НИЗЬКОРІВНЕВА СИНХРОНІЗАЦІЯ ПОТОКІВ

Питання 6.2.

Синхронізація потоків. Приклад з банківським рахунком подружжя

```
public class CheckingAccount
{
   private int balance;

   public CheckingAccount(int initialBalance)
   {
      balance = initialBalance;
   }

   public boolean withdraw(int amount)
   {
      if (amount <= balance)
      {
         try
         {
            Thread.sleep((int) (Math.random() * 200));
         }
         catch (InterruptedException ie)
         {
         }
         balance -= amount;
         return true;
      }
      return false;
   }
```

- Кожен потік у процесі свого виконання ізольований від інших, оскільки має власний method-call стек.
  - Проте потоки все ще можуть взаємодіяти між собою, коли вони отримують доступ та обробляють спільні (shared) дані.
  - Така взаємодія може пошкодити ці дані, що призведе, в свою чергу, до порушення роботи застосунку.

```
public static void main(String[] args)
{
   final CheckingAccount ca = new CheckingAccount(100);
   Runnable r = new Runnable()
                {
                   public void run()
                   {
                      String name = Thread.currentThread().getName();
                      for (int i = 0; i < 10; i++)
                          System.out.println (name + " withdraws $10: " +
                                              ca.withdraw(10));
                   }
                };
   Thread thdHusband = new Thread(r);
   thdHusband.setName("Husband");
   Thread thdWife = new Thread(r);
   thdWife.setName("Wife");
   thdHusband.start();
   thdWife.start();
}
}
```

Вивід

```
Wife withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Wife withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Wife withdraws $10: true
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false
```

- Додаток дозволяє зняти більше грошей, ніж знаходиться на рахунку.
  - Відповідний вивід показує видачу $110, хоча на рахунку – всього $100
  - Причиною цьому є ***стан гонитви (race condition)*** між потоками чоловіка та дружини.
- ***Стан гонитви*** – це сценарій, за якого кілька потоків відправляють запити на доступ до спільних даних і результат цих запитів залежить від таймінгу планування потоків.
  - Стани гонитви можуть призводити до багів, які важко відстежити, з непередбачуваними результатами.
  - У прикладі стан гонитви виникає при зніманні грошей, зокрема, перевірці залишку на рахунку.
- Стан гонитви існує, тому що дії не є *атомарними* (неділимими) операціями.
  - Присутній виклик методу Thread.sleep(), який змушує потік засинати на різні проміжки часу (до 199 мс).
  - Без цього виклику довелось би сотні раз тестувати додаток на цю проблему, оскільки планувальник рідко призупиняє потік в районі try-catch.

Сценарій проблем із синхронізацією

```
Husband withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Wife withdraws $10: true
Husband withdraws $10: true
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Husband withdraws $10: false
Wife withdraws $10: true
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false
Wife withdraws $10: false
```

1) Потік Husband виконує вираз amount <= balance (повертає true) методу withdraw(). Потім планувальник призупиняє потік Husband та виконує потік Wife.

2) Потік Wife аналогічно виконує вираз amount <= balance (повертає true) методу withdraw().

3) Потік Wife виконує знімання коштів. Планувальник призупиняє потік Wife та відновлює роботу потоку Husband.

4) Потік Husband виконує знімання коштів.

- Цю проблему можна виправити, синхронізуючи доступ до методу withdraw(), щоб лише один потік за раз міг виконувати цей метод.
  - Можна синхронізувати доступ до цього методу, додавши зарезервоване слово synchronized у заголовок методу перед типом повернення, наприклад, synchronized boolean withdraw(int amount).

