Porównanie metod IPC

Metoda	Szybkość	Zastosowanie	Wady
Pamięć współdzielona	Bardzo szybka	Wymiana dużych danych	Brak synchronizacji
Semafory	Średnia	Synchronizacja procesów	Możliwe zakleszczenia
Gniazda	Wolniejsza	Komunikacja sieciowa	Większy narzut

Tabela 6: Porównanie metod IPC

27.6 Podsumowanie

- Pamięć współdzielona jest szybka, ale wymaga synchronizacji.
- Semafory kontrolują dostęp do zasobów i zapobiegają konfliktom.
- Gniazda pozwalają na komunikację między procesami w różnych systemach.
- Wybór metody IPC zależy od potrzeb aplikacji i środowiska jej działania.

28 Rola i umiejscowienie wywołań systemowych w architekturze systemu operacyjnego, sposób ich uruchamiania, przykłady wywołania

28.1 Wprowadzenie

Wywołania systemowe (ang. *system calls*) to interfejs umożliwiający programom użytkownika komunikację z jądrem systemu operacyjnego. Zapewniają dostęp do zasobów sprzętowych i funkcji systemowych w kontrolowany sposób.

28.2 Rola wywołań systemowych

Wywołania systemowe pełnią kluczową rolę w operacjach takich jak:

- Zarządzanie procesami (tworzenie, synchronizacja, zakończenie).
- Obsługa plików (otwieranie, zamykanie, czytanie, zapis).
- Komunikacja międzyprocesowa (IPC).
- Zarządzanie pamięcią (alokacja, zwalnianie).
- Obsługa urządzeń wejścia/wyjścia.
- Sieciowa komunikacja.

28.3 Umiejscowienie wywołań systemowych w architekturze systemu operacyjnego

W architekturze systemu operacyjnego wywołania systemowe działają na granicy przestrzeni użytkownika (*user space*) i przestrzeni jądra (*kernel space*).

- **Przestrzeń użytkownika** – zawiera procesy aplikacji, które nie mają bezpośredniego dostępu do sprzętu.
- **Przestrzeń jądra** – kontroluje zasoby sprzętowe i wykonuje operacje niskopoziomowe.
- **Interfejs wywołań systemowych** – działa jako pośrednik, przekazując żądania użytkownika do jądra.

28.4 Sposób uruchamiania wywołań systemowych

Wywołania systemowe są wywoływane przez aplikacje użytkownika w następujący sposób:

1. Aplikacja użytkownika wywołuje funkcję biblioteczną (np. `open()` w języku C).
2. Funkcja biblioteczna przekazuje żądanie do jądra poprzez instrukcję pułapki (ang. *trap*).
3. Jądro przełącza kontekst na przestrzeń jądra i wykonuje odpowiednią funkcję.

4. Wynik jest zwracany do aplikacji użytkownika.

Schemat przepływu wywołania systemowego:

```
Program użytkownika → Biblioteka standardowa → Przerwanie → Jądro OS → Wynik
```

28.5 Przykłady wywołań systemowych

28.5.1 1. Wywołania systemowe do obsługi plików

- **open()** – otwiera plik.
- **read()** – odczytuje dane z pliku.
- **write()** – zapisuje dane do pliku.
- **close()** – zamyka plik.

Przykład w języku C:

```
int fd = open("plik.txt", O_RDONLY);
read(fd, buffer, sizeof(buffer));
close(fd);
```

28.5.2 2. Wywołania systemowe do zarządzania procesami

- **fork()** – tworzy nowy proces.
- **exec()** – uruchamia nowy program w bieżącym procesie.
- **wait()** – czeka na zakończenie procesu potomnego.
- **exit()** – kończy działanie procesu.

Przykład użycia `fork()`:

```
pid_t pid = fork();
if (pid == 0) {
    printf("Proces potomny\n");
} else {
    printf("Proces macierzysty\n");
}
```

28.5.3 3. Wywołania systemowe do zarządzania pamięcią

- **mmap()** – mapuje plik lub pamięć do przestrzeni adresowej.
- **brk()** – dynamicznie zmienia rozmiar sterty procesu.

Przykład użycia `mmap()`:

```
void *ptr = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1,
```

28.5.4 4. Wywołania systemowe do komunikacji międzyprocesowej

- **pipe()** – tworzy kanał komunikacyjny między procesami.
- **shmget()** – tworzy segment pamięci współdzielonej.
- **socket()** – tworzy gniazdo komunikacyjne.

28.6 Podsumowanie

- Wywołania systemowe są podstawowym mechanizmem komunikacji między aplikacjami a jądrem systemu operacyjnego.
- Są używane do zarządzania plikami, procesami, pamięcią oraz komunikacją międzyprocesową.
- Wywołania systemowe przechodzą z przestrzeni użytkownika do przestrzeni jądra, zapewniając kontrolowany dostęp do zasobów systemowych.

29 Główne wzorce architektoniczne oprogramowania

29.1 Wprowadzenie

Wzorce architektoniczne definiują wysokopoziomowe struktury organizacyjne systemów oprogramowania. Określają sposób podziału systemu na komponenty oraz ich interakcje, co pozwala na efektywne projektowanie skalowalnych, elastycznych i łatwych w utrzymaniu aplikacji.

29.2 Podstawowe wzorce architektoniczne

29.2.1 1. Architektura warstwowa (Layered Architecture)

Opis: System podzielony jest na warstwy, z których każda realizuje określoną funkcjonalność i komunikuje się z warstwami sąsiednimi.

Struktura:

- Warstwa prezentacji – interfejs użytkownika.
- Warstwa logiki biznesowej – przetwarzanie danych i reguły aplikacji.
- Warstwa dostępu do danych – komunikacja z bazą danych.
- Warstwa danych – baza danych i systemy przechowywania informacji.

Zalety:

- Modularność i łatwość testowania.
- Możliwość oddzielenia interfejsu od logiki biznesowej.

Wady:

- Może prowadzić do spadku wydajności z powodu licznych warstw pośrednich.

29.2.2 2. Architektura klient-serwer (Client-Server)

Opis: System podzielony jest na dwa główne komponenty: klienta, który żąda usług, oraz serwer, który te usługi udostępnia.

Zastosowanie:

- Aplikacje internetowe (np. HTTP, REST API).
- Systemy baz danych (np. MySQL, PostgreSQL).

Zalety:

- Centralizacja logiki aplikacji na serwerze.
- Łatwa aktualizacja klienta bez wpływu na serwer.

Wady:

- Wysokie obciążenie serwera przy dużej liczbie użytkowników.

29.2.3 3. Architektura mikroserwisowa (Microservices)

Opis: System składa się z wielu niezależnych usług (mikroserwisów), które komunikują się przez API.

Zastosowanie:

- Aplikacje chmurowe i rozproszone.
- Duże systemy internetowe (np. Netflix, Amazon).

Zalety:

- Skalowalność i łatwość wdrażania poszczególnych komponentów.
- Możliwość użycia różnych technologii w różnych mikroserwisach.

Wady:

- Złożoność zarządzania komunikacją między usługami.

29.2.4 4. Architektura zdarzeniowa (Event-Driven Architecture)

Opis: Komponenty systemu komunikują się poprzez zdarzenia asynchroniczne.

Zastosowanie:

- Systemy czasu rzeczywistego (np. IoT, giełda papierów wartościowych).

Zalety:

- Wysoka responsywność i skalowalność.

Wady:

- Złożoność obsługi komunikacji i zarządzania stanem.

