1. Omówić podział podział architektur komputerów według TAKSONOMI FLYNNA

- **SISD (Single Instruction stream, Single Data stream):**
    - Architektura sekwencyjna, gdzie pojedynczy procesor wykonuje jedną instrukcję na jednym zestawie danych w danym czasie.
- **SIMD (Single Instruction stream, Multiple Data streams):**
    - Architektura, w której jedna instrukcja jest wykonywana na wielu zestawach danych jednocześnie, przy użyciu wielu procesorów.
- **MISD (Multiple Instruction streams, Single Data stream):**
    - Rzadko spotykana architektura, w której wiele instrukcji operuje na jednym zestawie danych. Trudna do praktycznego zastosowania.
- **MIMD (Multiple Instruction streams, Multiple Data streams):**
    - Najbardziej powszechna architektura, w której wiele instrukcji jest wykonywanych na wielu zestawach danych jednocześnie, przy użyciu niezależnych procesorów.

W skrócie, SISD to pojedynczy procesor wykonujący jedną instrukcję na jednym zestawie danych, SIMD to wiele procesorów wykonujących tę samą instrukcję na wielu zestawach danych, MISD to rzadka architektura z wieloma instrukcjami operującymi na jednym zestawie danych, a MIMD to najbardziej powszechna architektura z wieloma procesorami i instrukcjami działającymi na wielu zestawach danych jednocześnie.

2. Omówić pojęcia - program sekwencyjny, równoległy oraz współbieżny

- **Program sekwencyjny:**
    - Program sekwencyjny to taki, który wykonuje się krok po kroku, wykonywanie kolejnych instrukcji następuje w jednym, ustalonym porządku.
    - Procesor wykonuje pojedynczą instrukcję na raz, a wykonanie jednej instrukcji musi zakończyć się przed rozpoczęciem wykonania kolejnej.
    - Jest to klasyczna forma programowania, gdzie instrukcje wykonywane są sekwencyjnie, co oznacza, że jedno zdarzenie następuje po drugim.
- **Program równoległy:**
    - Program równoległy to taki, w którym kilka instrukcji lub zestawów instrukcji jest wykonywanych jednocześnie przez różne jednostki przetwarzania.
    - W architekturze równoległej wiele procesorów lub rdzeni wykonuje różne części programu jednocześnie, co przyspiesza cały proces przetwarzania.
    - Przykładem może być równoczesne przetwarzanie danych w macierzowych operacjach lub jednoczesne wykonywanie niezależnych zadań.
- **Program współbieżny:**
    - Program współbieżny to taki, w którym różne części programu wykonują się równocześnie, ale niekoniecznie są one niezależne od siebie.

- Współbieżność dotyczy zarządzania jednoczesnym wykonywaniem wielu zadań, które mogą być powiązane ze sobą.
- Współbieżność może być realizowana na różne sposoby, takie jak korzystanie z wielu wątków w jednym programie, co pozwala na jednoczesne wykonywanie różnych operacji.

Podsumowując, program sekwencyjny wykonuje instrukcje po kolei, program równoległy wykorzystuje jednoczesne wykonanie niezależnych instrukcji, a program współbieżny zajmuje się jednoczesnym wykonywaniem zadań, które mogą być powiązane. Współbieżność i równoległość są kluczowymi koncepcjami w dziedzinie programowania równoległego i współbieżnego, które pozwalają efektywnie wykorzystać dostępne zasoby obliczeniowe.

3. Omówić różnice pomiędzy procesem a wątkiem

Procesy i wątki są dwiema fundamentalnymi koncepcjami w programowaniu wielozadaniowym, ale różnią się pod wieloma względami:

- **Definicja:**
  - **Proces:** Proces jest jednostką wykonywania programu, posiada własną przestrzeń adresową, deskryptory plików, obszar pamięci oraz zasoby systemowe. Procesy są zazwyczaj niezależne i izolowane od siebie.
  - **Wątek:** Wątek to lżejsza jednostka wykonywania niż proces, która działa w obrębie tego samego procesu. Wątki dzielą przestrzeń adresową i zasoby procesu, co ułatwia komunikację między nimi.
- **Izolacja vs. Współdzielenie:**
  - **Proces:** Procesy są izolowane, co oznacza, że każdy proces ma swoją własną przestrzeń pamięci i zasoby. Komunikacja między procesami wymaga mechanizmów takich jak inter-process communication (IPC).
  - **Wątek:** Wątki działają w ramach tego samego procesu i dzielą wspólną przestrzeń adresową oraz zasoby. Współdzielenie pamięci między wątkami ułatwia komunikację, ale może też prowadzić do problemów synchronizacyjnych.
- **Koszt:**
  - **Proces:** Tworzenie i zarządzanie procesami jest bardziej kosztowne w porównaniu do wątków ze względu na większe wymagania zasobowe.
  - **Wątek:** Tworzenie i zarządzanie wątkami jest zazwyczaj tańsze, ponieważ współdzielą one zasoby z procesem macierzystym.
- **Bezpieczeństwo:**
  - **Proces:** Błędy w jednym procesie zazwyczaj nie wpływają na inne procesy, co oznacza, że procesy są bardziej odporne na awarie.
  - **Wątek:** Błąd w jednym wątku może wpłynąć na cały proces, ponieważ współdzielą one przestrzeń adresową. To może prowadzić do trudniejszej analizy i obsługi błędów.
- **Synchronizacja:**
  - **Proces:** Synchronizacja między procesami wymaga specjalnych mechanizmów, takich jak semafory, komunikaty czy potoki.

- **Wątek:** Wątki współdzielą przestrzeń pamięci, co ułatwia synchronizację, ale może prowadzić do problemów związanych z dostępem do wspólnych danych (konieczność zastosowania mechanizmów synchronizacyjnych).

Podsumowując, procesy są bardziej niezależne, izolowane i bezpieczne, ale wymagają więcej zasobów, natomiast wątki są bardziej elastyczne, tańsze, ale wymagają ostrożności w synchronizacji i dzieleniu zasobów. Wybór między procesami a wątkami zależy od wymagań danego zadania oraz charakterystyki aplikacji.

4. Omówić poprawność programów równoległych (do omówienia zakleszczenie, poprawność częściowa, zagłodzenie)

- **Zakleszczenie (Deadlock):**
    - Zakleszczenie to sytuacja, w której dwa lub więcej wątków lub procesów są zablokowane, ponieważ każdy z nich oczekuje na zasób, który jest już używany przez inne zadanie zablokowane w tej samej sytuacji.
    - Aby uniknąć zakleszczenia, należy korzystać z mechanizmów synchronizacyjnych, takich jak blokady, z umiarem i zawsze w tej samej kolejności, aby unikać sytuacji, w której jeden wątek czeka na zasób, który jest już zajęty przez inny wątek, który z kolei czeka na zasób będący w posiadaniu pierwszego wątku.
- **Poprawność Częściowa:**
    - Poprawność częściowa dotyczy tego, czy program równoległy nadal działa zgodnie z założeniami, nawet jeśli nie zachodzi równoległe wykonywanie wszystkich instrukcji.
    - W wielu przypadkach poprawność częściowa jest akceptowalna, o ile program osiąga swoje cele, nawet jeśli nie wszystkie operacje są wykonywane równocześnie. W niektórych sytuacjach można akceptować pewien stopień synchronizacji i sekwencyjności w celu uniknięcia trudności związanych z programowaniem równoległym.
- **Zagłodzenie (Starvation):**
    - Zagłodzenie występuje, gdy pewne wątki lub procesy nie otrzymują wystarczającej ilości zasobów lub czasu procesora, aby zakończyć swoje zadania.
    - Może wystąpić w przypadku niewłaściwej implementacji mechanizmów zarządzania zasobami lub przy niewłaściwym planowaniu dostępu do zasobów.
    - Aby uniknąć zagłodzenia, ważne jest sprawiedliwe przydzielanie zasobów i czasu procesora wszystkim wątkom lub procesom, a także skuteczne zarządzanie kolejkami zadań oczekujących na dostęp do zasobów.

5. Opisać problem wzajemnego wykluczenia

- **Definicja:** Wzajemne wykluczenie to problem występujący w programowaniu równoległym, gdy dwa lub więcej wątków lub procesów jednocześnie próbują uzyskać dostęp do wspólnego zasobu (takiego jak zmienna lub obszar pamięci), co może prowadzić do nieprzewidywalnych rezultatów.
- **Cel:** Problem wzajemnego wykluczenia pojawia się z powodu potrzeby ochrony wspólnych zasobów przed jednoczesnym dostępem wielu wątków, co mogłoby prowadzić do niekonsystentnych lub błędnych wyników.

- **Przykład:** Załóżmy, że dwie funkcje równocześnie chcą modyfikować tę samą zmienną globalną. Bez odpowiednich mechanizmów synchronizacji, obie funkcje mogą odczytać obecną wartość zmiennej, a następnie jednocześnie ją zmienić, co może prowadzić do utraty danych lub innych problemów.
- **Mechanizmy rozwiązujące:**
    - **Bloki krytyczne:** Są obszarami kodu, w których dostęp do wspólnego zasobu jest kontrolowany przez mechanizmy synchronizacyjne, takie jak blokady.
    - **Semafory:** Mechanizmy synchronizacyjne pozwalające kontrolować dostęp wielu wątków do wspólnego zasobu. Semafory umożliwiają ograniczenie dostępu do jednego wątku na raz.
    - **Zmienne warunkowe:** Wykorzystywane do powiadamiania innych wątków o zmianach w stanie wspólnego zasobu, co pozwala na skuteczniejsze zarządzanie dostępem.
- **Ryzyko:** Bez właściwego rozwiązania problemu wzajemnego wykluczenia może wystąpić tzw. "wyścig", gdzie wiele wątków jednocześnie próbuje uzyskać dostęp do zasobu, co prowadzi do nieprzewidywalnych i potencjalnie błędnych wyników.
- **Konsekwencje:** Błędy wynikające z braku wzajemnego wykluczenia mogą prowadzić do trudnych do zdiagnozowania problemów, utraty danych, czy też niestabilnego działania programu równoległego.

6. Wyjaśnić co to jest sekcja krytyczna i po co ją stosujemy

- **Sekcja krytyczna:** Sekcja krytyczna to fragment kodu, w którym tylko jeden wątek może wykonywać operacje na wspólnych zasobach w danym czasie. Jest to stosowany mechanizm w programowaniu równoległym, mający na celu uniknięcie błędów wynikających z równoczesnego dostępu wielu wątków do tych samych danych.
- **Cel stosowania:** Sekcje krytyczne mają na celu ochronę wspólnych zasobów przed równoczesnym dostępem, co może prowadzić do błędów i utraty danych. Zapobiegają wyścigom o zasoby i utrzymują konsystencję danych.
- **Implementacja:** Mogą być realizowane za pomocą bloków krytycznych, gdzie dostęp do sekcji krytycznej jest kontrolowany przez mechanizmy synchronizacyjne.
- **Znaczenie:** Sekcje krytyczne są kluczowe dla poprawności programów równoległych, zapewniając bezpieczny dostęp do wspólnych zasobów i utrzymanie konsystencji danych.

