1 Programowanie współbieżne

Program współbieżny składa się ze skończonej liczby procesów sekwencyjnych. Każdy proces sekwencyjny wykonuje ciąg operacji atomowych.

1.1 Możliwe wykonania programu współbieżnego

1. Równoległe synchroniczne
   - Typowe dla architektur wielordzeniowych
   - Wspólny zegar taktujący
   - Wszystkie procesory wykonują w tym samym momencie kolejną fazę procesora
2. Równoległe asynchroniczne
   - Typowe dla architektur rozproszonych
   - Brak wspólnego zegara
   - Każdy procesor pracuje we własnym tempie
3. W przeplocie

Definicja (Równoległość)**.** Faktycznie jednoczesne wykonywanie wielu czynności.

Definicja (Współbieżność)**.** Abstrakcja równoległości. Wykonanie współbieżne może odbywać się równoległe lub w przeplocie

Definicja (Bezpieczeństwo)**.** Lub inaczej zapewnienie:

- nigdy nie dojdzie do sytuacji niepożądanej
- własność zapewniania pojawia się w specyfikacji problemu synchronizacyjnego
- odpowiednik częściowej poprawności programu sekwencyjnego

Definicja (Żywotność)**.** oznacza:

- każdy proces, który chce wykonać pewną akcję w skończonym czasie będzie mógł to zrobić
- własność żywotności jest związana z wykonaniem programu
- odpowiednik całkowitej poprawności programu sekwencyjnego (razem z własnością bezpieczeństwa)

Definicja (Zakleszczenie (deadlock; globalny brak żywotności))**.** Sytuacja, w której co najmniej dwie różne akcje czekają na siebie nawzajem, więc żadna nie może się zakończyć. Występuje gdy wiele zadań w tym samym czasie konkuruje o wyłączny dostęp do zasobów.

Definicja (Zagłodzenie (lokalny brak żywotności))**.** Proces czeka w nieskończoność, gdyż zdarzenie, na które czeka powoduje zawsze wznowienie innego procesu

Definicja (Uczciwość)**.** Każdy proces gotowy do wykonania stanie się w końcu aktywny. Zakładamy, że system operacyjny jest uczciwy.

2 Mechanizmy synchronizacji

2.1 Semafory

Definicja (Semafor ogólny)**.** Abstrakcyjny typ danych z operacjami:

- inicjacji (od razu przy deklaracji, poza procesami)
- opuszczenia P (próby przejścia przez semafor)
- podniesienia V

Dodatkowo:

- Operacje P i V są niepodzielne i wykluczają się nawzajem
- Nie są dopuszczalne żadne inne operacje na semaforze

2.1.1 Semantyka semafora ogólnego

- Semafor to zmienna całkowitoliczbowa, przyjmująca wartości nieujemne
- Semafor o wartości = 0 jest zamknięty
- Semafor o wartościach większych niż 0 jest otwarty
- Jeśli semafor jest otwarty, to operacja P powoduje zmniejszenie jego wartości o 1
- Operacja P na zamkniętym semaforze powoduje wstrzymanie wykonującego ją procesu na semaforze
- Operacja V powoduje zwiększenie wartości semafora o 1, jeśli nikt nie czeka na semaforze
- Szczegóły działania P i V można definiować na wiele sposobów

2.1.2 Rodzaje semaforów

Definicja (Semafor słaby)**.** działa w następjący sposób:

- ```
  def P(S): # opuszczenie
      if S > 0:
          S := S - 1
      else:
          wstrzymaj wykonujący proces na semaforze
  ```

- ```
  def V(S): # podniesienie
      if jakiś proces czeka na S:
          wznów dowolnie wybrany proces spośród oczekujących na S
      else:
          S := S + 1
  ```

Definicja (Semafor silny)**.** działa w następjący sposób:

- ```
  def P(S):
      if S > 0:
          S := S - 1
      else:
          wstrzymaj wykonujący proces w kolejce procesów związanych z semaforem S
  ```
  ```

- ```
  def V(S):
      if jakiś proces czeka na S:
          wznów pierwszy proces z kolejki oczekujących na S
      else:
          S := S + 1
  ```

Definicja (Semafor silnie uczciwy)**.** działa w następjący sposób:

- ```
  def P(S):
      if S > 0:
          S := S - 1
      else:
          wstrzymaj wykonujący proces na semaforze S
  ```

- ```
  def V(S):
      if jakiś proces czeka na S:
          wznów dowolny proces spośród oczekujących na S
      else:
          S := S + 1
  ```

- Jeśli operacja V będzie wykonana nieskończenie wiele razy, to w końcu każdy proces oczekujący na S zostanie wznowiony
- W dalszych rozważaniach przyjmujemy tę właśnie definicję!

Definicja (Semafor binarny)**.** Semafor, który może przyjmować jedynie wartości: 0 lub 1

```
if jakiś proces czeka na S:
    wznów dowolnie wybrany proces spośród oczekujących na S
else:
    if S = 1:
        błąd
    else:
        S := 1
```

Definicja (Dziedziczenie sekcji krytycznej)**.** Technika rozwiązywania problemów synchronizacyjnych. Proces, który faktycznie kogoś budzi nie podnosi semafora, obudzony proces zastaje zamknięty semafor (dziedziczy sekcję krytyczną) i musi podnieść go w imieniu procesu, który go obudził. Dzięki temu między obudzeniem a faktycznym wznowieniem działania przez proces nic nie może się zdarzyć.

```
binary semaphore mutex = 1;
binary semaphore delay = 0;
int ilu_czeka = 0;
```

```
# protokół wstępny
P(mutex)
if trzeba poczekać:
    ++ilu_czeka
    V(mutex)
    P(delay) # dziedziczenie
    --ilu_czeka
...
V(mutex)
```

```
# protokół końcowy
P(mutex)
...
if można kogoś wznowić and ilu_czeka > 0:
    V(delay)
else:
    V(mutex)
```
```

Definicja (Semafor uogólniony)**.** działa w następujący sposób:

- ```
  def P(S, n):
      czekaj, aż S ≥ n;
      S = S - n
  ```

- ```
  def V(S, n):
      S = S + n
  ```

- PROBLEM: co ma się stać przy próbie wykonania P(S, 2), jeśli S = 3 i jakiś proces oczekuje już na P(S, 4)?

