1. **Opisać różnicę między programem równoległym, sekwencyjnym i współbieżnym.**

Program sekwencyjny- poszczególne akcje procesu są wykonywane jedna po drugiej. Dokładniej: kolejna akcja rozpoczyna się po całkowitym zakończeniu poprzedniej. (np. wywołania rekurencyjne w algorytmie sortowania przez scalanie)

Program równoległy (paralel program) – odnosi się do systemów, w których wiele programów wykonywanych jest w tym samym czasie. Obliczenia mogą być realizowane np. na różnych procesorach.

Program współbieżny (concurrent program) – zbiór programów sekwencyjnych, które można wykonać równolegle (Dwa procesy współbieżne jeżeli jeden z nich rozpoczyna się przed zakończeniem drugiego). Jest to uogólnione pojęcie obejmujące zarówno wykonanie równoległe, jak i przeplatane.

Słowo współbieżność oznacza potencjalną równoległość tj. procesy mogą się wykonywać w tym samym czasie, ale nie muszą. Równoległość może być jedynie pozorna np. w przypadku jednego procesora. Współbieżność jest pewną abstrakcją.

W przypadku programów sekwencyjnych, gdy naraz wykonuje się jedna instrukcja i nie trzeba rozważać wszystkich możliwych interakcji z innymi działającymi w tym samym czasie programami.

Wprowadzenie współbieżności jeszcze bardziej utrudnia programowanie. Każde uruchomienie programu współbieżnego prowadzi do wykonania innego scenariusza. W programie współbieżnym niektóre scenariusze prowadzą do poprawnego wyniku, inne zaś nie.

2. **Co to jest proces. Jakie znasz stany procesu.**

Jako program należy rozumieć kod wykonywany przez proces. Ma on charakter statyczny.

Proces ma charakter dynamiczny. Oznacza wykonanie programu w pewnym środowisku lub wstrzymane wykonanie (w oczekiwaniu na jakiś zasób). Proces ma przydzielone zasoby (pamięć, urządzenia we/wy) i rywalizuje o dostęp do procesora. Procesy wykonują się pod kontrolą systemu operacyjnego.

Stany procesu:

Proces może być **gotowy** do wykonania. Oznacza to, że ma wszystkie potrzebne zasoby z wyjątkiem procesora. Gdy tylko otrzyma procesor, może rozpocząć wykonanie.

Proces może być **wykonywany**, jeśli jest gotowy i ma procesor. Jest tyle procesów wykonywanych, ile procesorów znajduje się w systemie.

Proces wykonywany może zrzec się procesora i stać się **wstrzymany**, jeśli próbuje wykonać jakąś operację, której nie może zrealizować natychmiast, bo np. nie ma niezbędnych zasobów. Klasycznym przykładem jest wykonywanie operacji we/wy i konieczność poczekania aż niezbędne informacje zostaną sprowadzone z pamięci zewnętrznej. System operacyjny po sprowadzeniu niezbędnych zasobów zmieni stan procesu na gotowy.

3. **Co to jest wątek.**

Wątek (thread) – część programu wykonywana współbieżnie w obrębie jednego procesu; w jednym procesie może istnieć wiele wątków

Różnica między zwykłym procesem a wątkiem polega na współdzieleniu przez wszystkie wątki działające w danym procesie przestrzeni adresowej oraz wszystkich innych struktur systemowych (np. listy otwartych plików, gniazd itp.) – z kolei procesy posiadają niezależne zasoby.

Wątki wymagają mniej zasobów do działania i też mniejszy jest ich czas tworzenia

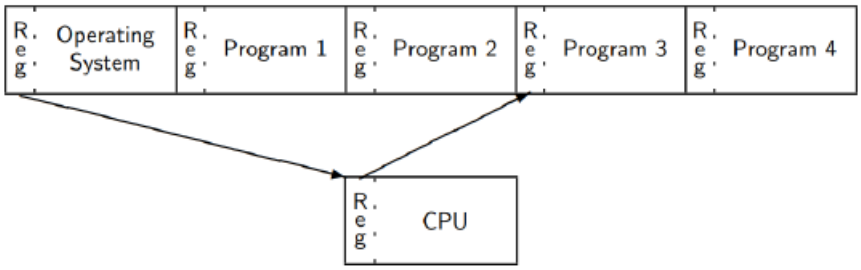
Dzięki współdzieleniu przestrzeni adresowej (pamięci) wątki jednego zadania mogą się miedzy sobą komunikować w bardzo łatwy sposób, niewymagający pomocy ze strony systemu operacyjnego.

Przekazanie dowolnie dużej ilości danych wymaga przesłania jedynie wskaźnika, zaś odczyt (a niekiedy zapis) danych o rozmiarze nie większym od słowa maszynowego nie wymaga synchronizacji (procesor gwarantuje atomowość takiej operacji).

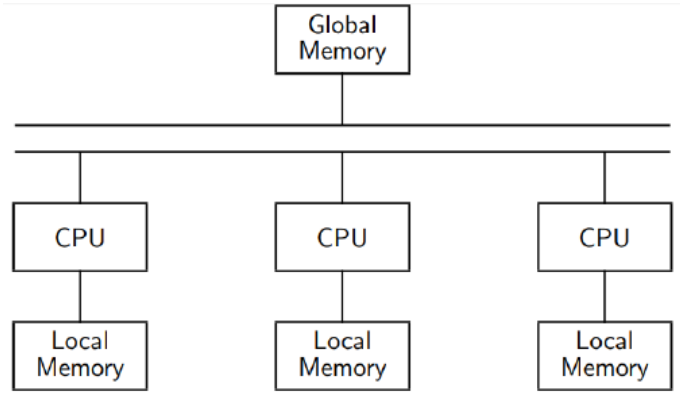
4. Wymień i opisz znane Ci architektury systemowe.