29.2.5 5. Architektura model-widok-kontroler (MVC – Model-View-Controller)

Opis: System podzielony na trzy komponenty:

- **Model** – logika biznesowa i dane.
- **Widok (View)** – interfejs użytkownika.
- **Kontroler (Controller)** – obsługuje interakcje użytkownika.

Zastosowanie:

- Aplikacje internetowe (np. frameworki jak Django, Spring MVC).

Zalety:

- Oddzielenie warstw poprawia organizację kodu.

Wady:

- Może prowadzić do złożoności w większych systemach.

29.2.6 6. Architektura repozytorium (Repository Architecture)

Opis: System składa się z centralnego repozytorium danych, z którego korzystają wszystkie komponenty.

Zastosowanie:

- Kompilatory (np. GCC).
- Systemy zarządzania wiedzą.

Zalety:

- Spójność danych w całym systemie.

Wady:

- Możliwy wąskie gardło w dostępie do repozytorium.

29.2.7 7. Architektura rurek i filtrów (Pipe-and-Filter)

Opis: System składa się z przetwarzających dane komponentów (*filtrów*), które są połączone kanałami przepływu danych (*rurkami*).

Zastosowanie:

- Przetwarzanie strumieni danych (np. potok w systemach UNIX).

Zalety:

- Możliwość łatwego komponowania procesów.

Wady:

- Może powodować wysoką latencję przy dużych zbiorach danych.

29.3 Podsumowanie

- Wzorce architektoniczne pomagają w organizacji systemów oprogramowania.
- Architektura warstwowa jest klasyczna i szeroko stosowana.
- Mikroserwisy i architektura zdarzeniowa są preferowane w nowoczesnych, skalowalnych systemach.
- Wybór wzorca zależy od wymagań projektu, wydajności i skalowalności systemu.

30 Refaktoryzacja oprogramowania i wybrane jej sposoby

30.1 Wprowadzenie

Refaktoryzacja oprogramowania to proces restrukturyzacji kodu w celu poprawy jego jakości, czytelności i utrzymania, bez zmiany zewnętrznego zachowania programu. Jest kluczowa dla długoterminowej efektywności projektu i zapobiega zjawisku długu technologicznego.

30.2 Cele refaktoryzacji

- Poprawa czytelności i zrozumiałości kodu.
- Zmniejszenie złożoności i redundancji.
- Ułatwienie przyszłego rozwoju i utrzymania systemu.
- Poprawa wydajności poprzez eliminację zbędnych operacji.
- Zwiększenie testowalności i redukcja błędów.

30.3 Podstawowe techniki refaktoryzacji

30.3.1 1. Ekstrakcja metod (*Extract Method*)

Polega na wydzieleniu fragmentu kodu do nowej metody w celu zwiększenia czytelności i unikania duplikacji.

Przykład przed refaktoryzacją:

```
void oblicz() {
    int suma = 0;
    for (int i = 0; i < lista.size(); i++) {
        suma += lista[i];
    }
    System.out.println("Suma: " + suma);
}
```

Po refaktoryzacji:

```
void oblicz() {
    int suma = sumujElementy();
    System.out.println("Suma: " + suma);
}

int sumujElementy() {
    int suma = 0;
    for (int i = 0; i < lista.size(); i++) {
        suma += lista[i];
    }
    return suma;
}
```

30.3.2 2. Zastąpienie magicznych liczb stałymi (*Replace Magic Number with Constant*)

Pozwala uniknąć nieczytelnych wartości liczbowych w kodzie.

Przed:

```
double obliczObwod(double promien) {
    return 2 * 3.14159 * promien;
}
```

Po:

```
static final double PI = 3.14159;

double obliczObwod(double promien) {
    return 2 * PI * promien;
}
```

30.3.3 3. Wprowadzenie obiektu parametru (*Introduce Parameter Object*)

Zamiast przekazywać wiele argumentów, można przekazać obiekt enkapsulujący dane.

Przed:

```
void ustawWymiary(int szerokosc, int wysokosc, int glebokosc) { ... }
```

Po:

```
class Wymiary {
    int szerokosc, wysokosc, glebokosc;
}

void ustawWymiary(Wymiary wymiary) { ... }
```

30.3.4 4. Usunięcie zbędnych komentarzy

Czysty kod powinien być samodokumentujący się, a nadmiar komentarzy może świadczyć o złej jakości kodu.

Przed:

```
// Dodaje produkt do listy
listaProduktow.add(produkt);
```

Po:

```
dodajProduktDoListy(produkt);
```

30.3.5 5. Podział dużych klas (*Extract Class*)

Jeśli klasa ma zbyt wiele odpowiedzialności, warto podzielić ją na mniejsze.

Przed:

```
class Zamowienie {
    List<Produkt> produkty;
    double obliczCene() { ... }
    void wyslijEmailPotwierdzajacy() { ... }
}
```

Po podziale:

```
class Zamowienie {
    List<Produkt> produkty;
    double obliczCene() { ... }
}

class Notyfikacja {
    void wyslijEmailPotwierdzajacy() { ... }
}
```

30.4 Automatyzacja refaktoryzacji

Wiele narzędzi wspiera refaktoryzację, np.:

- IntelliJ IDEA, Eclipse – refaktoryzacja kodu w językach obiektowych.
- SonarQube – analiza jakości kodu.
- Black, Prettier – formatowanie kodu w Pythonie i JavaScript.

30.5 Podsumowanie

- Refaktoryzacja poprawia jakość kodu i ułatwia jego utrzymanie.
- Wprowadzenie metod, eliminacja magicznych liczb i podział dużych klas zwiększają czytelność.
- Nowoczesne narzędzia wspomagają proces refaktoryzacji.
- Regularna refaktoryzacja zmniejsza ryzyko długu technologicznego.

31 Główne założenia metodyki eXtreme Programming

31.1 Wprowadzenie

eXtreme Programming (XP) to zwinna metodyka wytwarzania oprogramowania, która kładzie nacisk na adaptację do zmieniających się wymagań, wysoką jakość kodu oraz ścisłą współpracę zespołu. XP jest szczególnie skuteczna w dynamicznych projektach, gdzie wymagania klienta mogą ewoluować w trakcie prac.

31.2 Główne założenia eXtreme Programming

31.2.1 1. Komunikacja

XP promuje intensywną komunikację pomiędzy członkami zespołu, co minimalizuje nieporozumienia i zwiększa efektywność pracy.

Przykłady praktyk:

- Codzienne spotkania zespołu (*stand-up meetings*).
- Programowanie w parach (*pair programming*).
- Bezpośrednie konsultacje z klientem.

31.2.2 2. Prostota

XP zachęca do stosowania prostych rozwiązań, które spełniają wymagania projektu, eliminując zbędną złożoność.

Praktyki:

- Implementowanie tylko niezbędnej funkcjonalności.
- Stosowanie przejrzystego i łatwego w utrzymaniu kodu.

31.2.3 3. Informacje zwrotne

Ciągłe zbieranie informacji zwrotnych pozwala na szybkie dostosowanie systemu do zmieniających się wymagań.

Przykłady:

- Testy jednostkowe uruchamiane po każdej zmianie kodu.
- Regularne wersje oprogramowania dostarczane klientowi.

31.2.4 4. Odwaga

Programiści w XP powinni być gotowi na wprowadzanie zmian w kodzie i eliminację błędów bez obawy o negatywne skutki.

Praktyki:

- Ciągła refaktoryzacja.
- Śmiałe podejmowanie decyzji technicznych.