Definicja (Semafor dwustronnie ograniczony)**.** przyjmuje wartości z przedziału $[0, M]$, gdzie M jest wartością podawaną przy inicjalizacji semafora

- ```
  def P(S):
      czekaj, aż S ≥ 0;
      S = S - 1
  ```

- ```
  def V(S):
      czekaj, aż S < M;
      S = S + 1
  ```

- obydwie operacje są blokujące

Definicja (Semafor Agerwali)**.** niech $S_1, \cdots, S_n$ będą semaforami ogólnymi

- Argumentami operacji PA są dwa rozłączne zbiory semaforów $A, B \subseteq \{S_1, \cdots, S_n\}$, $A \cap B = \emptyset$

  ```
  def PA(A, B):
      czekaj aż (dla każdego S ∈ A będzie S > 0 and
                 dla każdego S ∈ B będzie S = 0);
      for S in A:
          S = S - 1
  ```

- Argumentami operacji VA jest zbiór semaforów A

  ```
  def VA(A):
      for S in A:
          S = S + 1
  ```

- zarówno PA jak i PV są niepodzielne i wzajemnie się wykluczają

Definicja (Semafor uniksowy)**.**

- najpierw trzeba utworzyć ZESTAW semaforów (`semget`)
- w zestawie może być wiele semaforóœ
- jednym wywołaniem systemowym `semop` można wykonać w sposób NIEPODZIELNY wiele operacji z tego samego zestawu
- operacje na pojedynczym semaforze to:
  - ogólnione opuszczenie P(S)
  - ogólnione podniesienie V(S)
  - wstrzymanie procesu w oczekiwaniu na zamknięcie semafora (Z(S)?)
- operacja P(S) występuje w wersji blokującej i nieblokującej

- można odczytać wartość semafora i ją zmienić
- operacja P(S) może powodować zagłodzenie
- semantyka operacji jednoczesnych:
  - operacje wykonują się po kolei
  - jeśli danej operacji nie można wykonać, to system wycofuje poprzednie

  to oznacza że zawsze wykonują się wszystkie operacje albo żadna

2.2 Monitor

Definicja (Monitor)**.** moduł programistyczny udostępniający na zewnątrz pewne procedury i funkcje, ukrywający wszystkie zmienne i stałe i typy. W danej chwili tylko jeden proces wykonuje funkcje monitora (JEST W MONITORZE).

Monitor jest SEKCJĄ KRYTYCZNĄ. Jeśli pewien proces wywoła funkcję monitora w czasie, gdy inny proces jest już w monitorze, to zostanie automatycznie wstrzymany. Wzajemne wykluczanie jest na poziomie CAŁEGO monitora, a nie poszczególnych funkcji. Monitor gwarantuje ochronę zmiennych globalnych, które są umieszczone w monitorze (bo dostęp do nich mają tylko funkcje monitora, a w funkcji może być tylko jeden proces na raz).

Definicja (Zmienna warunkowa)**.** Monitor oferuje dodatkowy abstrakcyjny typ danych - zmienne warunkowe (oznaczane typem `condition`). Z każdą zmienną warunkową związany jest zbiór procesów oczekujących na niej. Dostępne są trzy operacje na zmiennych warunkowych:

- `wait(c)` - bezwarunkowe wstrzymanie procesu wykonującego tę operację. Proces opuszcza monitor
- `signal(c)` - obudzenie jednego procesu oczekującego na zmiennej `c`. Wznowiony proces kontynuuje wykonanie od kolejnej instrukcji po `wait(c)`
- `empty(c)` zwraca `true` gdy żaden proces nie czeka na zmiennej

2.2.1 Różne semantyki wait i signal

Semantyka wait

- **Semantyka C.A.R. Hoare**, ze zmiennymi warunkowymi związane są KOLEJKI PROSTE (FIFO). Procesy oczekujące na wejście do monitora także czekają na kolejce prostej.
- **Semantyka języka Mesa**, nie zakłada się budzenia procesów w kolejności ich zawieszania - wznawiany proces jest wybierany przez moduł szeregujący zgodnie z zaimplementowaną strategią, można jednak ZAŁOŻYĆ SILNĄ UCZCIWOŚĆ

Semantyka signal

W związku z tym, że w monitorze może być co najwyżej jeden proces na raz, pojawia się problem gdy `signal(c)` nie jest ostatnią operacją wykonywaną przez proces w monitorze i na zmiennej warunkowej c ktoś czeka. Nie można po prostu obudzić procesu, bo wtedy w monitorze byłyby dwa procesy! Oto najpopularniejsze sposoby rozwiązywania problemu.

- **Semantyka Per Brinch Hansen**, `signal` musi być ostatnią operacją wykonywaną w monitorze, proces sygnalizujący wychodzi z monitora, a jego miejsce zajmuje obudzony proces.
  - Procesy budzone mają priorytet przed oczekującymi na wejście do monitora (WARUNEK NATYCHMIASTOWEGO WZNOWIENIA)

  - Nie zawsze jednak da się rozwiązać dany problem wykonując signal na końcu

- **Semantyka C.A.R. Hoare**, proces sygnalizujący wychodzi z monitora i ustawia się na początku kolejki procesów oczekujących na wejście do monitora, a jego miejsce w monitorze zajmuje obudzony proces. Procesy budzone mają priorytet przed oczekującymi na wejście do monitora (WARUNEK NATYCHMIASTOWEGO WZNOWIENIA).
  - Jeśli `signal` był ostatnią instrukcją wykonaną w monitorze przez proces lub nikt na zmiennej warunkowej nie czekał, to proces sygnalizujący wychodzi z monitora nie czekając (jak w semantyce Brinch Hansena)
  - Gdy obudzony proces wyjdzie z monitora nie budząc nikogo, to jego miejsce zajmie proces, który go obudził - procesy wstrzymane na skutek wykonania `signal` mają pierwszeństwo przed procesami oczekującymi na wejście do monitora
- **Semantyka języka Mesa**, proces sygnalizujący wykonuje się dalej. Po jego wyjściu z monitora wznawiamy dowolny proces: albo oczekujący na wejście albo obudzony z wait.
  - Jest łatwo w implementacji i przez to powszechnie stosowana w rzeczywistych implementacjach
  - Niestety proces wznawiający działanie po wait nie ma gwarancji, że warunek, na który czekał jest nadal spełniony

2.2.2 Semantyka Hoare'a

O ile nie mówimy wprost, że jest inaczej, przyjmujemy semantykę Hoare'a.

