Synchronizacja procesów

Podstawy

- Współbieżny dostęp do danych dzielonych może prowadzić do **niespójności** (***inconsistency***) danych.
- Utrzymywanie spójności danych wymaga mechanizmów zapewniających uporządkowane wykonywanie **procesów współpracujących** *(**cooperating processes**).*
- Przykład dla problemu producenta\konsumenta:
  Załóżmy, że do problemu dodajemy nową zmienną `counter = 0`, a do procedur producenta i konsumenta kolejno polecenia: `counter++, counter--`
  Powyższe instrukcje muszą być wykonywane **niepodzielnie** ***(atomically),*** czyli w całości bez przerwania,
  - ***Primitive* – operacja elementarna**,
  - Obowiązuje tutaj zasada ***"wszystko albo nic"***,
  - Gdyby producent i konsument zechcieli uaktualniać bufor współbieżnie, mogłoby dość do **przeplotu** wykonywanych rozkazów asemblera,
  - Przeplot ***(interleaving)*** zależy od planowania procesów producenta i konsumenta,
  - **Szkodliwa rywalizacja/wyścig** *(**race condition**)*: tak określa się sytuację, w której kilka procesów sięga po dane i operuje na nich współbieżnie. Ostateczna wartość danych dzielonych zależy od tego, który proces aktualizował je jako ostatni,
  - Aby zapobiec wyścigom, procesy współbieżne należy **synchronizować.**

Problem sekcji krytycznej

- Mamy n procesów ubiegających się o korzystanie z pewnych wspólnych danych,
- Każdy proces ma segment kodu, zwany **sekcją krytyczną *(critical section)***, w którym sięga do dzielonych danych,
- Problem polega na tym, aby zapewnić, że gdy któryś proces działa w sekcji krytycznej, żaden inny nie może wykonywać sekcji krytycznej udostępniającej te same obiekty danych,
- Każdy proces prosi o pozwolenie na wejście do sekcji krytycznej w s**ekcji wejściowej *(entry section)*,**
- Po sekcji wejściowej może występować **sekcja wyjściowa *(exit section),***
- Pozostały kod nazywa się **resztą *(remainder seciton),***
- ***mutual exclusion* = wzajemne {wykluczanie |wyłączanie}**
- ***shared data* = dane dzielone (współużytkowane)**
- **Rozwiązanie** problemu sekcji krytycznej:
  - **Wzajemne wykluczanie (*mutual exclusion*).** Jeśli proces P_i działa w sekcji krytycznej, to żaden inny proces nie może działać w swojej, związanej z tymi samymi danymi,
  - **Postęp *(progress)*.** Jeśli żaden proces nie działa w swojej sekcji krytycznej i istnieją procesy chcące wejść do swoich sekcji, to wybieranie procesów, które jako następne wejdą do sekcji krytycznej odnosi się do tych, które nie wykonują swoich reszt i nie może być odwlekane w nieskończoność,
  - **Ograniczone czekanie *(bounded waiting)*.** Liczba procesów wchodzących do swoich sekcji krytycznych po zgłoszeniu przez pewien proces zapotrzebowania na wejście do sekcji krytycznej i przed udzieleniem mu na to pozwolenia musi być ograniczona.

- Zakładamy, że każdy proces działa z niezerową szybkością,
- Nie czyni się żadnych założeń co do względnych szybkości n procesów.
- Rozwiązanie **wieloprocesowe** dla n procesów - **algorytm piekarni *(bakery algorithm):***
    - Przed wejściem do swojej sekcji krytycznej proces otrzymuje numer. Posiadacz najmniejszego numeru wchodzi do sekcji krytycznej,
    - Jeśli procesy P_i i P_j dostaną ten sam numer, to gdy $i < j$, wówczas najpierw jest odsługiwany P_i , a jak nie, to P_j ,
    - Schemat numerowania zawsze generuje numery w porządku niemalejącym, np. 1, 2, 3, 3, 3, 3, 4, 5,…

Semafory

- Mechanizm synchronizacji, który w znacznym stopniu eliminuje **czekanie aktywne *(busy waiting),***
- **Semafor S** jest realizowanym sprzętowo typem danych: na zmiennej całkowitej są wykonywane dwie specjalne operacje,
- Dostęp do zmiennej semaforowej jest możliwy tylko za pomocą takich oto niepodzielnych działań:

```
czekaj(S)
     while S <= 0 do nic;
     S := S – 1;
sygnalizuj(S)
     S := S + 1;
```

