Објектно орјентисано програмирање

Владимир Филиповић

vladaf@matf.bg.ac.rs

Вишенитно програмирање

Владимир Филиповић
vladaf@matf.bg.ac.rs

Нити у језику Јава

Преко нити (енг. thread) је обезбеђен један облик паралелизма (конкурентности) у програмском језику Јава.

У презентацији ће бити описани основни појмови који се односе не нити и начин рада са нитима на ниском нивоу.

Литература:

1. P. Hyde, Java Thread Programming.
2. M Adler, D. Herst, Mastering Java Threads.
3. S. Oaks, H. Wong, Java Threads.

Математички факултет

vladaf@matf.bg.ac.rs

Конкурентност у рачунарству

Конкурентност у оперативним системима: редна обрада (енг. batch processing), рад у раздељеном времену (енг. time-sharing), мултипрограмирање.

Логичка и физичка конкурентност.

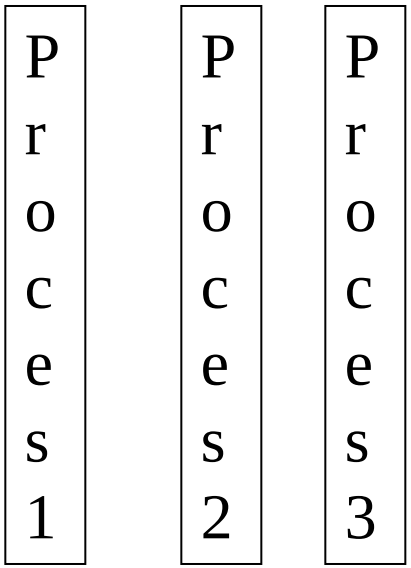
Програмски језици који подржавају мултипрограмирање обезбеђују логичку конкурентност.

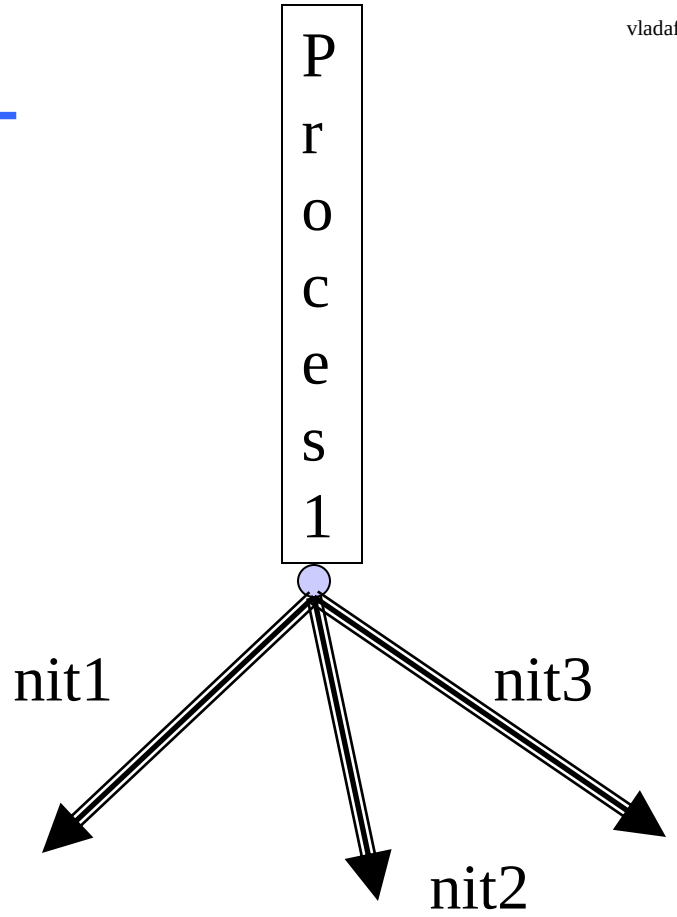
Процес - основни појам мултипрограмирања.

Математички факултет

vladaf@matf.bg.ac.rs

Процес и нит





Конкурентност у Јави је остваварена преко нити.

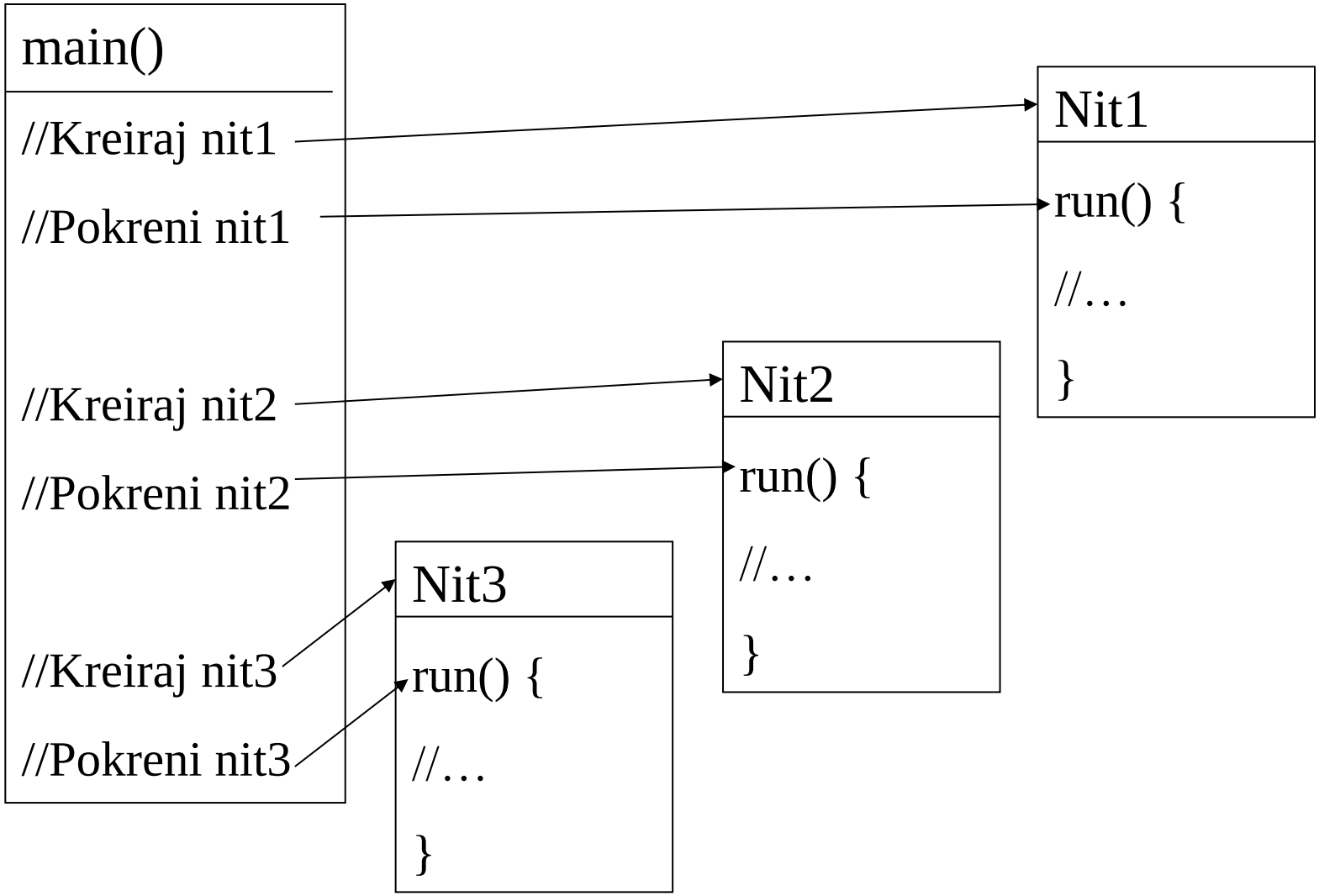
Нит је независан ток низа инструкција које се извршавају паралелно са неким другим токовима инструкција.