Architektury systemowe:

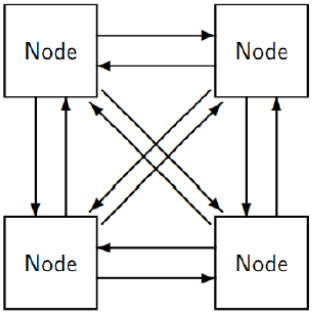
- Systemy wielozadaniowe

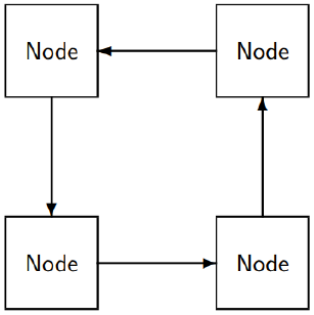


  - ❖ Nie ma możliwości wykonywania wielu rozkazów równolegle
  - ❖ O kolejności wykonywanych rozkazów decyduje procesor i system operacyjny
  - ❖ Występują przerwania przez urządzenie WE/WY
  - ❖ Po przerwaniu system operacyjny wybiera kolejny proces do wykonania
- Multiprocesory ze wspólną pamięcią



  - ❖ Posiada więcej niż jeden procesor
  - ❖ Dostęp do pamięci lokalnej mają tylko pojedyncze procesory, do globalnej wszystkie
  - ❖ Jeśli mamy dostateczną liczbę procesorów to można każdemu z nich przypisać jeden proces (brak przeplotu)
  - ❖ Jeśli procesy nie sięgają do pamięci globalnej to nie ma rywalizacji i obliczenia są w pełni równoległe
- Architektura rozproszona





- Składa się z wielu komputerów bez wspólnych zasobów
- Są one połączone kanałami komunikacyjnymi
- Często do ich opisu używa się analogi grafowej
- Mogą być rozproszone geograficznie
- Współpraca z innymi węzłami odbywa się poprzez wymianę komunikatów
- Brak pamięci globalnej wspólnej dla wszystkich maszyn
- Topologia pełna – efektywna, ale droga (alternatywa topologia pierścienia)

5. **Sformułować własność bezpieczeństwa programów równoległych.**

Nigdy nie dojdzie do niepożądanej sytuacji

Popatrzymy na przykład. Przypuśćmy, że rozpatrujemy ruchliwe skrzyżowanie w środku dużego miasta. Procesy to samochody usiłujące przez to skrzyżowanie przejechać (na wprost). Własność bezpieczeństwa wyraża fakt, że nie dojdzie do kolizji. Możemy ją wyrazić tak: nigdy na skrzyżowaniu nie będą jednocześnie samochody jadące w kierunku wschód-zachód i północ-południe.

Własność bezpieczeństwa zazwyczaj jest podawana w specyfikacji zadania, które należy rozwiązać. Stanowi ona odpowiednik częściowej poprawności.

W omawianym przykładzie skrzyżowania własność bezpieczeństwa można łatwo zagwarantować: nie wpuszczać na skrzyżowanie żadnego samochodu! Mamy wtedy bardzo bezpieczne rozwiązanie, ale … trudno uznać je za poprawne. Z tego powodu poprawny program współbieżny powinien mieć także inną własność.

6. **Sformułować własność żywotności programów równoległych.**

Jeśli proces chce coś zrobić, to w końcu mu się to uda

W przykładzie skrzyżowania oznacza to, że każdy samochód, który chce przez skrzyżowanie przejechać, w końcu przez to skrzyżowanie przejedzie.

7. **Jakie problemy mogą wynikać z braku żywotności programu.**

Zakleszczenie. Jest to globalny brak żywotności. Nic się nie dzieje, system nie pracuje, oczekując na zajście zdarzenia, które nigdy nie zajdzie.

Zagłodzenie. Jest to lokalny brak żywotności, dotyka tylko niektórych procesów. Przypuśćmy, że w przedstawionym problemie skrzyżowania, procesy jadące mogą wjechać na skrzyżowanie zawsze wtedy, gdy są na nim inne procesy jadące w tym samym kierunku lub gdy skrzyżowanie jest puste. Załóżmy, że jako pierwszy zjawia się proces jadący na kierunku wschód- zachód. Oczywiście wjeżdża na skrzyżowanie. Jeśli teraz pojawi się proces jadący z północy, to oczywiście musi poczekać.

8. **Opisać problem blokady i zagłodzenia programu równoległego.**

Blokada:

Inne nazwy zastój, zakleszczenie, martwy punkt

Blokada oznacza, że jeden lub więcej procesów się nie kończy. Może ono wystąpić wtedy, gdy procesy współbieżne synchronizują swoje działania.

Przyczyną mogą być wspólne zasoby, z których korzysta kilka procesów.

W programie współbieżnym blokada nie musi wystąpić przy każdym wykonaniu programu.

Unikanie blokady może być bardzo kosztowne.

Jeśli prawdopodobieństwo wystąpienia blokady jest małe, lepiej jest godzić się na nią stosując jednocześnie mechanizmy jej wykrywania i niwelowania.

Zagłodzenie:

Występuje jeśli jakiś proces jest nieskończenie długo wstrzymywany w oczekiwaniu na komunikat lub sygnał synchronizacyjny.

Zdarzenie, na które czeka proces, występuje dowolną liczbę razy, ale za każdym razem jest wybierany do wznowienia inny proces.

Zagłodzenie świadczy o braku żywotności programu.

Zjawisko zagłodzenia nie wystąpi jeśli procesy będą wznawiane w takiej kolejności w jakiej zostały zatrzymane.

W przypadku kolejki priorytetowej procesy o wyższym priorytecie mogą zagłodzić procesy o niższym priorytecie.

Wykrycie zagłodzenia jest trudniejsze od zjawiska blokady.

9. **Co to jest sekcja krytyczna i z jakimi problemami się ona wiąże**.

Procesy współbieżne mogą ze sobą konkurować o dostęp do wspólnych zasobów tj. takich, które mogą być wykorzystywane przez jeden proces lub ograniczoną ich liczbę.

Zasobem dzielonym nazywa się wspólny obiekt, z którego w sposób wyłączny może korzystać wiele procesów.

Sekcją krytyczną nazywa się ten fragment procesu, w którym korzysta się z zasobu dzielonego (między programami – pamięć dyskowa, między wątkami – pamięć operacyjna)

Proces wykonujący swoją sekcję krytyczną uniemożliwia wykonanie sekcji krytycznych innych procesów.

10. **Opisać problem wzajemnego wykluczenia.**

Problem wzajemnego wykluczenia

Zsynchronizować N procesów, z których każdy w nieskończonej pętli na przemian zajmuje się własnymi sprawami i wykonanie sekcję krytyczną, w taki sposób, aby wykonanie sekcji krytycznych dwóch lub więcej procesów nie pokrywało się w czasie.

Rozwiązanie powyższego problemu wymaga wprowadzania dodatkowych instrukcji poprzedzających (protokół wstępny) oraz występujących tuż po zakończeniu sekcji krytycznej (protokół końcowy).

