Изпитна тема № 18:

Конкурентно програмиране

*План-тезис*: Конкурентност. Изпълнение на програма. Процес. Блокиращи операции. Видове блокиращи операции. Нишка. Връзка между процес и нишка. Създаване на нишки. Управление на нишки. Синхронизация между нишки. Проблеми при работа с нишки

[1 Конкурентност. 1](#_Toc132707671)

[1.1 Конкурентно програмиране 2](#_Toc132707672)

[2 Изпълнение на програма 2](#_Toc132707673)

[3 Процес 3](#_Toc132707674)

[4 Блокиращи операции. 3](#_Toc132707675)

[4.1 Блокиращи операции в графичен потребителски интерфейс 4](#_Toc132707676)

[5 Нишка 4](#_Toc132707677)

[6 Връзка между процес и нишка. 5](#_Toc132707678)

[7 Създаване на нишки. 5](#_Toc132707679)

[7.1 C# 5](#_Toc132707680)

[7.2 Java 5](#_Toc132707681)

[8 Управление на нишки. 6](#_Toc132707682)

[9 Синхронизация между нишки. 7](#_Toc132707683)

[9.1 Синхронизационни примитиви 7](#_Toc132707684)

[9.1.1 C# 8](#_Toc132707685)

[9.1.2 Java 8](#_Toc132707686)

[10 Проблеми при работа с нишки 8](#_Toc132707687)

# Конкурентност.

В компютърните науки конкурентност е способността на различни части или единици на програма / алгоритъм да се изпълняват непоследователно или в частичен ред, без това да засяга крайния резултат. Това дава възможност за паралелно изпълнение на отделните конкурентни единици, което може значително да подобри общата производителност на приложението при многопроцесорните и многоядрените системи.

Казано с по-технически термини, конкурентността се отнася до възможността за декомпозиране (разделяне) на дадена програма (алгоритъм / проблем) на отделни компоненти (изчислителни единици), като реда на тяхното изпълнение може да е частично зависим от други компоненти или абсолютно независим.

Конкурентността е композицията от независимо изпълняващи се изчисления. Конкурентността не е паралелизъм: конкурентността е свързана с **работа** с много неща наведнъж, а паралелизмът е свързан с **извършването** на много неща наведнъж.

Конкурентността е свързана със структурата, а паралелизмът е свързан с изпълнението. Конкурентността осигурява начин за структуриране на решение за справяне с даден проблем, който може (но не непременно) да подлежи на паралелизация.

## Конкурентно програмиране

Конкурентно програмиране e програмна парадигма за създаване на компютърни програми, в които многобройни изчисления могат да се изпълняват в застъпващи се периоди от време (конкурентно), вместо последователно (където едно изчисление трябва да завърши преди друго да започне). Конкурентните изчисления могат да се изпълняват на един или повече процесори на едно компютърно устройство, или на процесори разпределени в мрежа от устройства.

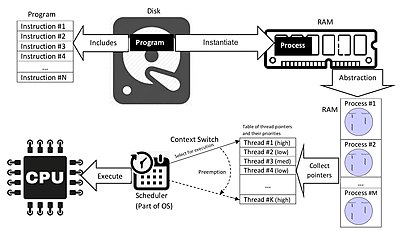
Предимства на този тип изчисления включват увеличена производителност на приложенията, където конкурентното изпълнение на процесите позволява броя на изпълнени задачи за определено време да се увеличи; способност да се изпълняват допълнителни задачи докато даден процес зарежда или записва обемни данни;

# Изпълнение на програма

Компютърната програма представлява набор от инструкции, които се изпълняват от компютъра, за да решават определена задача или да извършат определена функция. Инструкциите се изпълняват от централния процесор.

Програмата има и част, която можем да разберем, която се нарича изходен код. Изходният код на повечето програми се състои от списък с инструкции, които реализират определени алгоритми. Кодът може да бъде написан на различни езици, като C#, Java, Python и д.р. За да може изходния код да бъде изпълнен от процесора първо е необходимо той да бъде преобразуван в машинен код. Това става с помощта на транслиращи програми (компилатори или интерпретатори).

