Многонишково програмиране с JAVA

Multithreading

- За какво е необходимо многонишково програмиране?
  - Програмните езици са последователни, т.е. инструкциите се изпълняват една след друга(ред след ред)

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

    initializeArrays();
    downloadStocks();
    initalizeTimeSeriesModels();
    makePredictions();

}
```

  - При еднонишкова програма тези методи ще бъдат извикани един след друг: трябва да се изчака изпълнението да свърши едно по едно
    - Трябва да се отбележи, че не е най-доброто решение: операциите, които консумират много време трябва да се изчакват и потребителите няма да знаят какво се случва!

Multithreading

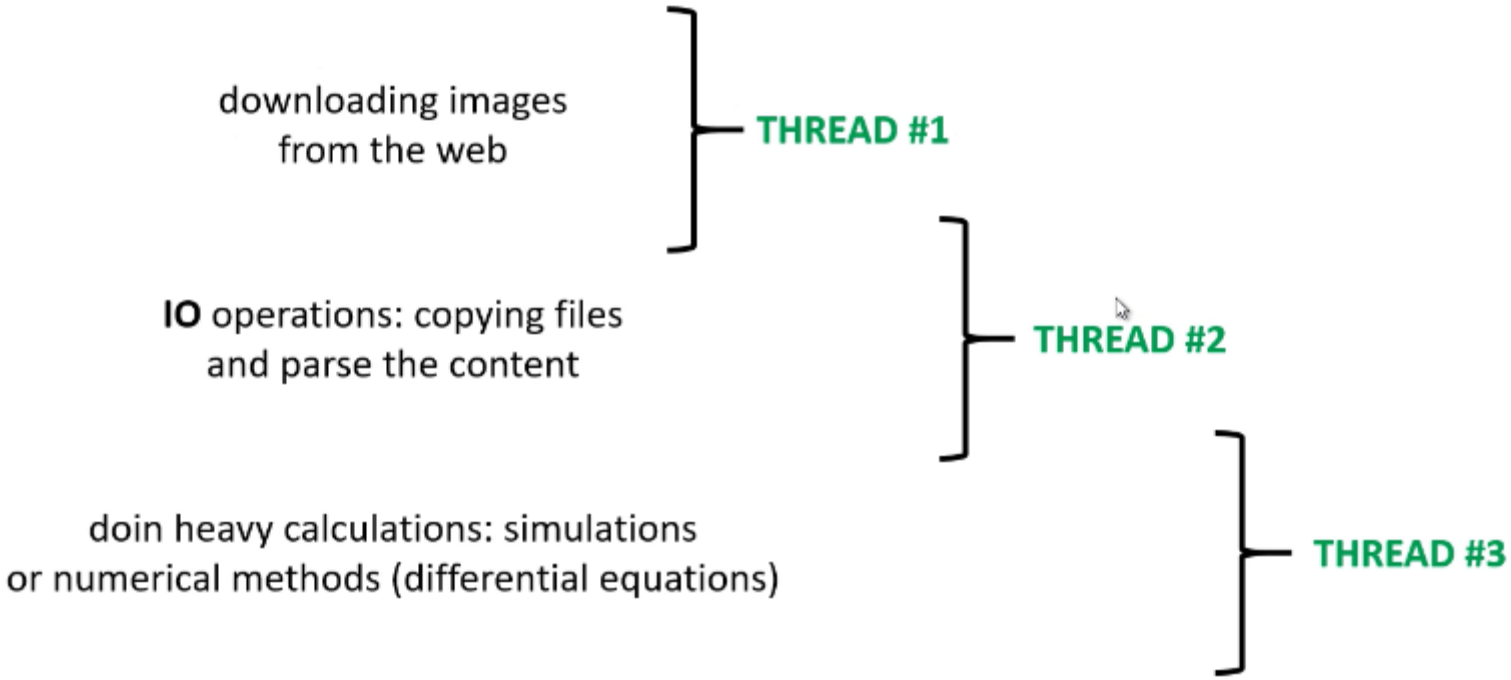
- Основният принцип в multithreading е да се раздели изпълнението на подзадачи: всяка една или група от подзадачи да се изпълняват едновременно.

For example: our stock market related software is able to download data from the web (*Yahoo Finance* for example)

→ it takes **2-3** mins to fetch the data BUT we want to make sure the application is not frozen !!!