- Procesy oczekują na wejście do monitora w kolejce prostej
- Z każdą zmienną warunkową jest związana osobna kolejka prosta
- `signal` na pustej kolejce nie robi nic
- `signal`, który faktycznie budzi jakiś proces, powoduje wstrzymanie procesu sygnalizującego i umieszczenie go na początku kolejki procesów oczekujących na wejście do monitora; w jego miejsce do monitora wraca obudzony proces, który wznawia wykonania od kolejnej instrukcji po `wait`
- `signal` wykonywany jako ostatnia instrukcja w monitorze nie wstrzymuje procesu

3 Dowodzenie żywotności i bezpieczeństwa

3.1 Semaforów

Slajdy 448 - 473

3.2 LTL :o

Slajdy 474 - 785

4 Modele współbieżności

Definicja (Model scentralizowany)**.** procesy wykonują się w środowisku ze wspólną pamięcią, w której w szczególności można umieścić zmienne globalne (współdzielone) (semafory, segmenty pamięci dzielonej)

Definicja (Model rozproszony)**.** synchronizacja odbywa się poprzez wymianę komunikatów. Naturalna w systemach z rodziny Unix, gdzie procesy mają osobne przestrzenie adresowe

4.1 Realizacja wymiany komunikatów

- **Synchroniczna vs asynchroniczna** - z komunikacją synchroniczną mamy do czynienia wtedy, gdy chcące się ze sobą skomunikować procesy są wstrzymywane do chwili, gdy komunikacja będzie się mogła odbyć. Przykładem komunikacji synchronicznej jest na przykład rozmowa telefoniczna. Nadawca przekazuje swój komunikat dopiero wówczas, gdy odbiorca chce go wysłuchać. 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.
- **Symetryczna vs asymetryczna** - procesy znają nawzajem swoje identyfikatory lub tylko jeden proces zna identyfikator drugiego (klient-serwer)
- **Pośrednia vs bezpośrednia**

4.2 SendMessage i GetMessage

- Wprowadzamy nowy, predefiniowany typ danych o nazwie `buffer`. Zmienne tego typu reprezentują bufory, do których wysyłamy i z których odbieramy komunikaty.
- Zmienne buforowe mogą być deklarowane jedynie poza procesami. Jedynymi operacjami na nich są operacja wyłania komunikatu i operacja odebrania komunikatu.
- Procesy nie mają dostępu do zmiennych globalnych, z wyjątkiem zmiennych buforowych.
- Bufory są nieograniczone i działają jak kolejki proste.
- Do wysłania komunikatu do bufora b służy operacja `SendMessage(b, m)`. Wobec nieskończoności bufora jest ona nieblokująca, tzn.: proces nie może zostać wstrzymany na skutek jej wywołania.
- Do odbierania komunikatu z bufora b służy operacja `GetMessage(b, m)`. Proces odbiera wtedy pierwszy komunikat z bufora i umieszcza go w zmiennej m. Typ zmiennej m musi być zgodny z typem odebranego komunikatu, w przeciwnym razie efekt operacji jest nieokreślony.
- Jeśli bufor jest pusty, to operacja `GetMessage` wstrzymuje wykonanie procesu, do momentu aż w buforze znajdą się jakieś dane.
- Jeśli wiele procesów oczekuje na operacji `GetMessage` i w buforze pojawi się jeden komunikat, to jeden z procesów oczekujących jest budzony. Nie zakładamy przy tym nic o kolejności budzenia. Oznacza to, że procesy nie muszą być budzone w kolejności wykonywania `GetMessage`
- Oczekiwanie jest jednak sprawiedliwe. Jeśli jakiś proces oczekuje na `GetMessage`, a bufor jest nieskończenie wiele razy niepusty, to proces ten zostanie w końcu obudzony.
- Zmienna m w powyższych operacjach może być dowolnego typu.
- Operacje `GetMessage` i `SendMessage` na tym samym buforze są niepodzielne i wykluczające się nawzajem.
- Procesy nie mają dostępu do zmiennych globalnych, z wyjątkiem zmiennych buforowych.
- Slajdy 796-809 - przykłady; Smurf

4.3 Linda

Inne podejście do asynchronicznego modelu komunikacji zaproponował w latach osiemdziesiątych Gelernter. Wprowadził on notację o nazwie Linda, która udostępnia tzw. przestrzeń krotek, czyli coś w rodzaju nieskończonego bufora, za pomocą którego porozumiewają się ze sobą procesy. Jest jedna wielka, globalna przestrzeń krotek. Procesy porozumiewają się ze sobą, umieszczając w przestrzeni krotek komunikaty i pobierając je z niej. Procesy nie muszą wiedzieć nic o sobie nawzajem, co więcej, mogą nie istnieć w tym samym czasie. Nadawca może odebrać komunikat od procesu, który już dawno zakończył swoje działanie.

Operacje:

- `void tsPut(char const* format, ...)` - umieszcza w przestrzeni krotkę opisaną formatem `format`. Operacja jest nieblokująca - proces umieszcza krotkę w przestrzeni i wykonuje się dalej

  ```
  tsPut("%d %f %c %d", 5, 3.14, 'c', 20-3);
  tsPut("KROTKA %d", 3*4);
  ```

  Operacja `tsPut` może umieścić w przestrzeni krotkę o nieokreślonej wartości pewnych składowych:

  ```
  tsPut ("%d %f ?c %d", 3.14, r, 2);
  ```

  Taka krotka ma jednoznacznie określoną postać i dziury na pewnych składowych. W operacjach selektywnego wyboru dziury pasują do każdej wartości, ale w przypadku zapisania dziury na zmienną, ma ona nieokreśloną wartość.