Sekcje krytyczne są nieodzownym elementem programowania równoległego, a ich właściwe zastosowanie jest kluczowe dla zapewnienia bezpiecznego i spójnego dostępu do wspólnych zasobów w środowisku wielowątkowym czy wieloprocesowym.

7. Opisać problem producentów-konsumentów

- **Problem Producentów-Konsumentów:** Problem Producentów-Konsumentów to klasyczny problem synchronizacyjny, występujący w programowaniu współbieżnym. Polega na skoordynowanym dostępie producentów, którzy produkują dane, i konsumentów, którzy konsumują te dane, do współdzielonego bufora lub magazynu.
- **Scenariusz:**
    1. Producent produkuje dane i umieszcza je w buforze.

2. Konsument pobiera dane z bufora i je konsumuje.
3. Procesy producentów i konsumentów są współbieżne, a bufor musi być zarządzany w taki sposób, aby uniknąć problemów, takich jak przekroczenie bufora (buffer overflow) lub pobranie danych z pustego bufora.

- **Wyzwania:**

    1. **Bezpieczny dostęp do bufora:** Aby uniknąć błędów związanych z równoczesnym dostępem, producenci i konsumenci muszą synchronizować swoje działania przy korzystaniu z bufora.

    2. **Zakleszczenie (Deadlock) i zagłodzenie (Starvation):** Niewłaściwa implementacja mechanizmów synchronizacyjnych może prowadzić do zakleszczeń, gdy procesy wzajemnie się blokują, oraz zagłodzenia, gdy jedne procesy są systematycznie pomijane w dostępie do bufora.

- **Mechanizmy rozwiązujące:**

    1. **Semafory:** Mechanizm synchronizacyjny, który kontroluje dostęp do wspólnego zasobu. Semafory mogą być używane do określenia liczby dostępnych miejsc w buforze.

    2. **Bloki krytyczne:** Wykorzystanie bloków krytycznych w celu bezpiecznego dostępu do współdzielonych zasobów, takich jak bufor.

- **Implementacja:** Przykładowa implementacja problemu Producentów-Konsumentów w języku Java przy użyciu semaforów:

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

#define BUFFER_SIZE 5

sem_t empty, full, mutex;
int buffer[BUFFER_SIZE];
int counter = 0;

void *producer(void *arg) {
    int item = 1;

    while (1) {
        sem_wait(&empty);
        sem_wait(&mutex);

        buffer[counter] = item;
        printf("Producent dodaje element %d do bufora\n", item);
        counter++;

        sem_post(&mutex);
        sem_post(&full);

        item++;
        sleep(1);  // Symulacja czasu produkcji
    }
```

```
    pthread_exit(NULL);
}

void *consumer(void *arg) {
    int item;

    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);

        item = buffer[counter - 1];
        printf("Konsument pobiera element %d z bufora\n", item);
        counter--;

        sem_post(&mutex);
        sem_post(&empty);

        sleep(2);  // Symulacja czasu konsumpcji
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t producer_thread, consumer_thread;

    // Inicjalizacja semaforów
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    // Utworzenie wątków producenta i konsumenta
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Oczekiwanie na zakończenie wątków
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Zniszczenie semaforów
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

- **Zastosowanie:** Problem Producentów-Konsumentów ma zastosowanie w sytuacjach, gdzie jedne procesy generują dane, a inne je konsumują, na przykład w systemach przetwarzania wsadowego, kolejkach zadań czy systemach komunikacji międzyprocesowej.

Rozwiązanie tego problemu wymaga skoordynowanego dostępu do współdzielonych zasobów oraz unikania zakleszczeń i zagłodzenia, co sprawia, że jest on popularnym tematem w kontekście programowania współbieżnego i równoległego.

8. Opisać problem czytelników i pisarzy

Problem Czytelników i Pisarzy to klasyczny problem synchronizacyjny, który występuje w kontekście współbieżnego dostępu do współdzielonych danych. W tym problemie mamy dwie grupy zadań: czytelników i pisarzy, którzy konkurują o dostęp do pewnego zasobu, na przykład do wspólnego obszaru pamięci.

- **Czytelnicy:**
  - Czytelnicy odczytują dane z zasobu.
  - Kilku czytelników może jednocześnie odczytywać dane.
  - Odczyt nie modyfikuje danych.
- **Pisarze:**
  - Pisarze modyfikują dane na zasobie.
  - Tylko jeden pisarz może jednocześnie modyfikować dane.
  - Pisarze blokują dostęp czytelników i innych pisarzy.
- **Problem:**
  - Czytelnicy nie powinni zakłócać się nawzajem podczas odczytu.
  - Pisarz nie powinien być zakłócany podczas modyfikowania danych.
  - Wszystkie operacje powinny być bezpieczne i chronione przed równoczesnym dostępem wielu czytelników lub pisarzy.
- **Wyzwania:**
  - Unikanie zagłodzenia pisarzy i czytelników, czyli zapewnienie, że każda grupa będzie miała szansę na dostęp.
  - Optymalizacja dostępu do zasobu, aby zminimalizować czas blokady.
  - Zabezpieczenie przed sytuacją, w której czytelnicy ciągle czytają, a pisarze nie mają szansy na modyfikację danych.
- **Rozwiązania:**
  - Używanie semaforów lub mutexów do kontrolowania dostępu do współdzielonych danych.
  - Zastosowanie liczników czytelników i pisarzy, aby śledzić ilość aktualnie korzystających zasobem.

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

sem_t mutex, wrt;
int read_count = 0;

void *reader(void *arg) {
    while (1) {
```

```
        sem_wait(&mutex);
        read_count++;
        if (read_count == 1) {
            sem_wait(&wrt);
        }
        sem_post(&mutex);

        // Odczyt danych

        sem_wait(&mutex);
        read_count--;
        if (read_count == 0) {
            sem_post(&wrt);
        }
        sem_post(&mutex);

        // Czas oczekiwania przed następnym odczytem
        sleep(1);
    }
}

void *writer(void *arg) {
    while (1) {
        sem_wait(&wrt);

        // Zapis danych

        sem_post(&wrt);

        // Czas oczekiwania przed następnym zapisem
        sleep(2);
    }
}

int main() {
    pthread_t reader_thread, writer_thread;

    // Inicjalizacja semaforów
    sem_init(&mutex, 0, 1);
    sem_init(&wrt, 0, 1);

    // Utworzenie wątków czytelników i pisarzy
    pthread_create(&reader_thread, NULL, reader, NULL);
    pthread_create(&writer_thread, NULL, writer, NULL);

    // Oczekiwanie na zakończenie wątków
    pthread_join(reader_thread, NULL);
    pthread_join(writer_thread, NULL);

    // Zniszczenie semaforów
    sem_destroy(&mutex);
    sem_destroy(&wrt);
```

```
    return 0;
}
```

Ten przykład ilustruje synchronizację dostępu do współdzielonych danych między czytelnikami a pisarzami za pomocą semaforów. Czytelnicy zwiększają licznik przed odczytem, a pisarze uzyskują dostęp tylko wtedy, gdy nie ma czytelników. To rozwiązanie zapewnia, że pisarze nie będą blokowani przez czytelników i vice versa.

10. Opisać problem ucztujących filozofów

Problem ucztujących filozofów to klasyczny problem synchronizacyjny, który ilustruje wyzwania związane z równoczesnym dostępem wielu wątków do współdzielonych zasobów. Problem ten został zaproponowany przez Edsgera Dijkstrę w 1965 roku i jest często używany do analizy i testowania mechanizmów synchronizacji w systemach operacyjnych i programowaniu równoległym.

- **Opis problemu:**
  - Pięciu filozofów siedzi wokół okrągłego stołu.
  - Każdy filozof ma przed sobą talerz z makaronem.
  - Między każdymi dwoma filozofami jest widelec.
  - Filozofowie mogą być w jednym z dwóch stanów: myślenia lub jedzenia.
  - Aby zjeść, filozof musi trzymać oba widelce po swojej lewej i prawej stronie.
  - Problem polega na takim zsynchronizowaniu działań filozofów, aby uniknąć zakleszczeń, zagłodzenia i zapewnić sprawiedliwy dostęp do widelców.
- **Wyzwania:**
  1. Unikanie zakleszczeń: Filozofowie nie powinni utknąć w sytuacji, gdzie każdy z nich trzyma jeden widelec i czeka na drugi.
  2. Unikanie zagłodzenia: Każdy filozof powinien mieć okazję do jedzenia, nawet jeśli inni filozofowie korzystają z widelców.
- **Rozwiązanie:**
  - Implementacje problemu ucztujących filozofów mogą korzystać z różnych mechanizmów synchronizacyjnych, takich jak semafory czy mutexy.
  - Jednym z popularnych rozwiązań jest użycie semaforów do zarządzania dostępem do widelców, zapewniając, że każdy filozof ma dostęp do obu widelców tylko wtedy, gdy są oba dostępne.
- **Przykładowa implementacja w języku C:**

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

#define N 5

sem_t forks[N];
```

```c
void *philosopher(void *arg) {
    int id = *(int *)arg;
    int left_fork = id;
    int right_fork = (id + 1) % N;

    while (1) {
        // Filozof czeka na oba widelce
        sem_wait(&forks[left_fork]);
        sem_wait(&forks[right_fork]);

        // Filozof je
        printf("Philosopher %d is eating\n", id);
        usleep(1000000); // Symulacja jedzenia

        // Filozof odkłada widelce
        sem_post(&forks[left_fork]);
        sem_post(&forks[right_fork]);

        // Filozof myśli
        printf("Philosopher %d is thinking\n", id);
        usleep(1000000); // Symulacja myślenia
    }
}

int main() {
    pthread_t philosophers[N];
    int phil_ids[N];

    // Inicjalizacja semaforów
    for (int i = 0; i < N; i++) {
        sem_init(&forks[i], 0, 1);
    }

    // Utworzenie wątków filozofów
    for (int i = 0; i < N; i++) {
        phil_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &phil_ids[i]);
    }

    // Oczekiwanie na zakończenie wątków
    for (int i = 0; i < N; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Zniszczenie semaforów
    for (int i = 0; i < N; i++) {
        sem_destroy(&forks[i]);
    }

    return 0;
}
```

W tym przykładzie, każdy filozof jest reprezentowany przez osobny wątek. Semafor `forks` jest używany do zarządzania dostępem do widelców. Filozofowie czekają na oba widelce, jedzą, a następnie odkładają widelce, co zapewnia, że każdy z nich może korzystać z widelców bezpiecznie. Symulowane jest jedzenie i myślenie przez opóźnienia czasowe.