Целокупно функционисање Јаве је засновано на нитима.

Математички факултет

vladaf@matf.bg.ac.rs

Креирање нити



Математички факултет

vladaf@matf.bg.ac.rs

Креирање нити (2)

Методе за рад са нитима обезбеђује класа Thread. Конкретне нити су примерци класе Thread, али се могу креирати на два начина:

1. Као примерци неке поткласе класе Thread, при чему онда наслеђују све методе класе Thread од којих је најважнији метод run.

2. Као елементи неких класа које имплеметирају интерфејс Runnable и метод run() овог интерфејса. Ово је једини метод интерфејса Runnable и служи за дефинисање шта ће дата нит радити у класи где се користи.

Други начин се чешће користи јер је прихватљивији са гледишта практичне примене.

Математички факултет

vladaf@matf.bg.ac.rs

Креирање нити (3)

java.lang.Thread

- Thread(Runnable target)
  креира нову нит која позива метод run() објекта target који имплементира интерфејс Runnable.
- void start()
  започиње ову нит, позива метод run(). Метод моментално враћа контролу текућој нити. Нова нит се извршава конкурентно.
- void run()
  позива метод run() придруженог Runnable објекта.

java.lang.Runnable

- void run()
  мора бити предевазиђен, а у телу се наводе наредбе које ће се извршити покретањем нити.

Математички факултет

vladaf@matf.bg.ac.rs

Прекидање нити

Нит се завршава када се заврши њен метод run()и то извршавањем наредбе return, након извршавања последње наредбе у телу метода или уколико се деси избацивање изузетка који се не ухвати унутар метода.

Метод interrupt() се може користити како би се затражио завршетак нити.

Када се за нит позове метод interrupt(), постави се тзв. статус прекида нити. То је маркер који поседује свака нит и који би требало повремено да проверава да ли је тај њен маркер постављен.

Међутим, уколико је нит блокирана, није могуће проверити њен статус прекида.

Прекидање нити (2)

Ту је од значаја изузетак InterruptedException. Када се метод interrupt() позове за нит која је блокирана при позиву попут sleep() или wait() (о којима ће бити речи касније), позив метода се завршава избацивањем изузетка типа InterruptedException.

Није обавезно да нит за која је „прекинута“ мора да се заврши. Прекид просто служи да привуче њену пажњу. „Прекинута“ нит може да одлучи како ће реаговати на прекид. Неке нити су веома важне, па могу руковати изузетком и наставити своје извршавање. Међутим, прилично је уобичајено да нит интерпретира прекид као захтев за завршетком.

Прекидање нити (3)

У том случају, метод run() је следећег облика:

```
public void run(){
   try {
      …
      while(!Thread.currentThread().isInterrupted() && има још посла)
      {
         ради
      }
   } catch(InterruptedException e) {
   // нит је прекинута за време позива sleep() или wait()
   } finally {
      завршно чишћење, ако је потребно
   }
   // излазак из метода run() завршава нит
}
```

Прекидање нити (4)

java.lang.Thread

- void interrupt()
  шаље нити захтев за прекид. Статус прекида нити се поставља на true. Ако је нит тренутно блокирана позивом метода sleep(), избацује се изузетак типа InterruptedException.
- static boolean interrupted()
  тестира да ли је текућа нит (тј. нит која извршава ову наредбу) прекинута. Метод је статички. Позив овог метода има бочни ефекат - ресетује статус прекида текуће нити на false.
- boolean isInterrupted()
  тестира да ли је нит прекинута. За разлику од статичког метода interrupted(), позив овог метода не мења статус прекида нити.
- static Thread currentThread()
  враћа Thread објекат који представља нит која се тренутно извршава.

Стање нити

Нит се може налазити у једном од 6 стања:

1. new
2. runnable
3. blocked
4. waiting
5. timed waiting
6. terminated

Како би се утврдило у ком стању је тренутно нит, може се позвати метод getState().

Математички факултет
vladaf@matf.bg.ac.rs

Стање нити (2)

1. New
Када се нит креира коришћењем оператора new, нпр. new Thread(r), она се још увек не извршава. Таква нит је у стању new.

2. Runnable
Након што се позове метод start, нит прелази у стање runnable. Заправо, runnable нит може, а не мора да се извршава. На оперативном систему је да дâ нити време за извршавање. Након што нит започне извршавање, она не мора наставити да се извршава.
У ствари, пожељно је да се нити повремено паузирају, како би и друге нити добиле шансу да се извршавају. Детаљи извршавања нити зависе од оперативног система. Системи који имају тзв. „preemptive scheduling“ дају свакој runnable нити делић времена за извршавање. Након што се то време потроши, оперативни систем даје другој нити шансу да ради. Приликом избора следеће нити, оперативни систем узима у обзир приоритете нити. Међутим, мали уређаји, попут мобилних телефона могу да користе „cooperative scheduling“. У таквом уређају, нит губи контролу једино када позове метод yieldили када блокира или чека.

Стање нити (3)

3. Blocked

Нит која је у овом стању је привремено неактивна. Таква нит не извршава никакав кôд и користи минималне ресурсе. Када нит покуша да добије катанац објекта који тренутно држи нека друга нит, она прелази у стање blocked. Нит постаје одблокирана када се све остале нити одрекну катанца и распоређивач нити допусти тој нити да га узме.

Када је нит у стању blocked (или, наравно, када се заврши), друга нит ће бити распоређена да се извршава.

4. Waiting

Нит која је у овом стању је привремено неактивна. Таква нит не извршава никакав кôд и користи минималне ресурсе. Када нит чека да друга нит обавести распоређивача нити о неком услову, она је у стању waiting. Ово се дешава позивањем метода wait() класе Object или метода join класе Thread или чекањем на Lock или Condition из пакета java.util.concurrent.

Када је нит у стању waiting (или, наравно, када се заврши), друга нит ће бити распоређена да се извршава.

Стање нити (4)

5. Timed waiting

Неколико метода за рад са нитима поседује параметар timeout. Њихово позивање узрокује да нит пређе у стање timed waiting. Нит излази из овог стања након истека задатог времена или ако прими одговарајућу нотификацију. Методи са параметром timeout су sleep() класе Thread, wait() класе Object, join() класе Thread, tryLock() класе Lock и await() класе Condition.

6. Terminated

Нит је у стању terminated из једног од следећа два разлога:

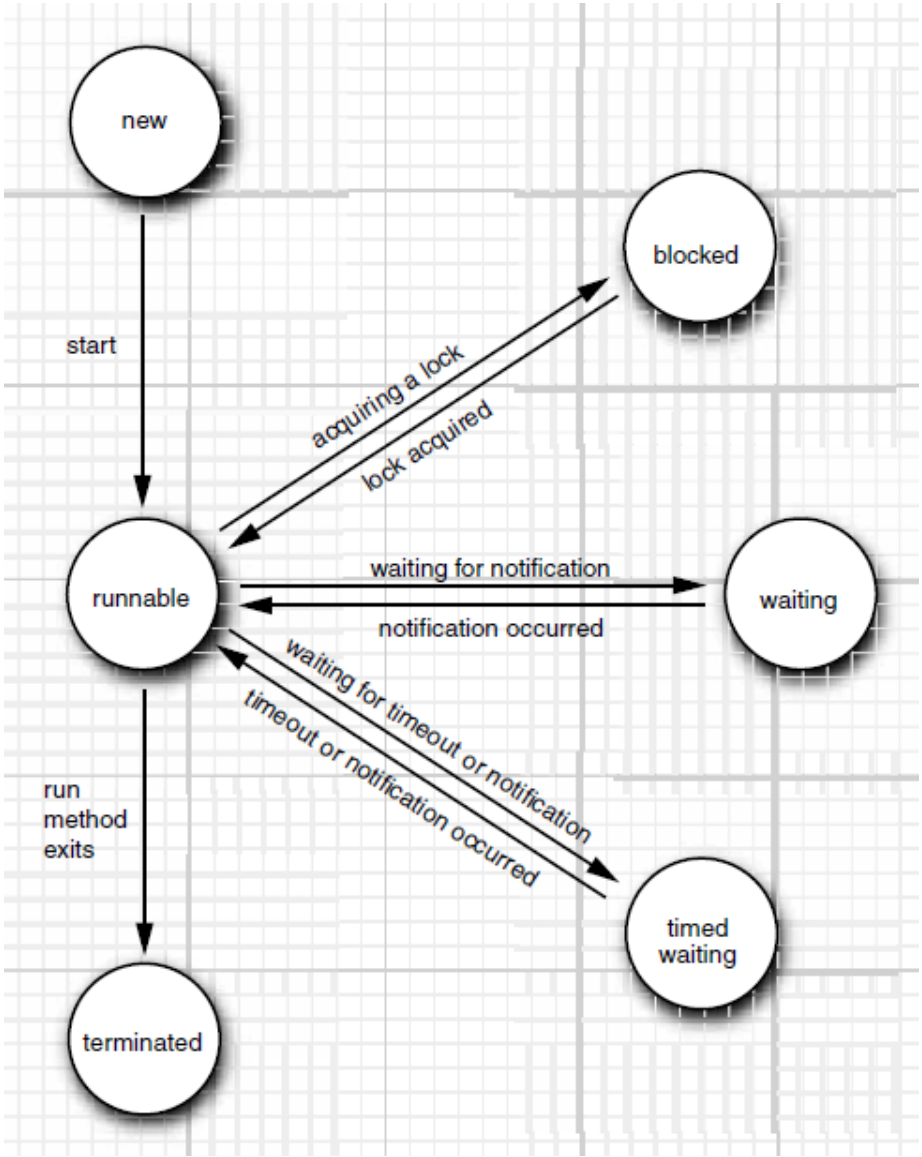
- умрла је природном смрћу јер је њен метод run() завршен нормално
- умрла је изненада јер је неухваћени изузетак завршио метод run().

Поред тога, могуће је убити нит позивом метода stop(). Међутим, овај метод је застарео и никада га не треба позивати из сопственог кôда.

Математички факултет

vladaf@matf.bg.ac.rs

Стање нити (5)



Математички факултет

vladaf@matf.bg.ac.rs

Стање нити (6)

java.lang.Thread

- void join()
  чека да се одређена нит заврши.
- void join(long millis)
  чека да се одређена нит заврши или да прође задати број милисекунди.
- Thread.State getState()
  враћа стање текуће нити: једно од NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
- void stop()
  зауставља нит. Застарео!
- void suspend()
  суспендује извршавање текуће нити тј. блокира текућу нит све док нека друга нит не позове метод resume. Застарео!
- void resume()
  наставља текућу нит. Валидан је само након што је позван метод suspend(). Застарео!

Приоритет нити

Нити могу имати разне приоритете. У принципу, нити које су интерактивне треба да имају виши приоритет у односу на нити које реализују обимна израчунавања.

Приоритет нити у Јави је одређен бројем. Овде нижи број одговара нижем приоритету, а виши број вишем приоритету.

Јава обезбеђује константе: Thread.MAX_PRIORITY, Thread.MIN_PRIORITY и Thread.NORM_PRIORITY за задавање приоритета.

Приоритет се задаје преко метода:

setPriority(int priority)

Математички факултет
vladaf@matf.bg.ac.rs

Приоритет нити (2)

java.lang.Thread

- void setPriority(int newPriority)
  поставља приоритет текуће нити. Приоритет мора бити између Thread.MIN_PRIORITY и Thread.MAX_PRIORITY. Користити Thread.NORM_PRIORITY за нормалан приоритет.
- static int MIN_PRIORITY
  минималан приоритет који нит може имати. Вредност је 1.
- static int NORM_PRIORITY
  подразумевани приоритет нити. Вредност 5.
- static int MAX_PRIORITY
  максималан приоритет који нит може имати. Вредност 10.
- static void yield()
  Уколико постоје друге runnable нити приоритета бар оноликог колики је приоритет текуће нити, оне ће бити распоређене следеће. Метод је статички.

Демон - нит

Нити које се извршавају у позадини да би подржали функционисање Јава окружења, називају се демон - нити.

Такве нити су: управљач часовником, скупљач отпадака, нит за ажурирање екрана итд.

Нити које креира програмер нису даемон - нити, већ корисничке нити. Може се подесити тако да се корисничке нити претворе у демон-нити.

java.lang.Thread

- void setDaemon(boolean isDaemon)
  означава текућу нит као демонску или корисничку. Овај метод мора се позвати пре него што се нит стартује.
- boolean isDeamon()
  враће true или false у зависности од тога да ли је текуће нит демон-нит

Груписање нити

Нити се организују по групама. Група нити је колекција нити којом је могуће манипулисати одједном.

Тако, на пример, нити једног аплета чине групу. Аплет може да манипулише само нитима из своје групе (не и другима!) Један аплет не може да стопира ажурирање екрана јер се то извршава помоћу нити друге групе.

Групе нити су хијерархијски организоване. Знају се родитељске нити и нити-деца (настали из родитељских). Постоји низ метода за манипулисање групама тредова:

getThreadGroup() из класе java.lang.Thread

getParent() из класе java.langGroup итд.

Груписање нити (2)

Подразумевано, све нити које креира програмер припадају истој групи нити, али могуће је извршити и другачија груписања.

Почев од Јаве 5.0 не треба користити групе нити у сопственим програмима.

Класа ThreadGroup имплементира интерфејс UncaughtExceptionHandler.

Њен метод uncaughtException() врши следећу акцију:

1. ако група нити има родитеља, позива се метод uncaughtException() родитељске групе.
2. иначе, ако метод getDefaultExceptionHandler() врати не-null handler, он се позива
3. иначе, ако је Throwable инстанца класе ThreadDeath, ништа се не дешава
4. иначе, име нити и запис са стека извршавања за Throwable се штампају на System.err.

Математички факултет

vladaf@matf.bg.ac.rs

Проблем синхронизације

Нити често деле неке ресурсе (податке) и уколико се није пажљиво приликом операција са њима, могу да се појаве нежељени резултати.

На пример, нека две нити штампају два различита документа. Ако те нити раде истовремено, мало штампа један, мало други (и тако наизменично), онда ћемо добити мешавину два документа, а не одвојено одштампана два документа.

Пример. Симулира се рад банке са рачунима корисника. На псеудослучајан начин се генеришу трансакције којима се премештају средства између ових рачуна. Сваки од рачуна има једну нит. Свака од трансакција премешта одређену (псеудослучајним путем генерисану) своту новца са рачуна који опслужује дата нит на други (псеудослучајним путем изабрани) рачун.

Математички факултет

vladaf@matf.bg.ac.rs

Проблем синхронизације (2)

Пример (наставак). Имплементација је једноставна. Креира се класа Bank која садржи метод transfer, који преноси неку количину новца са једног рачуна на други. У овој једноставној имплементацији се не разматра могућност неконзистенције стања на рачуну. Следи код метода:

```
public void transfer(int from, int to, double amount)
// CAUTION: unsafe when called from multiple threads
{
   System.out.print(Thread.currentThread());
   accounts[from] -= amount;
   System.out.printf(" %10.2f from %d to %d", amount, from, to);
   accounts[to] += amount;
   System.out.printf(" Total Balance: %10.2f%n", getTotalBalance());
}
```

Математички факултет

vladaf@matf.bg.ac.rs

Проблем синхронизације (3)

Пример (наставак). Класа TransferRunnable има следећу структуру. Њен метод run врши премештање средстава са датог (фиксираног) рачуна. У свакој од итерација, метод run псеудослучајно бира циљни рачуни и износ, позива метод transfer над објектом класе bank и потом спава током псеудослучајног периода:

```
class TransferRunnable implements Runnable {
   . . .
  public void run() {
    try {
      int toAccount = (int) (bank.size() * Math.random());
      double amount = maxAmount * Math.random();
      bank.transfer(fromAccount, toAccount, amount);
      Thread.sleep((int) (DELAY * Math.random()));
    }
    catch(InterruptedException e) {}
  }
}
```

Математички факултет

vladaf@matf.bg.ac.rs

Проблем синхронизације(4)

Пример (наставак). Као што смо видели, на крају сваке трансакције метод method рачуна укупан износ и приказује га на стандардном излазу.