Protokoły te są implementacją stosowanej w życiu zasady uprzejmości.

11. Opisać problem producentów i konsumentów.

W systemie działa P>0 procesów, które produkują pewne dane oraz K>0 procesów, które odbierają dane od producentów.

Między producentami a konsumentami znajduje się bufor o pojemności B, którego zadaniem jest równoważenie chwilowych różnic w czasie działania procesów.

Konsument oczekuje na pobranie danych w sytuacji, gdy bufor jest pusty. Gdybyśmy pozwolili konsumentowi odbierać dane z bufora zawsze, to w sytuacji, gdy nikt jeszcze nic w buforze nie zapisał, odebrana wartość byłaby bezsensowna.

Producent umieszczając dane w buforze nie nadpisywał danych już zapisanych, a jeszcze nie odebranych przez żadnego konsumenta. Wymaga to wstrzymania producenta w sytuacji, gdy w buforze nie ma wolnych miejsc.

Jeśli wielu konsumentów oczekuje, aż w buforze pojawiają się jakieś dane oraz ciągle są produkowane nowe dane, to każdy oczekujący konsument w końcu coś z bufora pobierze. Nie zdarzy się tak, że pewien konsument czeka w nieskończoność na pobranie danych, jeśli tylko ciągle napływają one do bufora.

Jeśli wielu producentów oczekuje, aż w buforze będzie wolne miejsce, a konsumenci ciągle co z bufora pobierają, to każdy oczekujący producent będzie mógł coś włożyć do bufora. Nie zdarzy się tak, że pewien producent czeka w nieskończoność, jeśli tylko ciągle z bufora jest coś pobierane.

Przykład tego problemu jest zapis danych do bufora klawiatury przez sterownik klawiatury i ich odczyt jest przez system operacyjny.

12. Jakie znasz warianty problem producentów i konsumentów.

Rozpatruje się różne warianty tego problemu:

- Bufor może być nieskończony
- Bufor cykliczny może mieć ograniczoną pojemność
- Może w ogóle nie być bufora
- Może być wielu producentów lub jeden
- Może być wielu konsumentów lub jeden
- Dane mogą być produkowane i konsumowane po kilka jednostek na raz
- Dane muszą być odczytywane w kolejności ich zapisu lub nie

13. Opisać problem czytelników i pisarzy.

W systemie działa C>0 procesów, które odczytują pewne dane oraz P>0 procesów, które zapisują te dane.

Procesy zapisujące będziemy nazywać pisarzami, a procesy odczytujące – czytelnikami, zaś moment, w którym procesy mają dostęp do danych, będziemy nazywać pobytem w czytelni.

Jednocześnie wiele procesów może odczytywać dane.

Jednak jeśli ktoś chce te dane zmodyfikować, to rozsądnie jest zablokować dostęp do danych dla wszystkich innych procesów na czas zapisu. Zapobiegnie to odczytaniu niespójnych informacji (na przykład danych częściowo zmodyfikowanych).

```
void Czytelnik() {
        do
        {
                własne_sprawy();
                protokół_wstępny_czytelnika();
                CZYTANIE();
                protokół_końcowy_czytelnika();
        } while (true);
}

void Pisarz() {
        do
        {
                własne_sprawy();
                protokół_wstępny_pisarza();
                PISANIE();
                protokół_końcowy_pisarza();
        } while (true);
}
```

Należy tak napisać protokoły wstępne i końcowe poszczególnych procesów, aby:

- Wielu czytelników powinno mieć jednocześnie dostęp do czytelni.
- Jeśli w czytelni przebywa pisarz, to nikt inny w tym czasie nie pisze ani nie czyta.
- Każdy czytelnik, który chce odczytać dane, w końcu je odczyta.
- Każdy pisarz, który chce zmodyfikować dane, w końcu je zapisze.

Rozpatruje się różne warianty tego problemu:

- W czytelni może przebywać dowolnie wielu czytelników
- Czytelnia może mieć ograniczoną pojemność.
- Pisarze mogą mieć pierwszeństwo przed czytelnikami (ale wtedy rezygnujemy z żywotności czytelników)
- Czytelnicy mogą mieć pierwszeństwo nad pisarzami (ale wtedy rezygnujemy z żywotności pisarzy)

14. Opisać problem ucztujących filozofów.

Pięciu filozofów siedzi przy stole i każdy wykonuje jedną z dwóch czynności – albo je, albo rozmyśla

Stół jest okrągły, przed każdym z nich znajduje się miska ze spaghetti, a pomiędzy każdą sąsiadującą parą filozofów leży widelec, a więc każdy osoba ma przy sobie dwie sztuki – po swojej lewej i prawej stronie.

Jedzenie potrawy jest trudne przy użyciu jednego widelca, zakłada się, że każdy filozof korzysta z dwóch.

Nie ma możliwości skorzystania z widelca, który nie znajduje się bezpośrednio przed daną osobą.

Problem ucztujących filozofów jest czasami przedstawiany przy użyciu ryżu, który musi być jedzony dwiema pałeczkami.

Gdy filozofowie nigdy nie rozmawiają ze sobą istnieje zagrożenie zakleszczenia w sytuacji, gdy każdy z nich zabierze lewy widelec i będzie czekał na prawy (lub na odwrót).

Może wystąpić również zagłodzenie, gdy zostanie wzięta pod uwagę kwestia czasu oczekiwania filozofa na dwa wolne widelce.

Należy tak napisać protokoły wstępne i końcowe, aby:

- Jednocześnie tym samym widelcem jadł co najwyżej jeden filozof.
- Każdy filozof jadł zawsze dwoma (i zawsze tymi, które leżą przy jego talerzu) widelcami.
- Żaden filozof nie umarł z głodu.

15. Opisać scentralizowany model współbieżności.

Model scentralizowany

- Zmienne globalne i problemy z nimi
- Niepodzielność instrukcji wysokopoziomowych
- Mechanizmy synchronizacyjne

Podstawową cechą modelu scentralizowanego jest możliwość stosowania zmiennych, które są dostępne do zapisu i odczytu przez wiele procesów. Takie zmienne będziemy nazywać zmiennymi dzielonymi lub zmiennymi współdzielonymi.