Когато дадена програма трябва да бъде изпълнена, операционната система я зарежда в паметта и стартира нов процес. След това централния процесор започва да извлича, декодира и изпълнява всяка машинна инструкция.



# Процес

В компютърната техника процесът е екземпляр на компютърна програма, която се изпълнява. Един процес винаги съществува в точно едно състояние. Състоянията могат да бъдат: създаден, стартиран, блокиран (изчакващ да настъпи определено събитие) и терминиран.

Всеки процес притежава уникален **идентификатор на процеса** (pid). Чрез този идентификатор потребителите на операционната система могат да го управляват или да получават информация за него. Също така, процесите имат и приоритет.

Докато компютърната програма е пасивна колекция от инструкции, обикновено съхранявани във файл на диск, процесът е реалното изпълнение на тези инструкции, след като са заредени от диска в паметта. Няколко процеса могат да бъдат свързани с една и съща програма. Например отварянето на няколко копия на една и съща програма, често води до изпълнение на повече от един процес.

В съвременните операционни системи, процесите притежават редица ресурси: памет, системни дескриптори, атрибути, касаещи сигурността, както и състоянието на процесорните регистри (контекст). Операционните системи изолират процесите и техните ресурси т.е. един процес няма достъп до паметта и ресурсите на друг процес. Повечето операционни системи предлагат механизми за комуникация между процесите (IPC – Inter-process-communication).

Процесите могат да създават други процеси (т.нар. дъщерни процеси).

# Блокиращи операции.

Блокираща операция е всяка операция , която блокира изпълнението на даден процес / нишка, до настъпване на определено събитие, изчакване на даден ресурс да стане наличен или до приключване на входно/изходна операция.

Примери:

* Четене или запис от / във файл;
* Достъп до отдалечен ресурс ;
* Извършване на тежък изчислителен процес;
* Изпълнение на асинхнонен метод.

Примери за блокиращи(синхронни) методи:

* C#: методите на класа Console: Read() и ReadLine() в C#;
* Java: методите на класа Scanner: nextLine(), nextInt(), nextLong(), nextFloat(), nextChar().

## Блокиращи операции в графичен потребителски интерфейс

Приложенията с графичен потребителски интерфейс (GUI) разполагат с една основна нишка, която се занимава с обработката на постъпилите от операционната система събития (натискане на клавиш или бутон на мишката), както и с опресняване на потребителския интерфейс. Блокирането на главната нишка би довело до застиване / замръзване на приложението.

При тези приложения, програмистът се абонира за определено събитие, например натискане на клавиш. Когато такова събитие настъпи, главната нишка изпълнява съответен метод (обработчик на събития). Дейностите, които изпълнява въпросния метод трябва да не отнемат много време, защото главната нишка няма да може да обработва съобщенията, свързани с пречертаване и опресняване на графичните елементи на приложението.

Поради тази причина, дейностите, които отнемат време, следва да се изпълняват в отделна нишка. По този начин главната нишка може спокойно да обработва настъпилите събития и без забавяне да опреснява графичния интерфейс на приложението.

Използването на допълнителни нишки в графичните приложения може да доведе и до проблеми. В случай, че две или повече нишки се опитат да променят състоянието на даден графичен компонент, могат да възникнат нежелани ефекти, като: странно поведение, възникване на софтуерни изключения, неконсистентност на данните и др. **Затова, по правило, елементите на графичния интерфейс трябва да променяни единствено в контекста на главната нишка**. Различните работни рамки предлагат механизми за синхронизация и комуникация между главната нишка и останалите нишки.

# Нишка

Нишката е най-малката последователност от програмирани инструкции, които могат да се управляват независимо от планировчика на задачите (scheduler), който обикновено е част от операционната система. Реализацията на нишките и процесите се различава между операционните системи, но в повечето случаи нишката принадлежи на даден процес.

Множеството нишки на даден процес могат да се изпълняват едновременно, като споделят ресурси като: памет, докато различните процеси не споделят тези ресурси. По-специално нишките на даден процес споделят неговия изпълним код и стойностите на неговите динамично разпределени променливи и глобални променливи, които не са локални за нишки във всеки даден момент.