31.2.5 5. Szacunek

XP podkreśla znaczenie wzajemnego szacunku w zespole, co wpływa na dobrą atmosferę pracy i skuteczną współpracę.

Praktyki:

- Wspólne podejmowanie decyzji projektowych.
- Docenianie wkładu każdego członka zespołu.

31.3 Podstawowe praktyki XP

31.3.1 1. Programowanie w parach (Pair Programming)

Dwóch programistów wspólnie pracuje nad tym samym fragmentem kodu, co poprawia jakość i redukuje liczbę błędów.

31.3.2 2. Ciągła integracja (Continuous Integration)

Kod jest regularnie integrowany z główną wersją systemu, a testy są automatycznie uruchamiane po każdej zmianie.

31.3.3 3. Test Driven Development (TDD)

Najpierw pisane są testy, a dopiero potem implementowana jest funkcjonalność, co zapewnia większą niezawodność systemu.

31.3.4 4. Refaktoryzacja

Regularne poprawianie kodu w celu zwiększenia jego czytelności i efektywności.

31.3.5 5. Małe wydania (Small Releases)

System jest dostarczany w krótkich iteracjach, co pozwala klientowi na bieżąco oceniać postęp prac.

31.4 Podsumowanie

- XP koncentruje się na komunikacji, prostocie, informacji zwrotnej, odwadze i szacunku.
- Kluczowe praktyki to programowanie w parach, TDD, ciągła integracja i refaktoryzacja.
- Metodyka ta pozwala na szybkie reagowanie na zmiany i dostarczanie wysokiej jakości oprogramowania.

32 Charakterystyka środków dostępnych w języku CUDA C/C++ i sposoby ich wykorzystania

32.1 Wprowadzenie

CUDA (Compute Unified Device Architecture) to platforma i model programowania opracowany przez firmę NVIDIA, umożliwiający wykorzystanie procesorów graficznych (GPU) do obliczeń równoległych. CUDA C/C++ to rozszerzenie języka C/C++, które pozwala na definiowanie i uruchamianie funkcji równoległych na GPU.

32.2 Podstawowe środki dostępne w języku CUDA

32.2.1 1. Hierarchia wątków CUDA

CUDA organizuje obliczenia w hierarchii wątków:

- **Wątki (threads)** – podstawowa jednostka obliczeniowa.
- **Bloki wątków (blocks)** – grupy wątków, które współdzielą pamięć współdzieloną (*shared memory*).
- **Siatka bloków (grid)** – zbiór bloków uruchamianych na GPU.

Przykład uruchomienia kernela CUDA:

```
kernel<<<numBlocks, numThreads>>>(d_data);
```

32.2.2 2. Model pamięci w CUDA

CUDA udostępnia różne rodzaje pamięci:

- **Pamięć globalna (global memory)** – dostępna dla wszystkich wątków, ale o wysokich opóźnieniach.
- **Pamięć współdzielona (shared memory)** – współdzielona w obrębie bloku wątków, szybka.
- **Pamięć lokalna (local memory)** – prywatna dla wątku, ale przechowywana w pamięci globalnej.
- **Pamięć rejestrów (register memory)** – najszybsza, ale o ograniczonym rozmiarze.
- **Pamięć stała (constant memory)** – zoptymalizowana dla niezmiennych danych.
- **Pamięć tekstur i powierzchni (texture & surface memory)** – zoptymalizowana dla operacji na obrazach.

Przykład alokacji pamięci globalnej:

```
cudaMalloc((void**)&d_array, size);
cudaMemcpy(d_array, h_array, size, cudaMemcpyHostToDevice);
```

32.2.3 3. Definiowanie i uruchamianie kerneli

Funkcje CUDA (kernels) są oznaczane kwalifikatorem `__global__` i wykonywane równolegle przez wiele wątków.

Przykład prostego kernela:

```
__global__ void add(int *a, int *b, int *c) {
    int idx = threadIdx.x;
    c[idx] = a[idx] + b[idx];
}
```

32.2.4 4. Synchronizacja wątków

CUDA udostępnia mechanizmy synchronizacji:

- `__syncthreads()` – synchronizacja wątków w obrębie bloku.
- **Atomiczne operacje** – zapewniają bezpieczne operacje na współdzielonych danych.

Przykład synchronizacji:

```
__shared__ int shared_data[256];
__syncthreads();
```

32.2.5 5. Strumienie i wielowątkowość

CUDA pozwala na równoczesne wykonywanie wielu operacji za pomocą strumieni (*streams*) i kolejek.

Przykład użycia strumieni:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
kernel<<<grid, block, 0, stream>>>();
```

32.2.6 6. Obsługa błędów w CUDA

CUDA zapewnia mechanizmy obsługi błędów za pomocą makr i funkcji `cudaGetErrorString()`.

Przykład sprawdzania błędów:

```
cudaError_t err = cudaMalloc((void**)&d_array, size);
if (err != cudaSuccess) {
    printf("CUDA error: %s\n", cudaGetErrorString(err));
}
```

32.3 Podsumowanie

- CUDA C/C++ udostępnia mechanizmy do programowania równoległego na GPU.
- Wątki organizowane są w bloki i siatki, które korzystają z różnych typów pamięci.
- Istnieją mechanizmy synchronizacji, obsługi błędów i optymalizacji wydajności.
- CUDA znajduje zastosowanie w obliczeniach naukowych, grafice, AI i modelowaniu numerycznym.

33 Charakterystyka środowiska sprzętowego NVIDIA CUDA i modelu wykonania SIMT

33.1 Wprowadzenie

CUDA (Compute Unified Device Architecture) to opracowana przez firmę NVIDIA platforma umożliwiająca wykorzystanie procesorów graficznych (GPU) do obliczeń równoległych. Środowisko sprzętowe CUDA charakteryzuje się specyficzną architekturą opartą na modelu równoległego wykonywania instrukcji SIMT (*Single Instruction Multiple Threads*).

33.2 Architektura sprzętowa NVIDIA CUDA

Architektura sprzętowa CUDA opiera się na hierarchicznej organizacji zasobów obliczeniowych:

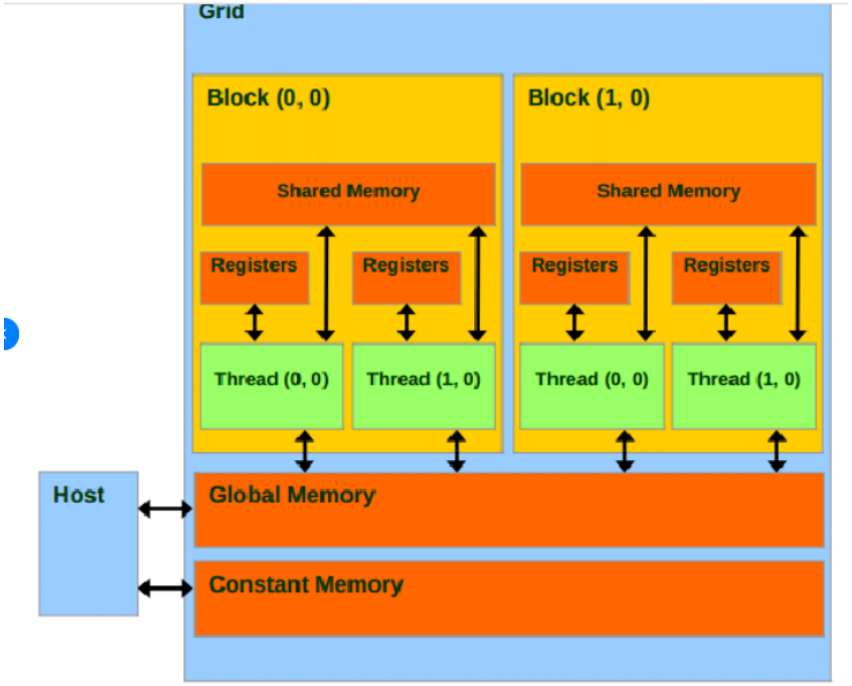
- **GPU (Graphics Processing Unit)** – jednostka obliczeniowa zawierająca wiele multiprocesorów strumieniowych.
- **Multiprocesory strumieniowe (SM – Streaming Multiprocessor)** – podstawowe jednostki obliczeniowe GPU, zawierające zestawy rdzeni.
- **Rdzenie CUDA (CUDA Cores)** – elementarne jednostki wykonawcze realizujące operacje arytmetyczne i logiczne.
- **Warp** – grupa 32 wątków wykonywanych równocześnie w jednym SM.
- **Blok wątków (Thread Block)** – zestaw wątków współdzielących pamięć lokalną.
- **Siatka bloków (Grid of Blocks)** – zbiór bloków realizujących obliczenia w danym kernelu CUDA.

33.3 Model wykonania SIMT (Single Instruction Multiple Threads)

Model **SIMT** stosowany w architekturze CUDA polega na jednoczesnym wykonywaniu tej samej instrukcji przez wiele wątków.

Cechy SIMT:

- Każdy multiprocesor SM wykonuje wiele wątków równocześnie.
- Wątki w ramach jednego warpa (32 wątki) wykonują te same instrukcje, ale na różnych danych.
- Gdy występuje rozgałęzienie warunkowe, wątki w warpie mogą wykonywać różne ścieżki kodu (*thread divergence*), co prowadzi do spadku wydajności.



Rysunek 1: Hierarchiczna organizacja zasobów CUDA.

33.4 Struktura wykonania w CUDA

33.4.1 1. Hierarchia wątków w modelu CUDA

- **Wątek (Thread)** – jednostka wykonawcza pracująca na określonych danych.
- **Blok wątków (Thread Block)** – grupa wątków, które mogą współdzielić pamięć współdzieloną.
- **Siatka bloków (Grid of Blocks)** – zestaw bloków, które wykonują równoległe obliczenia.

Przykład konfiguracji siatki i bloków:

```
kernel<<<numBlocks, threadsPerBlock>>>(d_data);
```

33.4.2 2. Organizacja pamięci w CUDA

CUDA udostępnia różne rodzaje pamięci:

- **Pamięć rejestrów** – najszybsza, ale ograniczona ilościowo.
- **Pamięć współdzielona (Shared Memory)** – szybka pamięć dostępna w obrębie jednego bloku wątków.
- **Pamięć globalna (Global Memory)** – dostępna dla wszystkich wątków, ale o dużym opóźnieniu.
- **Pamięć stała (Constant Memory)** – zoptymalizowana dla niezmiennych danych.
- **Pamięć tekstur i powierzchni (Texture & Surface Memory)** – wykorzystywana w operacjach graficznych i analizie obrazu.

33.5 Przykładowa implementacja kernela CUDA

Przykład prostego kernela sumującego dwa wektory:

```
__global__ void addVectors(int *a, int *b, int *c, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}
```

33.6 Zalety i ograniczenia modelu SIMT

Zalety:

- Wysoka wydajność w obliczeniach masowo-równoległych.
- Efektywne wykorzystanie zasobów sprzętowych GPU.
- Redukcja czasu obliczeń w porównaniu do CPU.

Ograniczenia:

- Problemy z *thread divergence*, gdy wątki wykonują różne ścieżki kodu.
- Konieczność optymalizacji dostępu do pamięci globalnej.
- Wysokie wymagania pamięciowe przy dużych rozmiarach danych.

33.7 Podsumowanie

- CUDA pozwala na programowanie GPU z wykorzystaniem hierarchii wątków.
- Model SIMT umożliwia jednoczesne wykonywanie tej samej instrukcji przez wiele wątków.
- Optymalizacja pamięci i unikanie *thread divergence* są kluczowe dla efektywnego programowania CUDA.
- CUDA znajduje zastosowanie w obliczeniach naukowych, sztucznej inteligencji i grafice komputerowej.

34 Techniki programowania procesorów graficznych wykorzystujące mechanizm wielowątkowości

34.1 Wprowadzenie

Programowanie procesorów graficznych (GPU) wykorzystuje mechanizmy wielowątkowości do równoległego przetwarzania dużych ilości danych. Model CUDA (Compute Unified Device Architecture) firmy NVIDIA umożliwia efektywne zarządzanie tysiącami wątków działających jednocześnie, co pozwala na znaczące przyspieszenie obliczeń w porównaniu do tradycyjnych procesorów CPU.

34.2 Podstawowe techniki programowania GPU

34.2.1 1. Model wykonania SIMT (Single Instruction Multiple Threads)

CUDA wykorzystuje model SIMT, który pozwala na jednoczesne wykonywanie tej samej instrukcji przez wiele wątków.

Cechy SIMT:

- Wątki są organizowane w grupy zwane **warpami** (po 32 wątki).
- Każdy multiprocesor strumieniowy (SM) zarządza wieloma warpami.
- Rozgałęzienia warunkowe mogą prowadzić do *thread divergence*, obniżając wydajność.

34.2.2 2. Hierarchia wątków w CUDA

CUDA organizuje wątki w strukturę hierarchiczną:

- **Wątki (Threads)** – podstawowa jednostka wykonawcza.
- **Bloki wątków (Thread Blocks)** – grupa wątków współdzielących pamięć lokalną.
- **Siatka bloków (Grid of Blocks)** – organizuje bloki dla większych obliczeń.

Przykład uruchomienia kernela CUDA:

```
kernel<<<numBlocks, threadsPerBlock>>>(d_data);
```

34.2.3 3. Wykorzystanie pamięci współdzielonej (Shared Memory)

Pamięć współdzielona pozwala wątkom w obrębie bloku na szybkie wymienianie danych.

Przykład użycia pamięci współdzielonej:

```
__shared__ int shared_data[256];

int idx = threadIdx.x;
shared_data[idx] = global_data[idx];

__syncthreads();  // Synchronizacja wątków bloku
```

34.2.4 4. Unikanie *thread divergence*

Podział warpa na różne ścieżki wykonania obniża wydajność. Unika się tego poprzez:

- Stosowanie jednolitych operacji dla wszystkich wątków w warpie.
- Unikanie instrukcji warunkowych w obrębie warpa.
- Grupowanie operacji na wspólnych danych.

34.2.5 5. Optymalizacja dostępu do pamięci globalnej

Dostęp do pamięci globalnej (Global Memory) jest wolny, dlatego stosuje się:

- **Koalescencję pamięci** – wątki powinny odczytywać dane w uporządkowany sposób.
- **Pamięć współdzieloną** jako bufor pośredni.

Przykład poprawnego dostępu do pamięci globalnej:

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
int value = global_data[idx];  // Odpowiednie wyrównanie dostępu
```

34.2.6 6. Wykorzystanie wielowątkowości na poziomie bloków i siatek

Obliczenia mogą być skalowane przez zwiększenie liczby bloków i siatek.