→ solution: we create a distinct thread for the download operation and during this procedure the user can do whatever he/she wants in the application

Multithreading



Multithreading

*„Multithreading is the ability of the **CPU** to execute multiple processes or threads concurrently"*

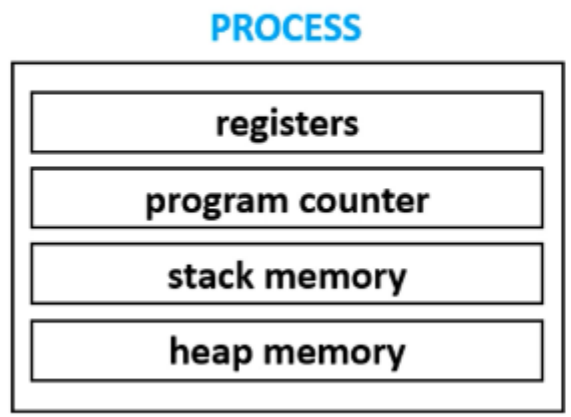
Both **processes** and **threads** are independent sequences of execution

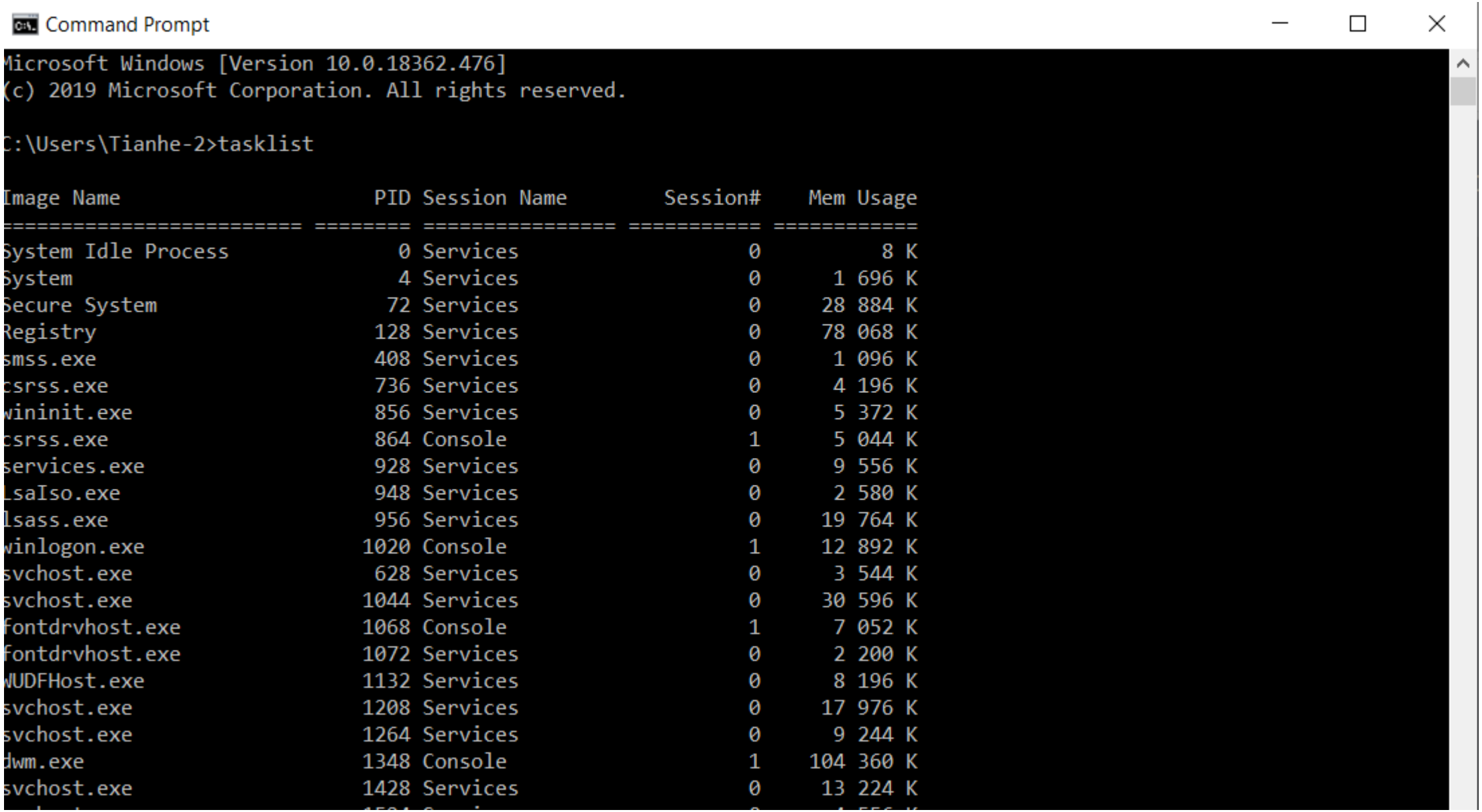
PROCESS: a process is an instance of program execution

→ when you open a software or a web browser: these are distinct processes

→ the **OS** assigns distinct registers, stack memory and heap memory to every single process

in **Java** we can create processes with the help of **ProcessBuilder** class




Command Prompt
Microsoft Windows [Version 10.0.18362.476]
(c) 2019 Microsoft Corporation. All rights reserved.
C:\Users\Tianhe-2>tasklist
Image Name PID Session Name Session# Mem Usage
System Idle Process 0 Services 0 8 K
System 4 Services 0 1 696 K
Secure System 72 Services 0 28 884 K
Registry 128 Services 0 78 068 K
smss.exe 408 Services 0 1 096 K
csrss.exe 736 Services 0 4 196 K
wininit.exe 856 Services 0 5 372 K
csrss.exe 864 Console 1 5 044 K
services.exe 928 Services 0 9 556 K
LsaIso.exe 948 Services 0 2 580 K
lsass.exe 956 Services 0 19 764 K
winlogon.exe 1020 Console 1 12 892 K
svchost.exe 628 Services 0 3 544 K
svchost.exe 1044 Services 0 30 596 K
fontdrvhost.exe 1068 Console 1 7 052 K
fontdrvhost.exe 1072 Services 0 2 200 K
WUDFHost.exe 1132 Services 0 8 196 K
svchost.exe 1208 Services 0 17 976 K
svchost.exe 1264 Services 0 9 244 K
dwm.exe 1348 Console 1 104 360 K
svchost.exe 1428 Services 0 13 224 K

Multithreading

*„Multithreading is the ability of the **CPU** to execute multiple processes or threads concurrently"*

Both **processes** and **threads** are independent sequences of execution

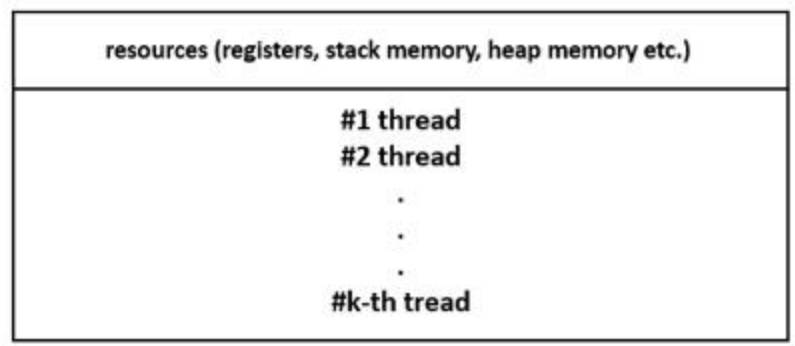
THREADS: a thread is a „light-weight" process

→ It is a unit of execution within a given process (a process may have several threads)

→ each thread in a process shares the memory and resources and this is why programmers have to deal with concurrent programming and multithreading

creating a new thread requires fewer resources than creating a new process

PROCESS



Multithreading

MULTITHREADING AND TIME-SLICING ALGORITHM

Assume we have **k** threads (so more than **1** thread in our application)

→ somehow the single processor has to deal with all of the **k** threads
~ one approach is to use time-slicing algorithm

„Processing time for a single core is shared among processes and threads. This is called time-slicing"

„multithreaded execution" (with time-slicing)

In this case the **CPU** will run **thread #1** for a small amount of time then **thread #2** then again **thread #1** and then **thread #2** and so on ...