- `void tsFetch(char const* format, ...)` - usuwa z przestrzeni krotkę opisaną formatem `format`, a wartości jej składowych przypisuje kolejnym argumentom (jak w funkcji `scanf`). Jeśli krotki o podanym formacie nie ma w przestrzeni krotek, to operacja `tsFetch` wstrzymuje proces. Format fmt konstruujemy jak w operacji wyjścia, ale zamiast znaku `%` używamy znaku `?`.

  ```
  int x, w; double y; char z;
  tsFetch("?d ?f ?c ?d", &x, &y, &z, &w);
  ```

  Nie jest określona kolejność budzenia procesów, ale zakłada się silną uczciwość, to samo z wyjmowaniem pasujących krotek. Elementem formatu w operacji tsFetch może być także pole rozpoczynające się od znaku %. W takiej sytuacji wyjmowana z przestrzeni krotka musi mieć w danym miejscu wskazaną wartość:

  ```
  tsFetch ("%d ?f %c ?d", 5*5, &y, 'a', &w);
  ```

- `void tsRead(char const* format, ...)` działa jak `tsFetch`, ale dodatkowo pozostawia krotkę w przestrzeni krotek

- `bool tsTryFetch(char const* format, ...)` nieblokująca wersja operacji `tsFetch`. Jeśli w przestrzeni jest krotka pasująca do podanego formatu, to funkcja przekazuje w wyniku `true` i krotka jest usuwana z przestrzeni krotek w przeciwnym razie proces nie jest wstrzymywany, funkcja przekazuje w wyniku `false`, a poszczególne elementy parametru są nieokreślone.

- `bool tsTryRead(char const* format, ...)` nieblokująca wersja operacji `tsRead`

4.4 Spotkania (randki) w Adzie

Procesy porozumiewają się ze sobą za pomocą komunikatów, które mogą mieć nazwy (ale nie muszą!) Komunikaty mają zero lub więcej argumentów, np:

```
Chcę(5)
Możesz()
(x+y, 5)
3*4
```

Wysyłając komunikat wskazujemy proces, do którego komunikat jest skierowany. Nadawca jest wstrzymywany do chwili, gdy odbiorca będzie gotowy do jego odbioru, a próba komunikacji z procesem, który się zakończył jest błędem. Przykłady:

```
send Bufor.5
send Czytelnia.Chcę()
send P.Dodaj(x, y+3)
send Q.(2, 3)
```

Odbierając komunikat można wskazać nadawcę. Odbiorca jest wstrzymywany do chwili, gdy nadawca będzie gotowy do jego wysłania. Parametrami komunikatu są lokacje (zmienne), w których zostaną umieszczone poszczególne argumenty odebranego komunikatu. Próba komunikacji z procesem, który się zakończył jest błędem. Przykłady:

```
receive x
receive Chcę()
receive Q.Dodaj(x, y)
receive P.z
```

- Instrukcje wysyłania i odbierania komunikatu są do siebie dopasowywane na podstawie:
  - adresu nadawcy i odbiorcy
  - nazwy komunikatu
  - liczby i typów argumentów
- Odbiorca i nadawca są wstrzymywani do chwili, gdy obaj będą gotowi do komunikacji (sterowanie w obu procesach osiągnie dopasowane do siebie instrukcje wysłania i odebrania)
- Komunikacja polega na przekazaniu wartości argumentów od nadawcy do odbiorcy
- Po zakończeniu komunikacji oba procesy kontynuują swoje działanie
- Zakładamy uczciwość komunikacji

Instrukcje pułapki:

```
select
{
    on A.x:
        suma += x;
    on B.x:
        suma += x;
    on Dodaj():
        ++ile;
    on Odejmij():
        --ile;
    if (w > 0)
        on P():
            ++ile;
}
```

- Instrukcja wyboru komunikatu składa się z jednej lub więcej instrukcji pułapek
- Jej wykonanie polega na czekaniu aż w co najmniej jedną z nich złapie się komunikat, tzn. nadawca jest gotowy do wysłania komunikatu dopasowanego do opisu w instrukcji pułapki

- Gdy to nastąpi, wybieramy jedną z pułapek ze złapanym komunikatem
- Odbieramy ten komunikat (co pozwoli wykonywać się dalej nadawcy) i wykonujemy instrukcję jego obsługi
- Pozostałe komunikaty uwalniamy z pułapek (nie dochodzi do komunikacji z nadawcami) i przechodzimy do wykonania kolejnej instrukcji za instrukcją wyboru
- Instrukcja pułapki może być poprzedzona dozorem `if (wyrażenie)`
- Wyrażenie nie może powodować efektów ubocznych
- Pułapka jest aktywna wtedy i tylko wtedy, gdy wyrażenie jest prawdziwe
- Wyrażenia we wszystkich instrukcjach pułapek w obrębie instrukcji wyboru są wyliczane raz na początku wykonania tej instrukcji
- Zakłada się słabą uczciwość: jeśli pułapka w obrębie instrukcji wyboru jest aktywna nieskończenie wiele razy i nieskończenie wiele razy złapie się w nią komunikat, to kiedyś zostanie wybrana do wykonania

4.5 System rozproszony

Slajdy 958 - 1294

5 Wydajność algorytmów współbieżnych

Używamy języka C rozszerzonego o operacje do wyrażania współbieżności:

- `spawn P(...)` wywołuje funkcję `P(...)` być może w nowym wątku wykonywanym równolegle z dotychczasowym
- `sync` wstrzymuje wątek do chwili zakończenia wszystkich jego dzieci
- `parallel for` jak zwykły `for`, ale poszczególne obroty mogą wykonywać się równolegle