- ***busy waiting = spin lock* = czekanie aktywne = wirująca blokada,**
- Używane np. do rozwiązywania **problemu sekcji krytycznej**. Wspólny semafor **`mutex`** jest dzielony na n procesów. Każdy proces używa **`czekaj(mutex)`** przed sekcją krytyczną oraz **`sygnalizuj(mutex)`** po niej.
- **Głodzenie *(starvation)***, czyli blokowanie nieskończone. Proces może nigdy nie opuścić kolejki semaforowej, w której go zawieszono.
- **Semafor zliczający *(counting semaphore)*** — jego wartość całkowita może się zmieniać bez ograniczeń.
- **Semafor binarny *(binary semaphore)*** — przyjmuje tylko wartości 0 lub 1; można go łatwie zrealizować. Inna nazwa: **mutex** (od mutual exclusion).
- **Semafor zliczający** S można zaimplementować z użyciem **semafora binarnego.**
- Klasyczne problemy synchronizacji:
    - **Problem ograniczonego buforowania *(bounded-buffer)*** – mamy n buforów, z których każdy mieści jedną jednostkę. Semafory `pusty` i `pełny` zawierają liczbę pustych i pełnych miejsc, na początku n i 0. Używa się dodatkowego semafora `mutex` (początkowo = 1), aby zapobiec współbieżnemu korzystaniu z sekcji krytycznej.
    - **Problem czytelników i pisarzy *(readers-writers)*** - mamy czytelników i pisarzy, którzy korzystają z tego samego pliku lub rekordu do kolejno: czytania i uaktualniania go. Nie ma problemu, jeżeli dwóch czytelników chce czytać jednocześnie, ale pisarz musi korzystać ze zmiennych sam. Dane dzielone: semafory `mutex` i `pisarz`, początkowo oba równe 1, dodatkowo liczba_czyt = 0 (zwiększamy, kiedy przyjdzie

chętny do czytania, zmniejszamy, kiedy skończy czytać. Czytelnicy czekają na pisarza, aż zwolni się miejsce, mutex pilnuje dostępu do licznika czytelników.

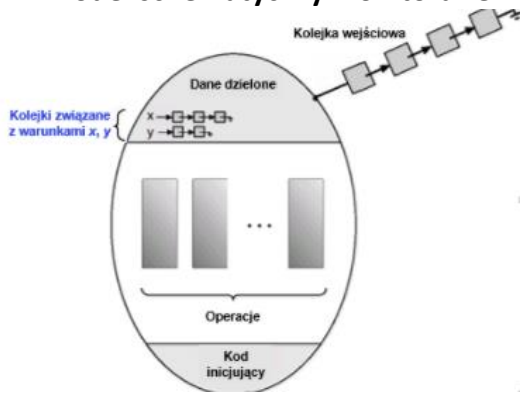
- **Problem obiadujących filozofów *(dining-philosophers)*** - pięciu filozofów siedzących przy okrągłym stole jedynie myślą i jedzą; między każdymi dwoma leży jedna pałeczka. Kiedy filozof poczuje głód, wtedy próbuje chwycić dwie pałeczki najbliżej niego. Za każdym razem może podnieść jedną na raz. Użycie jednego semafora dla każdej pałeczki i kazanie filozofowi czekać przy podnoszeniu i sygnalizować po odłożeniu **powoduje zakleszczenia**. Można **a)** pozwolić zasiadać tylko czwórce jednocześnie, **b)** pozwolić filozofowi podniesienie pałeczki jedynie, kiedy obie są dostępne (weryfikowane w sekcji krytycznej), **c)** filozof o numerze parzystym podnosi najpierw pałeczkę po prawej i na odwrót.
  Rozwiązanie wolne od zakleszczeń nie eliminuje automatycznie, że żadnej filozof się nie zagłodzi (tego wymagają sensowne rozwiązania).