Multithreading

- Предимства
  - Създават се по-бързи приложения: може да представят няколко операции едновременно

  - Постига се по-добра ресурсна натовареност (CPU utilization)

    По подразбиране всяка програма на Java е еднонишкова: има няколо ядра, които няма да се използват без многонишков режим.

  - Може да подобри производителността

    // подобрението на производителността ще има, когато се използват множество ядра и паралелно програмиране

Multithreading

MULTITHREADING AND TIME-SLICING ALGORITHM

Assume we have **k** threads (so more than **1** thread in our application)

→ somehow the single processor has to deal with all of the **k** threads
~ one approach is to use time-slicing algorithm

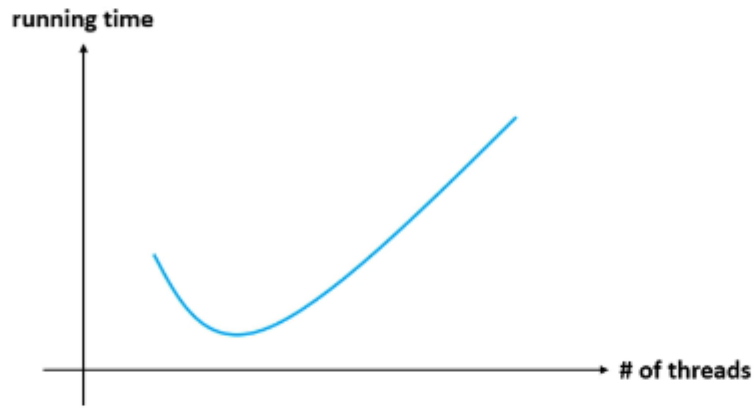
„Processing time for a single core is shared among processes and threads. This is called time-slicing"

„multithreaded execution" (with time-slicing)

In this case the **CPU** will run **thread #1** for a small amount of time then **thread #2** then again **thread #1** and then **thread #2** and so on ...

Multithreading

DISADVANTAGES