Przykład konfiguracji siatki i bloków:

```
dim3 blocksPerGrid(16, 16);
dim3 threadsPerBlock(32, 32);
kernel<<<blocksPerGrid, threadsPerBlock>>>(d_data);
```

34.2.7 7. Strumienie CUDA (CUDA Streams)

CUDA umożliwia wykonywanie wielu operacji jednocześnie poprzez użycie strumieni.

Przykład:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
kernel<<<grid, block, 0, stream>>>();
```

34.2.8 8. Operacje atomowe

CUDA obsługuje operacje atomowe, które pozwalają uniknąć konfliktów podczas modyfikowania współdzielonych danych.

Przykład operacji atomowej:

```
atomicAdd(&global_sum, value);
```

34.2.9 9. Wykorzystanie bibliotek CUDA

CUDA oferuje zestaw gotowych bibliotek, takich jak:

- **cuBLAS** – operacje na macierzach.
- **cuFFT** – szybka transformata Fouriera.
- **Thrust** – wysokopoziomowe operacje na wektorach.

34.3 Podsumowanie

- CUDA wykorzystuje model SIMT, organizując wątki w warpy i bloki.
- Efektywne wykorzystanie pamięci współdzielonej i optymalizacja dostępu do pamięci poprawiają wydajność.
- Strumienie, operacje atomowe i biblioteki CUDA umożliwiają dalsze przyspieszenie obliczeń.
- Unikanie *thread divergence* i stosowanie koalescencji pamięci jest kluczowe dla wysokiej wydajności.

35 Definicja programowania równoległego i współbieżnego; popularne języki; biblioteki i API

35.1 Wprowadzenie

Programowanie równoległe i współbieżne to dwa podejścia do realizacji obliczeń wielowątkowych, które różnią się sposobem wykonywania zadań i organizacją pracy procesów.

35.2 Definicja programowania równoległego i współbieżnego

35.2.1 1. Programowanie równoległe (Parallel Computing)

Programowanie równoległe polega na jednoczesnym wykonywaniu wielu operacji w celu przyspieszenia obliczeń.

Cechy:

- Wiele zadań wykonywanych jednocześnie na różnych procesorach lub rdzeniach.
- Typowe w obliczeniach naukowych, grafice komputerowej i AI.
- Wymaga synchronizacji i zarządzania współdzielonymi zasobami.

Przykłady zastosowań:

- Przetwarzanie dużych zbiorów danych (Big Data).
- Algorytmy sztucznej inteligencji (trenowanie sieci neuronowych).
- Obliczenia numeryczne i symulacje fizyczne.

35.2.2 2. Programowanie współbieżne (Concurrent Computing)

Programowanie współbieżne umożliwia wykonywanie wielu zadań w jednym czasie, ale niekoniecznie równolegle.

Cechy:

- Wiele procesów może być wykonywanych przeplatająco na jednym rdzeniu.
- Stosowane w systemach operacyjnych i aplikacjach sieciowych.
- Może wykorzystywać mechanizmy wątków, procesów i asynchronicznego wykonywania kodu.

Przykłady zastosowań:

- Obsługa wielu klientów w serwerach HTTP.
- Zarządzanie zadaniami w systemie operacyjnym.
- Aplikacje działające w tle (np. pobieranie plików).

35.3 Popularne języki programowania do programowania równoległego i współbieżnego

- **C/C++** – języki niskopoziomowe, szeroko stosowane w aplikacjach wysokowydajnościowych.
- **Python** – wspiera współbieżność dzięki bibliotekom takim jak `threading` i `multiprocessing`.
- **Java** – posiada wbudowaną obsługę wielowątkowości (`java.util.concurrent`).
- **Go** – zoptymalizowany pod kątem współbieżności dzięki lekkim wątkom (*goroutines*).
- **Rust** – zapewnia bezpieczne programowanie równoległe, eliminując błędy związane z dostępem do pamięci.
- **CUDA C/C++** – umożliwia programowanie równoległe na GPU.

35.4 Biblioteki i API do programowania równoległego i współbieżnego

35.4.1 1. Wątki i synchronizacja

- **POSIX Threads (pthread)** – standardowa biblioteka w C do obsługi wątków.
- **std::thread** (C++) – biblioteka do obsługi wątków w nowoczesnym C++.
- **Java Threads** – obsługa wielowątkowości wbudowana w język Java.
- **asyncio (Python)** – obsługa asynchronicznych operacji wejścia/wyjścia.

35.4.2 2. Równoległe przetwarzanie na wielu rdzeniach CPU

- **OpenMP** – standard do równoległego programowania w C, C++ i Fortranie.
- **TBB (Threading Building Blocks)** – biblioteka firmy Intel do równoległego programowania w C++.
- **multiprocessing (Python)** – umożliwia równoległe wykonywanie kodu na wielu procesorach.

35.4.3 3. Programowanie na GPU

- **CUDA** – platforma NVIDIA do programowania na GPU.
- **OpenCL** – otwarty standard do obliczeń równoległych na różnych platformach.
- **HIP (Heterogeneous-Compute Interface for Portability)** – API firmy AMD do programowania GPU.

35.4.4 4. Programowanie rozproszone

- **MPI (Message Passing Interface)** – standard komunikacji między procesami w klastrach komputerowych.
- **Apache Spark** – framework do obliczeń rozproszonych na dużych zbiorach danych.
- **Dask (Python)** – narzędzie do obliczeń równoległych na dużych zbiorach danych.

35.5 Podsumowanie

- Programowanie równoległe pozwala na jednoczesne wykonywanie wielu operacji na różnych procesorach lub rdzeniach.
- Programowanie współbieżne umożliwia wykonywanie wielu zadań jednocześnie, ale nie zawsze równolegle.
- Wiele języków programowania (C++, Python, Java, Go, CUDA) wspiera techniki wielowątkowe i równoległe.
- Biblioteki i API, takie jak OpenMP, MPI, CUDA, OpenCL i `pthread`, ułatwiają implementację obliczeń równoległych i współbieżnych.

36 Wydajność aplikacji równoległych: lokalność, przyspieszenie, efektywność, prawo Amdahla, prawo Gustafsona

36.1 Wprowadzenie

Ocena wydajności aplikacji równoległych obejmuje analizę kluczowych parametrów, takich jak przyspieszenie, efektywność oraz wpływ architektury pamięci na wydajność obliczeń. Dwa fundamentalne prawa – prawo Amdahla i prawo Gustafsona – opisują teoretyczne ograniczenia i potencjał skalowania aplikacji równoległych.

36.2 Lokalność danych

Lokalność odnosi się do sposobu, w jaki program odwołuje się do pamięci w czasie wykonywania. Optymalne zarządzanie lokalnością wpływa na wydajność aplikacji równoległych.

36.2.1 Rodzaje lokalności:

- **Lokalność czasowa (temporal locality)** – jeśli program odwołuje się do określonej komórki pamięci, istnieje duża szansa, że wkrótce ponownie się do niej odwoła.
- **Lokalność przestrzenna (spatial locality)** – jeśli program odwołuje się do określonej komórki pamięci, istnieje duża szansa, że wkrótce odwoła się do sąsiednich adresów pamięci.

Optymalizacja lokalności:

- Stosowanie buforowania i pamięci podręcznej (*cache*).
- Strukturyzacja danych i dostępu do pamięci w sposób zoptymalizowany pod kątem układu pamięci.
- Wektoryzacja kodu i unikanie nieuporządkowanego dostępu do pamięci.

36.3 Przyspieszenie i efektywność

36.3.1 1. Przyspieszenie (Speedup)

Przyspieszenie mierzy, jak bardzo poprawia się czas wykonania programu po zastosowaniu równoległości.

$$S(p) = \frac{T_1}{T_p} \tag{5}$$

gdzie:

- $S(p)$ – przyspieszenie dla p procesorów,
- T_1 – czas wykonania programu w wersji sekwencyjnej,
- T_p – czas wykonania programu przy p procesorach.

36.3.2 2. Efektywność (Efficiency)

Efektywność mierzy, jak dobrze wykorzystano dostępne procesory.

$$E(p) = \frac{S(p)}{p} = \frac{T_1}{pT_p} \tag{6}$$

gdzie:

- $E(p)$ – efektywność,
- p – liczba procesorów.

Interpretacja:

- Jeśli $E(p) \approx 1$, oznacza to optymalne wykorzystanie zasobów.
- Jeśli $E(p) \ll 1$, oznacza to, że dodanie procesorów nie przynosi oczekiwanej poprawy wydajności.

36.4 Prawo Amdahla

Prawo Amdahla określa teoretyczne ograniczenie przyspieszenia równoległego programu.

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}} \tag{7}$$

gdzie:

- f – część programu, która może zostać zrównoleglona,
- $(1-f)$ – część programu, która musi pozostać sekwencyjna.

Wnioski:

- Nawet jeśli $p \to \infty$, przyspieszenie jest ograniczone przez sekwencyjną część kodu.
- Im większy udział części sekwencyjnej, tym mniejsze możliwe przyspieszenie.

Przykład: Jeśli 90% kodu można zrównoleglić ($f = 0.9$), to dla $p = 10$ procesorów:

$$S(10) = \frac{1}{(1-0.9) + \frac{0.9}{10}} = \frac{1}{0.1 + 0.09} \approx 5.26 \tag{8}$$

36.5 Prawo Gustafsona

Prawo Gustafsona sugeruje, że można osiągnąć większe przyspieszenie, jeśli wraz ze wzrostem liczby procesorów rośnie również problem obliczeniowy.

$$S(p) = p - \alpha(p-1) \tag{9}$$

gdzie:

- α – proporcja części sekwencyjnej programu,
- p – liczba procesorów.

Wnioski:

- Prawo Gustafsona pokazuje, że skalowanie aplikacji jest możliwe, jeśli problem obliczeniowy rośnie wraz z liczbą procesorów.
- W praktyce pozwala to uzyskać lepsze przyspieszenie niż przewidywane przez prawo Amdahla.

36.6 Porównanie prawa Amdahla i Gustafsona

Cecha	**Prawo Amdahla**	**Prawo Gustafsona**
Podejście	Analizuje stały problem	Skalowalność dla rosnącego problemu
Ograniczenie	Narzucone przez część sekwencyjną	Problem obliczeniowy może rosnąć
Skalowanie	Przy dużej liczbie procesorów ograniczone	Możliwe duże przyspieszenie
Zastosowanie	Obliczenia o stałym rozmiarze	Obliczenia dynamicznie zwiększające się

Tabela 7: Porównanie prawa Amdahla i Gustafsona

36.7 Podsumowanie

- Wydajność aplikacji równoległych zależy od przyspieszenia, efektywności oraz lokalności dostępu do pamięci.
- Prawo Amdahla ogranicza maksymalne przyspieszenie przez część sekwencyjną kodu.
- Prawo Gustafsona pokazuje, że przy rosnącym rozmiarze problemu, można osiągnąć większe przyspieszenie.
- Optymalizacja lokalności pamięci i unikanie wąskich gardeł w dostępie do danych zwiększa efektywność obliczeń równoległych.

37 Mechanizmy przesyłania komunikatów w systemach rozproszonych

37.1 Wprowadzenie

Systemy rozproszone składają się z wielu węzłów komunikujących się poprzez sieć. Podstawowym mechanizmem wymiany informacji między procesami jest przesyłanie komunikatów (*message passing*). Efektywna komunikacja w systemach rozproszonych wymaga mechanizmów zapewniających niezawodność, synchronizację i efektywne zarządzanie przesyłanymi danymi.

37.2 Podstawowe modele komunikacji

37.2.1 1. Komunikacja synchroniczna i asynchroniczna

- **Komunikacja synchroniczna** – nadawca czeka na potwierdzenie odbioru komunikatu przed kontynuacją pracy.
- **Komunikacja asynchroniczna** – nadawca wysyła komunikat i natychmiast kontynuuje działanie, a odbiorca może go odebrać później.

Zastosowanie:

- Synchroniczna – systemy wymagające spójności, np. systemy bankowe.
- Asynchroniczna – systemy wymagające wysokiej przepustowości, np. przesyłanie wiadomości w mediach społecznościowych.

37.2.2 2. Komunikacja jednokierunkowa i dwukierunkowa

- **Jednokierunkowa (Unidirectional)** – komunikacja odbywa się w jednym kierunku (np. *UDP*).
- **Dwukierunkowa (Bidirectional)** – komunikacja wymaga odpowiedzi od odbiorcy (np. *TCP*).

37.2.3 3. Komunikacja jedno-do-jednego i jedno-do-wielu

- **Jedno-do-jednego (point-to-point)** – komunikacja między dwoma procesami.
- **Jedno-do-wielu (multicast, broadcast)** – komunikacja do wielu procesów jednocześnie.

37.3 Mechanizmy przesyłania komunikatów

37.3.1 1. Gniazda (Sockets)

Gniazda to podstawowy mechanizm komunikacji między procesami w sieci.

Rodzaje gniazd:

- **Gniazda strumieniowe (TCP)** – zapewniają niezawodny, uporządkowany przesył danych.

- **Gniazda datagramowe (UDP)** – zapewniają szybki, ale niegwarantowany przesył danych.

Przykład kodu w C (gniazdo TCP):

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr));
send(sockfd, message, strlen(message), 0);
```