Istnienie takich zmiennych w kontekście wykonujących się jednocześnie procesów może powodować pewne problemy. Trzeba na przykład odpowiedzieć na pytanie: co się dzieje, jeśli dwa procesy w tym samym momencie próbują zmodyfikować tę samą komórkę pamięci?

Z reguły taki konflikt jest rozstrzygany przez układ sprzętowy, noszący nazwę arbitra pamięci, który w jakiś sposób szereguje takie żądania. Mamy więc gwarancję, że na skutek wykonania takiego "jednoczesnego" przypisania, zmienna dzielona będzie miała jedną z wartości, które próbowano ustawić, a nie jakąś przypadkową nieznaną nam wartość.

Pisząc program współbieżny nie wolno nic założyć o niepodzielności instrukcji języka, w którym ten program tworzymy.

Model scentralizowany jest trudniejszy dla programisty. Trudniej jest zapewnić poprawność programu współbieżnego korzystającego ze zmiennych współdzielonych. Na dalszej części wykładu omówione będą mechanizmy synchronizacyjne rozwiązujące ten problem. Wśród nich omówione będą semafory, monitory i muteksy.

16. Opisać rozproszony model współbieżności.

Model rozproszony

- Komunikacja synchroniczna
- Komunikacja asynchroniczna
- Modele przekazywania komunikatów

Model rozproszony stosuje się najczęściej wtedy, gdy procesy składające się na program współbieżny wykonują się na różnych komputerach. Nie ma wtedy fizycznie miejsca, w którym można by składować zmienne współdzielone. Z tego powodu procesy w tym modelu komunikują się wyłącznie poprzez wymianę komunikatów.

O modelu rozproszonym można mówić również w sytuacji, gdy procesy działają na tym samym komputerze, ale ich przestrzenie adresowe tworzą logicznie odrębne jednostki i domyślnie procesy

nie mogą współdzielić zmiennych globalnych. Takie środowisko wykonania procesów udostępnia m.in. system operacyjny Unix.

W modelu rozproszonym możemy mówić o komunikacji synchronicznej i asynchronicznej. Różne są też modele przekazywania/adresowania komunikatów oraz identyfikacji procesów.

17. Jak jest różnica między komunikacją synchroniczną i asynchroniczną

Komunikacja synchroniczna

Z komunikacją synchroniczną mamy do czynienia wtedy, gdy chcące się ze sobą skomunikować procesy są wstrzymywane do chwili, gdy komunikacja będzie się mogła odbyć. Omówmy ten schemat na przykładzie procesów, z których jeden (zwany nadawcą) chce wysłać komunikat, a drugi (odbiorca) chce komunikat odebrać.

Jeśli odbiorca chce otrzymać komunikat od nadawcy, to wstrzymuje swoje działanie, aż nadawca będzie chciał go wysłać. Najczęściej oznacza to po prostu oczekiwanie do chwili, aż sterowanie w procesie nadawcy dojdzie do instrukcji wysyłania.

Podobnie dzieje się, jeśli nadawca chce wysłać komunikat do odbiorcy. Czeka wówczas, aż odbiorca będzie gotowy do odebrania tego komunikatu i dopiero go nadaje.

Zauważmy, że ponieważ procesy oczekują na siebie, to przesłanie komunikatu może się odbyć bezpośrednio między zainteresowanymi, tzn. nie potrzebny jest żaden bufor, który przechowa wysłany komunikat do momentu jego odbioru przez odbiorcę.

Komunikacja asynchroniczna

Komunikacja asynchroniczna nie wymaga współistnienia komunikujących się procesów w tym samym czasie. Polega na tym, że nadawca wysyła komunikat nie czekając na nic. Komunikaty są buforowane w jakimś miejscu (odpowiada za to system operacyjny lub mechanizmy obsługi sieci) i stamtąd pobierane przez odbiorcę.

Mamy więc następujący schemat:

- Nadawca wysyła komunikat nie czekając na nic. Komunikat wędruje do bufora.
- Odbiorca odbiera komunikat z bufora. Jeśli bufor jest pusty, to odbiorca jest wstrzymywany (wersja blokująca) lub przekazuje w wyniku błąd (wersja nieblokująca).

18. Wymień modele przekazywania komunikatów.

W modelu rozproszonym istnieje też wiele sposobów przekazywania komunikatów:

Komunikat może być przeznaczony dla konkretnego procesu. Nadawca zna nazwę odbiorcy, a odbiorca zna nazwę nadawcy. Mówimy wtedy o identyfikacji symetrycznej.

Komunikat może być przeznaczony dla konkretnego procesu. Jedna ze stron zna nazwę procesu, druga nie wie kogo obsługuje. Mówimy wtedy o identyfikacji asymetrycznej. Jest to model charakterystyczny dla architektury klient-serwer.

Komunikat może być wysyłany po kanale komunikacyjnym. Wysyłający zna nazwę kanału, ale nie wie, do kogo trafi komunikat, bo nie musi wiedzieć z kim jest ten kanał połączony.

Komunikat może być wstawiany do konkretnego bufora, z którego może go odebrać dowolny proces.

19. Podać definicję definicja klasyczna semafora ogólnego.

Definicja ta została podana przez Dijkstrę i zakłada ona, że semafor jest zmienną całkowitą z dwoma operacjami.

Jako Semaphore będzie dalej oznaczony typ odpowiadający semaforowi ogólnemu.

Opuszczenie semafora (wait lub P)

- Czekaj, aż S>0, S=S-1

Podniesienie semafor (signal lub V)

- S=S+1

Definicja ta nie spełnia warunku niepodzielności

20. Podać praktyczną definicję semafora ogólnego.

Podniesienie

If (są procesy wstrzymane podczas opuszczania S)
    Wznów jeden z nich;
Else
    S=S+1;

Opuszczenie

If (S>0)
    S=S-1;
Else
    Wstrzymaj;

Spełnia warunek niepodzielności

21. Wymienić typy semaforów oraz krótko je scharakteryzować.

Semafor binarny – zmienna semaforowa przyjmuje tylko wartości **true** (stan podniesienia, otwarcia) lub **false** (stan opuszczenia, zamknięcia). Przyjmuje się, że wielokrotne podnoszenie takiego semafora nie zmieni jego stanu – skutkiem będzie zawsze stan otwarcia. W niektórych systemach próba podniesienia otwartego semafora może doprowadzić do błędu.