It's expensive to switch between threads: this is why using threads is not always the fastest way possible (for example sorting algorithms)

RULE OF THUMB: for small problems it is unncecessary to use multithreading

Multithreading

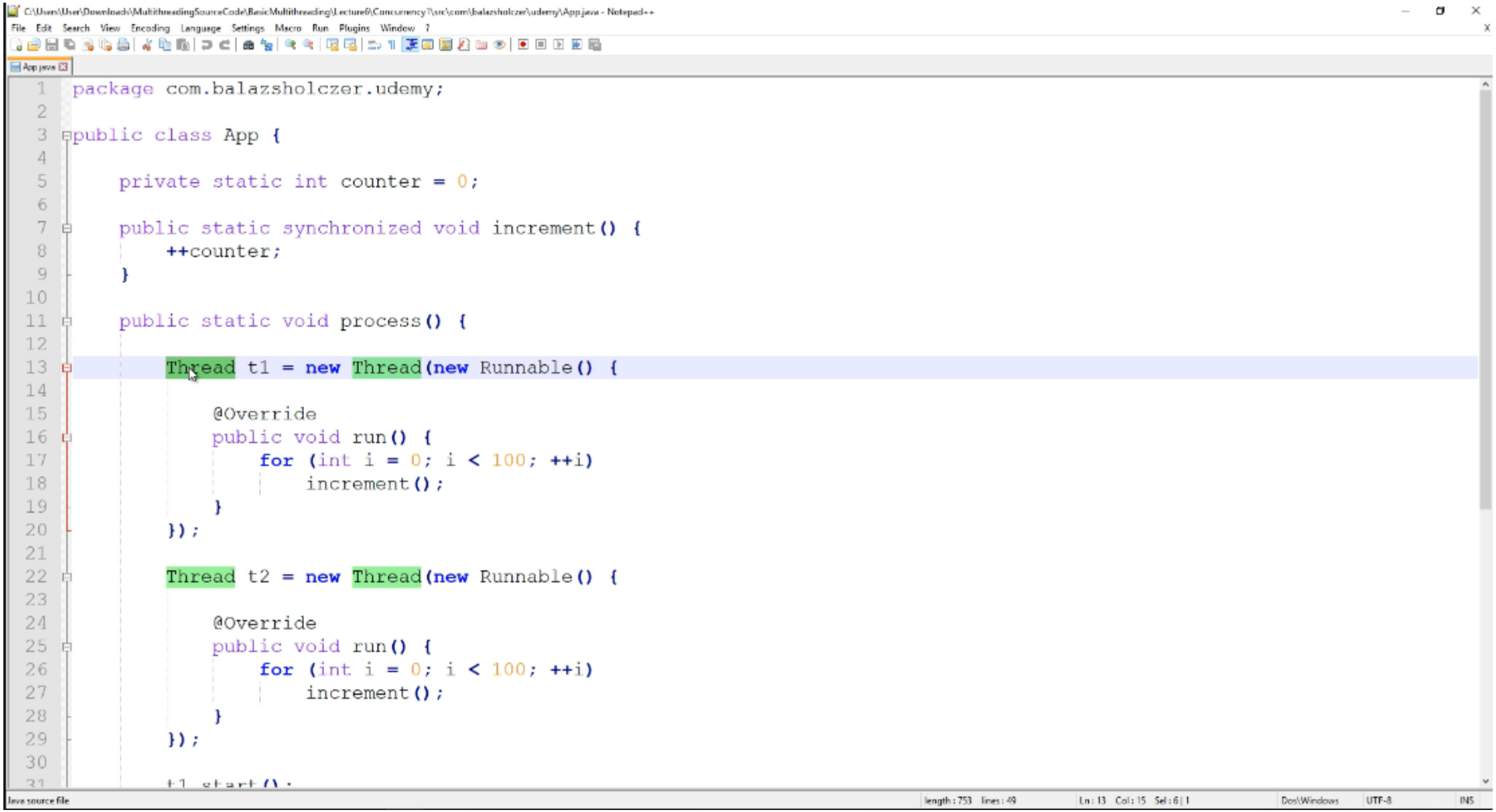
THREAD STATES

1.) ***NEW*** when we instantiate a thread
It is in this state until we start it
~ **start()** method

2.) ***RUNNABLE*** after we have started the thread
The thread is executing its task in this state

3.) ***WAITING*** when a thread is waiting: for example for another thread to finish its task
When other thread signals then this thread goes back to the **runnable** state
~ **wait()** and **sleep()** methods

4.) ***DEAD*** after the thread finishes its task


C:\Users\User\Downloads\MultithreadingSourceCode\BasicMultithreading\Lecture6\Concurrency7\src\com\balazsholczer\udemy\App.java - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
App.java
1 package com.balazsholczer.udemy;
2
3 public class App {
4
5 private static int counter = 0;
6
7 public static synchronized void increment() {