Овај програм никад не завршава са радом, па је потребно притиснути комбинацију тастера CTRL+C са завршетак.

Када се извршава програм, требало би да укупан износ новца (тј. сума стања на свим рачунима) буде непромењена, јер се новац само сели са једног рачуна на други.

Типичан излаз из који генерише програм је:

```
. . .
Thread[Thread-14,5,main] 521.51 from 14 to 22 Total Balance: 100000.00
Thread[Thread-13,5,main] 359.89 from 13 to 81 Total Balance: 100000.00
. . .
Thread[Thread-36,5,main] 401.71 from 36 to 73 Total Balance: 99291.06
Thread[Thread-35,5,main] 691.46 from 35 to 77 Total Balance: 99291.06
```

Математички факултет

vladaf@matf.bg.ac.rs

Проблем синхронизације (5)

Пример (наставак). Као што се може видети, постоји грешка. У првих неколико трансакција баланс банке остаје $100,000, што је тачна вредност када има 100 рачуна на којима се стање $1,000.

Међутим, после неког времена, баланс почиње да се полако мења. Некад се догоди да грешка настане брже, а некад треба много времена да баланс постане погрешан. У сваком случају, банка са оваквим софтвером не би била сигурно место за штедњу!

У овом примеру, проблем настаје када две нити симултано покушавају да ажурирају исти рачун. Претпоставимо да две нити симултано извршавају наредбу:

```
accounts[to] += amount;
```

Проблем је у томе што ова наредба није атомичка операција.

Математички факултет
vladaf@matf.bg.ac.rs

Проблем синхронизације (6)

Пример (наставак). Ова Јава наредба се може реализовати следећим наредбама бајт-кода (тј. следећим корацима):

1. Учитај accounts[to] у регистар JVM.
2. Изврши сабирање тј. увећање садржаја регистра JVM.
3. Врати резултат из регистра у accounts[to].

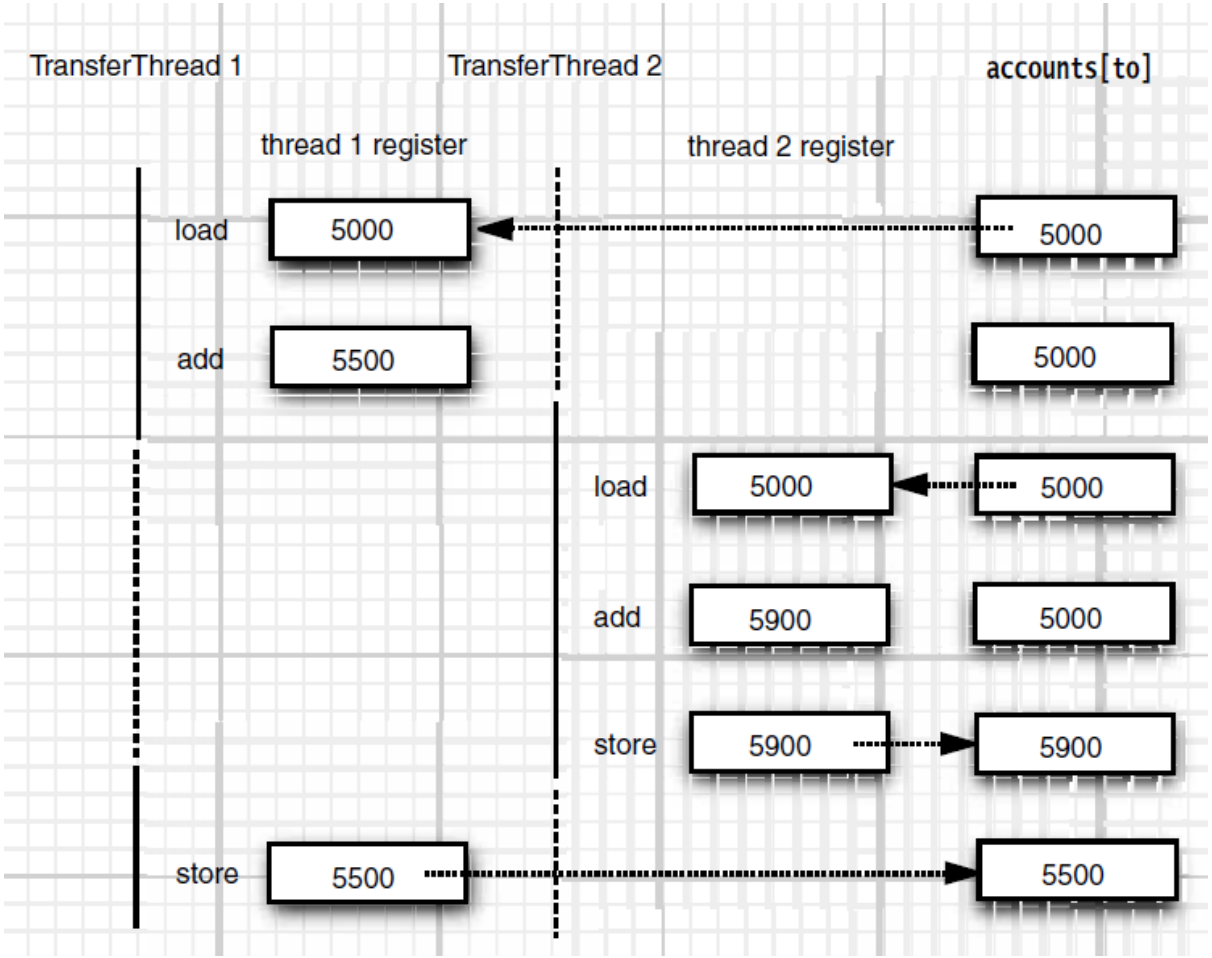
Претпоставимо да је прва нит извршила коараке 1 и 2, а да је потом прекинуто њено извршавање. Нека се онда пробудила друга нит и нека је ажурирала исти елеменат низа рачуна (извршила сва три корака). Потом се пробудила прва нит и извршила корак 3. Овим извршењем корака 3 су избрисане промене које је направила друга нит и рачуни су постали некоректни.

Математички факултет

vladaf@matf.bg.ac.rs

Проблем синхронизације (7)

Пример (наставак). Претходно описану ситуацију илуструје дијаграм активности:



Математички факултет
vladaf@matf.bg.ac.rs

Проблем синхронизације (8)

Пример (наставак). Шанса за уочавање проблема је у овом примеру повећана преплитањем наредби за приказ на стандардни излаз са наредбама којима се ажурира стање на рачуну.

Ако би се уклониле наредбе за приказ, битно се умањује ризик од настанка грешке, јер нит има јако мало посла пре него што заспе, па није много очекивано да ће распоређивач прекинути рад нити у сред израчунавања.

Међутим, ризик од грешке је смањен, али није нестао. Ако се извршава јако много нити на веома оптерећеном рачунару, и програм без приказа ће опет дати погрешне резултате.