Използването на нишки е полезно, защото позволява програмата да се раздели на независими части, които могат да се изпълняват едновременно, и така да увеличи скоростта на изпълнение на програмата. В зависимост от операционната система и конкретната програма, може да се използват различни начини за създаване и управление на нишки.

Всяка нишка притежава уникален идентификатор (thread id / tid) и приоритет.

Основно правило: в един момент, една нишка може да извършва само едно действие.

Нишките имат статус / състояние. В общия случай, статусите са:

1. Създадена – нишката е създадена и очаква сигнал за стартиране;
2. Готова за изпълнение – нишката изчаква планировчика на задачите да й назначи процесорно време;
3. Работеща – процесорът изпълнява програмните инструкции;
4. Блокирана / изчакваща – нишката изчаква настъпването на определено събитие, като например даден ресурс да стане достъпен, завършване на входно/изходна операция и др. В това състояние нишката не консумира процесорно време;
5. Неактивна / терминирана – изпълнението на нишката е приключило.

# Връзка между процес и нишка.

В съвременните многозадачни операционни системи, процесите не изпълняват инструкции, а са по-скоро контейнери, които съдържат оперативна информация. Нишките са тези, които изпълняват инструкциите.

Един процес може да има минимум една нишка, която наричаме **главна нишка**. Такива приложения наричаме еднонишкови. Ако процесът притежава повече от една нишка, то той се нарича многонишково приложение. И в двата случая главната нишка е тази, която изпълнява инструкциите в основния метод на приложението (обикновено main).

Всяка нишка споделя ресурсите на процеса, към който принадлежи, като: памет, манипулатори към файлове и т.н. Можем да направим извода, че нишките имат собствен стек, но споделят общата динамична памет на процеса. Те имат достъп и до глобалните променливи в програмата.

Нишките имат т. нар. вътрешно състояние (контекст), което включва стек и регистри. Когато процесорът превключва между изпълнението на различните нишки, възниква т. нар. превключване на контекста (context switch) – вътрешното състояние на текущата нишка се съхранява и се зарежда контекста на следващата нишка.

# Създаване на нишки.

## C#

В езика C# за работа с нишки се използва класа Thread, който се намира в пространството от имена System.Threading. Конструкторът на класа Thread очаква референция към метод (делегат), който нишката трябва да изпълни. Когато изпълнението на метода приключи, нишката бива терминирана. От тук можем да направим извода, че нишките не могат да се преизползват.

using System.Threading;

// Метод, който нишката ще изпълнява

public void WorkThreadFunction() { Console.WriteLine("I am child thread"); }

// Създаваме нишка и й подаваме референция към метода WorkThreadFunction

Thread thread = new Thread(new ThreadStart(WorkThreadFunction));

// Стартираме нишката

thread.Start();

Console.WriteLine("I am the main thread!");

// Текущата нишка блокира, докато нишката thread не приключи

thread.Join();

Можем да подаваме аргументи на нишките, чрез типа ParametrizedThreadStart:

using System.Threading;

// Параметризиран метод, който нишката ще изпълнява

public void WorkThreadFunction(object? arg)

{

var a = (int)arg;

Console.WriteLine($"Child thread: the number is: {a}");

}

// Създаваме нишка и й подаваме референция към метода WorkThreadFunction

Thread thread = new Thread(new ParametrizedThreadStart(WorkThreadFunction));

// Стартираме нишката и подаваме стойността (15), която искаме

// да бъде достъпна в метода WorkThreadFunction

thread.Start(15);

Console.WriteLine("I am the main thread!");

// Текущата нишка блокира, докато нишката thread не приключи

thread.Join();

Възможно е на конструктора да бъде подадена ламбда функция:

var thread = new Thread(**() =>**

**{**

Console.WriteLine("I’m a thread (via lambda)!");

**}**);

## Java

В Java нишките са обекти, подобни на всички останали обекти. Те са инстанции на класа java.lang.Thread или на негови наследници.

