Тема 4: Процеси (потоки). Стан процесу. Взаємодія процесів. Тупики
Питання:

1. **Процеси (потоки). Стан процесу. Взаємодія процесів.**
2. **Клас Thread і інтерфейс Runnable**
3. **Механізм рандеву. Примітиви Send/Recive**
4. **Взаємодія процесів. Тупики**

1. Процеси (потоки). Стан процесу. Взаємодія процесів

Поток (нить - англ. *thread*) або повніше нить виконання (англ. *thread of execution*), часто застосовується назва потік виконання) та англіцизм тред — в інформатиці так називається спосіб програми розщепити себе на дві чи більше одночасні (чи псевдо-одночасні) задачі. Реалізація нитей та процесіввідрізняються в різних операційних системах, але загалом нить міститься всередині процесу і різні ниті одного процесу спільно розподіляють деякі ресурси, в той час як різні процеси ресурси не розподіляють.

В системах з одним процесором багатонитевість реалізується загалом поділом часу виконання («кванти часу»), дуже подібно до паралельного виконання багатьох задач: процесор послідовно переключається між різними нитями. Це переключення контексту відбувається настільки швидко, що у кінцевого користувача створюється ілюзія одночасного виконання. На багатопроцесорних чи набагатоядерних системах робота нитей здійснюється справді одночасно, бо різні ниті і процесивиконуються буквально одночасно різними процесорами або ядрами процесора.

Багато сучасних операційних систем прямо підтримують квантування часу і багатопроцесорну роботу нитей через планувальник процесів.Ядро операційної системи дозволяє програмісту маніпулювати нитями через інтерфейс системних викликів. Деякі реалізації викликають*ниті ядра*, оскільки легковагові процеси (англ. *lightweight process, LWP*) є спеціальним типом нитей ядра, що розподіляють деякі стани і інформацію.

Поза тим, програма може емулювати роботу нитей, використовуючи таймер, сигнали або інші методи, щоб перервати власне виконання і послідовно виконувати різні задачі власним квантуванням часу. Такий спосіб іноді зветься *нитями користувацького простору* (англ. *user-space threads*) або волокнами.

Порівняння нитей з процесами

Ниті відрізняються від традиційних процесів багатозадачних операційних систем, в тому що процеси:

- на загал, незалежні,
- дублюють значну частину інформації про стан,
- мають окремий адресний простір,
- взаємодіють тільки через системні міжпроцесорні механізми комунікацій.

Ниті всередині процесу, з іншої сторони, розподіляють інформацію про стан процесу, і прямо доступаються до спільної пам'яті та інших ресурсів. Переключення контексту між нитями процеса на загал швидше, ніж переключення контексту між процесами. Описуючи ситуацію такі системи, як Windows NT та OS/2, кажуть, що мають «дешеві» ниті та «дорогі» процеси; в інших операційних системах ситуація не дуже відмінна.

Процеси, ниті і волокна

Процес є «найважчим» об'єктом для планувальника ядра операційної системи. Власні ресурси процесу розміщені операційною системою. Ресурси включають пам'ять, відкриті файли та пристрої, сокети, та вікна. Процеси не розділяють з кимось адресний простір чи залучені файли, за винятком явних методів, таких як успадкування файлів чи сегментів спільної пам'яті,

або робота з файлами в режимі спільного доступу. Типово, процеси запобіжно багатозадачні. Але Windows 3.1 і старі версії Mac OS використовували кооперативну багатозадачність.

Нить є «найлегшим» об'єктом для планувальника ядра. Щонайменше одна нить існує всередині кожного процесу. Якщо в процесі співіснують багато нитей, вони розподіляють одну пам'ять і файлові ресурси. Ниті мають запобіжну багатозадачність, якщо планувальник процесів операційної системи запобіжний. Ниті не мають своїх власних ресурсів, за винятком стеку, копії регістрів включно з лічильником задач, і власне-нитевим зберіганням даних (якщо така можливість забезпечується).

В деяких ситуаціях є різниця між «нитями ядра» та «нитями користувача» — перші управляються і плануються ядром, другі управляються і плануються в просторі користувача. В цій статті термін «нить» використовується для позначення поняття «нить ядра», а поняття «волокно»(англ. *fiber*) означає нить користувача. Волокна мають кооперативне планування: працююче волокно має явно зупинитися і передати управління іншому волокну. Волокно може бути запущено будь-якою ниттю в одному процесі.