Основни проблем је то што метод transfer може бити прекинут у сред рада. Ако би се могло обезбедити да се овај метод изврши до краја пре него што нит изгуби контролу, тада стање рачуна никада неће бити корумпирано.

Математички факултет

vladaf@matf.bg.ac.rs

Катанци

Разумевање синхронизације нити је једноставније када се прво проуче катанци и услови.

Пакет обезбеђује класе за реализацију ових фундаменталних механизама.

У основи, заштита блока кôда помоћу катанца, тј. помоћу објекта типа ReentrantLock из пакета java.util.concurrent изгледа овако:

```
mojKatanac.lock(); // ReentrantLock објекат
try{
        критична секција
}finally{
        mojKatanac.unlock();
}
```

Оваква конструкција гарантује да у сваком тренутку само једна нит може ући у критичну секцију. Оног тренутка када једна нит закључа катанац, ниједна друга нит не може проћи наредбу lock.

Катанци (2)

Када друге нити позову метод lock, оне се деактивирају све док нит која је позвала метод lock не откључа катанац.

Напомена. Од кључног је значаја да се операција unlock извршава унутар клаузе finally. Наиме, уколико кôд унутар критичне секције избаци изузетак, катанац се мора откључати - иначе, остале нити ће заувек остати блокиране.

Пример. Коришћење катанца ради заштите метода transfer класе Banka.

```
public class Banka1
{
  private Lock bankaLock = new ReentrantLock();
  // ReentrantLock implementira interfejs Lock
  …
```

Математички факултет

vladaf@matf.bg.ac.rs

Катанци (3)

Пример (наставак).

```
public void transfer(int sa, int na, int iznos)
{
   bankaLock.lock();
   try
   {
      System.out.println(Thread.currentThread());
      racuni[sa] -= iznos;
      System.out.printf(" %10.2f sa %d na %d", iznos, sa, na);
      racuni[na] += iznos;
      System.out.printf(" Ukupan iznos: %10.2f%n",
      getUkupnoStanje());
   }
   finally
   {
      bankaLock.unlock();
   }
}
}
```

Математички факултет

vladaf@matf.bg.ac.rs

Катанци (4)

Претпоставимо да једна нит позове метод transfer и буде прекинута пре него што заврши. Претпоставимо да друга нит такође позове метод transfer. Друга нит не може добити катанац и блокира у позиву метода lock. Она се деактивира и мора да чека да прва нит заврши извршавање метода transfer. Када прва нит откључа катанац, друга може наставити своје извршавање.

Када се у програм унесу ове измене, он се може извршавати заувек, а укупно стање у банци никада неће постати погрешно.

Приметимо да сваки Banka објекат поседује сопствени ReentrantLock објекат. Ако две нити покушају да приступе истом Banka објекту, катанац служи за серијализацију приступа. Међутим, ако две нити приступају различитим Banka објектима, онда свака нит добија различит катанац и ниједна не блокира, што и јесте жељено понашање.

Математички факултет

vladaf@matf.bg.ac.rs

Катанци (5)

Катанац се зове „са поновљеним уласцима“ (енг. reentrant), јер нит може у више наврата да добија катанац који већ поседује.

Катанац прати тзв. „бројач држања“, односно угњеждених позива метода lock. Нит мора да позове unlock за сваки позив lock како би ослободила катанац. Захваљујући овоме, кôд заштићен катанцем може позвати други метод који користи исте катанце.

Генерално, штите се блокови кôда који ажурирају дељени објекат или му приступају. Тако се постиже да се ове операције извршавају у потпуности пре него што друга нит може да користи исти објекат.

Напомена. Потребно је бити пажљив да се кôд у критичној секцији не заобиђе избацивањем изузетка. Уколико се избаци изузетак пре краја критичне секције, finally клауза ослобађа катанац, али објекат може бити у оштећеном стању.

Катанци (6)

java.util.concurrent.locks.Lock

- void lock()
  дохвата текући катанац, блокира ако нека друга нит већ поседује катанац.
- void unlock()
  ослобађа текући катанац.

java.util.concurrent.locks.ReentrantLock

- ReentrantLock()
  конструише катанац који се може користити за заштиту критичне секције.
- ReentrantLock(boolean fair)
  конструише катанац са задатом политиком праведности. Праведан катанац фаворизује нит која је чекала дуже. Међутим, гарантовање праведности може значајно утицати на перформансе. Према томе, подразумевано је да катанци нису праведни.

Напомена. Звучи лепше бити праведан, али праведни катанци су доста спорији од регуларних. Једино у случају да постоје специфични разлози због којих је праведност кључна за програм, само тада треба користити праведне катанце.

Математички факултет

vladaf@matf.bg.ac.rs

Условни објекти

Често нит уђе у критичну секцију само да би открила да не може да настави даље док неки услов не буде испуњен. За руковање нитима које су добиле катанац, али не могу радити нешто корисно користи се условни објекат. Из историјских разлога, условни објекти се често називају и условне променљиве.

Пример. У случају банкесе поставља додатни, логични захтев: не сме бити преноса новца са рачуна који нема довољно средстава. Приметимо да не можемо користити кôд попут

```
if(racuni[sa] >= iznos)
   banka.transfer(sa, na, iznos);
```

Наиме, потпуно је могуће да текућа нит буде деактивирана између успешне провере постојања довољних средстава и позива метода transfer, па да до тренутка када се нит поново активира, стање на рачуну постане ниже од потребног.

Математички факултет

vladaf@matf.bg.ac.rs

Условни објекти (2)

Пример (наставак). Дакле, неопходно је обезбедити да ниједна друга нит не може модификовати стање између провере услова и акције трансфера. То се чини заштитом и услова и акције трансфера катанцем.

```
public void transfer(int sa, int na, int iznos){
   bankaLock.lock();
   try{
      while(racuni[sa] < iznos){
         // cekanje
         …
   }
   // prenos sredstava
   …
   }finally{
       bankaLock.unlock();
   }
}
```

Математички факултет
vladaf@matf.bg.ac.rs

Условни објекти (3)

Пример (наставак). Шта да се ради када нема довољно новца на рачуну? Чекамо док га нека друга нит не дода. Али постоји проблем: нит која подиже новац је управо добила ексклузивни приступ катанцу bankaLock, па ниједна друга нит нема шансу да изврши улагање на рачун. Ту насупају условни објекти.

Сваки катанац може имати један или више придружених условних објеката. Условни објекат се добија позивом метода newCondition().

Пожељно је да име објекта асоцира на услов који он представља.

```
class Banka1{
   private Condition dovoljnoSredstava;
   …
   public Banka1(){
      …
      dovoljnoSredstava = bankaLock.newCondition();
   }
}
```

Математички факултет

vladaf@matf.bg.ac.rs

Условни објекти (4)

Пример (наставак). Уколико се при извршавању метода за пренос открије да није доступна довољна количина новца, онда се позива dovoljnoSredstava.await();

Текућа нит се онда деактивира и ослобађа се катанац bankaLock. Тиме се допушта да га узме нека друга нит која ће, евентуално, повећати стање на рачуну.

Након што нит позове метод await, та нит постаје део тзв. скупа чекајућих нити за тај услов. Нит се не реактивира када катанац постане доступан, већ остаје деактивирана све док друга нит не позове метод signalAll за исти услов.

Дакле, када друга нит изврши трансфер новца, треба да позове dovoljnoSredstava.signalAll();

Математички факултет

vladaf@matf.bg.ac.rs

Условни објекти (5)

Позив метода signalAll реактивира све нити које чекају на тај услов. Нити се уклоне из скупа чекајућих нити за тај услов, оне су поново у стању runnable и распоређивач их може активирати.

У том тренутку оне ће покушати да поново добију катанац. Чим катанац постане доступан, једна од њих ће га добити и наставити тамо где је стала, а то је завршетак позива метода await.