37.3.2 2. Kolejki komunikatów (Message Queues)

Mechanizm kolejek pozwala na przechowywanie i odbieranie komunikatów asynchronicznie.

Popularne implementacje:

- **POSIX Message Queues** – standard w systemach UNIX.
- **RabbitMQ, Apache Kafka** – systemy kolejkowania komunikatów w systemach rozproszonych.

37.3.3 3. Pamięć współdzielona

Procesy mogą wymieniać dane poprzez wspólną przestrzeń adresową.

Zalety:

- Bardzo szybka wymiana danych.
- Unikanie narzutu komunikacyjnego sieci.

Wady:

- Wymaga synchronizacji dostępu (semafory, muteksy).
- Ograniczona do systemów działających na jednym węźle.

37.3.4 4. Remote Procedure Call (RPC)

RPC pozwala na wywoływanie funkcji na zdalnym komputerze tak, jakby były lokalne.

Przykłady technologii:

- **gRPC** – nowoczesne RPC oparte na HTTP/2.
- **XML-RPC, JSON-RPC** – lekkie protokoły RPC.

Przykład RPC w Pythonie (gRPC):

```
import grpc
channel = grpc.insecure_channel('localhost:50051')
response = stub.MethodName(request)
```

37.3.5 5. Publish-Subscribe (Pub/Sub)

Wzorzec komunikacyjny, w którym nadawca (*publisher*) wysyła komunikaty do kanału, a subskrybenci (*subscribers*) je odbierają.

Przykłady implementacji:

- **Apache Kafka** – skalowalna platforma do przesyłania strumieni danych.
- **MQTT** – protokół komunikacji w IoT.

37.3.6 6. Strumieniowanie komunikatów (Message Streaming)

Mechanizm umożliwia przesyłanie dużych ilości danych w czasie rzeczywistym.

