­­МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

­­­

Лабораторна робота №12

з курсу «Об’єктно-орієнтоване програмування»

для студентів базового напрямку 6.08.04 "Комп’ютерні науки"

(заочна форма навчання)

Варіант 14

Виконав студент гр. КНз-2

Чалий Михайло

­­

Львів 2014

# Багатопотоковість у Java

## Мета роботи

Ознайомитися з багатопотоковоим програмування у Java.

## Теоретичні відомості

Основне поняття сучасних операційних систем — процесс(process). Як і всі загальні поняття, процес важко визначити. Можна розуміти під процесом виконувану (runnable) програму, але треба памятати про те, що у процесу єсть декілька станів. Процес може в будь-який момент перейти до виконання машинного коду іншої програми, а також "заснути" (sleep) на деякий час, призупинивши виконання програми. Він може бути вивантажений на диск. Кількість станів процесу і їх особливості залежать від операційної системи.

Всі сучасні операційні системи багатозадачні(multitasking), вони запускають і виконують зразу декілька процесів. Одночасно може працювати браузер, текстовий редактор, музичний програвач. На екрані дисплея відкриваються декілька вікон, кожне з яких звязано із своїм працюючим процесом. Якщо на компютері тільки один процессор, то він переключається з одного процесу на другий, створюючи видимість одночасної роботи. Переключення відбувається по закінченню одного або декількох "тиків" (ticks). Розмір тику залежить від тактової частоти процесора і звичайно має порядок 0,01 секунди. Процесам назначаються різні пріоритети(priority). Процеси з низьким пріоритетом не можуть перервати виконання процесу з більш високим пріоритетом, вони менше займають процесор, тому виконуються повільно, як говорять, "на фоні". Самий високий пріоритет у системних процесів, наприклад, у диспетчера(scheduler), який як раз і займається переключенням процесора з процесу на процесс. Такі процеси не можна переривати, поки вони не закінчать работу, інакше компютер швидко прийде в хаотичний стан.

Кожному процесу виділяється певна область оперативної памяті для розміщення коду програми і її даних — його адресний простір*.* В цю ж область записується частина даних про процес, складаюча його контекст(context). Дуже важливо розділить адресний простір різних процесів, щоб вони не могли змінити код і дані один одного. Операційні системи по-різному відносяться до забезпечення захисту адресних просторів процесів. MS Windows NT/2000 ретельно розділяють адресні простори, витрачаючи на це багато ресурсів і часу. Це підвищує надійність виконання програми, але утруднює створення процесу. Такі операційні системи погано справляються з управлінням великого числа процесів.

Операційні системи сімейства UNIX менше турбуються про захист памяті, але легше створюють процеси і здатні управляти сотнею одночасно працюючих процесів. Крім управління работою процесів операційна система повинна забезпечити засоби їх взаємодії: обмін сигналами і повідомленнями, створення спільних декільком процесам областей памяті і виконуваного коду програми. Ці засоби також вимагають ресурсів і уповільнюють роботу компютера.

Роботу багатозадачної системи можна спроститс і прискорити, якщо дозволити взаємодіючим процесам працювати в одному адресному просторі. Такі процесси називаються threads. Буквальний переклад - "нитка", але ми зупинимося на слові "підпроцес". Підпроцеси створюють нові труднощі для операційної системи — треба дуже уважно слідкувати за тим, щоб вони не заважали один одному при запису в спільні ділянки памяті, — але зате полегшують взаємодію підпроцесів. Створення підпроцесів і управління ними — це справа операційної системи, але в Java введені засоби для виконання цих дій. Оскільки програми, написані на Java, повинні працювати у всіх операційних системах, ці засоби дозволяють виконувати тільки самі загальні дії.

Коли операційна система запускає віртуальну машину Java для виконання додатку, вона створює один процес з декількома підпроцесами. *Головний* (main) підпроцес виконує байт-коди програми, а саме, він зразу ж звертається до методу main() додатку. Цей підпроцес може породити нові підпроцеси, які, в свою чергу, здатні породить підпроцеси і т. д. Головним підпроцесом аплета являеться один із підпроцесів браузера, в якому аплет виконується. Головний підпроцес не грає ніякої особливої ролі, просто він створюється першим.

Підпроцес в Java створюється і управляється методами класу Thread. Після створення обєкта цього класу одним із його конструкторів новий підпроцес запускається методом start (). Отримати посилку на поточний підпроцес можна статичним методом Thread.currentThread() ;

Клас Thread реалізує інтерфейс RunnabІe. Цей інтерфейс описує тільки один метод run(). Новий підпроцес буде виконувати те, що записано в цьому методі. Між іншим, клас Thread містить тільки пусту реалізацію методу run(), тому клас Thread не використовується сам по собі, він завжди розширюється. При його розширенні метод run() перевизначається. Метод run() не містить аргументів, так як нікому передавати їх значення в метод. Він не повертає значення, його нікуди передавати. До методу run() не можна звернутися із програми, це завжди робиться автоматично виконуючою системою Java при запуску нового підпроцесу методом start ().

