Programowanie współbieżne i rozproszone

WYKŁAD 2

dr inż. Krzysztof Pancerz

Problemy programowania współbieżnego

- Projektowanie i implementacja systemów współbieżnych wymaga niejednokrotnie rozwiązywania problemów niewystępujących w systemach sekwencyjnych.
- Problemy te z reguły związane są ze współpracą procesów (wątków), która jest niezbędna dla prawidłowego działania systemów.

Sekcja krytyczna

- **Sekcja krytyczna** występuje w przypadku gdy grupa procesów rywalizuje o zasób , przy czym w danym momencie tylko jeden z procesów może mieć do niego dostęp.

Problem producenta i konsumenta

- **Problem producenta i konsumenta** jest problemem wymagającym synchronizacji kolejności wykonywania instrukcji.
- W problemie tym występują dwa rodzaje procesów:
  - PRODUCENT - proces tworzący pewne dane, które przesyłane są procesom konsumentów.
  - KONSUMENT - proces pobierający i przetwarzający dane przesyłane z procesów producentów.

Problem producenta i konsumenta

- Przesyłanie danych z procesu producenta do procesu konsumenta może odbywać się na dwa sposoby:
  - *synchronicznie* (przesyłanie danych możliwe jest tylko wtedy gdy procesy producenta i konsumenta są do tego gotowe),
  - *asynchronicznie* (kanał komunikacyjny wyposażony jest w bufor do tymczasowego przechowywania danych, bufor może być nieskończony lub skończony).

Problem producenta i konsumenta

- Problemy synchronizacji:
  - Przy buforze nieskończonym:
    - proces konsumenta nie może pobierać danych z pustego bufora.
  - Przy buforze skończonym:
    - proces producenta nie może wysyłać danych do pełnego bufora.
    - proces konsumenta nie może pobierać danych z pustego bufora.