5.1 spawn

- Słowo kluczowe `spawn` oznacza logiczną równoległość - wywoływana funkcja może być wykonywana w osobnym wątku równolegle z wątkiem wywołującym, ale nie musi
- Decyduje o tym program szeregującego, przydzielający procesory podobliczeniom
- Zakładamy, że procesory są symetryczne

5.2 sync

- `sync` jest jedynym mechanizmem synchronizacji wątków
- Nie ma żadnych innych mechanizmów synchronizacyjnych, więc obliczenia wykonywane równolegle muszą być od siebie niezależne
- Zakładamy, że oprócz jawnych wystąpień `sync` każda instancja funkcja wykonuje `sync` niejawnie na zakończenie swojego działania
- Zatem wszystkie dzieci ko«czą się przed zakończeniem swojego rodzica

5.3 Model pamięci

- Wątki wykonują się we współdzielonej pamięci
- Arbiter pamięci szereguje zapisy i odczyty do tej samej komórki pamięci zachowując spójność sekwencyjną, czyli lokalną kolejność wykonywania rozkazów przez poszczególne procesy

5.4 Kolejna sekcja

Definicja (Serializacja)**.** Program otrzymany po usunięciu słów kluczowych `sync`, `spawn` i `parallel`. Serializacja jest poprawnym (choć sekwencyjnym) programem rozwiązującym ten sam problem, co oryginał.

Definicja (Graf obliczenia)**.** Skierowany graf acykliczny $G = (V, E)$, w którym:

- wierzchołki reprezentują poszczególne instrukcje
- krawędzie reprezentują kolejność wykonywania rozkazów, tj. krawędź $(u, v) \in E$ oznacza, że rozkaz u musi zostać wykonany przed v

Czasem wygodniej jest umieszczać w jednym wierzchołku ciągi instrukcji wykonywanych sekwencyjnie (nie zawierających konstrukcji `spawn` i `sync`) - takie ciągi to tzw. nici.

Definicja (Praca)**.** Łączny czas potrzebny do wykonania obliczenia na jednym procesorze. Inaczej: suma czasów wykonań wszystkich nici. Jeszcze inaczej: czas potrzebny na wykonanie serializacji obliczenia. Jeśli obliczenie każdej nici trwa jednostkę czasu, to praca jest równa liczbie wierzchołków w grafie.

Definicja (Rozpiętość)**.** Łączny czas wykonania wszystkich nici na najdłuższej ścieżce. Jeśli obliczenie każdej nici trwa jednostkę czasu, to rozpiętość jest liczbą wierzchołków na najdłuższej ścieżce w grafie.

Zasada pracy: $T_P \geq \frac{T_1}{P}$.
Zasada rozpiętości: $T_P \geq T_\infty$.

Definicja (Przyspieszenie)**.** - ile razy obliczenie wykonywane na P procesorach jest szybsze od wykonania na jednym procesorze, czyli: $\frac{T_1}{T_P}$.

Przyspieszenie jest LINIOWE, jeśli $\frac{T_1}{T_P} = \Theta(P)$.

DOSKONAŁE LINIOWE PRZYSPIESZENIE: $\frac{T_1}{T_P} = P$

Definicja (Równoległość obliczenia)**.** podatność obliczenia na zrównoleglenie, formalnie: $\frac{T_1}{T_\infty}$. Interpretacje:

- średnia ilość pracy, która może być wykonana równolegle w jednym kroku wzdłuż ścieżki krytycznej
- przyspieszenie, jakie można uzyskać mając do dyspozycji dowolnie wiele procesorów
- a z zasady rozpiętości także ograniczenie górne na możliwość przyspieszenia obliczenia, gdyż: $\frac{T_1}{T_P} \leq \frac{T_1}{T_\infty}$

5.5 Pętle równoległe

`parallel for (...; ...; ...)` - oznacza pętlę, której iteracje mogą wykonywać się równolegle. Przykład:

```
parallel for(int i = 0; i < n; i++)
    y[i] = 0;
```

Serializacja pętli równoległej - opuszczenie słowa kluczowego parallel. Tak jak przy `spawn` i `sync`, serializacja jest poprawnym, sekwencyjnym programem obliczającym to samo, co oryginał. Praca pętli równoległej = złożoność serializacji.

`parallel for` można zrealizować za pomocą `spawn` i `sync` metodą dziel i zwyciężaj. Powstaje wtedy zupełne drzewo binarne, w którego liściach znajdują się pojedyncze iteracje, ma ono wysokość $Theta(log(n))$, gdzie n jest liczbą iteracji. Zatem rozpiętość pętli równoległej z n iteracjami to:

$$T_\infty(n) = \Theta(logn) + \max_{0 \le i < n} iter_\infty(i)$$

Uwaga! Praca pętli jest w istocie większa niż praca jest serializacji, ale ponieważ liczba wierzchołków w drzewie jest $\Theta(n)$, więc nie ma to wpływu na rząd wielkości.

Definicja. Zachłanny program szeregujący - przydziela procesory do tylu nici, na ile to możliwe, tzn.: gdy w danej chwili jest co najmniej P gotowych do wykonania nici, procesory dostaje P dowolnych z nich w przeciwnym razie, każda gotowa do wykonania nić dostaje procesor

Twierdzenie. Na komputerze równoległym z P procesorami zachłanny program szeregujący wykonuje obliczenie wielowątkowe o pracy T_1 i rozpiętości T_∞ w czasie:

$$T_P \le \frac{T_1}{P} + T_\infty$$

Wniosek 1. czas działania T_P dowolnego obliczenia wielowątkowego, szeregowanego zachłannie na komputerze równoległym z P procesorami, jest co najwyżej dwukrotnie gorszy od optymalnego.

Wniosek 2. Niech T_P będzie czasem działania obliczenia wielowątkowego, szeregowanego zachłannie na komputerze równoległym z P procesorami. Wówczas, jeśli tylko równoległość obliczenia jest istotnie większa niż P, to przyspieszenie jest prawie doskonale liniowe.