11. Opisać modele współbieżności (scentralizowane, rozproszone)

Modele współbieżności odnoszą się do różnych podejść do organizacji i zarządzania równoczesnością w systemach komputerowych. Dwa główne modele to modele scentralizowane i rozproszone.

Model Współbieżności Scentralizowanej:

1. **Definicja:**
    - W modelu scentralizowanym istnieje jednostkowy punkt kontroli, zwany centrum kontroli równoczesności (ang. Concurrency Control Center).
    - To centrum decyduje, które zadania mogą być wykonywane równocześnie, a które powinny oczekiwać na dostęp do zasobów.
2. **Charakterystyka:**
    - Zasoby i zarządzanie równoczesnością są kontrolowane w jednym centralnym punkcie.
    - Cały proces podejmowania decyzji dotyczących równoczesności odbywa się w jednym miejscu.
3. **Zalety:**
    - Proste w implementacji i zrozumiałe.
    - Łatwe do utrzymania, gdy mamy jedno miejsce zarządzające równoczesnością.
4. **Wady:**
    - Punkt centralny może stać się wąskim gardłem i ograniczyć wydajność.
    - Ryzyko, że awaria centrum równoczesności może spowodować zatrzymanie całego systemu.

Model Współbieżności Rozproszonej:

1. **Definicja:**
    - W modelu rozproszonym zarządzanie równoczesnością jest rozproszone na różne jednostki kontrolne.
    - Każda jednostka kontrolna, na przykład węzeł w systemie rozproszonym, ma pewną autonomię w zarządzaniu równoczesnością.
2. **Charakterystyka:**
    - Zasoby są rozproszone, a zarządzanie równoczesnością odbywa się w rozproszony sposób.
    - Każda jednostka kontrolna podejmuje decyzje lokalnie.
3. **Zalety:**
    - Potencjalnie lepsza skalowalność, ponieważ zadania mogą być obsługiwane niezależnie przez różne jednostki kontrolne.

- Odporność na awarie, ponieważ awaria jednej jednostki kontrolnej nie zatrzymuje całego systemu.

4. **Wady:**
    - Skomplikowane zarządzanie równoczesnością, gdy wiele jednostek kontrolnych musi współpracować.
    - Możliwość konfliktów i trudniejsze do zdiagnozowania problemy równoczesności.

Podsumowanie:

- Model scentralizowany jest prostszy, ale może być mniej wydajny i bardziej podatny na awarie.
- Model rozproszony jest bardziej elastyczny, ale wymaga bardziej zaawansowanych mechanizmów zarządzania równoczesnością i może prowadzić do bardziej skomplikowanych problemów.

W praktyce często spotykamy się z hybrydowymi modelami, które łączą elementy zarówno modelu scentralizowanego, jak i rozproszonego, aby uzyskać optymalne rozwiązanie dla danego systemu.

12. Opisać różnice pomiędzy komunikacją synchroniczną a komunikacją asynchroniczną

Komunikacja synchroniczna i asynchroniczna to dwa główne podejścia do przekazywania informacji między procesami, wątkami lub systemami. Poniżej przedstawiam różnice pomiędzy komunikacją synchroniczną a komunikacją asynchroniczną:

Komunikacja Synchroniczna:

1. **Czas oczekiwania:**
    - **Synchroniczna:** Wymaga, aby wysyłający i odbierający procesy, wątki lub systemy były zsynchronizowane pod względem czasu. Wysyłający proces musi oczekiwać na odpowiedź od odbierającego przed kontynuacją działania.
    - **Przykład:** Wywołania funkcji w programowaniu synchronicznym, gdzie program oczekuje na zakończenie funkcji przed przejściem do następnego kroku.

2. **Prostota i Przejrzystość:**
    - **Synchroniczna:** Bardziej przejrzysta i prosta w zrozumieniu, ponieważ kolejność komunikatów odpowiada kolejności zdarzeń.

3. **Zakleszczenia:**
    - **Synchroniczna:** Może prowadzić do potencjalnych zakleszczeń, gdy jednostka czeka na odpowiedź, a jednocześnie blokuje zasoby, które inne jednostki mogą potrzebować.

Komunikacja Asynchroniczna:

1. **Czas oczekiwania:**
    - **Asynchroniczna:** Nie wymaga, aby jednostki były zsynchronizowane pod względem czasu. Wysyłający może kontynuować działanie bez oczekiwania na odpowiedź od odbierającego.
    - **Przykład:** Wysyłanie asynchronicznych żądań HTTP, gdzie program może kontynuować pracę, podczas gdy czeka na odpowiedź od zdalnego serwera.

2. **Prostota i Przejrzystość:**
    - **Asynchroniczna:** Może być bardziej złożona w zrozumieniu, ponieważ kolejność zdarzeń niekoniecznie odpowiada kolejności komunikatów.
3. **Efektywność:**
    - **Asynchroniczna:** Może być bardziej efektywna, ponieważ jednostki mogą pracować niezależnie, co zwiększa wydajność systemu.
4. **Zakleszczenia:**
    - **Asynchroniczna:** Mniej podatna na zakleszczenia, ponieważ jednostki mogą pracować niezależnie od siebie, a brak synchronizacji czasowej zmniejsza ryzyko blokowania zasobów.

Wybór Pomiędzy Synchroniczną a Asynchroniczną:

- **Synchroniczna:** Często stosowana w przypadku prostych zadań, gdzie ważna jest kolejność wykonywania operacji.
- **Asynchroniczna:** Bardziej skomplikowana, ale często stosowana w przypadku zadań, które można równolegle przetwarzać, co zwiększa wydajność systemu.

Obie formy komunikacji mają swoje zastosowania, a wybór między nimi zależy od wymagań konkretnego systemu lub aplikacji.

13. Co to jest semafor - wymienić rodzaje semaforów

W kontekście Unixowego języka C, semafory są dostępne poprzez interfejs semaforów systemu V. Oto krótkie omówienie semaforów dostępnych w Unix w kontekście C:

1. **Semafor Binarny:**
    - W Unixie można używać semaforów binarnych do synchronizacji między procesami lub wątkami.
    - Używane są funkcje takie jak `sem_init`, `sem_wait`, `sem_post` do zarządzania semaforem binarnym.
2. **Semafor Ogólny (Zliczający):**
    - Semafor ogólny, inaczej nazywany zliczającym, pozwala na przyjęcie dowolnej nieujemnej wartości całkowitej.
    - Używane są te same funkcje, co w przypadku semafora binarnego, ale pozwalają na bardziej elastyczne zarządzanie dostępem.
3. **Semafor Uogólniony:**
    - Semafor uogólniony (zliczający) jest semaforem, który może przyjmować wartości ujemne.
    - Dostęp do semafora i jego operacje oparte są na wartości semafora i mogą obejmować operacje na wartościach ujemnych.
4. **Semafor Dwustronnie Ograniczony:**

- Semafory dwustronnie ograniczone pozwalają na bardziej złożone operacje, takie jak oczekiwanie na sygnał od innego semafora przed wykonaniem operacji na danym semaforze.
- Umożliwiają bardziej zaawansowane zarządzanie dostępem do zasobów.

Przykład użycia semafora binarnego w języku C w systemie Unix:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>

int main() {
    sem_t *binary_semaphore;

    // Inicjalizacja semafora binarnego
    binary_semaphore = sem_open("/my_binary_semaphore", O_CREAT, 0644, 1);
    if (binary_semaphore == SEM_FAILED) {
        perror("sem_open");
        exit(EXIT_FAILURE);
    }

    // Przykładowe operacje na semaforze
    sem_wait(binary_semaphore);
    printf("Critical section: Process %d\n", getpid());
    sem_post(binary_semaphore);

    // Zamykanie i usuwanie semafora
    sem_close(binary_semaphore);
    sem_unlink("/my_binary_semaphore");

    return 0;
}
```

W tym przykładzie semafor binarny jest tworzony przy użyciu funkcji `sem_open`, a następnie są wykonywane operacje oczekiwania i zwalniania semafora za pomocą `sem_wait` i `sem_post`. Na koniec semafor jest zamykany i usuwany. Oczywiście, w prawdziwej aplikacji, trzeba byłoby dokładniej obsłużyć błędy i zajęcia bieżącego zasobu.

14. Zapisać definicję ogólną oraz praktyczną semafora ogólnego

Definicja Ogólna Semafora Ogólnego:

Semafor ogólny to mechanizm synchronizacji używany do kontrolowania dostępu do współdzielonych zasobów w środowisku wielowątkowym lub wieloprocesowym. Semafor ten może przyjmować dowolne nieujemne wartości całkowite, co umożliwia elastyczne zarządzanie dostępem do zasobów w zależności od aktualnej liczby dostępnych jednostek.

Definicja Praktyczna Semafora Ogólnego (w języku C w kontekście Unix):

W praktyce, w języku C na platformie Unix, semafor ogólny jest dostępny poprzez interfejs semaforów systemu V. Oto praktyczne definicje semafora ogólnego:

1. **Inicjalizacja Semafora Ogólnego:**

```
#include <semaphore.h>
#include <fcntl.h>

sem_t *general_semaphore;

// Inicjalizacja semafora ogólnego
general_semaphore = sem_open("/my_general_semaphore", O_CREAT, 0644,
initial_value);
if (general_semaphore == SEM_FAILED) {
    perror("sem_open");
    exit(EXIT_FAILURE);
}
```

   Semafor ogólny jest inicjalizowany przy użyciu funkcji `sem_open`. Parametr `initial_value` określa początkową wartość semafora.

2. **Operacje na Semaforze Ogólnym:**

```
// Operacja oczekiwania na semaforze
sem_wait(general_semaphore);

// Operacja zwalniania semafora
sem_post(general_semaphore);
```

   Standardowe operacje `sem_wait` i `sem_post` używane są do kontrolowania dostępu do zasobów, a semafor jest aktualizowany zgodnie z tymi operacjami.

3. **Zamykanie i Usuwanie Semafora Ogólnego:**

```
// Zamykanie semafora
sem_close(general_semaphore);