8 ++counter;
9 }
10
11 public static void process() {
12
13 Thread t1 = new Thread(new Runnable() {
14
15 @Override
16 public void run() {
17 for (int i = 0; i < 100; ++i)
18 increment();
19 }
20 });
21
22 Thread t2 = new Thread(new Runnable() {
23
24 @Override
25 public void run() {
26 for (int i = 0; i < 100; ++i)
27 increment();
28 }
29 });
30
31 t1.start();
Java source file
length : 753 lines : 49
Ln : 13 Col : 15 Sel : 6 | 1
Dos\Windows
UTF-8
INS


C:\Users\User\Downloads\MultithreadingSourceCode\BasicMultithreading\Lecture6\Concurrency7\src\com\balazsholczer\udemy\App.java - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
App.java
10
11 public static void process() {
12
13 Thread t1 = new Thread(new Runnable() {
14
15 @Override
16 public void run() {
17 for (int i = 0; i < 100; ++i)
18 increment();
19 }
20 });
21
22 Thread t2 = new Thread(new Runnable() {
23
24 @Override
25 public void run() {
26 for (int i = 0; i < 100; ++i)
27 increment();
28 }
29 });
30
31 t1.start();
32 t2.start();
33
34 try {
35 t1.join();
36 t2.join();
37 } catch (InterruptedException e) {