В Java съществуват два начина за създаване на нишки:

А) Чрез наследяване на класа Thread и предефиниране на метода run(). Методът run() е стартовата точка на всяка една нишка. В това отношение той може да се сравни с main() метода на главната нишка.

Б) Чрез имплементиране (реализиране) на интерфейса Runnable. Интерфейсът Runnable има единствен метод run.

class MyThread implements Runnable {

// Методът run ще бъде изпълнен в контекста на нишката

// т.е. нишката ще изпълни метода run()

public void run() {

System.out.println("I am a thread");

}

}

public class Test {

public static void main(String[] args) {

// Създаване на екземпляр на класа MyThread

MyThread ob = new MyThread();

// Създаване на нишка, базирана на ob, с име Thread1

Thread thread = new Thread(ob, "Thread1");

// Стартиране на нишката

thread.start();

System.out.println("I am the main thread");

// Текущата нишка изчаква нишката thread да приключи

thread.join();

}

}

# Управление на нишки.

Управлението на нишки включва няколко основни възможности:

1. Стартиране на нишки (Start) – в зависимост от езика за програмиране, в някои случаи създадените нишки не са стартирани и се изисква от програмиста да извика метод от класа, който да стартира изпълнението на нишката;
2. Изчакване на дадена нишка да приключи своята работа (Join) – когато главния метод на нишката приключи своето изпълнение, нишката бива терминирана (приключва своята работа). За да изчакаме нормалното спиране на дадена нишка можем да използваме методи, като Join. Този метод блокира извикващата нишка, докато втората нишка не приключи;
3. Принудително спиране на нишки (Termination) – някои езици за програмиране предоставят методи за принудително спиране на нишките. Обикновено това става чрез предизвикване на софтуерно изключение в основния метод на дадена нишка. Не е препоръчително да се използва тази техника за спиране на нишки, защото крие риск от нежелани ефекти.   
   Терминирането на нишки е кооперативен процес. Първата нишка известява втората нишка, че трябва да спре изпълнение. Втората нишка периодично проверява дали трябва да спре изпълнение. Ако такова е изискано, втората нишка излиза от главния си метод.
4. Временно спиране възстановяване на нишки (Suspend & Resume) – някои програмни езици позволяват временно спиране и последващо възстановяване работата на нишките. Използването на тези методи не е препоръчително, защото могат да доведат до неочаквани странични ефекти;
5. Yield – извикващата нишка отдава остатъка от процесорното си време на други нишки.

# Синхронизация между нишки.

При конкурентното програмиране едновременният достъп на няколко нишки до споделени ресурси (променливи, обекти и т.н.) може да доведе до неочаквано или погрешно поведение на програмата, или грешен краен резултат. Така че тези части от програмата, където се осъществява достъп до споделения ресурс, трябва да бъдат защитени по начини, които избягват едновременния достъп на няколко нишки. Един от начините да направите това е известен като критична секция или критичен регион. Този защитен раздел с програмен код не може да бъде изпълнен от повече от един процес или нишка наведнъж; другите нишки спират своята работа и изчакват, докато работещата нишка напусне критичната секция.

В случай, че едновременния достъп на няколко нишки до споделения ресурс не бъде синхронизиран, има опасност от възникване на състояние на състезание (race condition). Състезателно състояние е състоянието на софтуера, при което поведението на системата зависи от последователността или има времевата зависимост от неконтролируеми събития. Този тип софтуерни дефекти са трудни за откриване.

## Синхронизационни примитиви

Всяка операционна система и език за програмиране разполагат с т. нар. синхронизационните примитиви които можете да използвате, за да синхронизирате достъпа до споделен ресурс или да координирате взаимодействието между отделните нишки. Ето някои от най-често срещаните:

А) **Mutex** (mutual exclusion) - синхронизационна примитива, осигуряваща взаимно изключване при изпълнение на критичен участък от кода. Използва се за предотвратяване на едновременно изпълнение на блок код от множество нишки. В даден момент само една нишка може да заключи мутекса. Тя става негов собственик. Останалите нишки трябва да изчакат текущия собственик на мутекса да го отключи. При мутексите говорим за собственост.