Semafor ogólny (zliczający) – zmienna semaforowa przyjmuje wartości całkowite nieujemne , a jej bieżąca wartość jest zmniejszana lub zwiększana o jeden w wyniku wykonania odpowiednio operacji opuszczenia lub podniesienia semafora. Wartością tego semafora jest liczba operacji podniesienia . Liczba operacji opuszczania semafora powinna być równa liczbie operacji podnoszenia.

Semafor uogólniony – semafor zliczający , w przypadku, którego zmienną semaforową można zwiększać lub zmniejszać o dowolną wartość , podana jako argument operacji . Zmienna semaforowa musi być nieujemna . Jeśli w wyniku wykonanej operacji zmienna semaforowa przyjmie wartość ujemną proces zostanie zablokowany

Semafor dwustronnie ograniczony – semafor ogólny , w przypadku którego zmienna semaforowa, oprócz dolnego ograniczenia wartością 0 , ma górę ograniczenie , podane przy definicji semafora

22. **Podać praktyczną i klasyczną definicję semafora binarnego.**

Klasyczna :

Opuszczenie semafora (wait lub P od passeren)

*czekaj aż S= true , S=false

Podniesienie semafora (signal lub V od vrijmaken)

* S= true

Praktyczna :

- Podniesienie

if (j sa procesy wstrzymane podczas opuszczenia S):
    wznów jeden z nich
else:
    S =true

- Opuszczenie

if(S == true):
    S = false
else:
    wstrzymaj;

- Spełnia warunek niepodzielności

23. **Podać definicję definicja klasyczna i praktyczną semafora ogólnego.**

Klasyczna :

- Definicja ta została podana przez Dijkstrę i zakłada ona, że semafor jest zmienną całkowitą z dwoma operacjami.
- Jako Semphore będzie dalej oznaczony typ odpowiadający semaforowi ogólnemu.
- Opuszczenie semafora (wait lub P)
    - czekaj aż S>0 , S=S-1
- Podniesienie semafora (signal lub V)
    - S=S+1
- Definicja ta nie spełnia warunku niepodzielności

Praktyczna :

- Podniesienie

```
if (są procesy wstrzymane podczas opuszczania S)
        wznów jeden z nich;
else
        S=S+1;
```

- Opuszczenie

```
if (S>0)
        S=5-1;
else
        wstrzymaj;
```

- Spełnia warunek niepodzielności

24. Podać praktyczną i klasyczną definicję semafora binarnego.- było w 22

25. Podać definicję semafora dwustronnie ograniczonego.

Niech S jest zmienną całkowitą S e [0,N|

- **Podniesienie**

```
if (S==N)
        wstrzymaj;
else if (są procesy wstrzymane podczas opuszczania S)
        wznów jeden z nich;
else S = S+1;
```

- Opuszczenie

```
if (S==O)
        wstrzymaj;
else if (są procesy wstrzymane podczas opuszczania S)
        wznów jeden z nich;
else S = S-1;
```

26. Zapisać problem wzajemnego wykluczenia z użyciem semaforów.

Wzajemne wykluczenie (mutex) — jest stosowane w celu uniknięcia równoczesnego użycia wspólnego zasobu (np. zmiennej globalnej) przez różne wątki/procesy w częściach kodu zwanych sekcjami krytycznymi.

```
const int N = ...; // liczba procesów
BinarySemaphore S = true;

void P(int procid)
{
    while (true)
    {
        własne_sprawy();
        PB(S);  //  opuszczenie semafora binarnego
        sekcja_krytyczna();
        VB(S); //  podniesienie semafora binarnego
    }
}
```

27. Zapisać problem 1 producent – 1 konsument z użyciem semaforów.

Producenci i konsumenci - problem ograniczonego buforowania w komunikacji międzyprocesowej.

```
const int N = ...; // rozmiar bufora
Semaphore wolne = N;
Semaphore pełne = 0;
Produkt bufor[N];

void Producent()
{
    Produkt p;
    int j = 1;
    while (true)
    {
        produkuj(&p);
        P(wolne)
        bufor[j-1] = p;
        j = j % N + 1;
        V(pełne);
    }
}
```

```
void Konsument()
{
    Produkt p;
    int k = 1;
    while (true)
    {
        P(pełne)
        p = bufor[k-1];
        k = k % N + 1;
        V(wolne);
        konsumuj(p);
    }
}
```

Producent produkuje produkt p i czeka aż semafor wolne zostanie podniesiony gdy tak się stanie umieszcza produkt (lub produkty) w buforze . Po umieszczeniu podnosi semafor pełne.

Konsument czeka aż semafor pełne zostanie podniesiony , gdy tak się stanie pobiera z bufora produkt(lub produkty) , następnie podnosi semafor wolne i konsumuje produkt

28. Zapisać problem wiele producentów – wiele konsumentów z użyciem semaforów.

```
Semaphore wolne = N, pełne = 0;
Produkt bufor[N];
int j = 1, k = 1;  //  zmienne określające skąd dane są
pobierane
BinarySemaphore ochrona_j = true, ochrona_k = true;

void Producent(int idProducenta)
{
    Produkt p;
    while (true)
    {
        produkuj(&p);
        P(wolne)
        PB(ochrona_j);
        bufor[j-1] = p;
        j = j % N + 1;
        VB(ochrona_j);
        V(pełne);
    }
}
```

```
void Konsument(int idKonsumenta)
{
    Produkt p;
    while (true)
    {
        P(pełne)
        PB(ochrona_k);
        p = bufor[k-1];
        k = k % N + 1;
        VB(ochrona_k);
        V(wolne);
        konsumuj(p);
    }
}
```

Producenci czekają aż semafor wolne zostanie podniesiony , następnie aby do bufora produkty dodawał tylko jeden producent zostaje zamknięty semafor binarny ochrona-j – producent który został przepuszczony tym semaforem umieszcza produkty w buforze , gdy skończy otwiera semafor ochrona_j aby inni producenci również mogli dodać swoje produkty . Po wypełnieniu bufora zostaje otwarty semafor pełne.

Konsumenci czekają na otwarcie semafora pełne gdy to nastąpi rozpoczynają pobieranie produktów z bufora , aby nie pobierali naraz tych samych produktów został użyty semafor binarny ochrona_k który zapewnia dostęp do bufora tylko jednemu konsumentowi naraz . Gdy konsument pobierze produkt otwiera semafor dla kolejnego klienta . Po pobraniu produktów z bufora zostaje otwarty semafor wolne. Po tym konsumenci konsumują produkty.