Ścieżka krytyczna: T_∞,

Twierdzenie mistrza:

$$T(n) = a \cdot T(\left\lfloor \frac{n}{b} \right\rfloor) + f(n)$$

1. $f(n) = O(n^{log_b(a)-\varepsilon})$ dla wybranego $\varepsilon \in (0,1) \implies T(n) = \Theta(n^{log_b(a)})$ (O - co najwyżej)
2. $f(n) = \Theta(n^{log_b(a)}) \implies T(n) = \Theta(n^{log_b(a)} \cdot logn)$ (Θ - dokladnie)
3. $f(n) = \Omega(n^{log_b(a)+\varepsilon})$ dla $\varepsilon \in (0,1)$ oraz $a \cdot f(\frac{n}{b}) \le c \cdot f(n)$ dla $c < 1$ dddn $\implies T(n) = \Theta(f(n))$ (Ω - co najmniej)

6 Klasyczne problemy współbieżności

6.1 Problem wzajemnego wykluczania

Pierwszym klasycznym problemem synchronizacyjnym, chyba najczęściej pojawiającym się w praktyce, jest problem wzajemnego wykluczania. Przypuśćmy, że mamy procesy, z których każdy wykonuje następujący program:

```
process P():
    while true:
        # własne_sprawy
        protokół_wstępny
        sekcja_krytyczna
        protokół końcowy
```

Procedura `własne_sprawy` to fragment kodu, który nie wymaga żadnych działań synchronizacyjnych. Proces może wykonywać **własne sprawy dowolnie długo** (nawet nieskończenie długo), może ulec tam awarii. `Sekcja_krytyczna` to fragment programu, który może być jednocześnie wykonywany przez co najwyżej

jeden proces. Zakładamy, że **każdy proces, który wchodzi do sekcji krytycznej, w skończonym czasie z niej wyjdzie**. Oznacza to w szczególności, że podczas pobytu procesu w sekcji krytycznej nie wystąpi błąd, proces nie zapętli się tam. Zadanie polega na napisaniu protokołu wstępnego i protokołu końcowego tak, aby:

- W sekcji krytycznej przebywał co najwyżej jeden proces jednocześnie (**bezpieczeństwo**).
- Każdy proces, który chce wykonać sekcję krytyczną w skończonym czasie do niej wszedł (**żywotność**).

Protokoły wstępny i końcowy konstruuje się korzystając z dostępnych mechanizmów synchronizacyjnych. W przypadku braku jakiegokolwiek wsparcia ze strony systemu operacyjnego należy zastosować jeden z algorytmów synchronizacyjnych, np. algorytm Petersona.

Czasem rozpatruje się też wariant problemu wzajemnego wykluczania, w którym jednocześnie może przebywać więcej procesów. Mamy na przykład N procesów działających, z których w sekcji może przebywać jednocześnie 0<K<N.

6.1.1 Przykładowe rozwiązania

Listing1: Algorytm Petersona

```
chce[0] = false;
chce[1] = false;
kto_czeka = 1;

process P1():
    while true:
        # własne sprawy
        chce[0] = true
        kto_czeka = 1
        while kto_czeka == 1 and chce[1];
        # sekcja krytyczna
        chce[0] = false
```

```
process P2():
    while true:
        # własne sprawy
        chce[1] = true
        kto_czeka = 2
        while kto_czeka == 2 and chce[0];
        # sekcja krytyczna
        chce[1] = false
```

Listing2: Semafory

```
process P():
    # własne sprawy
    P(S)
    # sekcja krytyczna
    V(S)
```

To rozwiązanie nie jest żywotne bez dodatkowych założeń na rodzaj semafora. Dla semafora słabego to rozwiązanie działa poprawnie **tylko** dla dwóch procesów. Dla semaforów silnie uczciwych (a więc i silnego) to rozwiązanie jest poprawne.

6.2 Producenci i konsumenci

W systemie działa $P > 0$ procesów, które produkują pewne dane oraz $K > 0$ procesów, które odbierają dane od producentów. Między producentami a konsumentami może znajdować się bufor o pojemności B,

którego zadaniem jest równoważenie chwilowych różnic w czasie działania procesów. Procesy produkujące dane będziemy nazywać producentami, a procesy odbierające dane - konsumentami. Zadanie polega na synchronizacji pracy producentów i konsumentów, tak aby:

- Konsument oczekiwał 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 (**bezpieczeństwo**).
- 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 pojawią 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 (**żywotność**).
- Jeśli wielu producentów oczekuje, aż w buforze będzie wolne miejsce, a konsumenci ciągle coś 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 coś jest pobierane (**żywotność**).

Problem producentów i konsumentów jest abstrakcją wielu sytuacji występujących w systemach komputerowych, na przykład zapis danych do bufora klawiatury przez sterownik klawiatury i ich odczyt przez system operacyjny.

6.3 Przykładowe rozwiązanie

Początkowo w przestrzeni: `lprod 1`, `lkons 1` oraz `M` krotek `miejsce`.

```
process Producent (i : 1..P)
{
    int p, ile;
    while (true)
    {
        produkuj(&p);
        tsFetch("miejsce");
        tsFetch("lprod ?d", &ile);
        tsPut("%d %d", ile, p);
        tsPut ("lprod %d", ile+1);
    }
}
```

```
process Konsument (i : 1..K)
{
    int p, ile;
    while (true)
    {
        tsFetch("lkons ?d", &ile);
        tsFetch("%d ?d", ile, &p);
        tsPut("miejsce");
        tsPut("lkons ?d", ile + 1);
        konsumuj(p);
    }
}
```

Inny sposób: Początkowo w przestrzeni: `lprod 1`, `lkons 1`

```
process Producent (i : 1..P)
{
    int p, ile;
    while (true)
    {
        produkuj(&p);
        tsFetch("lprod ?d", &ile);
        tsPut("%d %d", ile, p);
        tsPut("lprod %d", ile+1);
    }
}
```

```
process Konsument (i : 1..K)
{
    int p, ile;
    while (true)
    {
        tsFetch("lkons ?d", &ile);
        tsFetch("%d ?d", ile, &p);
        tsPut("lkons ?d", ile+1);
        konsumuj(p);
    }
}
```

6.4 Czytelnicy i pisarze

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.