Потреба в синхронізації часто має свої тонкощі

```
class ID
{
   private static long nextID = 0;
   static long getNextID()
   {
      return nextID++;
   }
}
```

- у класі ID оголошується метод getNextID(), який повертає унікальний довгий цілий ID, наприклад, для іменування файлів.
  - навіть такий простий метод може призвести до пошкодження даних та повертати дубльовані значення на 32-бітних машинах.
  - Це відбувається тому, що 32-бітна віртуальна машина вимагає два кроки для оновлення 64-бітного довгого цілого, а додавання 1 до nextID – не атомарна операція.
  - Планувальник може перервати потік, який оновив лише половину nextID, що пошкоджує вміст змінної.

Проблема несинхронізованого контексту

- Змінні типу long і double можуть пошкоджуватись при записі в несинхронізованому контексті на 32-бітних машинах.
  - Дана проблема не трапляється зі змінними типу boolean, byte, char, float, int або short; кожен з цих типів займає 32 біти або менше.
- Дубльовані значення повертаються у зв’язку з постінкрементом (++), який зчитує та записує nextID у два етапи:
  - потік A читає nextID, проте не ікрементує значення до свого переривання планувальником
  - потік B використовує та зчитує те ж значення.
- Обидві проблеми можна виправити, синхронізуючи доступ до nextID, щоб лише один потік міг виконувати код методу.
  - Для цього потрібно додавати synchronized у заголовок методу перед типом повернення, наприклад, static synchronized int getNextID().
  - Як буде показано пізніше, також можна синхронізувати доступ до блоку операторів, використовуючи синтаксис, де object може бути довільним посиланням на об’єкт.

```
synchronized(object)
{
   /* statements */
}
```

Механізм взаємного виключення

- Незважаючи на те, синхронізуєте Ви метод чи набір інструкцій, ніякий потік не може ввійти в синхронізовану область доти, доки її не покине потік, який вже виконується в ній.
  - Ця властивість синхронізації називається ***взаємним виключенням (mutual exclusion)***.
  - Реалізація цього механізму відбувається на основі ***моніторів*** та ***замків (locks)***.
- ***Монітор*** – concurrency-конструкція для управління доступом до критичної секції – області коду, яка має виконуватись атомарно.
  - Він визначається на рівні коду як синхронізований метод або синхронізований блок.
- ***Замок*** – це токен, який має отримати потік до того, як монітор надасть можливість цьому потоку виконувати свій код всередині критичної секції.
  - Токен реалізується автоматично, коли потік виходить з монітора і дає змогу одержати токен та зайти іншому потоку.

Монітори та замки

- Потік, який встановив замок, не відкриває його при виклику одного з методів Thread.sleep().
- Потік, що заходить синхронізований метод екземпляру, встановлює замок, прив'язаний до об'єкту, чий метод викликався.
  - Вхід потоку в синхронізований метод класу встановлює замок, пов'язаний з об'єктом класу java.lang.Class.
  - Вхід потоку в синхронізований блок встановлює замок, пов'язаний з об'єктом, що управляє цим блоком.
- Клас Thread оголошує статичний метод holdsLock(Object o), який повертає true, якщо викликаючий потік утримує замок монітора над об'єктом o.
  - Метод корисний у твердженнях, на зразок assert Thread.holdsLock(o);

Видимість (Visibility)

- Для підвищення продуктивності кожен потік може мати власну копію спільної змінної, яка зберігається в локальному кеші.
  - Без синхронізації запис значення одного потоку в свою локальну копію змінної буде невидимим решті потоків.
  - В ідеалі потік повинен оновлювати значення і спільної змінної.
- Синхронізація також має властивість ***видимості***, для якої значення спільної змінної в основній пам'яті копіюється в кеш-пам'ять при вході в критичну секцію та копіюється з кешу в основну пам'ять при виході з неї.
  - Видимість стосується особливостей кешування та оптимізацій компілятора, які можуть убезпечити один потік від перегляду оновлень спільної змінної іншим потоком.

```
public class ThreadStopping
{
   public static void main(String[] args)
   {
      class StoppableThread extends Thread
      {
         private boolean stopped = false;

         @Override
         public void run()
         {
            while(!stopped)
              System.out.println("running");
         }

         void stopThread()
         {
            stopped = true;
         }
      }
      StoppableThread thd = new StoppableThread();
      thd.start();
      try
      {
         Thread.sleep(1000); // sleep for 1 second
      }
      catch (InterruptedException ie)
      {
      }
      thd.stopThread();
   }
}
```