Ниті одного процесу розподіляють один адресний простір. Це дозволяє паралельно працюючому коду бути тісно взаємопов'язаним і зручно обмінюватися даними без залучення складних методів міжпроцесорного обміну. Але, доступні всім нитям, навіть прості структури даних стають вразливі до помилок типу стану гонитви, якщо вони потребують більше однієї машинної команди під час присвоєння нових значень: дві ниті можуть одночасно спробувати змінити значення структури даних і внаслідок отримати зовсім не те, що бажалося. Такі помилки буває дуже важко виправити і ізолювати.

Щоб запобігти цьому, програмний інтерфейс нитей пропонує методи синхронізації, такі як *м'ютекс* (mutex) для блокування структур даних під час доступу при паралельному виконанні. На однопроцесорних системах, нить, що запитала закритий м'ютекс, має перейти в пасивний (сонний) режим і планувальник переключає контекст іншої ниті. В багатопроцесорних системах, нить (за відсутності інших конкуруючих нитей виконання на процесорі) стане запитувати м'ютекс, доки той на звільниться. Обидва ці способи зле впливають на продуктивність, а в випадку мультипроцесорності виникає навантаження на шину пам'яті, особливо якщо деталізація блокування висока.

Java реалізує вбудовану підтримку багатопотокового програмування. Багатопотокова програма містить дві або більше частин, які можуть виконуватися одночасно.

Кожна частина такої програми називається потоком (`Thread`), і кожний потік задає окремий потік виконання. Інакше кажучи, багатопотоковість - це спеціалізована форма багатозадачності.

У середовищі потокової багатозадачності найменшим елементом керованого коду є потік. Це означає, що одна програма може виконувати дві або більше задач одночасно.

Ще однією перевагою багатопоковості є зведення до мінімуму часу очікування. Це особливо важливо для інтерактивних мережних середовищ, у яких працює Java, тому що в них наявність очікування і простоїв звичайне явище.

В однопотокових середовищах програма змушена очікувати закінчення таких задач, перш ніж переходити до наступної, навіть якщо більшу частину часу програма простоює, очікуючи введення.

Багатопотоковість допомагає скоротити час простою, оскільки інші потоки можуть виконуватися, поки один очікує.

2. Клас Thread і інтерфейс Runnable

Багатопотокова система Java вбудована в клас `Thread`, що і доповнює його інтерфейс `Runnable`. Клас `Thread` інкапсулює потік виконання.

Щоб створити новий потік, програма повинна або розширити клас `Thread`, або реалізувати інтерфейс `Runnable`.

Клас `Thread` визначає кілька методів, які допомагають управляти потоками. Потік може знаходитися в одному зі станів, позначених наступними константами класу `Thread.State`:

NEW – потік створений, але ще не запущений;
RUNNABLE – потік виконується;
BLOCKED – потік блокований;
WAITING – потік чекає закінчення роботи іншого потоку;
TERMINATED – потік закінчений.

Клас Thread

У класі `Thread` вісім конструкторів. Основний з них,

```
Thread(Threadgroup group, Runnable target, String name, long stacksize);
```

створює потік з іменем `name`, що належить групі `group` і виконуючий метод `run()` об'єкта `target`. Останній параметр, `stacksize`, задає розмір стека і залежить від операційної системи.

Усі інші конструктори звертаються до нього з тим або іншим параметром, рівним `null`:

```
Thread() – створюваний потік буде виконувати свій метод run();
Thread(Runnable target);
Thread(Runnable target, String name);
Thread(String name);
Thread(Threadgroup group, Runnable target, String name);
Thread(Threadgroup group, Runnable target);
Thread(Threadgroup group, String name).
```

Ім'я потоку `name` не має ніякого значення, воно не використовується віртуальною машиною Java і застосовується тільки для розрізнення потоків програмі.

Після створення потоку його треба запустити методом `start()`. Віртуальна машина Java почне виконувати метод `run()` цього об'єкта-потоку. Потік завершить роботу після виконання методу `run()`.

Потік можна призупинити статичним методом

```
sleep(long ms);
```

на `ms` мілісекунд. Якщо обчислювальна система здатна відраховувати наносекунди, то можна призупинити потік з точністю до наносекунд методом

```
sleep(long ms, int nanosec);
```

Коли програма Java стартує, негайно починає виконуватися один потік. Зазвичай його називають головним потоком `(main thread)` програми

Потік виконання створюється і запускається в такий спосіб

```
Thread th = new Thread(task);
th.setName("Потік");
th.start();
```

Перед запуском у потоці повинна бути розміщене задача, яка повинна бути виконана.