Przykłady implementacji:

- **Apache Kafka, Apache Pulsar** – rozproszone przetwarzanie strumieniowe.
- **Google Cloud Pub/Sub** – system do strumieniowego przesyłania danych w chmurze.

37.4 Porównanie mechanizmów przesyłania komunikatów

Mechanizm	**Zalety**	**Wady**
Gniazda (Sockets)	Niski narzut, szybka komunikacja	Wymaga zarządzania połączeniami
Kolejki komunikatów	Buforowanie komunikatów	Większe opóźnienia
Pamięć współdzielona	Najszybsza wymiana danych	Ograniczona do jednego systemu
RPC	Transparentność wywołań	Opóźnienia sieciowe
Publish-Subscribe	Łatwa skalowalność	Opóźnienia propagacji
Strumieniowanie	Przetwarzanie w czasie rzeczywistym	Złożoność implementacji

Tabela 8: Porównanie mechanizmów przesyłania komunikatów

37.5 Podsumowanie

- Przesyłanie komunikatów w systemach rozproszonych może odbywać się synchronicznie lub asynchronicznie.
- Gniazda są podstawowym mechanizmem komunikacji w sieci.
- Kolejki komunikatów i strumieniowanie danych zapewniają niezależność nadawcy i odbiorcy.
- RPC upraszcza wywołania zdalnych funkcji, a Pub/Sub umożliwia komunikację wielu węzłów.
- Wybór mechanizmu zależy od wymagań dotyczących niezawodności, opóźnień i skalowalności systemu.

38 Topologie systemów rozproszonych

38.1 Wprowadzenie

Systemy rozproszone składają się z wielu węzłów komunikujących się w celu realizacji wspólnych zadań. Struktura połączeń między tymi węzłami, określana jako **topologia**, wpływa na wydajność, odporność na błędy oraz możliwości skalowania systemu. Wybór topologii zależy od charakterystyki aplikacji i wymagań dotyczących przepustowości, niezawodności i latencji komunikacyjnej.

38.2 Podstawowe topologie systemów rozproszonych

38.2.1 1. Topologia magistrali (Bus)

Wszystkie węzły są podłączone do jednej wspólnej linii komunikacyjnej (*magistrali*).

Cechy:

- Prosta implementacja i niski koszt.
- Ograniczona skalowalność – większa liczba węzłów prowadzi do przeciążeń komunikacyjnych.
- Awaria magistrali powoduje zatrzymanie całego systemu.

Zastosowanie:

- Systemy lokalne (LAN) o niewielkiej liczbie węzłów.
- Wczesne architektury systemów rozproszonych.

38.2.2 2. Topologia pierścienia (Ring)

Węzły są połączone w zamknięty łańcuch, gdzie każdy węzeł jest połączony z dwoma sąsiadami.

Cechy:

- Efektywne wykorzystanie przepustowości.
- Brak kolizji – komunikacja odbywa się w jednym kierunku.
- Awaria jednego węzła może przerwać komunikację, jeśli nie zastosowano mechanizmów redundancji.

Zastosowanie:

- Sieci Token Ring.
- Rozproszone systemy pamięci masowej.

38.2.3 3. Topologia gwiazdy (Star)

Wszystkie węzły są połączone z jednym centralnym węzłem.

Cechy:

- Centralizacja ułatwia zarządzanie ruchem sieciowym.
- Awaria centralnego węzła powoduje utratę komunikacji całej sieci.

Zastosowanie:

- Sieci Ethernet z przełącznikami.
- Systemy serwer-klient.

38.2.4 4. Topologia drzewa (Tree)

Hierarchiczna struktura, w której węzły są ułożone w postaci drzewa z centralnym węzłem głównym.

Cechy:

- Lepsza skalowalność niż topologia gwiazdy.
- Awaria węzła nadrzędnego może odciąć część systemu.

Zastosowanie:

- Rozproszone systemy plików (np. Hadoop HDFS).
- Sieci komunikacyjne.

38.2.5 5. Topologia siatki (Mesh)

Każdy węzeł jest połączony z kilkoma innymi, tworząc gęsto połączoną sieć.

Cechy:

- Wysoka odporność na awarie – redundancja połączeń.
- Duże koszty implementacji i zarządzania.

Zastosowanie:

- Sieci bezprzewodowe (np. Mesh WiFi).
- Systemy rozproszone o wysokiej dostępności.

38.2.6 6. Topologia hybrydowa (Hybrid)

Łączy cechy różnych topologii w celu uzyskania lepszej skalowalności i niezawodności.

Zastosowanie:

- Internet i sieci rozproszone na dużą skalę.
- Architektury chmurowe.

Topologia	Zalety	Wady	Zastosowanie
Magistrala	Prosta, tani koszt	Słaba skalowalność	Małe sieci LAN
Pierścień	Brak kolizji	Awaria może zatrzymać sieć	Sieci Token Ring
Gwiazda	Łatwa administracja	Awaria serwera wyłącza system	Sieci Ethernet
Drzewo	Skalowalne	Awaria węzła nadrzędnego izoluje część systemu	HDFS, sieci chmurowe
Siatka	Wysoka odporność	Wysoki koszt implementacji	Systemy wysokiej dostępności
Hybrydowa	Elastyczność i skalowalność	Złożoność konfiguracji	Internet, chmura

Tabela 9: Porównanie topologii systemów rozproszonych

38.3 Porównanie topologii

38.4 Podsumowanie

- Wybór topologii systemu rozproszonego wpływa na jego wydajność, niezawodność i skalowalność.
- Magistrala i pierścień są proste, ale mają ograniczoną odporność na awarie.
- Gwiazda i drzewo są efektywne dla systemów hierarchicznych.
- Siatka zapewnia wysoką niezawodność kosztem złożoności.
- Hybrydowe podejście łączy różne modele, aby uzyskać optymalne rozwiązanie.

39 Systemy ściśle i luźno powiązane

39.1 Wprowadzenie

Systemy komputerowe mogą być klasyfikowane na podstawie stopnia powiązania ich jednostek obliczeniowych. Wyróżnia się dwa główne modele: **systemy ściśle powiązane** (*tightly coupled systems*) oraz **systemy luźno powiązane** (*loosely coupled systems*). Wybór odpowiedniego modelu zależy od wymagań dotyczących wydajności, komunikacji oraz skalowalności.

39.2 Systemy ściśle powiązane (Tightly Coupled Systems)

39.2.1 Charakterystyka

- Współdzielona pamięć – wszystkie jednostki obliczeniowe korzystają ze wspólnej przestrzeni adresowej.
- Niski czas dostępu do pamięci – szybka komunikacja między procesorami.
- Zarządzanie procesami realizowane przez jeden system operacyjny.
- Mocno zintegrowane jednostki przetwarzające.

39.2.2 Przykłady

- Systemy wieloprocesorowe (SMP – Symmetric Multiprocessing).
- Superkomputery z pamięcią współdzieloną.
- Klasyczne serwery wieloprocesorowe.

39.2.3 Zalety

- Szybka komunikacja między procesorami.
- Wysoka wydajność w aplikacjach wymagających częstej wymiany danych.
- Jednolity dostęp do zasobów.

39.2.4 Wady

- Ograniczona skalowalność – dodanie kolejnych procesorów prowadzi do problemów z dostępem do pamięci.
- Możliwe wąskie gardła komunikacyjne.
- Wysoki koszt implementacji.