29. Zapisać problem czytelników i pisarzy z użyciem semaforów.

```
const int M = ...; // liczba czytelników
const int P = ...; // liczba pisarzy
Semaphore wolne = M;
BinarySemaphore W = true;

void Czytelnik(int idCzytelnika)
{
    while (true)
    {
        wlasne_sprawy();
        P(wolne);
        czytanie();
        V(wolne);
    }
}
```

```
void Pisarz(int idPisarza)
{
    while (true)
    {
        wlasne_sprawy();
        PB(W);
        for (int i = 1; i <= m; ++i) {
            P(wolne);
        }
        pisanie();
        for (int i = 1; i <= m; ++i) {
            V(wolne);
        }
        VB(W);
    }
}
```

Tworzymy semafor wolne o rozmiarze równym ilości czytelników oraz semafor binarny W odpowiadający za pisarza.

Czytelnik zajmuje się własnymi sprawami , gdy semafor wolne jest otwarty czytelnik rozpoczyna czytanie – gdy je zakończy otwiera z powrotem semafor .

Pisarz zajmuje się własnymi sprawami , jeśli semafor binarny odpowiedzialny za pisarza jest otwarty pisarz zamyka semafor wolne tak aby żaden czytelnik nie wszedł do czytelni , następnie rozpoczyna pisanie a po jego zakończeniu zwalnia wszystkie miejsca w semaforze wolne oraz otwiera semafor binarny W.

Możliwe jest zagłodzenie pisarza - nigdy nie wejdzie do czytelni.

30. **Zapisać problem pięciu filozofów z możliwością blokady stosując w zapisie semafory.**

```
BinarySemaphore widelec[] = {true, true, true, true, true};

void Filozof(int i)
{
    while (true)
    {
        myślenie();
        PB(widelec[i]);
        PB(widelec[(i+1)%5]);
        jedzenie();
        VB(widelec[i]);
        VB(widelec[(i+1)%5]);
    }
}
```

Tworzymy tablice semaforów binarnych odpowiadające za widelce.

Każdy filozof myśli , jeśli chce zjeść to blokuje semafor odpowiadający za jego widelec oraz widelec filozofa obok . Następnie rozpoczyna jedzenie . Po skończeniu odkłada oba widelce (otwiera semafory)

Istnieje możliwość zakleszczenia - wszyscy filozofowie podniosą naraz jeden widelec .

31. **Zapisać problem pięciu filozofów w wariancie z lokajem stosując w zapisie semafory.**

```
BinarySemaphore widelec[] = {true, true, true, true, true};
Semaphore lokaj = 4;

void Filozof(int idFilozofa)
{
    while (true)
    {
        myślenie();
        P(lokaj);
        PB(widelec[i]);
        PB(widelec[(i+1)%5]);
        jedzenie();
        VB(widelec[i]);
        VB(widelec[(i+1)%5]);
        V(lokaj);
    }
}
```

Ponownie tworzymy semafory binarne odpowiadające za widelce , następnie tworzymy semafor odpowiedzialny za lokaja . gdy dany filozof chce zjeść blokuje semafor lokaja a następnie blokuje oba widelce . gdy zje odkłada je , otwierając semafory oraz podnosi semafor lokaja .

32. **Podać przykład rozwiązanie problemu równoległego z zastosowaniem dekompozycji danych.**

Dekompozycja danych - podział danych na mniejsze porcje .

Przykład wyliczenie iloczynu wektorów - dekompozycja wejsciowa

```
static Task<int> LiczPart(int part, int num, int[] v1, int[] v2)
{
    var task = new Task<int>(() => {
        int suma = 0;
        for (int i = part*num; i < part*(num+1); i++)
        {
            suma += v1[i] * v2[i];
        }
        return suma;
    });
    task.Start();
    return task;
}
```

```
static void task_part(int[] v1, int[] v2, int ile)
{
    int ostateczna = 0;
    List<Task<int>> temp = new List<Task<int>>();
    int part = v1.Length / ile;
    for (int x = 0; x < ile; x++)
    {
        var temp2 = LiczPart(part, x, v1, v2);
        temp.Add(temp2);
    }

    foreach (Task<int> x in temp)
    {
        ostateczna += x.Result;
    }

}
```

Dekompozycja polega na podziale wektorów na równe części . Każdy wątek wylicza iloczyn wektorowy z danej części . Gdy wszystkie wątki zakończą prace ich wynik jest sumowany dzięki czemu otrzymujemy iloczyn całkowity obu wektorów .

Przykład 2 - dekompozycja wyjściowa - obliczenie iloczynu dwóch macierzy

Dekompozycja polega na podziale macierzy na sektory . Każdy wątek wylicza iloczyn danego sektora co daje nam pewna częśc macierzy wynikowej . Po wykonaniu pracy wszystkich wątków otrzymujemy wyniki z wszystkich sektorów które łączymy otrzymując macierz wynikową

33. **Podać przykład rozwiązanie problemu równoległego z zastosowaniem dekompozycji funkcjonalnej.**

Dekompozycja funkcjonalna - podział na zadania które realizuje odpowiednie funkcje.

Przykład :Sito Erastotenesa

Problem wyznaczania liczb pierwszych z przedziału [2..n], dla pewnej s liczby naturalnej n.

Wykonywanie algorytmu sita Eratostenesa kończy się z chwilą wykonania wszystkich zadań oraz znalezienia, przez przedostatnie zadanie, liczby pierwszej ry, spełniającej warunek rę > n albo inaczej nr, > yn(w przykładzie ry, = 7). Wynikiem rozwiązania problemu są wszystkie niewykreślone liczby z przedziału [2..n].

Pierwszy wątek programu znajduje liczbę pierwszą i wykreśla ją z listy - następnie wykreśla wszystkie jej wielokrotności . Następny wątek znajduje kolejna liczbe pierwsza - która jest nie wykreślona po czym postępuje tak samo wykreślając wszystkie jej wielokrotności .

Dekompozycja polega na tym że każdy wątek wykreśla wielokrotności liczby pierwszej której nie wykreśla żaden inny wątek - w tym przykładzie należy zapewnić komunikację między wątkami aby nie doszło do wykreślenia tych samych liczb przez dwa lub więcej wątków naraz .

34. Podać przykład rozwiązanie problemu równoległego z zastosowaniem dekompozycji spekulatywnej.

Dekompozycja spekulatywna - polega na zbadaniu kilku ścieżek a następnie porównaniu która z nich jest najbardziej optymalna.