Створюючи кілька потоків, у яких інкапсульовані виконувані задачі, створюється паралелізм обчислень.

Задачі створюються в такий спосіб:

```
Task<Void> task = new Task<Void>() {
    @Override protected Void call() throws Exception {
        while (true) {
            Platform.runLater(new Runnable() {
                @Override public void run() {
                    //необхідні обчислення
                }
            });
            //=== затримка
            Thread.sleep(10);
        }

    }
};
```

У прикладі задача має тип `<Void>`, тому не повертає значень. Блок обчислень розміщується в чергу подій платформи.

Перед закриттям вікна необхідно зупинити всі запущені потоки в оброблювачі події закриття вікна:

```
//====== закрыть потоки при закрытии окна
stage.setOnCloseRequest(new EventHandler<WindowEvent>() {
    @Override
    public void handle(WindowEvent event) {
        toConsole("Close");
        task_1.cancel();
        task_2.cancel();
        task_3.cancel();
        task_4.cancel();
        task_5.cancel();

    }
});
```

Використання інтерфейсу `Runnable` для створення потоку

Найпростіший спосіб створення потоку - це оголошення класу, який реалізує інтерфейс `Runnable`.

Можна створити потік з будь-якого об'єкта, який реалізує інтерфейс `Runnable`. Щоб реалізувати інтерфейс `Runnable`, клас повинен оголосити єдиний метод `run()`.

Усередині методу `run()` треба визначити код, який, буде виконуватися в потоці.

Метод `run()` встановлює крапку входу для іншого, паралельного потоку усередині програми. Цей потік завершиться, коли метод `run()` поверне керування.

Після того як буде оголошений клас, який реалізує інтерфейс `Runnable`, треба створити об'єкт типу `Thread` із цього класу. У класі `Thread` визначено кілька конструкторів.

Той, який повинен використовуватися в цьому випадку, виглядає таким чином.

```
Thread(Runnable об'єкт_потоку, String ім'я_потоку)
```

У цьому конструкторі ***об'єкт_потоку*** – це екземпляр класу, який реалізує інтерфейс ***Runnable.*** Він визначає, де почнеться виконання потоку. Ім'я нового потоку передається в параметрі ***імені_потоку.***

Після того як новий потік буде створений, він не запускається доти, поки не буде викликаний метод `start()`, оголошений у класі `Thread`.

Метод `start()` виконує виклик методу `run()`.

В JavaFx модель потоків більш розвинена і стала схожою на багатозадачність. Тому потоки для програміста знаходяться у двох станах: стан виконання і стан останова.

Більш того, стало можливим виконувати в потоці не тільки класи і методи, але і окремі оператори. Наприклад, обчислення суми площ прямокутників простіше задати у вигляді задачі `Task` і помістити задачу в потік виконання `Thread`:

```
//////////////////////////////////////////////
private void CreateTask_5() {
    //====== создать задачу 5
    task_5 = new Task<Void>() {
        String msg = new String("Общая площадь: ");
        @Override
        protected Void call() throws Exception {
            //====== постоянно
            while (true) {
                //===== поставить в очередь
                Platform.runLater(new Runnable() {
                    @Override
                    public void run() {
                        //=== вычислить сумму площадей
                        square_sum = (int) square_1 + (int) square_2 + (int) square_3 + (int) square_4;
                        //=== отобразить
                        square_display_sum.setText(msg + Double.toString((int) square_sum) + " кв.см");
                    }
                });
                //=== задержка
                Thread.sleep(10);
            }
        }
    };
    flag_created_5 = true;
    //======== запустить поток
    Thread t_5 = new Thread(task_5);
    t_5.setName("Поток 5");
    t_5.start();
}
```

3. Механізм рандеву. Примітиви Send/Recive

Багато реалізацій волокон лежать повністю в області користувача. Як наслідок, переключення контексту між волокнами всередині процесу, є максимально ефективним, бо воно не вимагає ніякої взаємодії з ядром: переключення контексту може бути зроблене локальним збереженням регістрів процесора, що використані у волокні, і завантаженням регістрів наступного волокна, що виконуватиметься . Оскільки планування відбувається в області користувача, політика планування може бути простіше скроєна найефективнішим чином для швидкодії програми загалом.