39.3 Systemy luźno powiązane (Loosely Coupled Systems)

39.3.1 Charakterystyka

- Każdy węzeł posiada własną pamięć lokalną i zasoby obliczeniowe.
- Komunikacja odbywa się przez sieć komputerową (np. Ethernet, InfiniBand).
- Może obejmować systemy działające pod kontrolą różnych systemów operacyjnych.

39.3.2 Przykłady

- Klastry obliczeniowe (np. Beowulf Cluster).
- Systemy gridowe (np. Grid Computing).
- Chmurowe systemy obliczeniowe (np. Amazon AWS, Google Cloud).

39.3.3 Zalety

- Wysoka skalowalność – można łatwo dodawać kolejne węzły.
- Lepsza odporność na awarie – pojedynczy węzeł może się wyłączyć bez zatrzymania całego systemu.
- Możliwość wykorzystania heterogenicznych zasobów (różne procesory, systemy operacyjne).

39.3.4 Wady

- Wyższe opóźnienia komunikacyjne w porównaniu do systemów ściśle powiązanych.
- Konieczność zarządzania rozproszonymi zasobami.
- Problemy z równomiernym podziałem obciążeń.

39.4 Porównanie systemów ściśle i luźno powiązanych

- W tabeli 10

39.5 Podsumowanie

- Systemy ściśle powiązane oferują wysoką wydajność i niskie opóźnienia, ale mają ograniczoną skalowalność.
- Systemy luźno powiązane są bardziej elastyczne i skalowalne, ale wymagają efektywnych mechanizmów komunikacji i synchronizacji.
- Wybór odpowiedniego modelu zależy od specyfiki aplikacji – systemy ściśle powiązane są preferowane w obliczeniach naukowych, a systemy luźno powiązane w aplikacjach chmurowych i rozproszonych.

Cecha	**Systemy ściśle powiązane**	**Systemy luźno powiązane**
Pamięć	Współdzielona	Każdy węzeł ma własną
Komunikacja	Szybka, poprzez magistralę	Wolniejsza, przez sieć
Skalowalność	Ograniczona	Wysoka
Odporność na awarie	Niska – awaria procesora wpływa na system	Wysoka – awaria pojedynczego węzła nie zatrzymuje systemu
Koszt	Wysoki	Niższy
Przykłady	Superkomputery, serwery SMP	Klastry, chmura, systemy gridowe

Tabela 10: Porównanie systemów ściśle i luźno powiązanych

40 Pojęcie skalowalności w przetwarzaniu rozproszonym

40.1 Wprowadzenie

Skalowalność to zdolność systemu do utrzymania lub zwiększenia wydajności wraz ze wzrostem liczby zasobów lub obciążenia. W kontekście przetwarzania rozproszonego oznacza to możliwość dodawania nowych węzłów do systemu bez znaczącego pogorszenia jego efektywności.

40.2 Rodzaje skalowalności

40.2.1 1. Skalowalność wertykalna (pionowa, *vertical scaling*)

Polega na zwiększaniu mocy pojedynczego węzła poprzez dodanie lepszego sprzętu (więcej pamięci RAM, szybszy procesor, wydajniejszy dysk).

Cechy:

- Prostota – zmiana konfiguracji pojedynczego serwera.
- Ograniczona skalowalność – istnieje granica sprzętowa dla jednego węzła.
- Wysokie koszty – im mocniejszy sprzęt, tym większy koszt jednostkowy.

Zastosowanie:

- Bazy danych o wysokich wymaganiach sprzętowych.
- Systemy wymagające niskich opóźnień i wysokiej wydajności na jednym węźle.

40.2.2 2. Skalowalność horyzontalna (pozioma, *horizontal scaling*)

Polega na dodawaniu nowych węzłów do systemu w celu rozłożenia obciążenia.

Cechy:

- Dobra skalowalność – system może obsługiwać większą liczbę użytkowników przez dodanie kolejnych serwerów.
- Wymaga zarządzania rozproszonymi zasobami.
- Zwykle tańsza w długoterminowej perspektywie niż skalowanie pionowe.

Zastosowanie:

- Chmura obliczeniowa (np. AWS, Google Cloud).
- Systemy Big Data (np. Hadoop, Apache Spark).
- Aplikacje webowe o wysokim ruchu (np. serwery CDN, load balancing).

40.2.3 3. Skalowalność funkcjonalna

Odnosi się do możliwości dodawania nowych funkcji do systemu bez wpływu na jego stabilność i wydajność.

Przykłady:

- Mikroserwisy – dodawanie nowych komponentów bez przerywania działania innych usług.
- Architektura modułowa w aplikacjach webowych.

40.2.4 4. Skalowalność geograficzna

Dotyczy systemów rozproszonych działających w różnych lokalizacjach.

Zastosowanie:

- Sieci CDN do dostarczania treści w różnych regionach świata.
- Rozproszone bazy danych (np. Google Spanner).

40.3 Metody poprawy skalowalności

40.3.1 1. Load Balancing (Równoważenie obciążenia)

Polega na dynamicznym rozkładaniu ruchu pomiędzy wiele serwerów.

Techniki:

- Round Robin – kolejność przydzielania zapytań do serwerów.
- Least Connections – kierowanie zapytań do najmniej obciążonego serwera.
- IP Hash – przypisanie użytkownika do konkretnego serwera na podstawie adresu IP.

40.3.2 2. Sharding (Podział danych)

Technika polegająca na podziale bazy danych na mniejsze fragmenty (*shardy*), które są przechowywane na różnych serwerach.

Przykład:

- Podział użytkowników serwisu społecznościowego według regionów.

40.3.3 3. Caching (Buforowanie)

Przechowywanie często używanych danych w szybkiej pamięci w celu zmniejszenia obciążenia głównych zasobów.

Przykłady:

- Redis – buforowanie zapytań do bazy danych.
- Content Delivery Networks (CDN) – przechowywanie treści stron internetowych w wielu lokalizacjach.

40.3.4 4. Asynchroniczna komunikacja

Umożliwia efektywniejszą wymianę danych między komponentami systemu.

Przykłady:

- Kolejki wiadomości (RabbitMQ, Apache Kafka).
- Mechanizmy Publish-Subscribe (Pub/Sub).

40.4 Porównanie skalowalności pionowej i poziomej

Cecha	**Skalowalność pionowa**	**Skalowalność pozioma**
Metoda	Zwiększanie mocy jednego węzła	Dodawanie nowych węzłów
Koszt	Wysoki przy dużej rozbudowie	Można skalować stopniowo
Skalowalność	Ograniczona przez sprzęt	Praktycznie nieograniczona
Awaryjność	Awaria jednego węzła zatrzymuje system	Awaria pojedynczego węzła nie wpływa na całość
Zastosowanie	Systemy bazodanowe, aplikacje monolityczne	Chmura, mikroserwisy, aplikacje webowe

Tabela 11: Porównanie skalowalności pionowej i poziomej

40.5 Podsumowanie

- Skalowalność to zdolność systemu do utrzymania wydajności przy wzroście obciążenia.
- Skalowalność pionowa polega na ulepszaniu pojedynczych węzłów, podczas gdy pozioma na dodawaniu kolejnych jednostek.
- Mechanizmy takie jak load balancing, sharding, caching i asynchroniczna komunikacja pozwalają na efektywne zarządzanie rosnącym obciążeniem.
- Wybór metody skalowania zależy od specyfiki systemu – aplikacje chmurowe preferują skalowanie poziome, a aplikacje wymagające dużej mocy obliczeniowej mogą korzystać ze skalowania pionowego.