// Usuwanie semafora
sem_unlink("/my_general_semaphore");
```

   Po zakończeniu pracy z semaforem, konieczne jest jego zamknięcie przy użyciu `sem_close`. Jeśli semafor nie jest już potrzebny, można go usunąć z systemu przy użyciu `sem_unlink`.

Semafor ogólny jest używany do elastycznego zarządzania dostępem do zasobów w środowisku współbieżnym, gdzie wieloprocesowość lub wielowątkowość wymaga synchronizacji dostępu do współdzielonych zasobów.

15. Zapisać definicję ogólną oraz praktyczną semafora binarnego

Definicja Ogólna Semafora Binarnego:

Semafor binarny to mechanizm synchronizacji używany do kontrolowania dostępu do współdzielonych zasobów w środowisku wielowątkowym lub wieloprocesowym. Semafor ten może przyjmować tylko dwie wartości: 0 i 1. Służy do implementacji wzajemnego wykluczania, gdzie 0 oznacza, że zasób jest zajęty, a 1, że jest dostępny.

Definicja Praktyczna Semafora Binarnego (w języku C w kontekście Unix):

W praktyce, w języku C na platformie Unix, semafor binarny jest dostępny poprzez interfejs semaforów systemu V. Oto praktyczne definicje semafora binarnego:

1. **Inicjalizacja Semafora Binarnego:**

```
#include <semaphore.h>
#include <fcntl.h>

sem_t *binary_semaphore;

// Inicjalizacja semafora binarnego
binary_semaphore = sem_open("/my_binary_semaphore", O_CREAT, 0644,
initial_value);
if (binary_semaphore == SEM_FAILED) {
    perror("sem_open");
    exit(EXIT_FAILURE);
}
```

   Semafor binarny jest inicjalizowany przy użyciu funkcji `sem_open`. Parametr `initial_value` określa początkową wartość semafora, która zazwyczaj wynosi 1.

2. **Operacje na Semaforze Binarnym:**

```
// Operacja oczekiwania na semaforze (czekanie na dostępność)
sem_wait(binary_semaphore);

// Operacja zwalniania semafora (oznaczanie dostępności)
sem_post(binary_semaphore);
```

   Standardowe operacje `sem_wait` i `sem_post` używane są do kontrolowania dostępu do zasobów. `sem_wait` blokuje wywołujący proces (lub wątek) do momentu, gdy semafor stanie się dostępny (wartość 1), a `sem_post` oznacza, że zasób jest dostępny (ustawia wartość semafora na 1).

3. **Zamykanie i Usuwanie Semafora Binarnego:**

```

```

```
// Zamykanie semafora
sem_close(binary_semaphore);

// Usuwanie semafora
sem_unlink("/my_binary_semaphore");
```

Po zakończeniu pracy z semaforem binarnym, konieczne jest jego zamknięcie przy użyciu `sem_close`. Jeśli semafor binarny nie jest już potrzebny, można go usunąć z systemu przy użyciu `sem_unlink`.

Semafor binarny jest często wykorzystywany do implementacji mechanizmu wzajemnego wykluczania, co jest istotne w synchronizacji dostępu do krytycznych sekcji w programowaniu współbieżnym.

16. Zapisać definicję ogólną oraz praktyczną semafora dwustronnie ograniczonego

Definicja Ogólna Semafora Dwustronnie Ograniczonego:

Semafor dwustronnie ograniczony to mechanizm synchronizacji używany do kontrolowania dostępu do współdzielonych zasobów w środowisku wielowątkowym lub wieloprocesowym. W odróżnieniu od semafora binarnego, semafor ten pozwala na bardziej złożone operacje synchronizacyjne, takie jak oczekiwanie na sygnał od innego semafora przed wykonaniem operacji na danym semaforze.

Definicja Praktyczna Semafora Dwustronnie Ograniczonego (w języku C w kontekście Unix):

W praktyce, w języku C na platformie Unix, semafor dwustronnie ograniczony jest dostępny poprzez interfejs semaforów systemu V. Oto praktyczne definicje semafora dwustronnie ograniczonego:

1. **Inicjalizacja Semafora Dwustronnie Ograniczonego:**

```
#include <semaphore.h>
#include <fcntl.h>

sem_t *two_way_semaphore;

// Inicjalizacja semafora dwustronnie ograniczonego
two_way_semaphore = sem_open("/my_two_way_semaphore", O_CREAT, 0644,
initial_value);
if (two_way_semaphore == SEM_FAILED) {
    perror("sem_open");
    exit(EXIT_FAILURE);
}
```

Semafor dwustronnie ograniczony jest inicjalizowany przy użyciu funkcji `sem_open`. Parametr `initial_value` określa początkową wartość semafora.

2. **Operacje na Semaforze Dwustronnie Ograniczonym:**

```
// Operacja oczekiwania na semaforze
sem_wait(two_way_semaphore);
```

```
// Wykonanie operacji (np. dostęp do zasobów)

// Operacja zwalniania semafora
sem_post(two_way_semaphore);
```

Standardowe operacje `sem_wait` i `sem_post` używane są do kontrolowania dostępu do zasobów. Semafor dwustronnie ograniczony pozwala na bardziej złożone operacje, takie jak oczekiwanie na sygnał od innego semafora przed operacją na bieżącym semaforze.

3. **Zamykanie i Usuwanie Semafora Dwustronnie Ograniczonego:**

```
// Zamykanie semafora
sem_close(two_way_semaphore);

// Usuwanie semafora
sem_unlink("/my_two_way_semaphore");
```

Po zakończeniu pracy z semaforem dwustronnie ograniczonym, konieczne jest jego zamknięcie przy użyciu `sem_close`. Jeśli semafor nie jest już potrzebny, można go usunąć z systemu przy użyciu `sem_unlink`.

Semafor dwustronnie ograniczony pozwala na bardziej zaawansowane operacje synchronizacyjne niż semafor binarny i jest stosowany w sytuacjach, gdzie konieczne jest skomplikowane zarządzanie dostępem do zasobów.

17. Zapisać problem wzajemnego wykluczenia z użyciem semaforów

Problem wzajemnego wykluczenia, znany również jako problem sekcji krytycznej, polega na tym, że kilka procesów próbuje jednocześnie korzystać z wspólnego zasobu lub obszaru pamięci, a naszym celem jest zapewnienie, aby tylko jeden proces mógł korzystać z tego zasobu w danym czasie. Semafor binarny jest często używany do rozwiązania tego problemu. Oto prosty przykład przedstawiający problem wzajemnego wykluczenia i jego rozwiązanie przy użyciu semaforów w języku C w kontekście systemu Unix:

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

#define NUM_THREADS 2

int shared_resource = 0;  // Wspólny zasób, który wymaga ochrony
sem_t mutex;              // Semafor binarny do zapewnienia wzajemnego wykluczenia

void *thread_function(void *thread_id) {
    int tid = *(int *)thread_id;

    for (int i = 0; i < 5; i++) {
```

```
        sem_wait(&mutex);  // Wejście do sekcji krytycznej

        // Sekcja krytyczna: dostęp do wspólnego zasobu
        shared_resource++;
        printf("Thread %d: Shared Resource = %d\n", tid, shared_resource);

        sem_post(&mutex);  // Wyjście z sekcji krytycznej

        // Pozostała część kodu (operacje poza sekcją krytyczną)
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    sem_init(&mutex, 0, 1);  // Inicjalizacja semafora binarnego

    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, thread_function, (void
*)&thread_ids[i]);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&mutex);  // Zniszczenie semafora

    return 0;
}
```

W tym przykładzie, `sem_init` służy do inicjalizacji semafora binarnego o nazwie `mutex` z wartością początkową 1. Każdy wątek przed wejściem do sekcji krytycznej wykonuje operację `sem_wait`, co skutkuje zablokowaniem semafora, jeśli jest równy 0 (czyli jeśli inny wątek już jest w sekcji krytycznej). Po wykonaniu operacji w sekcji krytycznej, wątek wykonuje `sem_post`, aby zwolnić semafor, pozwalając innym wątkom na wejście. W ten sposób zapewnione jest wzajemne wykluczanie, a sekcja krytyczna jest chroniona przed równoczesnym dostępem wielu wątków.

18. Zapisać problem producentów konsumentów z użyciem semaforów

Problem producentów i konsumentów to klasyczny problem synchronizacji w programowaniu współbieżnym. Polega on na współdzieleniu ograniczonego bufora przez jednocześnie działające producentów, którzy produkują dane, i konsumentów, którzy konsumują te dane. Semafor binarny i semafor licznikowy są często używane do rozwiązania tego problemu. Poniżej znajdziesz przykładową implementację problemu producentów i konsumentów w języku C przy użyciu semaforów w kontekście systemu Unix:

```
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 2

int buffer[BUFFER_SIZE];   // Wspólny bufor
int in = 0;                // Indeks do dodawania elementu
int out = 0;               // Indeks do pobierania elementu
sem_t empty, full, mutex;  // Semafory do synchronizacji

void produce(int item) {
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}

int consume() {
    int item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}

void *producer(void *producer_id) {
    int tid = *(int *)producer_id;

    for (int i = 0; i < 5; i++) {
        int item = rand() % 100;  // Przykładowy element do wyprodukowania

        sem_wait(&empty);  // Czekaj na dostępne miejsce w buforze
        sem_wait(&mutex);  // Wejście do sekcji krytycznej

        produce(item);
        printf("Producer %d produced item %d\n", tid, item);

        sem_post(&mutex);  // Wyjście z sekcji krytycznej
        sem_post(&full);   // Zasygnalizuj, że bufor jest pełny
    }

    pthread_exit(NULL);
}

void *consumer(void *consumer_id) {
    int tid = *(int *)consumer_id;

    for (int i = 0; i < 5; i++) {
        sem_wait(&full);   // Czekaj na dostępne elementy w buforze
        sem_wait(&mutex);  // Wejście do sekcji krytycznej

        int item = consume();
        printf("Consumer %d consumed item %d\n", tid, item);
```

```
        sem_post(&mutex);  // Wyjście z sekcji krytycznej
        sem_post(&empty);  // Zasygnalizuj, że jest dostępne miejsce w buforze
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t producers[NUM_PRODUCERS], consumers[NUM_CONSUMERS];
    int producer_ids[NUM_PRODUCERS], consumer_ids[NUM_CONSUMERS];

    sem_init(&empty, 0, BUFFER_SIZE);  // Inicjalizacja semafora empty
    sem_init(&full, 0, 0);             // Inicjalizacja semafora full
    sem_init(&mutex, 0, 1);            // Inicjalizacja semafora mutex

    for (int i = 0; i < NUM_PRODUCERS; i++) {
        producer_ids[i] = i;
        pthread_create(&producers[i], NULL, producer, (void *)&producer_ids[i]);
    }

    for (int i = 0; i < NUM_CONSUMERS; i++) {
        consumer_ids[i] = i;
        pthread_create(&consumers[i], NULL, consumer, (void *)&consumer_ids[i]);
    }

    for (int i = 0; i < NUM_PRODUCERS; i++) {
        pthread_join(producers[i], NULL);
    }

    for (int i = 0; i < NUM_CONSUMERS; i++) {
        pthread_join(consumers[i], NULL);
    }

    sem_destroy(&empty);  // Zniszczenie semafora empty
    sem_destroy(&full);   // Zniszczenie semafora full
    sem_destroy(&mutex);  // Zniszczenie semafora mutex

    return 0;
}
```

W tym przykładzie używane są trzy semafory: `empty` śledzi ilość dostępnych miejsc w buforze, `full` śledzi ilość dostępnych elementów w buforze, a `mutex` kontroluje dostęp do sekcji krytycznej. Producenci czekają na dostępne miejsce w buforze (`sem_wait(&empty)`) przed dodaniem nowego elementu, a konsumenci czekają na dostępne elementy w buforze (`sem_wait(&full)`) przed pobraniem elementu. Semafor `mutex` jest używany do ochrony sekcji krytycznej, w której następuje operacja na buforze.