Видимість дозволяє потокам взаємодіяти

- Можна спроектувати власний механізм зупинки потоку
  - використовувати методи stop() з класу Thread для цього небезпечно
- Локальний клас StoppableThread
  - субкласує Thread,
  - оголошує булеве поле stopped = false
  - описує методи stopThread() для позначення зупиненого потоку та run(), чий нескінченний цикл перевіряє на кожній ітерації стан потоку.
- Після інстанціювання StoppableThread, головний потік за замовчуванням запускає потік, пов'язаний з даним об'єктом Thread.
  - Далі він змушує заснути цей потік на 1с та викликає метод stop() класу StoppableThread перед смертю.
- При запуску додатку на одноядерному процесорі додаток, скоріше за все, припинить роботу.
  - На мультипроцесорних чи багатоядерних платформах ця зупинка може відбутись непомітно завдяки кешованим копіям змінної stopped.
  - Коли один потік змінює свою копію цього поля, його копія в іншому потоці залишається незмінною.

```
public class ThreadStopping
{
   public static void main(String[] args)
   {
      class StoppableThread extends Thread
      {
         private boolean stopped = false;

         @Override
         public void run()
         {
            while(!isStopped())
              System.out.println("running");
         }

         synchronized void stopThread()
         {
            stopped = true;
         }

         private synchronized boolean isStopped()
         {
            return stopped;
         }
      }
      StoppableThread thd = new StoppableThread();
      thd.start();
      try
      {
         Thread.sleep(1000); // sleep for 1 second
      }
      catch (InterruptedException ie)
      {
      }
      thd.stopThread();
   }
}
```

Виконаємо рефакторинг для гарантії коректності роботи

- Методи stopThread() та isStopped() синхронізовані для підтримки видимості, тому головний потік, що викликав stopThread(), може комунікувати із запущеним потоком, який виконується всередині run().
  - Коли потік заходить в один з цих методів, гарантується доступ до спільної (некешованої) копії поля stopped.

```
public class ThreadStopping
{
   public static void main(String[] args)
   {
      class StoppableThread extends Thread
      {
         private volatile boolean stopped = false;

         @Override
         public void run()
         {
            while(!stopped)
              System.out.println("running");
         }

         void stopThread()
         {
            stopped = true;
         }
      }
      StoppableThread thd = new StoppableThread();
      thd.start();
      try
      {
         Thread.sleep(1000); // sleep for 1 second
      }
      catch (InterruptedException ie)
      {
      }
      thd.stopThread();
   }
}
```

Подальший рефакторинг

- Синхронізація дає взаємне виключення та видимість.
  - Оскільки в цьому прикладі взаємне виключення не потрібне (немає race condition), можна виконати рефакторинг попереднього лістингу, використавши зарезервоване слово volatile, яке підтримує ***лише видимість***.
  - Потоки, що отримують доступ до цього поля, будуть завжди мати доступ лише до єдиної shared копії (не кешовані копії на мультипроцесорних/багатоядерних системах).
- Коли поле оголошено як volatile, його не можна також оголосити final.
  - Для детальної інформації розгляньте статтю Brian Goetz "Java theory and practice: Fixing the Java Memory Model, Part 2".
- Використовуйте volatile лише в контексті взаємодії потоків та оголошень полів.
  - Хоч Ви можете оголосити double та long поля як volatile, краще уникати цього на 32-бітних ВМ.
  - Для цього виконується дві операції для доступу до значень таких змінних, що вимагає взаємного виключення (синхронізації) для підтримки безпеки.