Б) **Семафор** (semaphore) – примитив за синхронизация на работата на процеси и нишки, който се основава на брояч, върху който могат да се извършват две атомарни операции: увеличаване и намаляване на стойността с единица. Когато стойността достигне нула, следващия опит за намаляване с единица блокира извикващата нишка.

Може да мислим за семафора, като за журнал, в който държим информация за това колко единици от определен ресурс са налични.

Семафорите, които позволяват произволен брой ресурси, се наричат броячни семафори. Семафорите, които са ограничени до стойностите 0 и 1 (или заключени/отключени, недостъпни/налични), се наричат двоични семафори и се използват за прилагане на заключвания. Двоичните семафори приличат на мутексите. Основната разлика между тях е, че при семафорите не говорим за собственост, а мутексите имат собственици.

Б) За синхронизиране може да се използва и **цикъл на празен ход** (busy-waiting, busy-looping or spinning). Това е техника, при която дадена нишка периодично проверява дали дадено условие е удовлетворено или дали има достъп до критична секция.

Езиците за програмиране разполагат със запазени думи, чрез които можем да заключим определен регион от код (критична секция).

### C#

### Java

Запазената дума synchronized, зададена при декларацията на метод предизвиква синхронизиране на изпълнението на този метод по обекта, на който той принадлежи, а при статични методи – по класа, на който той принадлежи. Синхронизацията на програмен код по някакъв обект предизвиква заключване на този обект при започване на изпълнението на синхронизирания код и отключване на обекта при завършване на изпълнението на кода. Когато някоя нишка се опита да изпълни синхронизиран код, чийто обект е заключен, тя принудително изчаква отключването на този обект. Така код, синхронизиран по един и същ обект, не може да се изпълнява от две нишки едновременно и заявките за изпълнението му се изпълняват една след друга в някакъв ред.

public static void deposit1(

Account aAcc, double aSum) { … }

**synchronized** public static void deposit2(

Account aAcc, double aSum) { … }

public static void deposit3(Account aAccount, double aSum) {

**synchronized** (aAccount) {

double oldAmmount = aAccount.getAmmount();

double newAmmount = oldAmmount + aSum;

aAccount.setAmmount(newAmmount);

}

}

# Проблеми при работа с нишки

А) **Deadlock** (самоблокиране)

Самоблокиране (безизходна ситуация) възниква, когато две или повече нишки не могат да напреднат, тъй като ресурсът, изискван от първата нишка, се държи от втората, а ресурсът, изискван от втората нишка, се държи от първата. В този случай всички нишки са блокирани.

Б) **Liveness** (жизненост)

Способността на програма или приложение да се изпълнява своевременно се нарича жизнеспособност. Ако дадена програма претърпи самоблокиране, тогава тя не проявява жизненост.

В) **Безкрайно отлагане** (Live-Lock)

Live-Lock възниква, когато две нишки непрекъснато реагират в отговор на действията на другата нишка, без да постигнат реален напредък. Най-добрата аналогия е да си представим двама души, които се опитват да се пресекат един друг в коридора. Джон се движи наляво, за да позволи на Арън да мине, а Арън се движи надясно, за да пусне Джон да мине. И двамата се блокират взаимно. Джон вижда, че блокира Арън отново и се премества надясно, а Арън се премества наляво, виждайки, че блокира Джон. Никога не се пресичат и продължават да се блокират. Този сценарий е пример за Live-Lock. Изглежда, че процесът се изпълнява и не е в задънена улица, но в действителност не постига никакъв напредък.

Основната разлика, между самоблокиране и live-lock е, че в първия случай нишките са напълно блокирани. Във втория случай нишките функционират и консумират процесорно време.

Г) **Starvation** (гладуване)

Освен самоблокирането, нишка на приложение може също да изпита глад (starvation), когато никога не получава процесорно време или достъп до споделени ресурси. Други „алчни“ нишки непрекъснато отнемат споделени системни ресурси, като не позволяват на гладуващата нишка да напредне.

Изготвен от: Петър Георгиев