Проте, використання системних викликів блокування у волокнах може бути проблематичним. Якщо волокно виконує системний виклик блокування, інші волокна процесу не можуть виконуватися, доки системний виклик не відпрацює. Типовим прикладом цієї проблеми є виконання введення-виведення: більшість програм пише у вивід синхронно. Коли операція вводу-виводу розпочалася, системний виклик зроблено, і він не поверне керування, доки операція вводу-виводу не буде завершена. На весь цей час робота всього процесу блокована ядром, процес «стоїть», бо інші волокна процесу не можуть отримати керування.

Загальним рішенням цієї проблеми є забезпечення програмного синхронного інтерфейсу з використанням внутрішнього неблокуючого введення-виведення, так, щоб під час виконання запису-читання інші волокна могли виконуватися. Таке рішення може бути запроваджене для інших блокуючих викликів. Інший спосіб — писати програму без використання синхронного введення-виведення та інших блокуючих системних викликів.

Win32 забезпечує програмний інтерфейс волокон за допомогою механізму рандеву і примітивів Send/Recive Використання нитей ядра спрощує код програми, ховаючи деякі найскладніші аспекти нитей в ядро. Програміст вже не піклується про планування нитей та явну передачу процесора, код може бути написаний у звичному процедурному стилі, включаючи виклик функцій блокування. Проте, ниті ядра на однопроцесорних системах можуть бути змушені переключити контекст будь-якої миті, що може призвести до стану гонитви і помилок

паралелізму, які можуть бути прихованими. На мультипроцесорних системах це може бути ще гостріше, оскільки ниті ядра можуть в дійсності виконуватися на різних процесорах.

Звісно, волокна можуть бути реалізовані поза підтримкою з боку операційної системи, хоча деякі операційні системи або бібліотекизабезпечують їхню явну підтримку. Наприклад, Microsoft Windows (Windows NT 3.51 SP3 і пізніші) підтримують програмний інтерфейс волокон для застосунків, тож за бажання, можна спробувати підвищити продуктивність програми самому, замість покладатися на планувальник ядра (який може бути не налаштований для застосунку). Прикладом може бути планувальник користувацького режиму вMicrosoft SQL Server 2000, який працює в просторі волокон.

4. Взаємодія процесів. Тупики

У багатонитевому середовищі часто виникають проблеми, зв'язані з використанням паралельними виконуваними нитями одних і тих же даних або пристроїв. Для вирішення подібних проблем використовуються такі методи взаємодії нитей, як взаємовиключення (м'ютекси),семафори, критичні секції і події.

взаємовиключення (mutex, м'ютекс) — це об'єкт синхронізації, який встановлюється в особливий сигнальний стан, коли не зайнятий якоюсь ниттю. Тільки одна нить володіє цим об'єктом у будь-який момент часу, звідси і назва таких об'єктів (від англійського mutually exclusive access — взаємно виключний доступ) — одночасний доступ до загального ресурсу виключається. Після всіх необхідних дій м'ютекс звільняється ниттю, надаючи іншим нитям доступ до загального ресурсу.

Семафори є доступні ресурси, які можуть займатися кількома нитями в один і той же час, поки обсяг ресурсів не спустіє. Тоді додаткові ниті повинні чекати, поки необхідна кількість ресурсів не буде знову доступна. Семафори дуже ефективні, оскільки вони дозволяють одночасний доступ до ресурсів.

Події. Події корисні в тих випадках, коли необхідно послати повідомлення ниті, що відбулося певна подія. Наприклад, при асинхронних операціях вводу/виводу з одного пристрою, система встановлює подію в сигнальний стан коли закінчується якась з цих операцій. Одна нить може використовувати кілька різних подій в декількох операціях, що перекриваються, а потім чекати приходу сигналу від будь-якого з них.

Критичні секції забезпечують синхронізацію подібно м'ютексам за винятком того, що об'єкти, що представляють критичні секції, доступні в межах одного процесу. Події, м'ютекси і семафори також можна використовувати в однопроцесному застосунку, проте критичні секції забезпечують швидший і ефективніший механізм синхронізації взаємного виключення.

<u>Вправи і завдання до теми №4</u>

1. **Дати визначення потокам і процесам.**
2. **Опишіть роботу класу Thread і інтерфейсу Runnable**
3. **Опишите работу механізму рандеву**
4. **Взаємодія процесів. Тупики**