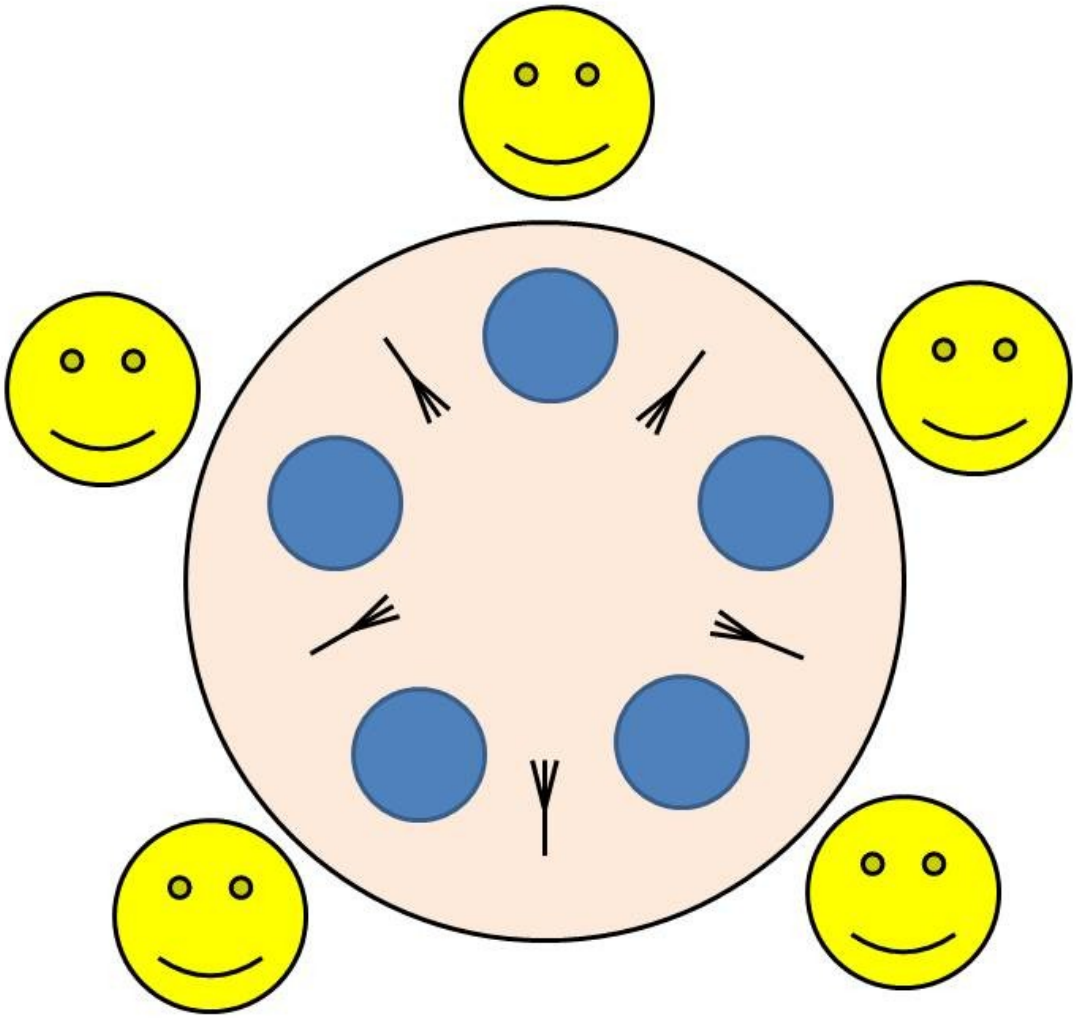
Problem ucztujących filozofów

- **Problem ucztujących filozofów** jest problemem synchronizacji, w którym procesy (wątki) wymagają jednoczesnego dostępu do więcej niż jednego zasobu.

Problem ucztujących filozofów

- Problem ucztujących filozofów jest sformułowany następująco:
  - Pięciu filozofów siedzi dookoła okrągłego stołu spędzając czas na wykonywaniu dwóch czynności: myślenia i jedzenia.
  - Na stole, przed każdym z filozofów znajduje się talerz ze spaghetti stale uzupełnianym.
  - Pomiędzy talerzami znajduje się pięć widelców.

Problem ucztujących filozofów



Problem ucztujących filozofów

- Problem ucztujących filozofów jest sformułowany następująco (cd.):
  - Aby jeść, każdy filozof potrzebuje dwóch widelców, po jego prawej i lewej stronie.
  - Każdy z filozofów nie może podnieść jednocześnie obu widelców.

Problem ucztujących filozofów

- Problem synchronizacji polega na takiej organizacji dostępu do widelców, aby każdy z filozofów mógł co jakiś czas realizować czynność jedzenia (aby nie wystąpiło tzw. zagłodzenie).

Problem czytelników i pisarzy

- **Problem czytelników i pisarzy** dotyczy wzajemnego wykluczania w systemach, w których wiele procesów rywalizuje o dostęp do sekcji krytycznej, ale wyróżnione są dwie klasy procesów:
  - CZYTELNICY - procesy wykluczające pisarzy, ale nie pozostałych czytelników,
  - PISARZE - procesy wykluczające każdy inny proces, zarówno czytelnika jak i pisarza.

Problem czytelników i pisarzy

- Kilka procesów współbieżnych odczytujących dane nie stanowi problemu.
- Zapis lub modyfikacja danych wymagają wzajemnego wykluczania, aby zapewnić spójność danych.

Współdzielenie zasobów przez wątki

- Kilka równolegle wykonujących się wątków może korzystać z tych samych zasobów (metod lub obiektów).
- Aby uniknąć równoczesnego działania wątków na tych samych metodach lub obiektach należy je zsynchronizować przez zastosowanie tzw. rygli.
- Ryglowanie dokonuje się automatycznie. Wówczas dany zasób, z którego korzysta wątek jest zablokowany i inne wątki nie mają do niego dostępu, dopóki nie zostanie odblokowany.

Współdzielenie zasobów przez wątki (cd.)

- Do ryglowania wspólnych zasobów wykorzystuje się słowo kluczowe **synchronized**.
- Synchronizowane mogą być metody lub bloki.

Synchronizacja (ryglowanie) metod

- Metoda synchronizowana oznaczana jest w deklaracji słowem kluczowym **synchronized**, np.:

```
synchronized void metoda()
{
   //...
}
```

Synchronizacja (ryglowanie) metod (cd.)

- Wywołanie metody synchronizowanej przez dany wątek powoduje zablokowanie obiektu, na rzecz którego była ona wywołana, uniemożliwiając innym wątkom dostęp do tego obiektu.
- Inne wątki odwołujące się w tym czasie do tego obiektu są blokowane i muszą oczekiwać na zdjęcie rygla z obiektu.
- Dowolne zakończenie metody synchronizowanej (także na skutek powstania wyjątku) zwalnia rygiel, przez co inne wątki uzyskują dostęp do obiektu.

