PROGRAMOWANIE OBIEKTOWE I GUI

dr inż. Michał Tomaszewski

katedra Metod Programowania
Polsko-Japońska Akademia Technik Komputerowych

WYKŁAD 05

Poruszyliśmy zagadnienia:

- Strumienie cd

WYKŁAD 05

Poruszyliśmy zagadnienia:

- Strumienie cd
- Współbieżność

Plan wykładu:

- Współbieżność cd

PROGRAMOWANIE WSPÓŁBIEŻNE

WSPÓŁBIEŻNOŚĆ

Program jest **współbieżny**, jeśli jego wykonanie wiąże się z więcej niż jednym **przepływem strowania**.

WSPÓŁBIEŻNOŚĆ

Program jest **współbieżny**, jeśli jego wykonanie wiąże się z więcej niż jednym **przepływem strowania**.
Każdy przepływ sterowania jest realizowany przez odrębny **wątek**.

WSPÓŁBIEŻNOŚĆ

Program jest **współbieżny**, jeśli jego wykonanie wiąże się z więcej niż jednym **przepływem strowania**.

Każdy przepływ sterowania jest realizowany przez odrębny **wątek**.

Początkowo wykonuje się tylko wątek **główny** i wątki **systemowe**.

WSPÓŁBIEŻNOŚĆ

Program jest **współbieżny**, jeśli jego wykonanie wiąże się z więcej niż jednym **przepływem strowania**.

Każdy przepływ sterowania jest realizowany przez odrębny **wątek**.

Początkowo wykonuje się tylko wątek **główny** i wątki **systemowe**.

Niezależnie od liczby procesorów, program wielowątkowy zachowuje się tak, jakby każdy przepływ sterowania był realizowany przez oddzielny procesor.

TWORZENIE WĄTKU

Użycie klasy pochodnej od `Thread`:

```
class Runner
    extends Thread {

    public void run(){
        // ...
    }
}

// ...

public static void main(String[] args){
    new Runner().start();
}
```

Użycie klasy `Thread` przez implementację interfejsu `Runnable`:

```
class Program
    implements Runnable {

    public static void main(String[] args){
        new Thread(this).start();
    }

    public void run(){
        // ...
    }
}
```

WYKLUCZENIE DOSTĘPU

Program współbieżny musi zapewnić **wykluczanie** jednoczesnego dostępu do wspólnego zasobu dowolnej pary wątków korzystających z tego zasobu.

WYKLUCZENIE DOSTĘPU

Program współbieżny musi zapewnić **wykluczanie** jednoczesnego dostępu do wspólnego zasobu dowolnej pary wątków korzystających z tego zasobu.

Zaniedbanie tego wymagania może spowodować, że stan zasobu stanie się **nieokreślony**.

Wykluczenie współbieżnego wykonania wycinka programu odbywa się za pomocą instrukcji:

```
synchronized (lock){
    block
}
```

w której **lock** jest odnośnikiem do obiektu **synchronizatora**, a **block** jest sekwencją instrukcji stanowiących **sekcję krytyczną**

Funkcją **synchronizowaną** jest metoda zadeklarowana ze specyfikatorem **synchronized**.

```
Type synchronized fun(par, par, .., par){
    block
}
```

to jest niejawnie zastępowane definicją:

```
Type fun(par, par, .., par){
    synchronized (lock){
        block
    }
}
```

Koordynowanie ma na celu **wstrzymanie** przepływu sterowania wątku od chwili, gdy zostanie on **uwolniony** przez inny wątek.

```
synchronized (readyLock) {
    // ...
    try {
        while(!gameReady)
            readyLock.wait();
    } catch(InterruptedException) {
        return;
    }
    // ...
}
```

Koordynowanie ma na celu **wstrzymanie** przepływu sterowania wątku od chwili, gdy zostanie on **uwolniony** przez inny wątek. Wstrzymanie wątków odbywa się za pomocą metody **wait**, a ich uwolnienie za pomocą metod **notify** i **notifyAll**.

```
synchronized (readyLock) {
    // ...
    try {
        while(!gameReady)
            readyLock.wait();
    } catch(InterruptedException) {
        return;
    }
    // ...
}
```

KOORDYNOWANIE WĄTKÓW

Koordynowanie ma na celu **wstrzymanie** przepływu sterowania wątku od chwili, gdy zostanie on **uwolniony** przez inny wątek.