Przykład - algorytm pełzającego sympleksu

Algorytm posiada punkt (wektor) początkowy od którego rozpoczyna szukanie wartości minimalnej - problem algorytmu polega na tym że możemy znaleźć minimum lokalne a nie globalne w tym celu używamy kilku punktów (wektorów) początkowych.

Dekompozycja polega na tym że każdy wątek szuka minimum z innego wektora początkowego . Następnie z danych zebranych ze wszystkihch watków wybieramy najmniejsza otrzymana z algorytmu wartość która traktujemy jako minimum .

35. Podać przykład rozwiązanie problemu równoległego z zastosowaniem dekompozycji rekursywnej.

Dekompozycja rekursywna - polega na podziale problemu na zbór podobnych niezależnych podproblemów , które będą mogły być rozwiązane przez podobną dekompozycje rekursywnie

Przykład - Algorytmy sortowania oparte na strategii dziel i zwyciężaj np sortowanie przez scalanie , quicksort

Zbiór liczb do posortowania dzielimy na dwie części - jeden wątek jedna część . Następnie w każdej części wykonujemy rekurencyjnie dzielenie zbioru na pół aż dojdziemy do wektora dwuelementowego - który jest sortowany . Następnie scalamy wszystkie posortowane podzbiory dzięki czemu otrzymujemy posortowany posortowany zbiór.

36. Opisać prawo Amdahla.

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{p}{1+(p-1)f}$$

Prawo to jest stosowane do do znalezienia maksymalnego przyspieszenia w programach równoległych . Określa czas przyśpieszenia ze względu na ilość wątków a zarazem czas opóźnienia który powstaje w momencie utworzenia nowego wątku/procesu . Stosując to prawo możemy dowiedzieć się ile wątków użyć w programie aby otrzymać maksymalne przyśpieszenie .

W rzeczywistych problemach skalowalność jest gorsza od wynikającej z prawa Amdanhla.

- Koszty komunikacji procesorów (rosną wraz ze wzrostem ich liczby)
- Niezrównoważenie obciążenia (ang. load imbalance)
- Potrzeba duplikacji obliczeń na różnych procesorach.

37. Co to jest przyspieszenie superliniowe i w jakich warunkach jest możliwe.

- Przyspieszenie definiujemy jako $S(p)=T_s/T_p(p)$.

Ts -> czas pracy wersji szeregowej programu

Tp -> czas funkcji procesorów p

Przyspieszenie większe od p jest przyspieszeniem super liniowym

Przyspieszenie superliniowe jest wtedy czas gdy wykonania jednego wątku w zadaniu równoległym jest krótszy niż czas wykonania programu sekwencyjnego.

W jakich warunkach :

- Gdy korzystamy z pamięci podręcznej casch
- Może wynikać z struktury algorytmu - np. przeszukiwanie w głąb (w drzewie szukamy pewnego wierzchołka - algorytm kończy się gdy go znajdziemy, korzystając z kilku procesów każdy z nich podaży inną ścieżki z tego samego punktu startowego dzięki czemu możliwe jest odnalezienie wierzchołka bardzo szybko podczas gdy program sekwencyjny czasami musi przeszukać całe drzewo żeby go znaleźć)

38. Miary efektywności w przetwarzaniu współbieżnym.

- Czas pracy (ang. wall clock time). Czas mierzony od momentu startu pierwszego procesora do momentu gdy ostatni procesor w grupie rozwiązującej problem zakończy pracę.
- Przyspieszenie (ang. speedup): ile razy wersja równoległa jest szybsza.
- Wydajność jest miarą użytecznego wykorzystania procesora. Definiuje się ją jako stosunek przyspieszenia do idealnego przyspieszenia liniowego równego p:

$$E(p)=\frac{S(p)}{p}=\frac{T_s}{p*T_p(p)}$$

- Koszt odzwierciedla sumę czasu spędzonego przez wszystkie procesory pracujące nad rozwiązaniem problemu.

$$C(p) = p * T_p(p)$$

39. Opisać metody realizacji obliczeń równoległych i rozproszonych w językach C/C++.

- biblioteka unistd.h - polecenia fork() , exec() , pipe() , wait()
- biblioteka zarządzania wątkami pthread.h - pthread_create() , pthread_join() , ptrhread_exit()
- biblioteka openMPI - MPI_Init() , MPI-Send , MPI_Recv
- OpenMP - opiera sie na dyrektywach pragma np #pragma omp parallel for
- pamięć współdzielona IPC - shget()- utworzenie segmentu ,shmat() - dodanie segmentu , shmdt()- odłączenie segmentu,shmctl()-usunięcie segmentu,

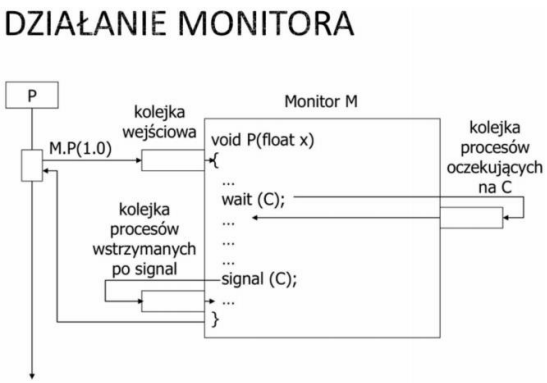
40. Opisać metody realizacji obliczeń równoległych i rozproszonych w technologii .NET.