19. Zapisać problem czytelników i pisarzy z użyciem semaforów

Problem czytelników i pisarzy to klasyczny problem synchronizacji w programowaniu współbieżnym. Polega na synchronizacji dostępu do współdzielonego zasobu, który może być jednocześnie czytany przez wielu czytelników lub zapisywany przez jednego pisarza, ale nie jednocześnie. Semafor binarny oraz semafor

licznikowy są używane do rozwiązania tego problemu. Poniżej znajdziesz przykładową implementację problemu czytelników i pisarzy w języku C z użyciem semaforów w kontekście systemu Unix:

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

#define NUM_READERS 3
#define NUM_WRITERS 2

int shared_data = 0;  // Współdzielony zasób
int readers_count = 0;  // Licznik aktywnych czytelników

sem_t mutex, write_mutex, readers_mutex;  // Semafor dla sekcji krytycznej, dla
zapisywania i dla czytelników

void *reader(void *reader_id) {
    int tid = *(int *)reader_id;

    for (int i = 0; i < 3; i++) {
        sem_wait(&readers_mutex);  // Wejście do sekcji krytycznej dla czytelników
        readers_count++;
        if (readers_count == 1) {
            sem_wait(&write_mutex);  // Zablokuj zapisywacza, jeśli to pierwszy
czytelnik
        }
        sem_post(&readers_mutex);  // Wyjście z sekcji krytycznej dla czytelników

        // Czytanie
        printf("Reader %d reads shared data: %d\n", tid, shared_data);

        sem_wait(&readers_mutex);  // Wejście do sekcji krytycznej dla czytelników
        readers_count--;
        if (readers_count == 0) {
            sem_post(&write_mutex);  // Odblokuj zapisywacza, jeśli to ostatni
czytelnik
        }
        sem_post(&readers_mutex);  // Wyjście z sekcji krytycznej dla czytelników
    }

    pthread_exit(NULL);
}

void *writer(void *writer_id) {
    int tid = *(int *)writer_id;

    for (int i = 0; i < 3; i++) {
        sem_wait(&write_mutex);  // Wejście do sekcji krytycznej dla zapisywaczy

        // Zapisywanie
        shared_data++;
```

```
        printf("Writer %d writes shared data: %d\n", tid, shared_data);

        sem_post(&write_mutex);  // Wyjście z sekcji krytycznej dla zapisywaczy
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t readers[NUM_READERS], writers[NUM_WRITERS];
    int reader_ids[NUM_READERS], writer_ids[NUM_WRITERS];

    sem_init(&mutex, 0, 1);  // Inicjalizacja semafora dla sekcji krytycznej
    sem_init(&write_mutex, 0, 1);  // Inicjalizacja semafora dla zapisywaczy
    sem_init(&readers_mutex, 0, 1);  // Inicjalizacja semafora dla czytelników

    for (int i = 0; i < NUM_READERS; i++) {
        reader_ids[i] = i;
        pthread_create(&readers[i], NULL, reader, (void *)&reader_ids[i]);
    }

    for (int i = 0; i < NUM_WRITERS; i++) {
        writer_ids[i] = i;
        pthread_create(&writers[i], NULL, writer, (void *)&writer_ids[i]);
    }

    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(readers[i], NULL);
    }

    for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writers[i], NULL);
    }

    sem_destroy(&mutex);          // Zniszczenie semafora dla sekcji krytycznej
    sem_destroy(&write_mutex);    // Zniszczenie semafora dla zapisywaczy
    sem_destroy(&readers_mutex);  // Zniszczenie semafora dla czytelników

    return 0;
}
```

W tym przykładzie, semafor `mutex` jest używany do synchronizacji dostępu do sekcji krytycznej, semafor `write_mutex` chroni zapisywacza, a semafor `readers_mutex` zapewnia, że operacje inkrementacji/dekrementacji `readers_count` są atomowe. Czytelnicy zwiększają `readers_count` przed czytaniem i zmniejszają go po czytaniu. Pisarze zamykają semafor `write_mutex` przed zapisywaniem.

Zastosowanie semaforów w tym przypadku pomaga w zapewnieniu, że jednocześnie czytelnicy czytają lub jeden pisarz zapisuje, co eliminuje konflikty dostępu do współdzielonego zasobu.

20. Zapisać problem ucztujących filozofów z użyciem semaforów

Problem ucztujących filozofów to klasyczny problem synchronizacji w programowaniu współbieżnym, który ilustruje trudności związane z zarządzaniem dostępem do współdzielonych zasobów. W problemie tym filozofowie siedzą przy okrągłym stole, a między każdą parą sąsiadujących filozofów leży widelec. Aby zjeść posiłek, filozof musi podnieść oba widelce po swojej lewej i prawej stronie. Jednak aby uniknąć zakleszczeń, filozofowie muszą dostosować swoje działania. Oto przykładowa implementacja problemu ucztujących filozofów w języku C z użyciem semaforów w kontekście systemu Unix:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_PHILOSOPHERS 5

sem_t forks[NUM_PHILOSOPHERS];  // Semafor dla każdego widelca

void *philosopher(void *philosopher_id) {
    int tid = *(int *)philosopher_id;
    int left_fork = tid;
    int right_fork = (tid + 1) % NUM_PHILOSOPHERS;

    while (1) {
        // Filozof oczekuje na dostęp do obu widełek
        sem_wait(&forks[left_fork]);
        sem_wait(&forks[right_fork]);

        // Filozof je posiłek
        printf("Philosopher %d is eating\n", tid);
        usleep(rand() % 1000000);  // Symulacja jedzenia

        // Filozof odkłada widelce
        sem_post(&forks[left_fork]);
        sem_post(&forks[right_fork]);

        // Filozof myśli
        printf("Philosopher %d is thinking\n", tid);
        usleep(rand() % 1000000);  // Symulacja myślenia
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_ids[NUM_PHILOSOPHERS];

    // Inicjalizacja semaforów dla każdego widelca
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&forks[i], 0, 1);
    }
```

```
    // Inicjalizacja wątków dla każdego filozofa
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, (void
*)&philosopher_ids[i]);
    }

    // Oczekiwanie na zakończenie wątków
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Zniszczenie semaforów
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_destroy(&forks[i]);
    }

    return 0;
}
```

W tym przykładzie, każdy filozof jest reprezentowany przez osobny wątek. Każdy widelec jest chroniony przez semafor, a operacje podnoszenia i opuszczania semaforów zapewniają, że tylko jeden filozof naraz może podnieść dwa widelce i zacząć jeść. Aby uniknąć zakleszczeń, filozofowie muszą podnosić widelec po lewej stronie przed widełkiem po prawej stronie, a następnie odkładać widelce w odwrotnej kolejności.

Warto zauważyć, że ten kod nie jest zaprojektowany do zakończenia – filozofowie kontynuują jedzenie, myślenie i odkładanie widelców w nieskończoność. W praktyce, program ten byłby zakończony przez zewnętrzny mechanizm, na przykład sygnał przerwania (`Ctrl+C`).

21. Czym są monitory w programowaniu współbieżnym i po co je stosujemy

W systemie Unix, standard POSIX definiuje mechanizmy monitorów jako część biblioteki PThreads (POSIX Threads). Poniżej przedstawiam przykład prostego monitora w języku C z użyciem PThreads:

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;

// Struktura reprezentująca monitor
typedef struct {
    // Zasoby monitora
    int shared_data;
} Monitor;

// Inicjalizacja monitora
void initMonitor(Monitor *monitor) {
    monitor->shared_data = 0;
}
```

```c
// Operacja w monitorze - zwiększa wartość współdzielonego zasobu
void monitorOperation(Monitor *monitor) {
    pthread_mutex_lock(&mutex);

    // Zakres krytyczny monitora
    monitor->shared_data++;

    pthread_cond_signal(&condition); // Sygnalizacja warunku

    pthread_mutex_unlock(&mutex);
}

// Wątek korzystający z monitora
void *threadFunction(void *arg) {
    Monitor *monitor = (Monitor *)arg;

    while (1) {
        pthread_mutex_lock(&mutex);

        // Oczekiwanie na spełnienie warunku
        while (monitor->shared_data == 0) {
            pthread_cond_wait(&condition, &mutex);
        }

        // Wykonanie operacji na współdzielonym zasobie
        printf("Thread %ld: Shared data = %d\n", pthread_self(), monitor-
>shared_data);

        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[5];
    Monitor monitor;

    initMonitor(&monitor);

    for (int i = 0; i < 5; i++) {
        pthread_create(&threads[i], NULL, threadFunction, (void *)&monitor);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

W powyższym przykładzie `pthread_mutex_t` służy jako zamek monitora, a `pthread_cond_t` to warunek, który pozwala wątkom na oczekiwanie na spełnienie pewnego warunku. W funkcji `monitorOperation` jest przykład operacji wykonywanej w monitorze, po której sygnalizowany jest warunek, co powiadamia inne wątki, że mogą kontynuować. Wątki w funkcji `threadFunction` oczekują na spełnienie warunku, wykonują operację i ponownie oczekują.

Ważne jest, aby monitor był inicjalizowany przed użyciem, na przykład za pomocą funkcji `initMonitor` w przykładzie. Ponadto, w monitorze należy stosować zamek (`pthread_mutex_t`) do synchronizacji dostępu do współdzielonych zasobów.

22. Co to jest zmienna warunkowa i jak działa

Zmienna warunkowa (conditional variable) to mechanizm synchronizacyjny używany w programowaniu współbieżnym do umożliwienia wątkom czekanie na spełnienie pewnego warunku, a także sygnalizowania innym wątkom o wystąpieniu tego warunku. W systemie Unix, mechanizm ten jest często używany w kontekście monitorów, które pozwalają na bezpieczny dostęp do współdzielonych zasobów.