Multithreading

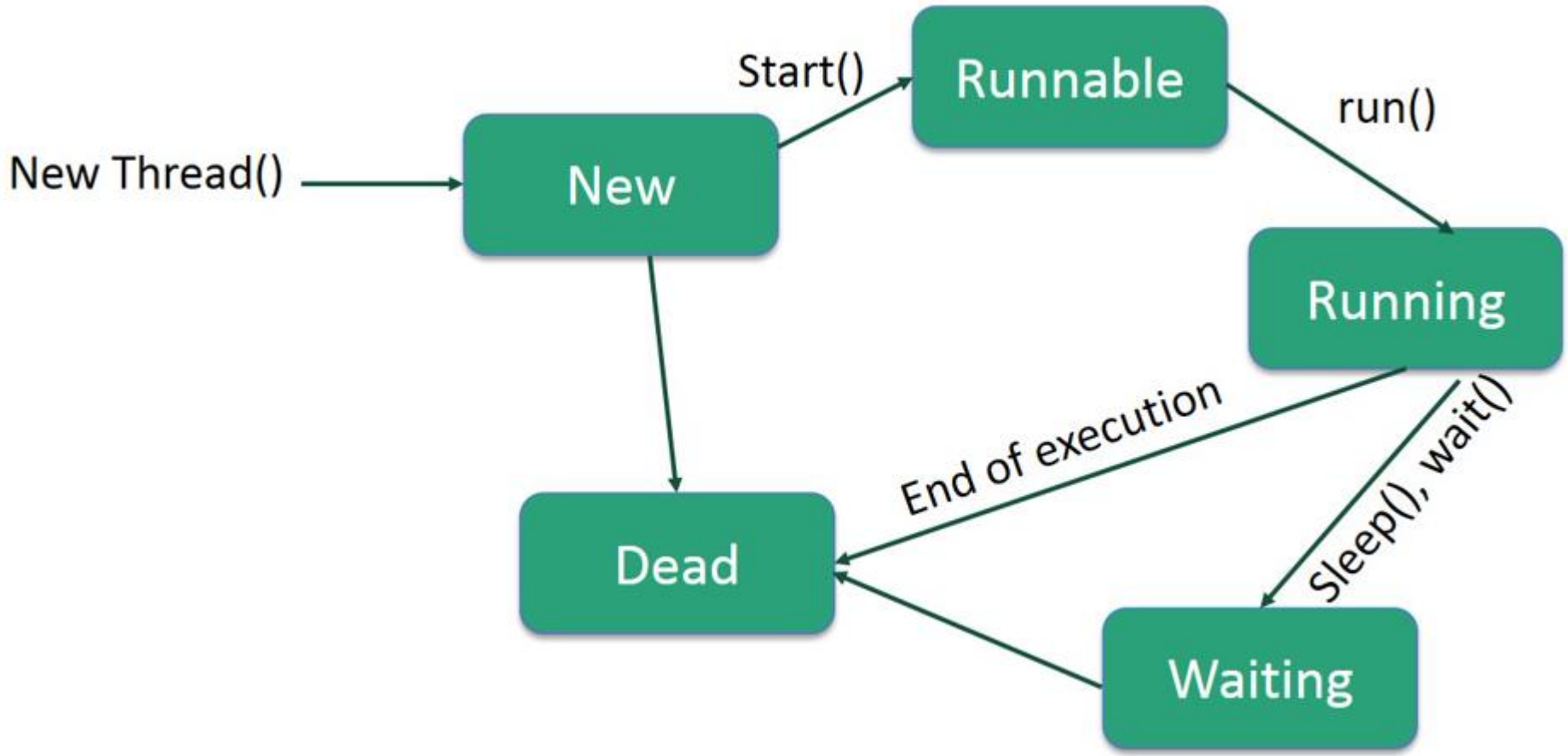
THREAD STATES

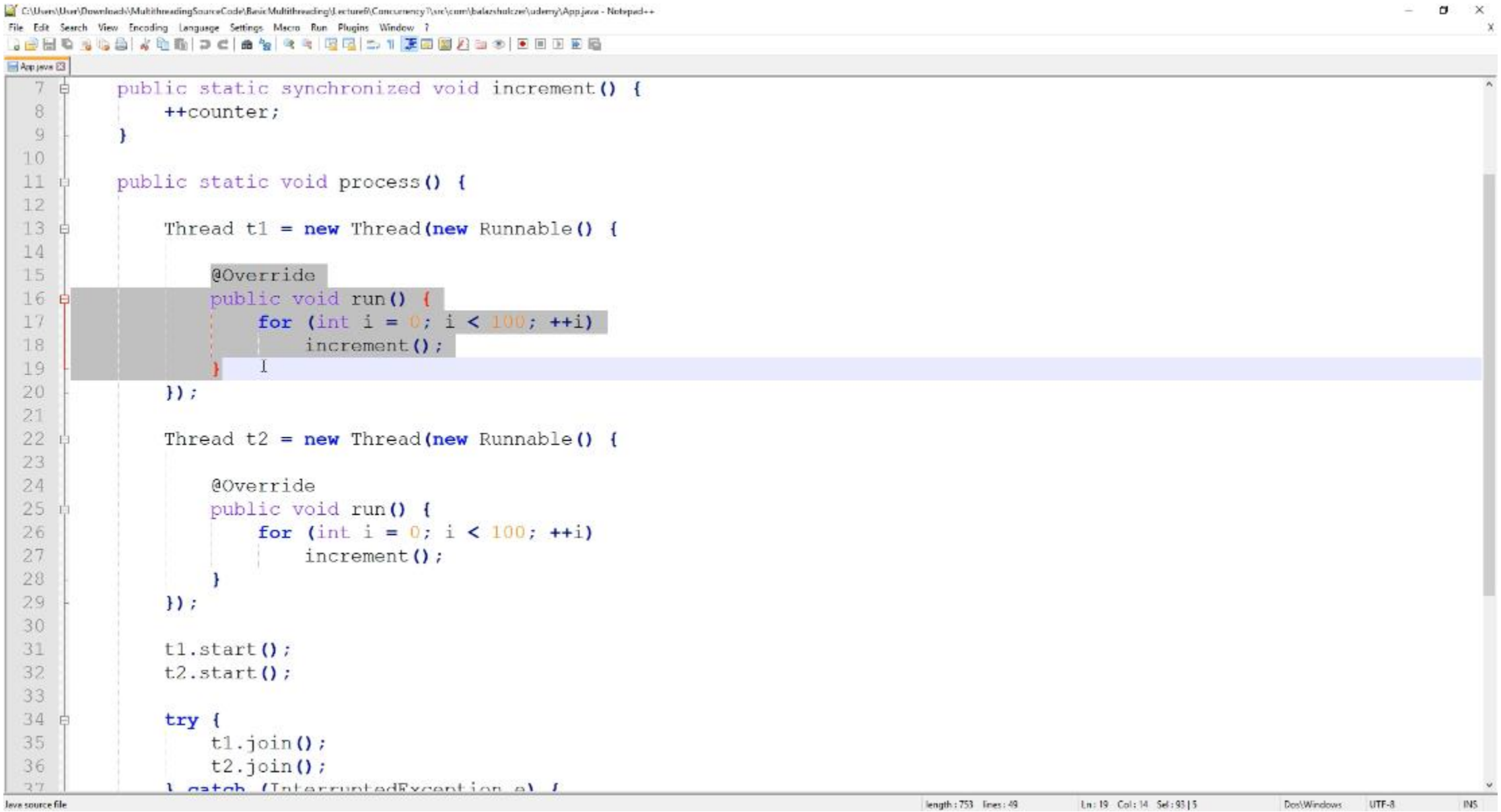
1.) ***NEW*** when we instantiate a thread
It is in this state until we start it
~ **start()** method

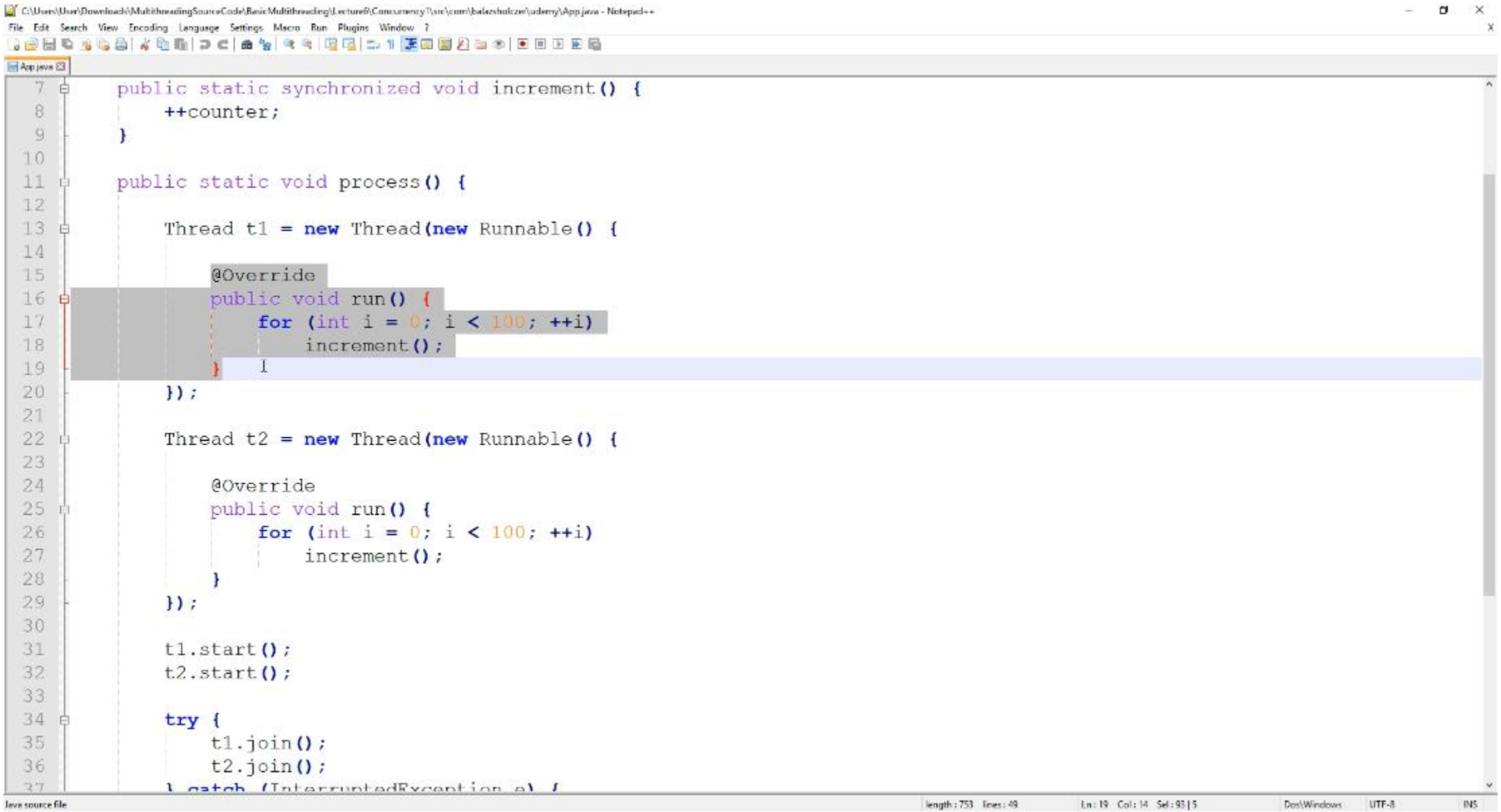
2.) ***RUNNABLE*** after we have started the thread
The thread is executing its task in this state

3.) ***WAITING*** when a thread is waiting: for example for another thread to finish its task
When other thread signals then this thread goes back to the **runnable** state
~ **wait()** and **sleep()** methods

4.) ***DEAD*** after the thread finishes its task


New Thread()
New
Start()
Runnable
run()
Running
End of execution
Sleep(), wait()
Dead
Waiting


C:\Users\User\Downloads\MultithreadingSourceCode\BasicMultithreading\Lecture8\Concurrency7\src\com\balazshulcze\udemy\App.java - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
App.java
7 public static synchronized void increment() {
8 ++counter;
9 }
10
11 public static void process() {
12
13 Thread t1 = new Thread(new Runnable() {
14
15 @Override
16 public void run() {