Очікування

- Клас java.lang.Object постачає методи wait(), notify() і notifyAll() для підтримки однієї з форм міжпоточної взаємодії, коли потік змушують чекати на виконання деякої умови (передпосилання для продовження виконання потоку).
  - У момент, коли умова виконалась, інший потік повідомляє про можливість продовження потоку, який знаходиться в стані очікування.
  - Метод wait() змушує потік, що його викликав, очікувати монітор об'єкта, а методи notify() та notifyAll() «будять» один або всі потоки, що очікують монітора.
- Оскільки wait(), notify() і notifyAll() залежать від замка, їх не можна викликати за межами синхронізованого методу або блоку.
  - Інакше виникне виняток java.lang.IllegalMonitorStateException.
  - Також потік, який встановив замок, відкриває його, коли в потоці буде виклик одного з методів Object.wait().

Відношення виробник-споживач

- Класичний приклад взаємодії потоків.
  - Потік виробника створює елементи даних, що споживаються потоком споживача.
  - Кожен продукований елемент даних зберігається в спільній змінній.
- Уявіть, що потоки запущені з різною швидкістю.
  - Виробник може створювати нові елементи даних та записувати їх у спільну змінну до отримання даних споживачем.
  - Споживач може отримувати контент спільної змінної до того, як елементи даних будуть виробленими.
- Для подолання цих проблем потік-виробник (producer thread) має очікувати, поки його повідомлять, що попередньо створені дані були спожиті.
  - Потік-споживач має чекати, поки його не повідомлять, що новий елемент даних було створено.

```
class Shared
{
   private char c = '\u0000';
   private boolean writeable = true;

   synchronized void setSharedChar(char c)
   {
      while (!writeable)
         try
         {
            wait();
         }
         catch (InterruptedException e) {}
      this.c = c;
      writeable = false;
      notify();
   }

   synchronized char getSharedChar()
   {
      while (writeable)
         try
         {
            wait();
         }
         catch (InterruptedException e) {}
      writeable = true;
      notify();
      return c;
   }
}
```

Відношення виробник-споживач

- Додаток створює об’єкт класу Shared і два потоки, що отримують копію посилання на об’єкт.
  - Виробник викликає метод setSharedChar() об’єкту, щоб зберегти кожну з 26 великих букв;
  - Споживач викликає метод getSharedChar() об’єкту для отримування кожної літери.

```
public class PC
{
   public static void main(String[] args)
   {
      Shared s = new Shared();
      new Producer(s).start();
      new Consumer(s).start();
   }
}
```

```
class Producer extends Thread
{
   private Shared s;

   Producer(Shared s)
   {
      this.s = s;
   }

   @Override
   public void run()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
         s.setSharedChar(ch);
         System.out.println(ch + " produced by producer.");
      }
   }
}
```

```
class Consumer extends Thread
{
   private Shared s;

   Consumer(Shared s)
   {
      this.s = s;
   }

   @Override
   public void run()
   {
      char ch;
      do
      {
         ch = s.getSharedChar();
         System.out.println(ch + " consumed by consumer.");
      }
      while (ch != 'Z');
   }
}
```

- Поле екземпляру writeable відстежує дві умови:
  - Виробник очікує, що споживач обробить елемент даних
  - Споживач очікує виробника для створення нового елементу даних.
- Поле допомагає координувати виконання виробника та споживача.