Zmienna warunkowa działa w następujący sposób:

1. **Inicjalizacja:**
    - Zmienna warunkowa musi być zainicjalizowana przed użyciem. W systemie Unix, często jest to osiągane za pomocą funkcji `pthread_cond_init`.

```
pthread_cond_t condition_variable = PTHREAD_COND_INITIALIZER;
```

2. **Czekanie (czekanie na warunek):**
    - Wątek, który chce czekać na spełnienie pewnego warunku, używa funkcji `pthread_cond_wait`.
    - Wątek ten blokuje się i przekazuje kontrolę do innego wątku, który może zmienić stan współdzielonych zasobów.

```
pthread_mutex_lock(&mutex);
while (!warunek_spełniony) {
    pthread_cond_wait(&condition_variable, &mutex);
}
// Kontynuacja działania po spełnieniu warunku
pthread_mutex_unlock(&mutex);
```

3. **Sygnalizacja (sygnalizowanie warunku):**
    - Wątek, który wprowadzi zmiany, które mogą wpłynąć na warunek oczekiwania, używa funkcji `pthread_cond_signal` lub `pthread_cond_broadcast`.
    - `pthread_cond_signal` sygnalizuje jednemu z oczekujących wątków.
    - `pthread_cond_broadcast` sygnalizuje wszystkim oczekującym wątkom.

```
pthread_mutex_lock(&mutex);
zmiana_w_zasobach(); // Wprowadzenie zmian, które mogą wpłynąć na warunek
oczekiwania
```

```
pthread_cond_signal(&condition_variable); // Lub pthread_cond_broadcast, jeśli
chcemy sygnalizować wszystkim
pthread_mutex_unlock(&mutex);
```

Ważne jest, aby sygnalizować warunek w sekcji krytycznej, a także korzystać z tego samego zamka (`pthread_mutex_t`) do zarządzania zmienną warunkową. To zapewnia bezpieczne operacje na warunku.

Zastosowanie zmiennych warunkowych umożliwia efektywne oczekiwanie wątków na pewne warunki, zamiast bezczynnego oczekiwania w pętli, co może prowadzić do nieefektywnego zużycia zasobów systemowych.

23. Zapisać problem producentów konsumentów z użyciem zmiennej warunkowej

Poniżej znajdziesz przykładową implementację problemu producentów i konsumentów w języku C z użyciem zmiennych warunkowych. W tym przypadku, korzystamy z biblioteki PThreads, która dostarcza mechanizmy do pracy z wątkami i zmiennymi warunkowymi.

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

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 2

int buffer[BUFFER_SIZE];
int in = 0, out = 0, count = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;

void produce(int item) {
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}

int consume() {
    int item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    return item;
}

void* producer(void* arg) {
    for (int i = 0; i < 5; i++) {
        int item = rand() % 100;  // Przykładowy element do wyprodukowania

        pthread_mutex_lock(&mutex);

        // Czekaj, jeśli bufor jest pełny
```

```
        while (count == BUFFER_SIZE) {
            pthread_cond_wait(&not_full, &mutex);
        }

        produce(item);
        printf("Producer produced item %d\n", item);

        // Powiadom odblokowane wątki konsumentów
        pthread_cond_signal(&not_empty);

        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(NULL);
}

void* consumer(void* arg) {
    for (int i = 0; i < 5; i++) {
        pthread_mutex_lock(&mutex);

        // Czekaj, jeśli bufor jest pusty
        while (count == 0) {
            pthread_cond_wait(&not_empty, &mutex);
        }

        int item = consume();
        printf("Consumer consumed item %d\n", item);

        // Powiadom odblokowane wątki producentów
        pthread_cond_signal(&not_full);

        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t producers[NUM_PRODUCERS], consumers[NUM_CONSUMERS];

    for (int i = 0; i < NUM_PRODUCERS; i++) {
        pthread_create(&producers[i], NULL, producer, NULL);
    }

    for (int i = 0; i < NUM_CONSUMERS; i++) {
        pthread_create(&consumers[i], NULL, consumer, NULL);
    }

    for (int i = 0; i < NUM_PRODUCERS; i++) {
        pthread_join(producers[i], NULL);
    }

    for (int i = 0; i < NUM_CONSUMERS; i++) {
        pthread_join(consumers[i], NULL);
```

```
    }

    return 0;
}
```

W tym przykładzie, zmienne warunkowe `not_empty` i `not_full` są używane do informowania wątków producentów i konsumentów o stanie bufora. Wątek producenta czeka na sygnał `not_full`, jeśli bufor jest pełny, a wątek konsumenta czeka na sygnał `not_empty`, jeśli bufor jest pusty. Po każdej operacji produkcji lub konsumpcji, wątki sygnalizują drugą grupę wątków, aby kontynuowały pracę. To eliminuje konieczność bezczynnego oczekiwania (busy waiting) i skutecznie zarządza zasobami.

Warto zaznaczyć, że w tym przykładzie zastosowano blokady mutex (`pthread_mutex_t`) wokół sekcji krytycznych, a zmienne warunkowe są używane do synchronizacji i powiadamiania o zmianach w stanie bufora.

24. Zapisać problem ucztujących filozofów z użyciem zmiennej warunkowej

Problem ucztujących filozofów to klasyczny problem synchronizacji w programowaniu współbieżnym, gdzie filozofowie siedzą przy okrągłym stole, a między każdą parą sąsiadujących filozofów leży widelec. Aby zjeść posiłek, filozof musi podnieść oba widelce po swojej lewej i prawej stronie. Jednak aby uniknąć zakleszczeń, filozofowie muszą dostosować swoje działania.

Poniżej znajdziesz przykładową implementację problemu ucztujących filozofów w języku C z użyciem zmiennych warunkowych:

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

#define NUM_PHILOSOPHERS 5

pthread_mutex_t forks[NUM_PHILOSOPHERS];
pthread_cond_t philosophers_condition[NUM_PHILOSOPHERS];

void *philosopher(void *philosopher_id) {
    int tid = *(int *)philosopher_id;
    int left_fork = tid;
    int right_fork = (tid + 1) % NUM_PHILOSOPHERS;

    while (1) {
        pthread_mutex_lock(&forks[left_fork]);
        pthread_mutex_lock(&forks[right_fork]);

        // Filozof je posiłek
        printf("Philosopher %d is eating\n", tid);

        pthread_mutex_unlock(&forks[left_fork]);
        pthread_mutex_unlock(&forks[right_fork]);

        // Filozof myśli
```

```
            printf("Philosopher %d is thinking\n", tid);

            // Odczekaj na zmniejszenie obciążenia systemu
            usleep(rand() % 1000000);
        }

        pthread_exit(NULL);
    }

    int main() {
        pthread_t philosophers[NUM_PHILOSOPHERS];
        int philosopher_ids[NUM_PHILOSOPHERS];

        for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
            pthread_mutex_init(&forks[i], NULL);
            pthread_cond_init(&philosophers_condition[i], NULL);
        }

        for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
            philosopher_ids[i] = i;
            pthread_create(&philosophers[i], NULL, philosopher, (void
    *)&philosopher_ids[i]);
        }

        for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
            pthread_join(philosophers[i], NULL);
        }

        for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
            pthread_mutex_destroy(&forks[i]);
            pthread_cond_destroy(&philosophers_condition[i]);
        }

        return 0;
    }
```

W tym przykładzie, każdy widelec jest chroniony przez mutex (`pthread_mutex_t`), a zmienne warunkowe (`pthread_cond_t`) są używane do informowania filozofów o gotowości do jedzenia. Filozofowie używają blokad mutexów do uniknięcia jednoczesnego dostępu do widełek przez wielu filozofów. Gdy filozof podnosi oba widelce, informuje o tym zdarzeniu inne filozofy, a następnie odkłada widelce i przechodzi do stanu myślenia. To podejście eliminuje potencjalne zakleszczenia, gdy wszyscy filozofowie jednocześnie chcieliby podnieść swoje lewe widelce.

25. Zapisać problem czytelników -pisarzy z użyciem zmiennej warunkowej

Problem czytelników i pisarzy to inny klasyczny problem synchronizacji w programowaniu współbieżnym, gdzie wielu czytelników i pisarzy współdzieli zasób, na przykład wspólną bazę danych. Czytelnicy czytają zasób, podczas gdy pisarze zapisują do niego. Jednak jednoczesny dostęp czytelników do zasobu nie wpływa na spójność danych, ale zapisywanie przez jednego pisarza powinno blokować zarówno czytelników, jak i innych pisarzy.

Poniżej znajdziesz przykładową implementację problemu czytelników i pisarzy w języku C z użyciem zmiennych warunkowych:

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

#define NUM_READERS 3
#define NUM_WRITERS 2

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t readers_condition = PTHREAD_COND_INITIALIZER;
pthread_cond_t writers_condition = PTHREAD_COND_INITIALIZER;

int readers_count = 0; // Licznik aktywnych czytelników

void *reader(void *reader_id) {
    int tid = *(int *)reader_id;

    while (1) {
        pthread_mutex_lock(&mutex);

        // Czekaj, jeśli jest pisarz lub inny czytelnik zapisuje
        while (readers_count == -1) {
            pthread_cond_wait(&readers_condition, &mutex);
        }

        readers_count++;

        pthread_mutex_unlock(&mutex);

        // Czytanie zasobu
        printf("Reader %d is reading\n", tid);

        pthread_mutex_lock(&mutex);
        readers_count--;

        // Jeśli to ostatni czytelnik, informuj pisarzy, że mogą zapisywać
        if (readers_count == 0) {
            pthread_cond_signal(&writers_condition);
        }

        pthread_mutex_unlock(&mutex);

        // Odczekaj na zmniejszenie obciążenia systemu
        usleep(rand() % 1000000);
    }

    pthread_exit(NULL);
}

void *writer(void *writer_id) {
```

```
    int tid = *(int *)writer_id;

    while (1) {
        pthread_mutex_lock(&mutex);

        // Czekaj, jeśli jest czytelnik lub inny pisarz zapisuje
        while (readers_count != 0) {
            pthread_cond_wait(&writers_condition, &mutex);
        }

        readers_count = -1; // Blokuj dostęp czytelników i innych pisarzy

        pthread_mutex_unlock(&mutex);

        // Zapisywanie do zasobu
        printf("Writer %d is writing\n", tid);

        pthread_mutex_lock(&mutex);
        readers_count = 0;

        // Informuj czytelników, że mogą czytać
        pthread_cond_broadcast(&readers_condition);

        pthread_mutex_unlock(&mutex);

        // Odczekaj na zmniejszenie obciążenia systemu
        usleep(rand() % 1000000);
    }

    pthread_exit(NULL);
}

int main() {
    pthread_t readers[NUM_READERS], writers[NUM_WRITERS];
    int reader_ids[NUM_READERS], writer_ids[NUM_WRITERS];

    for (int i = 0; i < NUM_READERS; i++) {
        reader_ids[i] = i;
        pthread_create(&readers[i], NULL, reader, (void *)&reader_ids[i]);
    }

    for (int i = 0; i < NUM_WRITERS; i++) {
        writer_ids[i] = i;
        pthread_create(&writers[i], NULL, writer, (void *)&writer_ids[i]);
    }