17 for (int i = 0; i < 100; ++i)
18 increment();
19 }
20 });
21
22 Thread t2 = new Thread(new Runnable() {
23
24 @Override
25 public void run() {
26 for (int i = 0; i < 100; ++i)
27 increment();
28 }
29 });
30
31 t1.start();
32 t2.start();
33
34 try {
35 t1.join();
36 t2.join();
Java source file
length : 753 lines : 49
Ln : 19 Col : 14 Sel : 93 | 5
Dos\Windows
UTF-8
INS


C:\Users\User\Downloads\MultithreadingSourceCode\BasicMultithreading\Lecture8\Concurrency7\src\com\balazshulcze\udemy\App.java - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
App.java
7 public static synchronized void increment() {
8 ++counter;
9 }
10
11 public static void process() {
12
13 Thread t1 = new Thread(new Runnable() {
14
15 @Override
16 public void run() {
17 for (int i = 0; i < 100; ++i)
18 increment();
19 }
20 });
21
22 Thread t2 = new Thread(new Runnable() {
23
24 @Override
25 public void run() {
26 for (int i = 0; i < 100; ++i)
27 increment();
28 }
29 });
30
31 t1.start();
32 t2.start();
33
34 try {
35 t1.join();
36 t2.join();
Java source file
length : 753 lines : 49