Ілюстрація координування: споживач виконується першим

1. Споживач виконує s.getSharedChar() для отримання літери.
2. Всередині синхронізованого методу споживач викликає wait(), оскільки writeable дорівнює true. Тепер споживач чекає на надходження нотифікації (сповіщення) від виробника.
3. Виробник в кінці кінців виконує s.setSharedChar(ch);.
4. Коли виробник входить у синхронізований метод, значення writeable відповідає true, і виробник не викликає wait().
    - Можливість входу є, оскільки споживач відкрив замок (release the lock) у методі wait() перед початком очікування
5. Виробник зберігає символ, присвоює writeable значення false (змусить виробника чекати наступного виклику setSharedChar(), коли споживач на той час ще не спожив символ), а потім викликає notify(), щоб розбудити споживача (припускаємо, що споживач в очікуванні).
6. Виробник виходить з setSharedChar(char c).
7. Споживач просинається (і встановлює замок), присвоює writeable значення true (змусить споживача чекати на наступний виклик getSharedChar(), коли виробник ще не відтворив символ), будить виробника (передбачається, що він у стані очікування), а потім повертає спільний символ.

```
public class PC
{
   public static void main(String[] args)
   {
      Shared s = new Shared();
      new Producer(s).start();
      new Consumer(s).start();
   }
}

class Shared
{
   private char c = '\u0000';
   private boolean writeable = true;

   synchronized void setSharedChar(char c)
   {
      while (!writeable)
         try
         {
            wait();
         }
         catch (InterruptedException e) {}
      this.c = c;
      writeable = false;
      notify();
   }

   synchronized char getSharedChar()
   {
      while (writeable)
         try
         {
            wait();
         }
         catch (InterruptedException e) {}
      writeable = true;
      notify();
      return c;
   }
}
```

Неатомарність викликів продукування/споживаннята System.out.println()

- Хоч синхронізація працює коректно, на деяких платформах можна спостерігати вивід, який показує багато повідомлень щодо продукування перед багатьма повідомленнями про споживання.
  - Наприклад, на початку виводу можна помітити спродукований символ A, далі – спродукований символ B, потім спожитий символ A.
  - Такий дивний вивід спричинений неатомарністю виклику setSharedChar(), за яким слідує відповідний виклик System.out.println().
  - Аналогічно і для getSharedChar().
- Порядок виводу можна скоригувати, обгорнувши (wrapping) кожну пару викликів методів у синхронізованому блоці, який синхронізує Shared-об'єкт s.

Код та результати виводу

```
A produced by producer.
A consumed by consumer.
B produced by producer.
B consumed by consumer.
C produced by producer.
C consumed by consumer.
D produced by producer.
D consumed by consumer.
```

```
class Producer extends Thread
{
   private Shared s;
   Producer(Shared s)
   {
      this.s = s;
   }

   @Override
   public void run()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
         synchronized(s)
         {
            s.setSharedChar(ch);
            System.out.println(ch + " produced by producer.");
         }
      }
   }
}
```

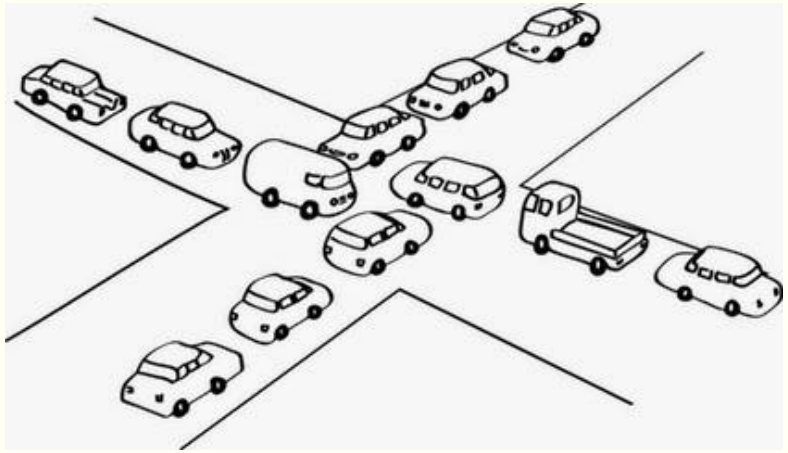
```
class Consumer extends Thread
{
   private Shared s;

   Consumer(Shared s)
   {
      this.s = s;
   }

   @Override
   public void run()
   {
      char ch;
      do
      {
         synchronized(s)
         {
            ch = s.getSharedChar();
            System.out.println(ch + " consumed by consumer.");
         }
      }
      while (ch != 'Z');
   }
}
```