    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(readers[i], NULL);
    }

    for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writers[i], NULL);
    }
```

```
    return 0;
}
```

W tym przykładzie, czytelnicy i pisarze używają zmiennej warunkowej, aby czekać na swoją kolej na dostęp do zasobu. Zmienna `readers_count` pełni rolę licznika aktywnych czytelników, a jej wartość `-1` oznacza, że jest aktywny pisarz. Gdy pisarz zaczyna pisać, blokuje dostęp czytelników i innych pisarzy, a gdy kończy pisanie, informuje czytelników, że mogą czytać. Czytelnicy sprawdzają warunek i czekają, gdy jest aktywny pisarz, ale mogą czytać, gdy nie ma żadnych aktywnych pisarzy.

26. Omówić monitor Hoara oraz Massy

Monitor Hoare'a i monitor Massy to dwie różne koncepcje monitorów w programowaniu współbieżnym. Obydwa są mechanizmami synchronizacyjnymi, ale różnią się w swoim podejściu do obsługi warunków i sygnalizacji. Poniżej znajdziesz krótkie omówienie obu monitorów:

Monitor Hoare'a:

1. **Warunki:**
    - Monitor Hoare'a obsługuje warunki (ang. conditions), które pozwalają wątkom czekać na spełnienie określonego warunku przed kontynuacją działania.
    - Wątek może wywołać `wait` na warunku, aby tymczasowo zwolnić zasoby monitora i oczekiwać na sygnał, że warunek został spełniony.
2. **Sygnalizacja:**
    - `signal` w monitorze Hoare'a powiadamia jedynie jeden wątek oczekujący na warunku.
    - W przypadku zakończenia działania wątku, który wywołał `signal`, nowy wątek może przejąć kontrolę, nawet jeśli warunek nie został jeszcze spełniony.
3. **Zakleszczenia:**
    - Teoretycznie, monitor Hoare'a może prowadzić do zakleszczeń w pewnych warunkach, szczególnie jeśli jest używany nieostrożnie.
    - Wykorzystanie `wait` i `signal` musi być starannie zaplanowane, aby unikać potencjalnych problemów z zakleszczaniem.

Monitor Massy:

1. **Warunki:**
    - Monitor Massy również obsługuje warunki, ale różni się w tym, że używa jednocześnie dwóch rodzajów warunków: warunku zmiennego i warunku zdarzenia.
    - Warunek zmienny jest używany do warunkowego wejścia do monitora, podobnie jak w przypadku Hoare'a.
    - Warunek zdarzenia jest używany do określenia, czy warunek został spełniony.
2. **Sygnalizacja:**

    - `signal` w monitorze Massy sygnalizuje jedynie warunek zdarzenia, co oznacza, że wątek oczekujący może przejąć kontrolę tylko wtedy, gdy warunek zdarzenia jest spełniony.
    - To ograniczenie zmniejsza ryzyko zakleszczeń.

3. **Zakleszczenia:**

    - Monitor Massy został zaprojektowany tak, aby minimalizować ryzyko zakleszczeń.
    - Korzystanie z warunków zmiennych i zdarzeń wraz z bardziej restrykcyjną sygnalizacją pomaga w unikaniu problemów z zakleszczaniem.

Podsumowanie:

- Monitor Hoare'a oferuje bardziej ogólne podejście, ale wymaga od programisty ostrożności w użyciu `wait` i `signal`, aby uniknąć zakleszczeń.
- Monitor Massy wprowadza dodatkową warstwę sygnalizacji, co może zwiększyć bezpieczeństwo w kontekście zakleszczeń.

Obie koncepcje monitorów zostały opracowane, aby ułatwić synchronizację dostępu do współdzielonych zasobów w programowaniu współbieżnym. Wybór między nimi zależy od preferencji programisty i konkretnego kontekstu aplikacji.

27. Omówić etapy projektowania algorytmu równoległego

Projektowanie algorytmu równoległego jest procesem skomplikowanym, który wymaga uwzględnienia wielu czynników, takich jak natura problemu, architektura sprzętowa, dostępność zasobów i efektywność obliczeń równoległych. Poniżej przedstawiam etapy projektowania algorytmu równoległego:

1. **Analiza problemu:**

- Zrozumienie charakterystyki problemu, który ma być rozwiązany równolegle.
- Identyfikacja obszarów, które można skutecznie zrównoleglić, takich jak niezależne obliczenia, powtarzalne operacje, czy operacje na niezależnych danych.

2. **Podział problemu:**

- Zidentyfikowanie części zadania, które mogą być wykonane równolegle.
- Podział problemu na mniejsze zadania, które mogą być przetwarzane niezależnie od siebie.
- Określenie, czy podział zadania może być zrealizowany statycznie czy dynamicznie.

3. **Określenie modelu równoległego:**

- Wybór modelu równoległego, czyli sposób, w jaki zadania będą wykonywane równolegle (np. równoległość danych, równoległość zadań, równoległość funkcji).
- Określenie, czy algorytm będzie działał asynchronicznie czy synchronicznie.

4. **Projektowanie algorytmu:**

- Opracowanie algorytmu, który realizuje podział zadania i określony model równoległy.
- Uwzględnienie synchronizacji wątków lub procesów, jeśli to konieczne.
- Optymalizacja algorytmu pod kątem równoległego wykonania, unikanie zależności między zadaniami i minimalizacja konieczności komunikacji między jednostkami obliczeniowymi.

5. **Wybór struktury danych:**

- Wybór odpowiednich struktur danych, które umożliwią efektywne przechowywanie i udostępnianie danych między jednostkami obliczeniowymi.
- Uwzględnienie dostępu do pamięci współdzielonej lub dystrybuowanej w zależności od architektury systemu równoległego.

6. **Implementacja i debugowanie:**

- Implementacja algorytmu na wybranej platformie równoległej.
- Testowanie i debugowanie kodu równoległego, identyfikacja ewentualnych zakleszczeń, wyścigów czy innych problemów synchronizacyjnych.

7. **Optymalizacja:**

- Optymalizacja algorytmu pod kątem efektywnego wykorzystania zasobów sprzętowych.
- Uwzględnienie specyfikacji danego systemu równoległego, tak aby zminimalizować narzut związanego z komunikacją między jednostkami obliczeniowymi.

8. **Ewaluacja:**

- Pomiar i analiza efektywności algorytmu równoległego pod kątem czasu wykonania, zużycia zasobów i skalowalności.
- Weryfikacja, czy osiągnięto oczekiwane przyspieszenie w stosunku do wersji sekwencyjnej.

9. **Dokumentacja:**

- Dokumentacja algorytmu równoległego, w tym opis problemu, modelu równoległego, struktury danych, implementacji i wyników ewaluacji.

10. **Adaptacja i doskonalenie:**

- W razie potrzeby dostosowanie algorytmu do ewentualnych zmian w problemie, architekturze sprzętowej czy wymaganiach wydajnościowych.
- Ciągłe doskonalenie algorytmu w oparciu o nowe odkrycia w dziedzinie równoległego programowania.

Proces projektowania algorytmu równoległego wymaga zrozumienia zarówno charakterystyki problemu, jak i architektury system

28. Omówić metody dekompozycji stosowane w przetwarzaniu współbieżnym

Dekompozycja to proces podzielenia zadania na mniejsze części, które mogą być przetwarzane równolegle. W kontekście przetwarzania współbieżnego istnieje kilka głównych metod dekompozycji, z których każda ma swoje zastosowanie w zależności od charakterystyki problemu. Poniżej omówione są główne metody dekompozycji stosowane w przetwarzaniu współbieżnym:

1. **Dekompozycja funkcjonalna:**

- Polega na podziale zadania na mniejsze funkcje lub podzadania.
- Każda funkcja lub podzadanie może być przetwarzane równolegle.

- Często używana w algorytmach podziału i zrównoleglania, gdzie różne funkcje są przypisane różnym jednostkom obliczeniowym.

2. **Dekompozycja danych:**

- Opiera się na podziale danych na fragmenty, które są przetwarzane równolegle przez różne jednostki obliczeniowe.
- Bardzo efektywna w problemach, gdzie dane można podzielić na niezależne fragmenty, a wyniki są łączone na końcu.
- Przykłady to podział macierzy, wektorów, tablic czy innych struktur danych.

3. **Dekompozycja dziedzinowa:**

- Polega na podziale obszaru problemu na podobszary, z których każdy jest przetwarzany równolegle.
- Często stosowana w problemach, gdzie istnieje naturalny podział przestrzeni lub dziedziny problemu.
- Przykłady to podział przestrzeni obliczeniowej, przestrzeni grafów czy innych struktur dziedzinowych.

4. **Dekompozycja geograficzna:**

- Polega na podziale obszaru geograficznego lub przestrzeni fizycznej na fragmenty, które są przetwarzane równolegle.
- Często stosowana w problemach związanych z analizą danych przestrzennych, geoinformatyką czy symulacjami związanymi z przestrzenią.

5. **Dekompozycja hierarchiczna:**

- Kombinuje różne metody dekompozycji w strukturę hierarchiczną.
- Zadanie jest dzielone na mniejsze części, a każda z tych części może być dalej dekomponowana według innej metody.
- Stosowana w problemach, gdzie istnieje hierarchia struktury lub zależności.

6. **Dekompozycja zadaniowa:**

- Każde zadanie jest traktowane jako odrębna jednostka, która może być przetwarzana równolegle.
- Bardzo elastyczna, ale wymaga skoordynowanego zarządzania i komunikacji między jednostkami obliczeniowymi.
- Stosowana w systemach, gdzie zadania są niewielkie, a ich ilość jest duża.

7. **Dekompozycja czasowa:**

- Polega na przyspieszeniu przetwarzania poprzez równoległe wykonywanie różnych etapów zadania w tym samym czasie.
- Często stosowana w przypadku złożonych operacji, gdzie różne etapy mogą być przetwarzane równolegle.

Wybór odpowiedniej metody dekompozycji zależy od charakterystyki problemu, dostępnych zasobów, architektury sprzętowej i wymagań dotyczących wydajności. Często projektanci algorytmów równoległych stosują kombinacje różnych metod, aby uzyskać optymalne wyniki.

29. Opisz Prawo Amdahla oraz wnioski z niego wynikające

Prawo Amdahla to teoretyczna formuła opracowana przez Gene'a Amdahla w 1967 roku, która opisuje, jak maksymalne przyspieszenie można osiągnąć przy użyciu równoległego przetwarzania w kontekście danego zadania. Prawo Amdahla pozwala ocenić, jak bardzo efektywne stanie się przetwarzanie równoległe w zależności od stopnia zrównoleglenia danego zadania.