KOORDYNOWANIE WĄTKÓW

Koordynowanie ma na celu **wstrzymanie** przepływu sterowania wątku od chwili, gdy zostanie on **uwolniony** przez inny wątek. Wstrzymanie wątków odbywa się za pomocą metody **wait**, a ich uwolnienie za pomocą metod **notify** i **notifyAll**.

Koordynowanie ma na celu **wstrzymanie** przepływu sterowania wątku od chwili, gdy zostanie on **uwolniony** przez inny wątek. Wstrzymanie wątków odbywa się za pomocą metody **wait**, a ich uwolnienie za pomocą metod **notify** i **notifyAll**.

```
synchronized (readyLock) {
    // ...
    readyLock.notify();
    gameReady = true;
    // ...
}
```

PROBLEMY WSPŁBIERZNOŚCI

- Wyścigi warunków

PROBLEMY WSPŁBIERZNOŚCI

- Wyścigi warunków
- Zakleszczenia

PROBLEMY WSPŁBIERZNOŚCI

- Wyścigi warunków
- Zakleszczenia
- Zagłodzenie

PROBLEMY WSPŁBIERZNOŚCI

- Wyścigi warunków
- Zakleszczenia
- Zagłodzenie
- Livelock

PROBLEMY WSPŁBIERZNOŚCI

- Wyścigi warunków
- Zakleszczenia
- Zagłodzenie
- Livelock
- Interferencja wątków

PROBLEMY WSPŁBIERZNOŚCI

- Wyścigi warunków
- Zakleszczenia
- Zagłodzenie
- Livelock
- Interferencja wątków
- Błędy w spójności pamięci

PROBLEM WYŚCIGU WARUNKÓW

Wyścigi warunków występują, gdy dwa lub więcej wątków jednocześnie uzyskuje dostęp do wspólnego zasobu, a wynik zależy od kolejności wykonania.

PROBLEM ZAKLESZCZENIA

Zakleszczenie występuje, gdy dwa lub więcej wątków oczekuje na zwolnienie zasobów potrzebnych do kontynuowania działania przez siebie nawzajem. Może to prowadzić do sytuacji, w której oba wątki są zablokowane i nie mogą kontynuować działania.

PROBLEM ZAGŁODZENIA

Zagłodzenie występuje, gdy wątek nie może uzyskać dostępu do procesora lub zasobów, których potrzebuje do wykonania swojego zadania, ponieważ wątki o wyższym priorytecie stale korzystają z tych zasobów.

PROBLEM LIVELOCK

Livelocki występują, gdy dwa lub więcej wątków ciągle zmienia swoje stany w odpowiedzi na zmiany stanu innych, ale żaden z nich nie może kontynuować działania.

PROBLEM INTERFERENCJI WĄTKÓW

Interferencja wątków występuje, gdy dwa lub więcej wątków jednocześnie uzyskuje dostęp do wspólnego zasobu i zakłócają wykonanie siebie nawzajem, co prowadzi do nieprzewidywalnych wyników.

PROBLEM BŁĘDÓW SPÓJNOŚCI PAMIĘCI

Wielowątkowość może prowadzić do błędów w spójności pamięci, gdy wątki uzyskują dostęp do współdzielonych zmiennych bez odpowiedniej synchronizacji. Może to prowadzić do nieprawidłowych danych lub awarii programu.

`Concurrency` API umożliwiło wysokopoziomową pracę z wątkami za pomocą **ExecutorService**.

`Concurrency` API umożliwiło wysokopoziomową pracę z wątkami za pomocą `ExecutorService`. Dzięki temu zadania mogą być tworzone, uruchamiane i zarządzane asynchronicznie bez potrzeby tworzenia wątków.

`Concurrency` API umożliwiło wysokopoziomową pracę z wątkami za pomocą `ExecutorService`. Dzięki temu zadania mogą być tworzone, uruchamiane i zarządzane asynchronicznie bez potrzeby tworzenia wątków.

```
ExecutorService executor =
    Executors.newSingleThreadExecutor();
executor.submit(
    () -> {
        // ...
    }
);
```

Dziękuję