Обережно

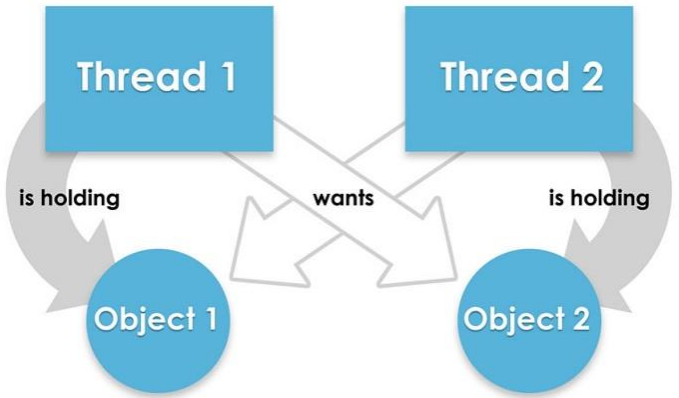
- Ніколи не викликайте wait() ззовні циклу.
  - Умова циклу (!writeable або writeable у попередньому прикладі) перевіряється до та після виклику wait().
- Тестування умови до виклику wait() забезпечує ***живучість (liveness)***.
  - Якби тест не було представлено, умова виконувалась, а notify() викликався до виклику wait(), очікуючий потік міг ніколи не прокинутись.
- Перетестування умови після виклику wait() забезпечує ***безпеку***.
  - Якби перетестування не відбувалось, а умова не виконувалась після того, як потік прокинувся від виклику wait() (можливо, інший потік ненароком викликав notify(), коли умова не виконувалась), потік продовжував би знищувати інваріанти, захищені замком.

Deadlock

- Надто багато синхронізацій може бути проблематичним.
- Якщо бути необережним, можна потрапити в ситуацію, коли замки отримуються багатьма потоками:
  - Жоден потік не має власного замка, проте тримає замок, потрібний іншому потоку.
  - Жоден потік не може ввійти, а потім вийти зі своєї критичної секції, щоб відчинити замок, оскільки інший потік тримає замок від критичної секції.



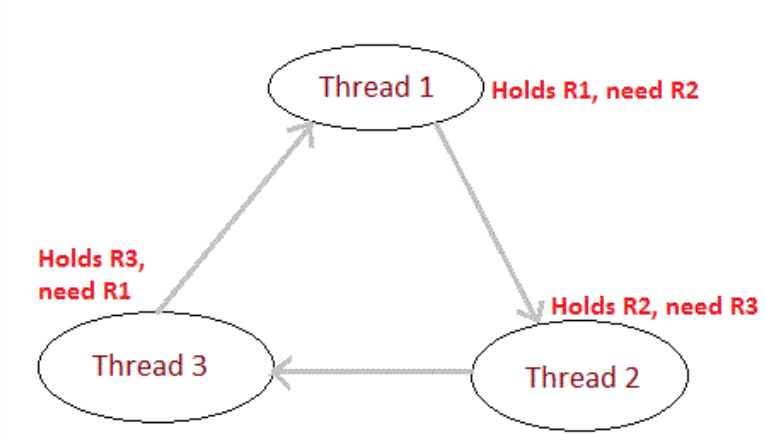
```
public class DeadlockDemo
{
   private Object lock1 = new Object();
   private Object lock2 = new Object();

   public void instanceMethod1()
   {
      synchronized(lock1)
      {
         synchronized(lock2)
         {
            System.out.println("first thread in instanceMethod1");
            // critical section guarded first by
            // lock1 and then by lock2
         }
      }
   }

   public void instanceMethod2()
   {
      synchronized(lock2)
      {
         synchronized(lock1)
         {
            System.out.println("second thread in instanceMethod2");
            // critical section guarded first by
            // lock2 and then by lock1
         }
      }
   }
```




```
   public static void main(String[] args)
   {
      final DeadlockDemo dld = new DeadlockDemo();
      Runnable r1 = new Runnable()
                    {
                       @Override
                       public void run()
                       {
                          while(true)
                          {
                             dld.instanceMethod1();
                             try
                             {
                                Thread.sleep(50);
                             }
                             catch (InterruptedException ie)
                             {
                             }
                          }
                       }
                    };
      Thread thdA = new Thread(r1);
      Runnable r2 = new Runnable()
                    {
                       @Override
                       public void run()
                       {
                          while(true)
                          {
                             dld.instanceMethod2();
                             try
                             {
                                Thread.sleep(50);
                             }
                             catch (InterruptedException ie)
                             {
                             }
                          }
                       }
                    };
      Thread thdB = new Thread(r2);
      thdA.start();
      thdB.start();
   }
}
```