Нит која се сада извршава поново тестира услов одређен условним објектом. Нема гаранције да је он сада испуњен – метод signalAll просто даје сигнал нитима које чекају да је услов **можда** сада испуњен и да има смисла поново извршити проверу.

Генерално, позив метода await треба да буде у петљи облика:

```
while( !(ok_nastaviti) )
    uslovni_objekat.await();
```

Математички факултет

vladaf@matf.bg.ac.rs

Условни објекти (6)

Од кључног је значаја да нека друга нит у неком тренутку позове метод signalAll. Када нит позове await, нема начина да саму себе реактивира. Она се узда у друге нити. Уколико ниједна од њих не реактивира нит која чека, она се никада више неће извршавати. То може водити неугодној ситуацији смртоносног блокирања.

Уколико су све остале нити блокиране и последња активна позове await не одблокирајући ниједну од осталих нити, она такође бива блокирана. Ниједна нит не остаје да одблокира остале и програм стаје (не завршава се извршавање, али ништа се ни не дешава).

Математички факултет

vladaf@matf.bg.ac.rs

Условни објекти (7)

Када је потребно позивати signalAll? Увек када се стање објекта промени на начин који може утицати на нити које чекају.

Пример. У случају банке, кад год се стање на рачуну промени, нитима које чекају треба дати нову шансу да провере стање. Дакле, позива се метод signalAll када се заврши са преносом средстава.

```
public void transfer(int sa, int na, int iznos){
   bankaLock.lock();
   try{
      while(racuni[sa] < iznos)
           dovoljnoSredstava.await();
      // prenos sredstava
      …
      dovoljnoSredstava.signalAll();
   }finally{
      bankaLock.unlock();
   }
}
```

Математички факултет

vladaf@matf.bg.ac.rs

Условни објекти (8)

Позив метода signalAll не активира моментално нит која чека. Он једино одблокира нити које чекају како би се могле надметати за катанац након што га текућа нит ослободи.

Други метод, signal, одблокира само једну, случајно одабрану, нит из скупа чекајућих нити sa тај услов. То је ефикасније него одблокиравање свих нити, али постоји опасност: ако случајно изабрана нит и даље не може да настави, она поново постаје блокирана, па ако ниједна друга нит не позове signal, настаје смртоносно блокирање.

Напомена. Нит може звати методе await, signalAll и signal за условне објекте само када поседује катанац придружен том условном објекту.

Цена која мора да се плати када се користи механизам синхронизације за заштиту приступа дељеним подацима јесте успорење.

Математички факултет
vladaf@matf.bg.ac.rs

Условни објекти (9)

java.util.concurrent.locks.Lock

- Condition newCondition()
  враћа condition објекат придружен текућем катанцу.

java.util.concurrent.locks.Condition

- void await()
  ставља текућу нит у скуп чекајућих нити за текући услов.
- void signalAll()
  одблокира све нити из скупа чекајућих нити за текући услов.
- void signal()
  одблокира једну случајно изабрану нит из скупа чекајућих нити за текући услов.

Математички факултет
vladaf@matf.bg.ac.rs

Синхронизовани објекти и методи

Интерфејси Lock и Condition додати су у Java SE 5.0 како би омогућили програмерима висок ниво контроле над закључавањем.

Међутим, у већини ситуација толика контрола није неопходна, већ се може се користити механизам који је уграђен у Јаву. Сваки објекат у Јави има интерни катанац.

Уколико је метод декларисан са кључном речју synchronized, катанац текућег објекта штити читав метод.

То значи да нит, да би позвала метод, мора прибавити унутрашњи катанац објекта.

Математички факултет
vladaf@matf.bg.ac.rs

Синхронизовани објекти и методи (2)

Другим речима,

```
public synchronized void metod()
{
   telo metoda
}
```

је еквивалентно са

```
public void metod()
{
   this.intrinsicLock.lock();
   try
   {
      telo metoda
   }
   finally
   {
      this.intrinsicLock.unlock();
   }
}
```

Математички факултет
vladaf@matf.bg.ac.rs

Синхронизовани објекти и методи (3)

Унутрашњи катанац објекта има само један придружени услов. Метод wait додаје нит у скуп чекајућих нити, а методи notifyAll и notify одблокиравају нити које чекају.

Другим речима, позиви wait и notifyAll еквивалентни су са: intrinsicCondition.await и intrinsicCondition.signalAll

Методи wait, notifyAll и notify су final методи класе Object. Методи условних објеката (тј. класе Condition) зову се await, signalAll и signal.

Математички факултет
vladaf@matf.bg.ac.rs

Синхронизовани објекти и методи (4)

Пример. Класа Banka може се имплементирати на следећи начин:

```
class Banka2
{
   private double[] racuni;

   public synchronized void transfer(int sa, int na, double iznos)
   throws InterruptedException
   {
      while(racuni[sa] < iznos)
         wait();
      // ceka na jedini uslov unutrasnjeg katanca objekta
      racuni[sa] -= iznos;
      racuni[na] += iznos;
      notifyAll(); // obavestava sve niti koje cekaju na uslov
   }

   public synchronized double getUkupnoStanje()
   {…}
}
```

Математички факултет
vladaf@matf.bg.ac.rs

Синхронизовани објекти и методи (5)

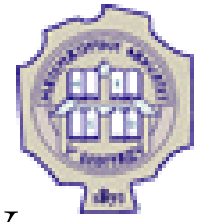
Као што се може видети, коришћење кључне речи synchronized води много концизнијем кôду.

Наравно, да би се тај кôд разумео, неопходно је знати да сваки објекат поседује унутрашњи катанац, коме је придружен унутрашњи услов. Катанац управља нитима које покушавају да уђу у synchronized метод. Услов управља нитима које су позвале метод wait.

Допуштено је и статички метод класе декларисати као synchronized. Уколико се такав метод позове, он добија унутрашњи катанац придруженог Class објекта.

Пример. Да класа Banka има статички synchronized метод, тада би катанац објекта Banka.class био закључан приликом његовог позива. Последица би била да ниједна друга нит не би могла да позове нити тај нити било који други synchronized статички метод исте класе.

Математички факултет
vladaf@matf.bg.ac.rs



Синхронизовани објекти и методи (6)

Унутрашњи катанаци и услови имају извесна ограничења. Између осталог:

- није могуће прекинути нит која покушава да добије катанац
- није могуће задати timeout период за покушај добијања катанца
- поседовање само једног услова по катанцу може бити неефикасно.

Препорука да ли у кôду користити Lock и Condition објекте или synchronized методе:

- најбоље је не користити ни једно ни друго. У многим ситуацијама може се користити неки од механизама тј. колекција из пакета java.util.concurrent
- уколико кључна реч synchronized „ради“ у конретној ситуацији, треба је користити. Тако се пише мање кôда, па је мање грешака.
- користити катанце и условне објекте ако постоји специфична потребу за додатном моћи коју ове конструкције пружају.

Математички факултет
vladaf@matf.bg.ac.rs

Синхронизовани објекти и методи (7)

java.lang.Object

Сви методи описани на овом слајду могу бити позивани искључиво из синхронизованог метода или блока. Уколико текућа нит не поседује катанац текућег објекта ови методи избацују изузетак IllegalMonitorStateException.

- void notifyAll()
  одблокира нити које су позвале wait() за текући објекат.
- void notify()
  одблокирава једну, случајно изабрану нит од оних које су позвале wait() за текући објекат.
- void wait()
  узрокује да нит чека да буде обавештена (енг. notified).
- void wait(long millis)
- void wait(long millis, int nanos)
  узрокују да нит чека док не буде обавештена или док не истекне задата количина времена.
  Параметри: millis број милисекунди, nanos број наносекунди < 1,000,000

Математички факултет

vladaf@matf.bg.ac.rs

Проблеми при извршењу нити

Катанци и услови не могу решити све проблеме који могу настати у вишенитном програмирању.