Synchronizacja (ryglowanie) bloków

- Bloki synchronizowane wprowadzane są za pomocą instrukcji **synchronized**, np.:

```
synchronized(ref_do_rygl_obiektu)
{
   //...
   //instrukcje
   //...
}
```

Synchronizacja (ryglowanie) bloków (cd.)

- W nawiasie podawana jest referencja do obiektu, którego blokada ma być używana do synchronizacji otoczonego blokiem kodu. Oznacza to, że kod w bloku jest dostępny tylko dla wątku, który zablokował dany obiekt.
- W bloku instrukcji nie musi występować odwołanie do blokowanego obiektu.
- Synchronizacja bloków jest ogólniejsza od synchronizacji metod.

Synchronizacja (ryglowanie) bloków (cd.)

- Instrukcja **synchronized** może zostać użyta tylko w stosunku do obiektów klas wyprowadzonych z klasy **Object**. Nie można jej więc zastosować np. w stosunku do zmiennych typów prostych (**float**, **int**, ...).

Porozumiewanie się wątków

- Ryglowanie służy do zapobiegania niepożądanej interakcji wątków. W większości przypadków nie jest to jednak wystarczające. Zachodzi bowiem potrzeba skoordynowania działań między wątkami aby można było efektywniej wykorzystać korzyści programowania wielowątkowego.

Porozumiewanie się wątków (cd.)

- Skoordynowanie działań między wątkami uzyskuje się za pomocą finalnych metod klasy **Object**:

  wait(),

  notify(),

  notifyAll().

- Metody te mogą być wywoływane tylko z bloku synchronizowanego.

Porozumiewanie się wątków (cd.)

- Metoda **wait** wywoływana jest przez wątek na rzecz obiektu, na którego zmianę stanu wątek oczekuje.
- Wywołanie metody **wait** blokuje dany wątek i jednocześnie powoduje otwarcie rygla, umożliwiając dostęp do obiektu innym wątkom.
- Inny wątek może zmienić stan obiektu i powiadomić o tym wątek czekający za pomocą metody **notify**.

Porozumiewanie się wątków (cd.)

- Po wywołaniu metody **notify** następuje odblokowanie wątku czekającego na danym obiekcie.
- Metoda **notifyAll** odblokowuje wszystkie wątki czekające na danym obiekcie.
- Metoda **wait** wywołana z argumentem będącym liczbą całkowitą określającą milisekundy powoduje wstrzymanie wątku na wskazany okres czasu.

Porozumiewanie się wątków (cd.)

- Odblokowanie wątku następuje po wywołaniu metod **notify** lub **notifyAll** albo po upływie określonego czasu oczekiwania.
- Warunek czekania na danym obiekcie powinien być sprawdzany w pętli tak, aby po odblokowaniu wątku czekającego mógł on sprawdzić czy nadal spełniony jest oczekiwany warunek.

Wzajemna blokada wątków

- Wzajemna blokada wątków polega na tym, że warunkiem dalszego działania dwóch wątków jest uzyskanie dostępu do wzajemnie blokowanych synchronizowanych obiektów.
- Np. *wątek 1* wywołuje na rzecz obiektu *o1* synchronizowaną metodę *m1*. Obiekt *o1* zostaje zaryglowany. W tym samym czasie *wątek 2* wywołuje na rzecz obiektu *o2* synchronizowaną metodę *m2*. Obiekt *o2* zostaje także zaryglowany.

Wzajemna blokada wątków (cd.)

- W metodzie *m1* występuje wywołanie metody na rzecz obiektu *o2*. Ponieważ obiekt ten jest zaryglowany *wątek 1* zostaje zablokowany i oczekuje na odblokowanie tego obiektu. W metodzie *m2* występuje wywołanie metody na rzecz obiektu *o1*. Ponieważ obiekt ten też jest zaryglowany *wątek 2* zostaje zablokowany i oczekuje na odblokowanie tego obiektu. Oba wątki więc wzajemnie się blokują.
- Jedynym sposobem zapobieżenia wzajemnej blokadzie wątków jest odpowiednie zaprojektowanie programu.