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

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 (**bezpieczeństwo**).
- Każdy czytelnik, który chce odczytać dane, w końcu je odczyta (**żywotność**).
- Każdy pisarz, który chce zmodyfikować dane, w końcu je zapisze (**żywotność**).

6.4.1 Przykładowe rozwiązanie

```
process Czytelnik():
    while true:
        # własne sprawy
        Czytelnia.ChcęCzytać()
        # czytanie
        Czytelnia.KoniecCzytania()
```

```
process Pisarz():
    while true:
        # własne sprawy
        Czytelnia.ChcęPisać()
        # pisanie
        Czytelnia.KoniecPisania()
```

```
monitor Czytelnia:
    def __init__():
        czytelnicy = 0 # liczba czytelników w czytelni; to jest pythonlike, załóżmy, że
 ↪  tam wszędzie jest self.
        pisanie = False # czy jest pisarz w czytelni
        można_pisać = Condition()
        można_czytać = Condition()

    def ChcęCzytać():
        if pisanie or not empty(można_pisać):
            wait(można_czytać)
        czytelnicy += 1
        signal(można_czytać)

    def ChcęPisać():
        if czytelnicy ≠ 0 or pisanie:
            wait(można_pisać)
        pisanie := True

    def KoniecCzytania():
        czytelnicy -= 1
        if czytelnicy == 0:
            signal(można_czytać)

    def KoniecPisania():
        pisanie = False
        if not empty(można_czytać):
```

```
            signal(można_czytać)
        else:
            signal(można_pisać)
```

W Lindzie:
Początkowo w przestrzeni: `0 0`

```
process Czytelnik (i : 1..C)
{
    int c, p;
    while (true)
    {
        własne_sprawy();
        tsFetch("?d %d", &c, 0);
        tsPut("%d %d", c+1, 0);
        czytanie();
        tsFetch("?d ?d", &c, &p);
        tsPut("%d %d", c-1, p);
    }
}
```

```
process Pisarz (i : 1..P)
{
    int c, p;
    while (true)
    {
        // własne_sprawy();
        tsFetch("?d ?d", &c, &p);
        if (c > 0)
        {
            tsPut("%d %d", c, p+1);
            tsFetch("%d ?d", 0, &p);
            p--;
        }
        pisanie();
        tsPut("%d %d", 0, p);
    }
}
```

Inny sposób w Lindzie:
Początkowo w przestrzeni: `0`, `nie piszę`, `można`

```
process Czytelnik (i : 1..C)
{
    int c;
    while (1)
    {
        // własne_sprawy();
        tsFetch("można");
        tsFetch("?d", &c);
        tsPut("%d", c+1);
        tsPut("można");
        tsRead("nie_piszę");
        czytanie();
        tsFetch("?d", &c);
        tsPut("%d", c-1);
    }
}
```

```
process Pisarz (i : 1..P)
{
    while (1)
    {
        // własne_sprawy();
        tsFetch("można");
        tsRead("%d", 0);
        tsFetch("nie_piszę");
        tsPut("można");
        pisanie();
        tsPut("nie_piszę");
    }
}
```

W Adzie:

```
process Czytelnik[i : 1..C]
{
    while (1)
    {
        send Czytelnia.czytam();
        czytanie();
        send Czytelnia.koniecCzytania();
    }
}
```

```
process Pisarz[i: 1..P]
{
    while (1)
    {
        send Czytelnia.chcęPisać();
        receive możeszPisać();
        pisanie();
        send Czytelnia.koniecPisania();
    }
}
```

```
process Czytelnia
{
    int czytający = 0;
    while (1) {
        select
        {
            on czytam():
                czytający++;
            on koniecCzytania():
                czytający--;
            for (i in 1..P)
                on Pisarz[i].chcęPisać():
                {
                    while (czytający > 0)
                    {
                        receive koniecCzytania();
                        czytający--;
                    }
                    send Pisarz[i].możeszPisać();
                    receive koniecPisania();
                }
        }
    }
}
```

6.5 Pięciu filozofów

Pięciu filozofów siedzi przy okrągłym stole. Przed każdym stoi talerz. Między talerzami leżą widelce. Pośrodku stołu znajduje się półmisek z rybą. Każdy filozof myśli. Gdy zgłodnieje sięga po widelce znajdujące się po jego prawej i lewej stronie, po czym rozpoczyna posiłek. Gdy się już naje, odkłada widelce i ponownie oddaje się myśleniu. Schemat działania filozofa jest więc następujący:

```
process Filozof(i : 0..4):
    while true:
        myśli
        protokół_wstępny
        je
        protokół_końcowy
```

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.

Chcemy ponadto, aby każdy filozof działał w ten sam sposób.

Spróbujmy zastanowić się, jak mogłyby wyglądać przykładowe protokoły wstępne i końcowe filozofa. Pierwszy pomysł polega na tym, aby każdy filozof, gdy zgłodnieje, sprawdzał czy widelec po jego lewej stronie jest wolny. Jeśli tak, to filozof powinien podnieść ten widelec i oczekiwać na dostępność drugiego. Jak widać, po pewnym czasie dochodzi do zakleszczenia (można przyspieszyć jego wystąpienie skracając suwakiem czas jedzenia i rozmyślania filozofów). Dzieje się tak wówczas, gdy filozofowie jednocześnie zgłodnieją i każdy z nich w tej samej chwili podniesie lewy widelec. Każdy oczekuje teraz na prawy widelec, ale nigdy się na niego nie doczeka.

A co stałoby się, jeśli każdy filozof podnosiłby na raz oba widelce, jeśli oba są wolne? Wtedy do zakleszczenia nie dochodzi. Tym razem doszło do zagłodzenia filozofa. Mamy przykład braku żywotności, różny od zakleszczenia. Jeśli dwaj sąsiedzi pewnego filozofa źmówią się"przeciw niemu, to mogą nie dopuścić do sytuacji, w której obydwa widelce będą dostępne. Poprawne rozwiązanie tego problemu przedstawimy w następnym wykładzie.