Приликом паралелног извршавања више нити могу појавити разни проблеми, као што су:

Смртоносно блокирање, смртоносни загрљај (енг. deadlock): ситуација кад се све нити нађу у стању блокирања.

Живо блокирање (енг. livelock): кад све нити нешто раде, али нема напретка јер су превише заузете одговарањем једна другој да би могле да наставе користан рад.

Изгладњивање (енг. starvation): нит није у могућности да добије регуларни приступ дељеном ресурсу.

Да би се проблеми избегли, треба озбедити узајамно искључивање критичних секција, приступачност ресирсима за све нити, поштеност у извршавању нити.

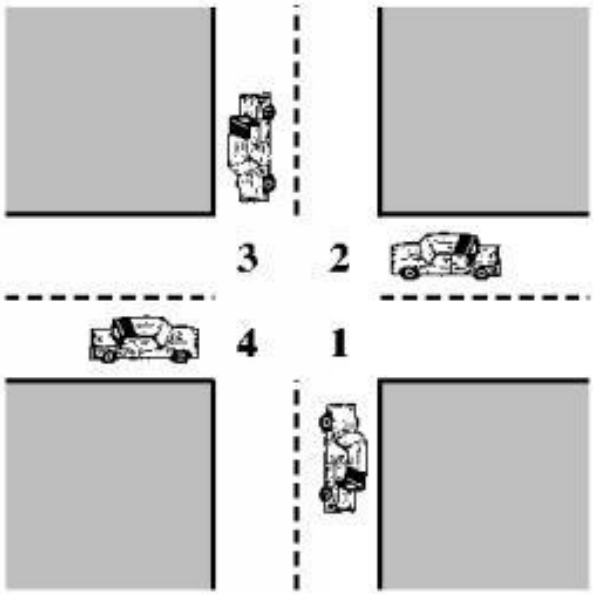
Математички факултет

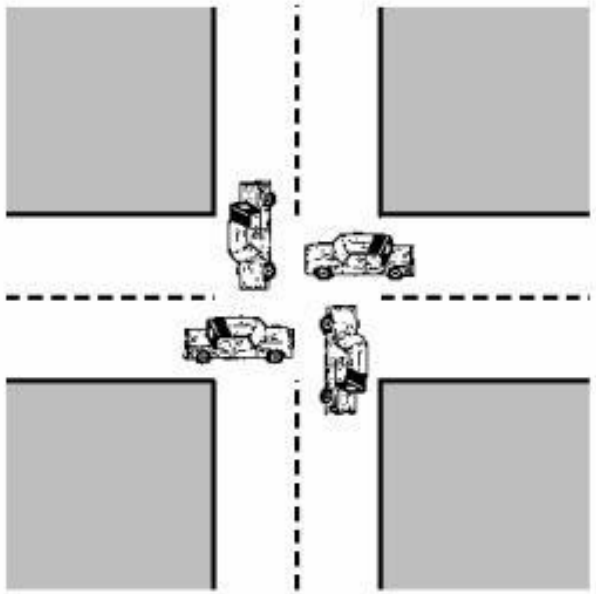
vladaf@matf.bg.ac.rs

Смртоносно блокирање

Смртоносно блокирање описује ситуацију у којој су две или више нити заувек блокиране, јер свака од њих чека неку другу.

Пример смртоносног блокирања: раскрсница.





Математички факултет
vladaf@matf.bg.ac.rs

Смртоносно блокирање (2)

Пример. Илуструје смртоносно блокирање.

```
public class SmrtonosniZagrljaj {
   static class Japanac {
      private final String name;
      public Japanac( String name ) {
         this.name = name;
      }
      public String getName() {
         return this.name;
      }
      public synchronized void nakloniSe( Japanac kolega ) {
         System.out.format( "%s: %s" + "  se naklonio!%n", this.name, kolega.getName() );
         kolega.uzvratiNaklon( this );
      }
      public synchronized void uzvratiNaklon( Japanac kolega ) {
          System.out.format( "%s: %s" + " je uzvratio naklon!%n",
                                    this.name, kolega.getName() );
      }
   }
```

Математички факултет

vladaf@matf.bg.ac.rs

Смртоносно блокирање (3)

Пример (наставак).

```
  public static void main( String[] args ) {
    final Japanac katsuki = new Japanac( "Tacuaki Katsuki" );
    final Japanac honda = new Japanac( "Keisuke Honda" );
    new Thread( new Runnable()
    {
      public void run()
      {
        katsuki.nakloniSe( honda );
      }
    } ).start();
    new Thread( new Runnable()
    {
      public void run()
      {
        honda.nakloniSe( katsuki );
      }
    } ).start();
  }
}
```

Математички факултет

vladaf@matf.bg.ac.rs

Смртоносно блокирање (4)

Пример. Односи се на банку, а илуструје смртоносно блокирање.

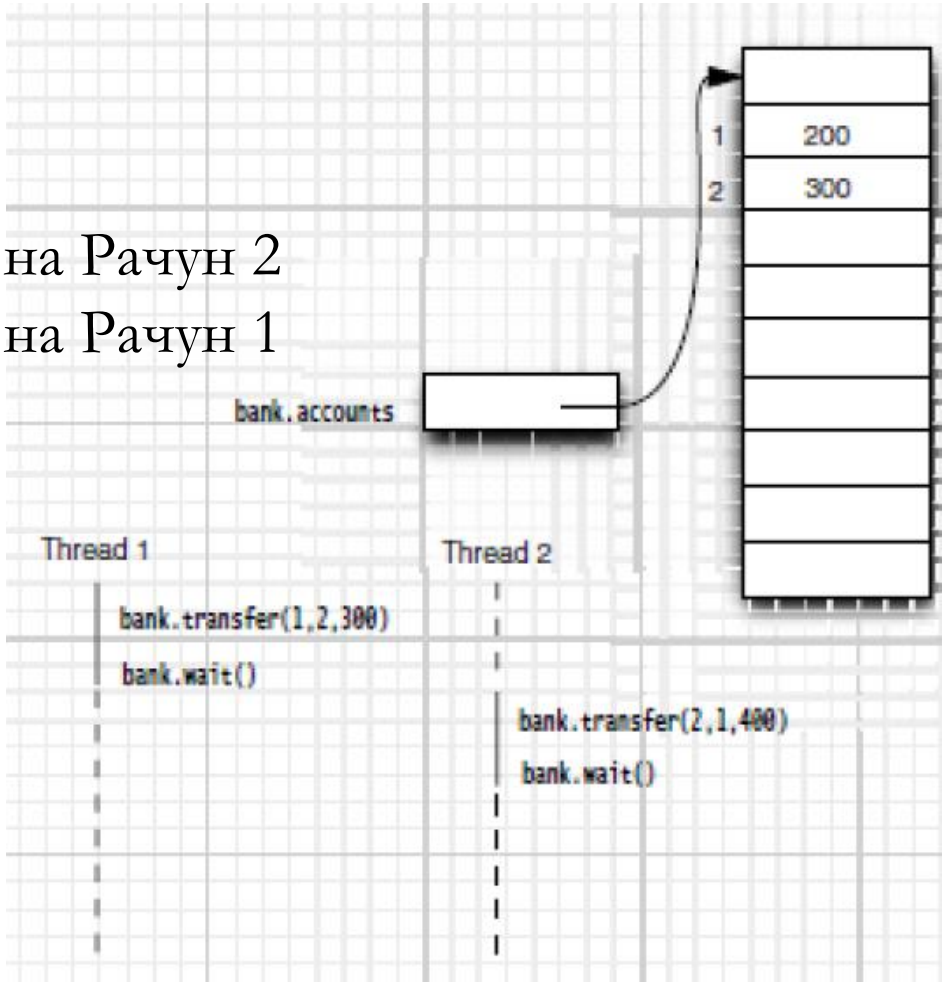
Размотримо следећу ситуацију:

Рачун 1: $200

Рачун 2: $300

Нит 1: Пренос $300 са Рачун 1 на Рачун 2

Нит 2: Пренос $400 са Рачун 2 на Рачун 1



Математички факултет

vladaf@matf.bg.ac.rs

Смртоносно блокирање (5)

Пример (наставак). Јасно је да су нити 1 и 2 блокиране. Ниједна не може да настави јер на рачунима 1 и 2 нема довољно средстава. Да ли је могуће да све нити буду блокиране јер свака чека на још новца? У првој верзији програма смртоносно блокирање се не може десити из једноставног разлога: износ сваког трансфера је највише $1,000. Како имамо 100 рачуна и укупно $100,000 на њима, бар један од рачуна мора имати бар $1,000 у сваком тренутку. Према томе, нит која преноси новац са тог рачуна може наставити са радом.