- **Region krytyczny *(critical region)*** jest wysokopoziomową konstrukcją synchronizacji.
- Zmienną dzieloną v typu T deklaruje się tak:
  `v: shared T;`
- Dostęp do zmiennej v jest możliwy tylko wewnątrz instrukcji
  `region v when (B) do S;`
  gdzie B jest wyrażeniem boolowskim.
- Dopóki jest wykonywana instrukcja S, żaden inny proces nie ma dostępu do zmiennej v.
- Regiony odwołujące się do tej samej zmiennej dzielonej wykluczają się w czasie. Ich instrukcje S nie mogą działać jednocześnie.
- Gdy proces próbuje wykonać instrukcję regionu, oblicza się wyrażenie boolowskie B. Jeśli B jest prawdziwe, to S jest wykonywana. Jeśli jest fałszywe, proces jest opóźniany do czasu, aż B stanie się prawdziwe i żaden inny proces nie będzie przebywał w regionie związanym z v.
- Ze zmienną dzieloną x kojarzymy następujące zmienne:
  ```
  semaphore mutex, pierwsza_zwłoka, druga_zwłoka;
  int pierwszy_licznik, drugi_licznik;
  ```
- Dostęp do sekcji krytycznej na zasadzie wyłączności jest zapewniany przez **`mutex`**.
- Jeśli proces nie może wejść do sekcji krytycznej, gdyż wyrażenie boolowskie B jest fałszywe, to najpierw czeka pod semaforem **`pierwsza_zwłoka`**, nim zezwoli mu się na ponowne obliczenie B, przesuwa się go pod drugi semafor: **`druga_zwłoka.`**
- W licznikach **`pierwszy_licznik i drugi_licznik`** utrzymuje się liczbę procesów czekających pod semaforami **`pierwsza_zwłoka i druga_zwłoka.`**
- Kolejka procesów pod semaforem jest porządkowana w tym algorytmie na zasadzie **FIFO**.

Monitory

- Monitor jest **wysokopoziomową konstrukcją synchronizacji** umożliwiającą bezpieczne dzielenie abstrakcyjnego typu danych przez procesy współbieżne.
- Aby zezwolić procesowi na czekanie w monitorze, trzeba zadeklarować zmienną typu **`condition:`**
- Na zmiennej warunkowej można wykonywać tylko operacje **`wait i signal.`**
- Proces wywołujący **`x.wait()`** czeka aż inny proces wywoła **`x.signal()`**

- Operacja **`x.signal()`** wznawia tylko jeden z zawieszonych procesów. Jeśli żaden proces nie jest zawieszony, operacja **`signal`** nie wywołuje żadnych skutków.
- **Model schematyczny monitora ze zmiennymi warunkowymi** (bez nich nie ma kolejek x, y):



Jeśli proces P wywołuje **`x.signal()`**, a proces Q jest zawieszony w **`x.wait()`**, to P musi zaczekać. Mamy do wyboru:
- sygnalizować i czekać — P czeka, aż Q opuści monitor lub zacznie czekać na spełnienie innego warunku,
- **sygnalizować i kontynuować** — Q czeka, aż P opuści monitor lub zacznie czekać na spełnienie innego warunku.

- Konstrukcja **x.wait(c) czekania warunkowego** (***conditional-wait***):
  - **c** — wyrażenie całkowite obliczane podczas wykonywania operacji **wait**,
  - wartość **c** (**numer priorytetu**, ***priority number***) jest pamiętana wraz z nazwą zawieszonego procesu,
  - podczas wykonywania **x.signal** następuje wznowienie procesu o najmniejszym numerze (najwyższym priorytecie).
- Aby zapewnić poprawność systemu, sprawdza się dwa warunki:
  - procesy użytkownika muszą zawsze dokonywać wywołań **monitora we właściwej kolejności,**
  - należy zagwarantować, że proces **niewspółpracujący nie zignoruje bramki wzajemnego wykluczania** tworzonej przez monitor, próbując bezpośrednio dobrać się do wspólnego zasobu, z pominięciem protokołu dostępu.

Synchronizacja w systemie Solaris

- System realizuje spory asortyment zamków do organizacji wielozadaniowości, wielowątkowości (w tym wątków czasu rzeczywistego) i wieloprzetwarzania.
- **Zamki adaptacyjne *(adaptive mutexes)*** służą do ochrony krótkich segmentów kodu w systemach jedno- i wieloprocesorowych.
- Semafory i zmienne warunkowe są używane do ochrony dłuższych segmentów kodu.
- **Zmienne warunkowe (*condition variables*)** i **blokady** (zamki) typu czytelnicy-pisarze (readers-writers locks) są stosowane do ochrony danych używanych często, lecz głównie do czytania.
- Poza tym używa się **turniketów(*turnstile):***
  - jest strukturą typu **kolejka,**
  - **służy do porządkowania wykazu wątków** czekających na nabycie zamka adaptacyjnego lub blokady typu czytelnik-pisarz,
  - mamy po **jednym turnikecie na wątek**, a nie na obiekt synchronizacji co powoduje, że wątek może się blokować w danej chwili **tylko na jednym obiekcie**