Пояснення до лістингу

- Розглянемо таку послідовність виконання:
  1. Потік A викликає instanceMethod1(), отримує замок lock1 (referenced object) та заходить у свою outer critical section (ще не отримавши замок lock2-referenced object).
  2. Потік B викликає instanceMethod2(), отримує замок lock2 (referenced object), та входить у свою outer critical section (ще не отримавши замок lock1-referenced object).
  3. Потік A намагається отримати замок, associated with lock2. Віртуальна машина змушує потік чекати ззовні внутрішньої критичної секції, оскільки потік B тримає замок.
  4. Потік B намагається отримати замок, associated with lock1. Віртуальна машина змушує потік очікувати ззовні внутрішньої критичної секції, оскільки потік A тримає замок.
  5. Жоден потік не може працювати, оскільки інший потік тримає потрібний замок. Отримується ситуація deadlock, а програма (принаймні, в контексті цих 2 потоків) зависає.



Часто дедлок важко ідентифікувати

- Наприклад, код може містити кругове відношення серед різних класів:
  - Синхронізований метод класу A викликає синхронізований метод класу B.
  - Синхронізований метод класу B викликає синхронізований метод класу C.
  - Синхронізований метод класу C викликає синхронізований метод класу A.
- Нехай потік A викликає синхронізований метод класу A, а потік B викликає синхронізований метод класу C.
  - Потік B заблокується при спробі виклику синхронізованого методу класу A, а потік A досі всередині цього методу.
  - Потік A продовжить роботу, поки не викличе синхронізований метод класу C, а далі блокуватиметься.
  - У результаті – дедлок.

Зауважте!

- Ні мова Java, ні віртуальна машина не забезпечують способи запобігання deadlock, тому це лежить на розробнику.
  - Найпростіший спосіб – уникати виклику синхронізованого методу/блоку з іншого синхронізованого методу/блоку.
  - Хоч це буде запобігати появі deadlock, даний підхід непрактичний, оскільки Ваш синхронізований метод/блок може потребувати синхронізованого методу з Java API.

Локальні для потоку змінні (Thread-Local Variables)

- Інколи виникає потреба прив’язати дані (наприклад, user ID) до потоку.
  - Це завдання можна виконати за допомогою локальної змінної, але лише при її існуванні.
  - Можна використати поле екземпляру, щоб подовше тримати ці дані, проте буде потрібна синхронізація.
- Java постачає клас ThreadLocal в якості зручної альтернативи.
  - Кожен екземпляр класу ThreadLocal описує *локальну для потоку змінну*, що забезпечує окрему комірку (storage slot) для зберігання значень для кожного потоку, який отримує доступ до змінної.
  - Локальну для потоку змінну можна розглядати як multislot variable, в якій кожен потік може зберігати різне значення однієї змінної.
  - Кожен потік бачить лише власне значення і не знає про наявність інших потоків та їх змінних.