Пример. Међутим, ако се промени метод run тако што се уклони ограничење од $1,000 по трансакцији, може се брзо десити смртоносни загрљај. Ако се нпр. постави BROJ_RACUNA на 10, конструише сваки TransferRunnable са вредношћу max постављеном на 2 * INICIJALNO_STANJE и покрене програм, он ће радити извесно време, а потом стати.

Математички факултет

vladaf@matf.bg.ac.rs

Смртоносно блокирање (6)

Напомена. Када програм стоји, укуцајте комбинацију тастера CTRL+\. Добићете листу свих нити. Свака нит има стек извршавања, који говори где је тренутно блокирана.

Пример. Још један начин како би се могао изазвати настанак смртоносног блокирања у примеру банке: учини се i-та нит одговорна за стављање новца на i-ти рачун уместо за подизање новца са њега. У овом случају, постоји шанса да све нити „нападну“ један рачун, свака покушавајући да са њега подигну више новца него што на њему има.

Да би се то урадило, само је потребно у оквиру метода run класе Runnable, у позиву метода transfer да се размене аргументи saRacun и naRacun.

Математички факултет

vladaf@matf.bg.ac.rs

Смртоносно блокирање (7)

Пример. Још једна ситуација у којој једноставно настаје смртоносно блокирање: промене се позиви signalAll у signal у програму. Програм ће у неком тренутку стати. Поново, ако се промени и BROJ_RACUNA на 10 то заустављање ће се брже десити.

За разлику од signalAll који обавештава све нити које чекају на додавање средстава, метод signal одблокира само једну нит. Ако та нит не може наставити, све нити могу постати блокиране.

Математички факултет

vladaf@matf.bg.ac.rs

Живо блокирање

Нит често делује тако што одговара на активности друге нити. Ако је акција те друге нити такође одговор на акцију неке нити, тада може доћи до живог блокирања.

Слично као код смртоносног блокирања, нити које су „живо блокиране“ нису у могућносто да даље напредују. Међутим, те нити нису блокиране, већ су само превише заузете одговарањем једна другој да би могле да наставе са корисним послом.

Живо блокирање се може упоредити са следећим ситуацијом: двојица учтивих људи (Јапаната – нпр. Катсуки и Хонда ;)) који су се сучелице срели на пролазу: Катсуки се помера улево да пропусти Хонду, а истовремено се Хонда помера удесно да пропусти Катсукија па су њих двојица и даље блокирани. Како виде да су и даље блокирани, Катцуки се помера дасно а истовремено Хонда иде улево. Они и даље блокирају један другог, па стога...

Математички факултет

vladaf@matf.bg.ac.rs

Изгладњивање

Изгладњивање описује ситуацију када нит није у могућности да добије регуларан приступ дељеном ресурсу и стога нема напретка.

Ово се догађа када је дељени ресурс постао недоступан током дугог временског периода, јер га користе „похлепне“ нити.

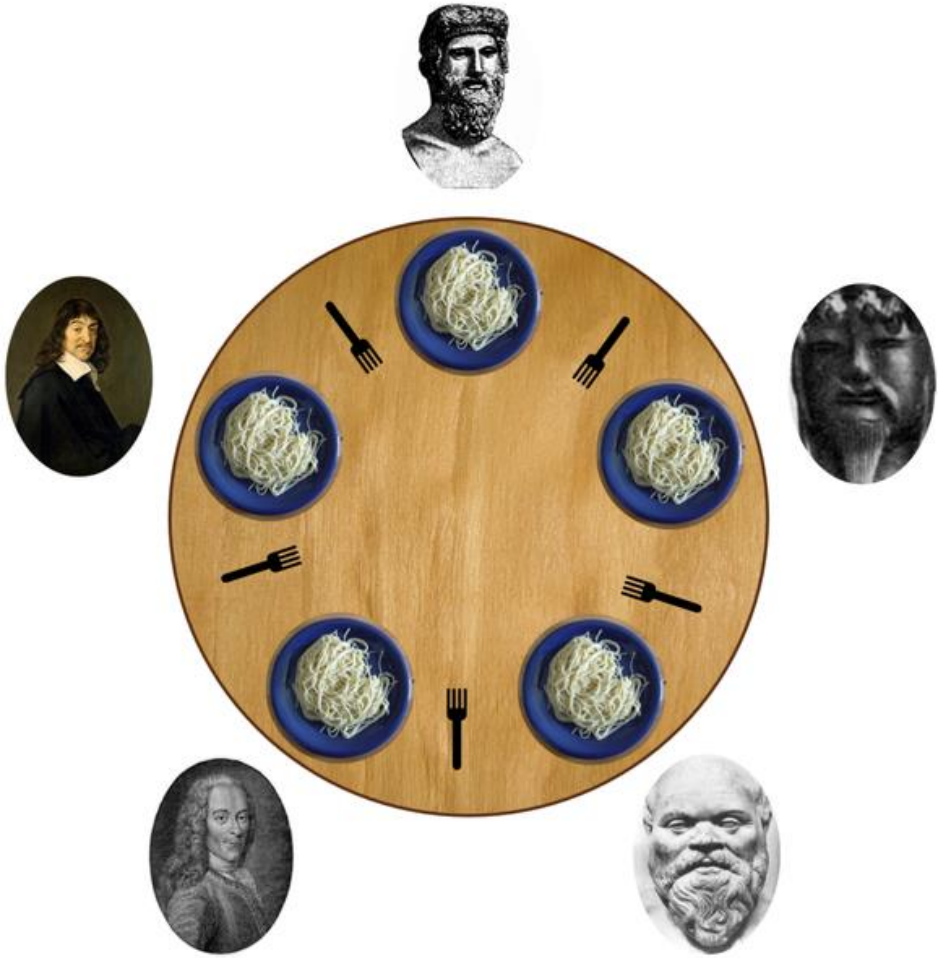
На пример, претпоставимо да објекат садржи синхронизовани метод које ис дуго извршава. Ако један нит често позива тај метод, бдуге нити које такође захтевају синхронизовани приступ истом објекту ће често бити блокиране.

Математички факултет

vladaf@matf.bg.ac.rs

Изгладњивање (2)

Проблем филозофа који вечерају Hoare, 1985.



Benjamin D. Esham / Wikimedia Commons. Преузето са адресе http://en.wikipedia.org/wiki/File:Dining_philosophers.png

Математички факултет

vladaf@matf.bg.ac.rs

Захвалница

Велики део материјала који је укључен у ову презентацију је преузет из презентације коју је раније (у време када је он држао курс Објектно орјентисано програмирање) направио проф. др Душан Тошић.

Хвала проф. Тошићу што се сагласио са укључивањем тог материјала у садашњу презентацији, као и на помоћи коју ми је пружио током конципцирања и реализације курса.

Надаље, један део материјала је преузет од колегинице Марије Милановић.

Хвала Марији Милановић на помоћи у реализацији ове презентације.