Formuła Prawa Amdahla jest następująca:

[S(n) = \frac{1}{(1 - P) + \frac{P}{n}}]

Gdzie:

- (S(n)) to przyspieszenie całkowite (speedup),
- (P) to ułamek pracy, który może być zrównoleglony,
- (n) to liczba procesorów (jednostek obliczeniowych).

Wnioski z Prawa Amdahla:

1. **Przyrost przyspieszenia jest ograniczony:**
    - Prawo Amdahla pokazuje, że maksymalne przyspieszenie jest ograniczone przez część sekwencyjną programu. Nawet jeśli bardzo dużo pracy można by było zrównoleglić, to sekwencyjna część programu ogranicza korzyści z równoległego przetwarzania.
2. **Zrównoleglenie ma sens tylko dla dużych danych:**
    - Im większa jest wartość (P) (ułamek pracy zrównoleglonej), tym większe przyspieszenie można osiągnąć. Jednakże, aby to miało znaczący efekt, (P) musi być duże. Dlatego równoległe przetwarzanie najbardziej opłaca się dla dużych zbiorów danych lub zadań o dużym stopniu zrównoleglenia.
3. **Konieczność równoważenia obciążenia:**
    - Jeśli zadanie jest nierównomiernie podzielone między procesory, to nieefektywności mogą obniżyć przyspieszenie. Konieczne jest równomierne podzielenie pracy między jednostki obliczeniowe.
4. **Równoległe przetwarzanie ma sens tylko dla istotnych czasowo operacji:**
    - Jeśli sekwencyjna część programu wymaga bardzo mało czasu, to nawet duża ilość zrównoleglonej pracy może nie przynieść istotnego przyspieszenia.
5. **Wartość (n) ma ograniczenie górne:**
    - Wzrost liczby procesorów (n) zwiększa przyspieszenie, ale tylko do pewnego momentu. Wartość (n) jest ograniczona przez ilość dostępnych zasobów, a także przez ewentualne koszty komunikacji i synchronizacji.

Prawo Amdahla przypomina, że zrównoleglenie nie zawsze prowadzi do liniowego przyspieszenia, a przemyślane podejście do zrównoleglenia oraz analiza skali zadania są kluczowe dla uzyskania korzyści z równoległego przetwarzania.

30. Wyjaśnij pojęcia przyspieszenia oraz wydajności algorytmu równoległego

Przyspieszenie algorytmu równoległego:

Przyspieszenie ((S)) algorytmu równoległego to miara, która ilustruje, ile razy algorytm zrównoleglony jest szybszy niż jego wersja sekwencyjna, działająca na jednym procesorze. Można je obliczyć za pomocą wzoru:

[S = \frac{T_{\text{sekwencyjny}}}{T_{\text{równoległy}}}]

gdzie:

- (T_{\text{sekwencyjny}}) to czas wykonania algorytmu w wersji sekwencyjnej,
- (T_{\text{równoległy}}) to czas wykonania algorytmu w wersji równoległej na (n) procesorach.

Im większe przyspieszenie, tym efektywniejsze zrównoleglanie. Idealnie, dla doskonałego zrównoleglenia, przyspieszenie byłoby równe liczbie procesorów ((n)).

Wydajność algorytmu równoległego:

Wydajność algorytmu równoległego to miara efektywności wykorzystania dostępnych zasobów obliczeniowych. Można ją obliczyć za pomocą wzoru:

[\text{Wydajność} = \frac{S}{n}]

gdzie:

- (S) to przyspieszenie algorytmu równoległego,
- (n) to liczba procesorów.

Wydajność informuje nas, jak dobrze algorytm wykorzystuje potencjał równoległy w porównaniu do ilości użytych procesorów. Optymalna wydajność wynosi 1, co oznacza, że zasoby są używane w pełni, a algorytm jest idealnie zrównoleglony. Wydajność poniżej 1 oznacza, że zrównoleglenie nie jest efektywne, a nadmiar procesorów może prowadzić do narzutu związanego z komunikacją i synchronizacją.

Podsumowując, przyspieszenie mierzy, ile razy algorytm jest szybszy w wersji równoległej, a wydajność uwzględnia to przyspieszenie w kontekście liczby użytych procesorów, dostarczając informacji o efektywności wykorzystania zasobów.

31. Wyjaśnij kiedy możemy mieć do czynienia z przyspieszeniem superliniowym

Przyspieszenie superliniowe to sytuacja, w której przyspieszenie algorytmu równoległego jest większe niż liczba użytych procesorów. Jest to nietypowy, ale teoretycznie możliwy scenariusz, który może wystąpić w pewnych warunkach. Istnieje kilka sytuacji, które mogą prowadzić do przyspieszenia superliniowego:

1. **Czas dostępu do pamięci:**
    - W przypadku, gdy algorytm równoległy wymaga intensywnego korzystania z pamięci, a dostęp do pamięci jest czasochłonny, zwiększenie liczby procesorów może pomóc w redukcji czasu oczekiwania na dane. W rezultacie przyspieszenie może być większe niż sama liczba procesorów.
2. **Uwspólnienie zasobów:**
    - Jeśli algorytm równoległy korzysta z zasobów wspólnych, takich jak wspólna pamięć podręczna, wzrost liczby procesorów może prowadzić do lepszej współpracy w dostępie do wspólnych

danych, co przyspiesza wykonanie zadania.

3. **Optymalizacje kosztów obliczeń:**
    - W niektórych przypadkach, zwiększenie liczby procesorów może prowadzić do bardziej efektywnego zarządzania kosztami obliczeń. Na przykład, jeśli zrównoleglenie pozwala na lepsze wykorzystanie dostępnych jednostek obliczeniowych, to przyspieszenie może być większe niż liczba procesorów.

Warto jednak zauważyć, że przyspieszenie superliniowe jest rzadko spotykane i zazwyczaj jest efektem szczególnych warunków problemu oraz architektury systemu. Ponadto, w praktyce, zwykle istnieją pewne ograniczenia, takie jak narzuty związane z komunikacją i synchronizacją między procesorami, które mogą ograniczać skokowe zwiększenie wydajności wraz ze wzrostem liczby procesorów. Dlatego w większości przypadków oczekuje się, że przyspieszenie będzie subliniowe lub liniowe w najlepszym przypadku.

32. Wymień źródła narzutów przetwarzania równoległego

Narzuty przetwarzania równoległego to dodatkowe koszty i opóźnienia związane z organizacją i zarządzaniem równoległymi zadaniami. W kontekście przetwarzania równoległego można wyróżnić kilka źródeł narzutów:

1. **Narzut komunikacyjny:**
    - Wynika z konieczności wymiany danych między jednostkami obliczeniowymi. Komunikacja może wprowadzać opóźnienia związane z przesyłaniem danych przez sieć, wspólną pamięć czy inne mechanizmy komunikacyjne.
2. **Narzut synchronizacyjny:**
    - Związany z potrzebą synchronizacji działań równoległych jednostek obliczeniowych. Oczekiwanie na zakończenie innych zadań, utrzymanie porządku wykonania, czy blokowanie dostępu do wspólnych zasobów to przykłady operacji synchronizacyjnych, które mogą wprowadzać opóźnienia.
3. **Narzut planowania (scheduling):**
    - Wynika z konieczności zarządzania rozkładem zadań na dostępnych jednostkach obliczeniowych. Procesy planowania, alokacji zasobów i dystrybucji zadań między jednostki obliczeniowe mogą wprowadzać dodatkowy narzut czasowy.
4. **Narzut zarządzania zasobami:**
    - Związany z zarządzaniem zasobami sprzętowymi i programowymi, takimi jak dostęp do pamięci, wspólnych struktur danych czy innych zasobów systemowych. Mechanizmy zarządzania zasobami mogą wprowadzać dodatkowy koszt czasowy.
5. **Narzut tworzenia i zakończenia wątków/procesów:**
    - Tworzenie nowych jednostek obliczeniowych, takich jak wątki czy procesy, oraz zakończenie ich działania to operacje, które również wymagają zasobów i czasu.
6. **Narzut obciążenia (load balancing):**

- W przypadku, gdy zadania nie są równomiernie rozdzielane między jednostki obliczeniowe, może wystąpić narzut związany z nierównomiernym wykorzystaniem zasobów, co obniża efektywność przetwarzania równoległego.

7. **Narzut kosztów synchronizacji przy współdzielonej pamięci:**
   - W systemach z współdzieloną pamięcią narzut ten wynika z konieczności użycia mechanizmów synchronizacji, takich jak semafory czy blokady, aby uniknąć konfliktów i czytelność danych.

Rozważanie i minimalizacja tych narzutów jest kluczowe podczas projektowania efektywnych algorytmów równoległych, aby uzyskać zrównoważoną dystrybucję pracy i uniknąć zbędnych opóźnień.

33. Jak obliczamy koszt algorytmu równoległego oraz całkowity narzut równoległy

Koszt algorytmu równoległego i całkowity narzut równoległy to kategorie używane do analizy efektywności przetwarzania równoległego. Poniżej opisuję, jak można je obliczyć:

Koszt algorytmu równoległego ((C_p)):

Koszt algorytmu równoległego to suma kosztów wykonania poszczególnych części sekwencyjnej i zrównoleglonej wersji algorytmu. Można to wyrazić wzorem:

$$ C_p = f \cdot T_{\text{sekwencyjny}} + \frac{T_{\text{równoległy}}}{p} $$

gdzie:

- (C_p) to koszt algorytmu równoległego,
- (f) to ułamek pracy, który jest sekwencyjny,
- (T_{\text{sekwencyjny}}) to czas wykonania algorytmu w wersji sekwencyjnej,
- (T_{\text{równoległy}}) to czas wykonania algorytmu w wersji równoległej na (p) procesorach.

Całkowity narzut równoległy ((O_p)):

Całkowity narzut równoległy to stosunek czasu wykonania sekwencyjnej wersji algorytmu do czasu wykonania równoległej wersji na (p) procesorach. Można go wyrazić wzorem:

$$ O_p = \frac{T_{\text{sekwencyjny}}}{T_{\text{równoległy}}/p} $$

gdzie:

- (O_p) to całkowity narzut równoległy,
- (T_{\text{sekwencyjny}}) to czas wykonania algorytmu w wersji sekwencyjnej,
- (T_{\text{równoległy}}) to czas wykonania algorytmu w wersji równoległej na (p) procesorach.

Jeśli (O_p) jest bliskie 1, oznacza to, że algorytm ma niewielki narzut związany z przetwarzaniem równoległym. Jednak dla efektywnego przetwarzania równoległego koszt algorytmu równoległego ((C_p)) powinien być mniejszy niż czas wykonania sekwencyjnej wersji ((T_{\text{sekwencyjny}})).

Analiza tych wskaźników pozwala ocenić efektywność przetwarzania równoległego w kontekście danego algorytmu i architektury systemu.