Отже, задати дії створюваного підпроцесу можна двома способами: розширити клас Thread або реалізувати інтерфейс RunnabІe. Перший спосіб дозволяє використовувати методи класу Thread для управління підпроцесом. Другий спосіб застосовується в тих випадках, коли треба тільки реалізувати метод run(), або клас, створюючий підпроцес, уже розширяє якийсь інший клас.

## Завдання

Програма моделює обслуговування одного потоку процесів двома центральними процесорами комп'ютера із загальною чергою. Якщо черговий процес генерується в мить, коли будь-який з процесорів вільний, процес поступає на обробку, інакше процес стає в чергу. Якщо один з процесорів звільняється, і в черзі є процеси, процес віддаляється з черги. Визначте максимальний розмір черги для двох однакових процесорів.

## Реалізація

Лістінг 3. Broker.java

**import** java.util.Scanner;

**import** java.util.concurrent.LinkedBlockingQueue;

**import** java.util.concurrent.atomic.AtomicBoolean;

**public** **class** Broker {

**public** **static** **void** main(String[] args) {

// Програма моделює обслуговування одного потоку процесів двома

// центральними процесорами комп'ютера із загальною чергою. Якщо

// черговий процес генерується в мить, коли будь-який з

// процесорів вільний, процес поступає на обробку, інакше процес

// стає в чергу. Якщо один з процесорів звільняється, і в черзі є

// процеси, процес віддаляється з черги. Визначте максимальний

// розмір черги для двох однакових процесорів.

**final** LinkedBlockingQueue<String> workQueue = **new** LinkedBlockingQueue<String>();

**final** AtomicBoolean shouldRun = **new** AtomicBoolean(**true**);

Thread t1 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**while**(shouldRun.get()){

String workDef;

**try** {

workDef = workQueue.take();

System.*out*.println("Worker #1: " + workDef + " started");

Thread.*sleep*(5000);

System.*out*.println("Worker #1: " + workDef + " stopped");

} **catch** (InterruptedException e) {

System.*out*.println("Worker #1 failed");

}

}

}

});

Thread t2 = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**while**(shouldRun.get()){

String workDef;

**try** {

workDef = workQueue.take();

System.*out*.println("Worker #2: " + workDef + " started");

Thread.*sleep*(20000);

System.*out*.println("Worker #2: " + workDef + " stopped");

} **catch** (InterruptedException e) {

System.*out*.println("Worker #2 failed");

}

}

}

});

t1.start();

t2.start();

System.*out*.println("Enter command to execute, stop to exit");

Scanner in = **new** Scanner(System.*in*);

**while**(**true**){

String command = in.nextLine();

**if** ("stop".equalsIgnoreCase(command)){

**break**;

}

**try** {

workQueue.put(command);

} **catch** (InterruptedException e) {

System.*out*.println("Failed to queue " + command);

}

}

in.close();

System.*out*.println("Stopping threads...");

shouldRun.set(**false**);

**try** {

t1.join();

t2.join();

} **catch** (InterruptedException e) {

System.*out*.println("Failed to stop threads...");

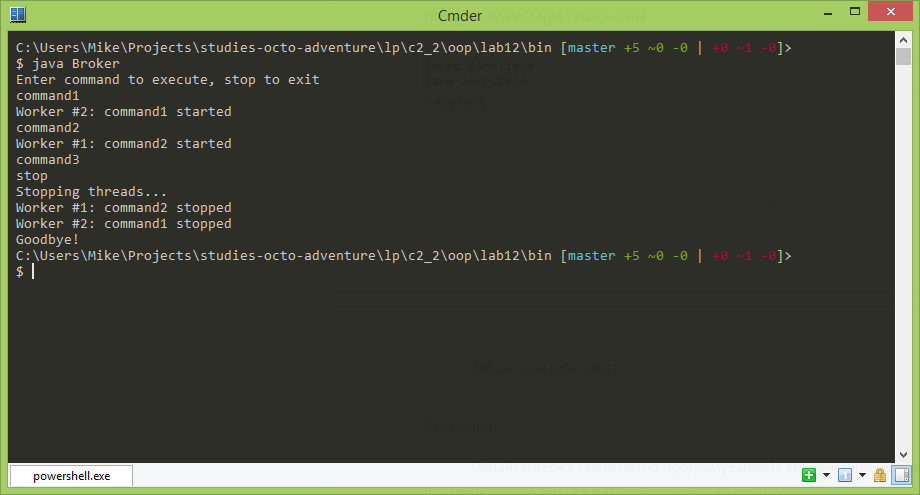
}

System.*out*.println("Goodbye!");

}

}

## Результат



## Висновки

Ознайомитиdcz з багатопотоковоим програмування у Java