- Thread - wątek , Task - obiekty które realizują pewne zadania , TaskWaitAll() , TaskWaitAny()
- Parallel for - pętla tworzące wątek przy każdej iteracji - może zawierać sekcje lock która jest miejscem synchronizacji danych
- async - funkcje asynchroniczne , await
- paralizacja zapytań LINQu
- biblioteka MS MPI

41. Wyjaśnić koncepcję i realizację monitorów.

- Semafor nie jest mechanizmem strukturalnym
- Trudno jest analizować programy współbieżne i ogarnąć wszystkie możliwe przeploty
- Mogą dojść błędy wynikające z niezamierzonej zamiany ze sobą operacji P i V.
- Monitor stanowi połączenie modułu programistycznego z sekcją krytyczną.
- Monitor jest modułem zawierającym deklaracje stałych, zmiennych, funkcji.
- Wszystkie te elementy, za wyjątkiem jawnie wskazanych funkcji, są lokalne w monitorze.

```
class Monitor
{
private:
  definicje stałych i typów;
  deklaracje lokalnych pól;
  deklaracje i definicje funkcji;
public:
    Monitor()
    {
        instrukcje inicjujące monitor;
    }
}
```



42. Implementacja monitorów w C#.

public static class Monitor

- public static void Enter (Object obj) — wejście do monitora

- public static void Exit (Object obj) — wyjście z monitora
- public static bool Wait (Object obj) — oczekiwanie
- public static void Pulse (Object obj)- sygnalizacja
- public static void PulseAll (Object obj) - sygnalizacja do wszystkich

43. Implementacja monitorów w języku C/C++.

Monitor to obiekt którego dane są współdzielone między wątkami a jego metody są ograniczone mutexami tak aby dostęp do nich miał tylko jeden wątek - w języku c++ nie mamy obiektów Monitor , musimy je utworzyć ręcznie za pomocą semaforów .

```
class Monitor
{
        int suma;
        Mutex m;
        funkcja jakaśtam()
        {
                mutex.lock()
                //tutaj coś sie robi
                mutex.open()
        }
}
```

44. Wyjaśnić koncepcję równoważenia obciążenia (ang. Load Balancing).

Każdy proces wykonuje prace wymagającej tyle samo obliczeń - nie występuje sytuacja że procesy zakończą prace a potem czekają na zakończenie pracy innych - procesy tak samo obciążone powinny kończyć pracę w tym samym momencie .

45. Co to jest rozdrobnienie obliczeń.

- Dekompozycja gruboziarnista (ang. coarse grained) - problem podzielony na niewielką liczbę stosunkowo dużych zadań- np iloczyn wektorów na dwóch taskach
- Dekompozycja drobnoziarnista (ang. fine grained) - problem podzielony na dużą liczbę niewielkich zadań - np iloczyn wektorów w parallel for

Żadna z dekompozycji nie jest lepsza od drugiej - to której należy użyć zależy od problemu zadania.

46. Omówić systemy kolejkowania RabbitMQ

RabitMQ jest Message broker akceptuje oraz przekierowuje wiadomości.

Podstawowe elementy :

- Producentem jest program, który wysyła wiadomości.
- Konsumentem jest program, który oczekuje na odebranie wiadomości.
- Kolejka to kontener („skrzynka na listy”) w postaci bufora, która zawiera wiadomości przesyłane pomiędzy producentami a konsumentami.

Wielu producentów może wysyłać wiadomości do jednej kolejki, a także wielu konsumentów może próbować odebrać wiadomości znajdujące się w niej.

Producent oraz konsument w większości przypadków nie są umieszczeni na tym samym hoście (maszynie) jednakże aplikacja może pracować jednocześnie zarówno jako producent i konsument.

Dodatkowe elementy

- wiadomość zwrotna - informuje o otrzymaniu wiadomości
- flagi durable - zapewniają trwałość wiadomości i kolejek , w razie zaprzestania działania brokera wiadomości nie zostaną stracone
- Fair Dispatch pozwala na zmianę domyślnej metody przydzielania wiadomości do danego konsumenta. Dzięki wykorzystaniu metody BasicQos można sterować ilością przydzielonych jednocześnie wiadomości danemu klientowi.

Exchange jest mechanizmem pośredniczącym w transferze informacji. Odbiera wiadomości wysyłane przez producentów i kieruje je do wskazanych kolejek na podstawie danych zawartych w publikacji - producent wysylajac wiadomosc nie wie do jakiej kolejki ja wysyła

Kolejki tymczasowe sprawdzają się, gdy zawsze po połączeniu z Rabbit potrzebujemy świeżej, pustej kolejki natomiast po rozłączeniu konsumenta kolejka ma zostać usunięta. Możemy tworzyć nietrwałą, automatycznie kasowaną kolejkę

Binding jest to relacja pomiędzy exchange a kolejką określająca powiązanie kolejki z danym exchange.

Exchange może działać na kilka sposobów:

- Direct -kieruje wiadomości do kolejek, których routingKey bindingu jest identyczny z routingKey przesyłanej wiadomości.
- Topic - jest rozszerzeniem Direct Exchange w którym to binding routingKey nie jest jednym słowem lecz listą słów pozwalających na zdefiniowanie nie 1 lecz wielu kryteriów przekierowania wiadomości.
- Headers
- Fanout -kieruje wiadomości do wszystkich kolejek o istnieniu których ma informacje
- Default -zawsze kieruje wiadomości do kolejki o nazwie wprowadzonej do pola routingKey.

RPC (Remote Procedure Call) jest to zdalne wywołanie procedury. W przypadku brokera wiadomości RabbitMQ wykorzystywane jest do zadysponowania poprzez wiadomość zadania do wykonania przez odbiorcę, a następnie po określonym czasie odebrania z kolejki zwrotnej uzyskanego wyniku.

- Każdy klient tworzy po swojej inicjalizacji wyłączną kolejkę zwrotną.
- Każdy request, który klient wysyła do serwera identyfikowany jest przez 2 właściwości: ReplyTo oraz CorrelationId.
- Requesty wysyłane są do kolejki jednostki wykonawczej RPC (servera). 38
- Kiedy server odczyta wiadomość, przystąpi do wykonywania zdefiniowanego w requeście zadania.
- Po zakończeniu pracy server wysyła na podstawie parametru ReplyTo wyniki do kolejki zwrotnej konkretnego klienta.
- Klient sprawdza swoją kolejkę i dopasowuje otrzymane wiadomości do konkretnego requesta na podstawie Correlation Id

47. Omówić zastosowanie biblioteki OpenMp

OpenMP jest zadeklarowane w kompilatorze - nie jest biblioteka .

Wykorzystuje dyrektywy #pragma do zrównoleglenia funkcji

`#pragma omp nazwa_funkcji [opcje]`

`#pragma omp parallel` –tworzy tyle wątków ile ma procesor

`shared(count)` - wyznacza zmienna która jest współdzielona między wątkami

`#pragma omp critical` - wyznacza sekcje krytyczną

`#pragma omp atomic` - wyznacza operacje atomową (działanie jak sekcja krtyczna)

`#pragma omp barrier` - deklaracja bariery

`num_threads(3)` - informacje ile wątków ma zostać utworzone

`private (data)` - informacja ze zmienna jest prywatne , kazdy wątek utworzy jej kopie