Клас ThreadLocal (узагальнений тип ThreadLocal<T>)

- Конструктори та методи:
  - **ThreadLocal()** – створює нову змінну, локальну для потоку.
  - **T get()** повертає значення зі storage slot для викликаючого потоку. Якщо такого немає, get() викликає initialValue().
  - **T initialValue()** створює комірку зберігання для викликаючого потоку та зберігає початкове (default = null) значення в цій комірці. Необхідно субкласувати ThreadLocal та переозначити protected-метод, щоб задавати більш корисне початкове значення.
  - **void remove()** видаляє комірку зберігання викликаючого потоку. Якщо цей метод слідує після get() без втручання set(), get() викликає initialValue().
  - **void set(T value)** задає значення для комірки зберігання викликаючого потоку.

Різні User ID для різних потоків

```
public class ThreadLocalDemo
{
   private static volatile ThreadLocal<String> userID =
      new ThreadLocal<String>();

   public static void main(String[] args)
   {
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         String name = Thread.currentThread().getName();
                         if (name.equals("A"))
                            userID.set("foxtrot");
                         else
                            userID.set("charlie");
                         System.out.println(name + " " + userID.get());
                      }
                   };
      Thread thdA = new Thread(r);
      thdA.setName("A");
      Thread thdB = new Thread(r);
      thdB.setName("B");
      thdA.start();
      thdB.start();
   }
}
```

- Після інстанціювання ThreadLocal та присвоєння посилання volatile полю класу userID, головний потік за замовчуванням створює ще два потоки, які містять різні String-об'єкти в userID, та виводить їх об'єкти.
  - поле volatile, оскільки доступ до нього відбувається з різних потоків
- Можливий вивід:
  A foxtrot
  B charlie

Клас InheritableThreadLocal

- Значення локальних для потоку змінних не пов’язані між собою.
  - Коли створюється новий потік, він отримує нову комірку (storage slot), яка містить значення initialValue().
- Можливо, Ви віддасте перевагу передачі значення від батьківского потоку (*parent thread)* до створеного ним дочірнього потоку (*child thread*).
  - Зробити це можна за допомогою InheritableThreadLocal (підкласу ThreadLocal).
- Як і конструктор InheritableThreadLocal(), цей клас оголошує protected метод T childValue(T parentValue), який обчислює початкове значення для дочірнього потоку як функцію від значення для батьківського потоку під час створення дочірнього потоку.
  - Викликається з батьківського потоку до запуску дочірнього потоку.
  - Повертає аргумент, що передано в parentValue, і має бути переозначеним, якщо хочете отримувати інше число.

```
public class InheritableThreadLocalDemo
{
   private static volatile InheritableThreadLocal<Integer> intVal =
      new InheritableThreadLocal<Integer>();

   public static void main(String[] args)
   {
      Runnable rP = new Runnable()
                    {
                       @Override
                       public void run()
                       {
                          intVal.set(new Integer(10));
                          Runnable rC = new Runnable()

                                        {
                                           @Override
                                           public void run()
                                           {
                                              Thread thd;
                                              thd = Thread.currentThread();
                                              String name = thd.getName();
                                              System.out.println(name + " " +
                                                                 intVal.get());
                                           }
                                        };
                          Thread thdChild = new Thread(rC);
                          thdChild.setName("Child");
                          thdChild.start();
                       }
                    };
      new Thread(rP).start();
   }
}
```

- Після інстанціювання InheritableThreadLocal головний потік за замовчуванням створює батьківський (parent) потік, який зберігає об'єкт Integer із значенням 10 в intVal.
- Батьківський потік створює дочірній (child) потік, який отримує доступ до intVal та виокремлює його Integer-об'єкт.

Вивід додатку: Child 10

ДЯКУЮ ЗА УВАГУ!

Для глибшого розуміння ThreadLocal розгляньте пост з блогу Patson Luk “A Painless Introduction to Java's ThreadLocal Storage”.