6.5.1 Przykładowe rozwiązania

```
w = Semaphore(1) * 5; # widelce
lokaj = Semaphore(4)

process F (i : 0..4):
    while true:
        # myśli
        P(lokaj)
        P(w[i])
        P(w[(i+1) % 5])
        # je
        V(w[(i+1) % 5])
        V(w[i])
        V(lokaj)
```

7 Zadania

7.1 Linda

7.1.1 Centrala telefoniczna

W systemie działa `N` nadawców, `M` odbiorców oraz centrala, która ma `K` łączy. Nadawca cyklicznie losuje numer odbiorcy wywołując procedurę `losuj(m)`, następnie przekazuje ten numer centrali, od której otrzymuje numer łącza `k` (jeśli odbiorca jest zajęty lub brakuje wolnego łącza, to k = 0. W tym przypadku nadawca ponawia wysyłanie numeru odbiorcy do centrali). Nadawanie jest realizowane za pomocą procedury `nadaj(k)`. Po zakończeniu nadawania nadawca zwalnia łącze. Odbiorca czeka, aż centrala dostarczy mu numer łącza, a następnie odbiera informacje ze wskazanego łącza wywołując procedurę `odbierz(k)`. Zapisz treści procesów `NADAWCA(1...n)` oraz `ODBIORCA(1...m)`. Krotki wolnyOdbiorca, wolneŁącze

```
process Nadawca(id : 1..n)
{
    int łącze;
    int id_odbiorcy = losuj();

    tsFetch("MOGE GADAC %d", id_odbiorcy);
    tsFetch("WOLNE ŁĄCZE ?d", &łącze);
    tsPut("DZWONIE %d %d", łącze, id_odbiorcy);
    gadam(łącze);
    tsFetch("SKOŃCZYLIŚMY %d", id_odbiorcy);
    tsPut("WOLNE ŁĄCZE %d", łącze);
}
```

```
process Odbiorca(id : 1..n)
{
    int id_nadawcy, łącza;

    tsPut("MOGE GADAC %d, id);
    tsFetch("DZWONIE ?d %d, &łącze, id);
    gadamy(łącze);
    tsPut("SKOŃCZYLIŚMY %d", id);
}
```

7.2 Ada

7.2.1 Zadanie o politykach

Link: Egzamin 2000/2001

```
process Dziennikarz(id):
    int p = polityk();
    int sala;
    send Info.chceDo(id, p);
    select:
        on obudź(id_budzonego):
            receive Sala(sala);
            send Dziennikarz[id_budzonego].Sala(sala);
            idźNaSalę(sala);

        on wypierdalaj():;
```

```
process Polityk(id)
{
    send Info.siema(id);
    receive czeka(liczba);
    if (liczba > M):
        send Info.chceDuza();
    else if (liczba > 0):
        send Info.chceMała();
    else:
        return;
    receive sala(id_sali);
    zrób_wykład(id_sali);
    Info.zwrócSalę();
}
```

```
process Info():
    bool lampki[P];
    int dużeSale, małeSale;
    int ostatni[P]; // = - 1;
    int ileCzeka[P]; // = 0;
    int sala[P];
    bool przybył[P];

    dużeSale = małeSale = S;

    select:
        on chceDo(id, p):
            if (!lampki[p]):
                send Dziennikarz[id].wypierdalaj();
            else if (!przybył(p)):
                send Dziennikarz[id].obudź(ostatni[p]);
                ostatni[p] = id;
                ileCzeka[p]++;
            else if (przybył[p]):
                if (ileCzeka[p] < rozmiar(sala[p])):
                    send Dziennikarz[id].obudź(ostatni[p]);
```

```
                ostatni[p] = id;
                ileCzeka[p]++;
            else:
                send Dziennikarz[id].wypierdalaj();

    on siema(id):
        przybył[p] = true;
        if (ileCzeka[id] == 0):
            lampka[id] = false;
        send Polityk.czeka(ileCzeka[id]);

    on chceDuza:
        // ... do dokończenia w domu :v
```

7.2.2 Hotel Ada

Link: Kolos 2003/2004

```
process Boy(id : 1..B):
    while (true):
        send Recepcja.Gotowy(id);
        select:
            on Zanieś(pokój, gość):
                zanoszę(pokój, gość);

            on Sprzątaj(pokój):
                sprzątaj(pokój);
```

```
def ZamieszkajIOddajPokoj(id):
    send Recepcja.chcęBoya(id);
    receive Boy(boy);
    idęDoPokoju(pokój);
    Recepcja.OddajPokój(pokój);

process Gość(id : 1...G):
    send Recepcja.ChcęPokój(id);
    select:
        on MaszPokój(pokój):
            ZamieszkajIOddajPokój(id);
            return;

        on brak():;

    send Recepcja.ZnowuChcePokój(id);
    receive MaszPokój(pokój);
    ZamieszkajIOddajPokój(id);
```

```
def obsloz():
    int nr = pierwszy_wolny();
    ileZajętych++;
    zajętyPokój[nr] = true;
    czekaGość[id] = true;
    send Gość[id].maszPokój(nr);

process Recepcja:
    bool zajętyPokój[P];
    bool zajętyBoy[B];
    int ilePokojówZajętych = 0;
    int ileBoyówZajętych = B;
    int ileNaBoya = 0;

    select:
        if !czyCzekaPriorytet and ileZajętch < P:
            on chcęPokój(id):
                obsłóż();
```

```
if ileZajętych == P:
    on chcePokój(id):
        send Gosć[id].brak();
if ktoś_czeka(czekaGość):
    on Gotowy(id):
        czekaGość[pierwszy_czeka()] = 0;
        send Boy[id].Zanieś(pierwszy_czeka());
on ChcePrioritet(id):
    czyPriorytet[id] = true;
if ileZajętych < P:
    on chceZnowuPokój(id):
        chcePriorytet[id] = false;
        obsłóż();
on ZwolnijPokoj(pokój):
    --ileZajętych;
    # i tak dalej
```

8 Egzaminy

Link: Egzamin poprawykowy SO 2009

1. ABC
2. B
3. ABC
7. C
8. BC
9. A