Ln : 19 Col : 14 Sel : 93 | 5
Dos\Windows
UTF-8
INS

Multithreading

THREAD STATES

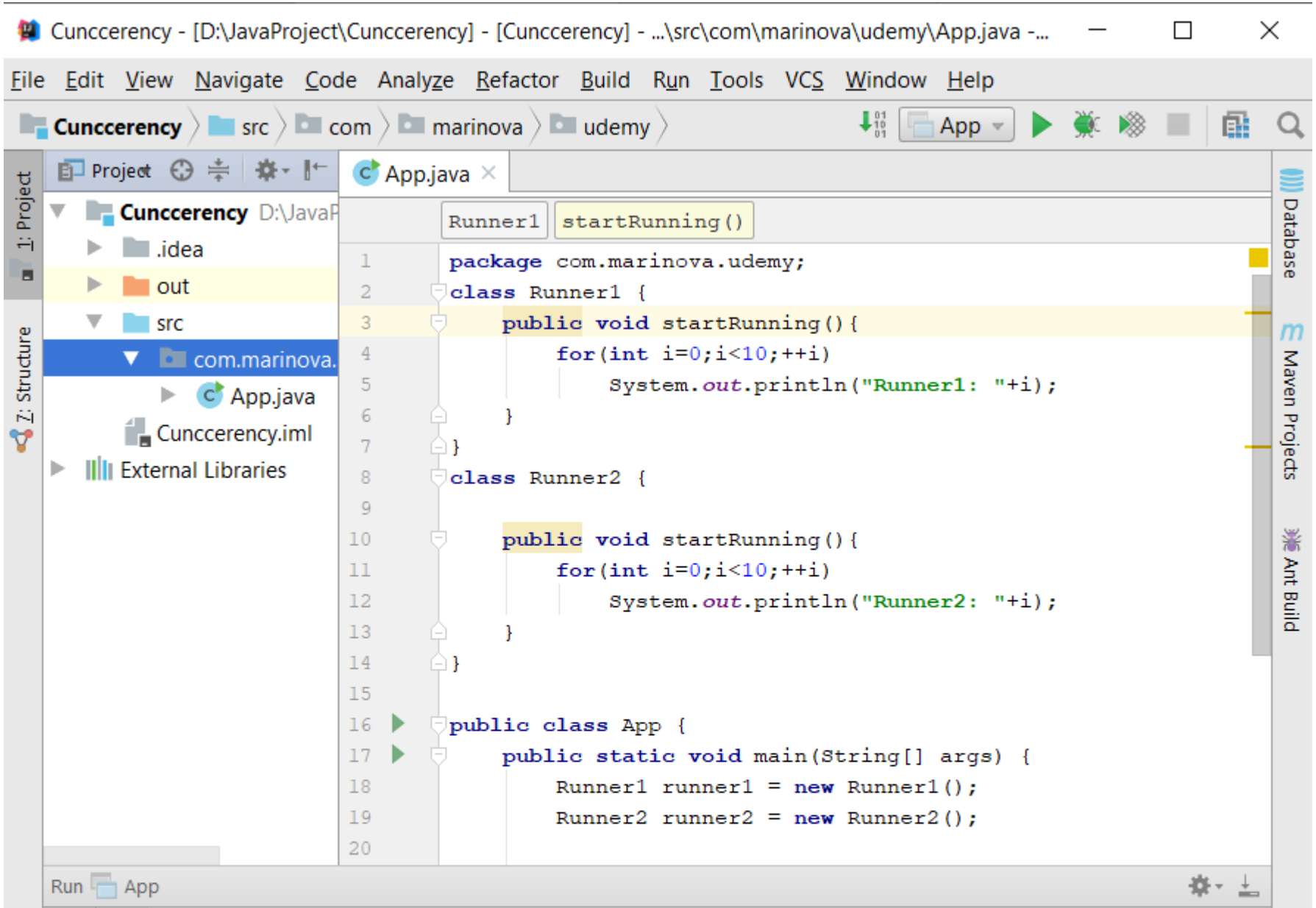
1.) *NEW* when we instantiate a thread
It is in this state until we start it
~ **start()** method

2.) *RUNNABLE* after we have started the thread
The thread is executing its task in this state

3.) *WAITING* when a thread is waiting: for example for another thread to finish its task
When other thread signals then this thread goes back to the **runnable** state
~ **wait()** and **sleep()** methods

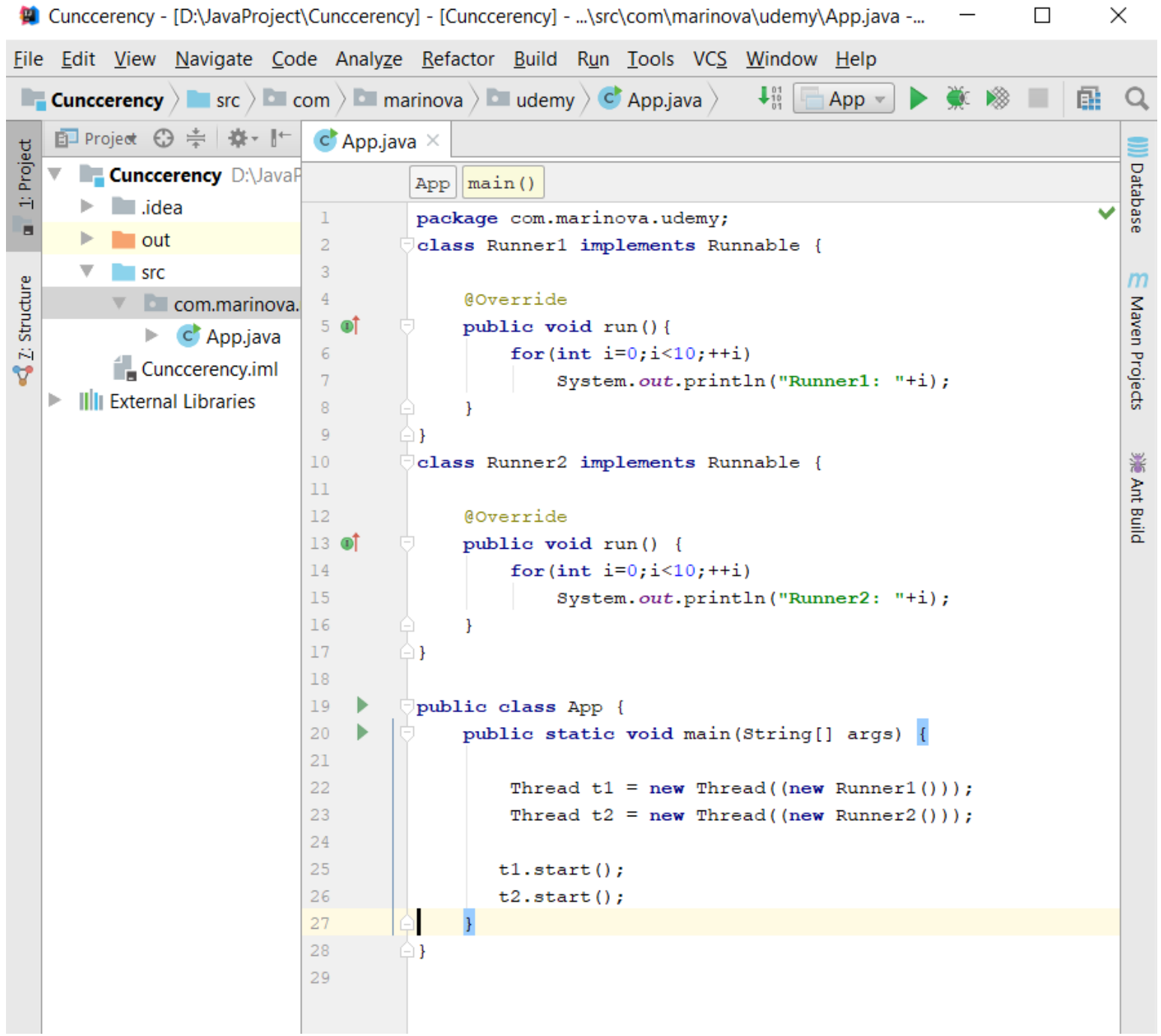
4.) *DEAD* after the thread finishes its task

Sequential processing

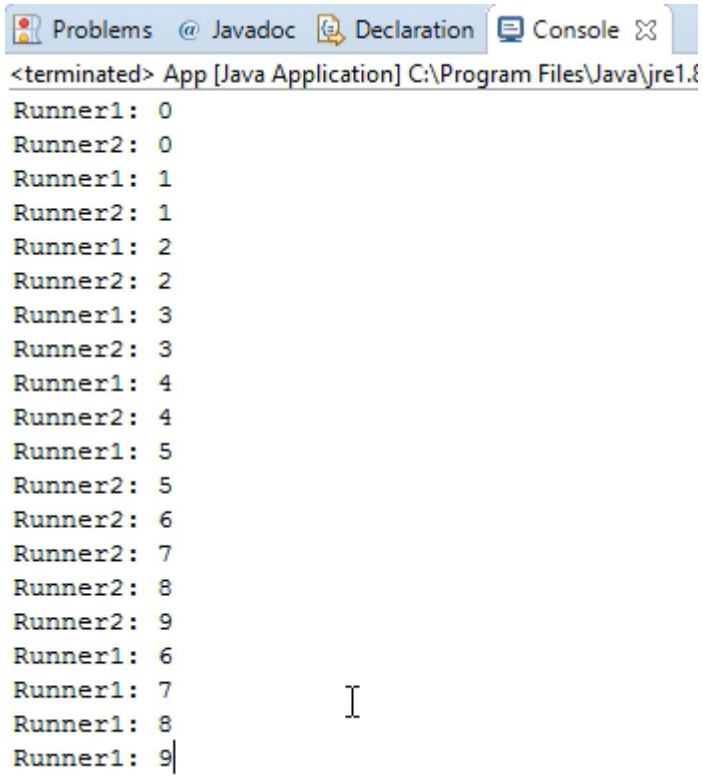


Резултат от изпълнението на последователната програма:

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java" ...
Runner1: 0
Runner1: 1
Runner1: 2
Runner1: 3
Runner1: 4
Runner1: 5
Runner1: 6
Runner1: 7
Runner1: 8
Runner1: 9
Runner2: 0
Runner2: 1
Runner2: 2
Runner2: 3
Runner2: 4
Runner2: 5
Runner2: 6
Runner2: 7
Runner2: 8
Runner2: 9

Process finished with exit code 0
```


Cunccerency - [D:\JavaProject\Cunccerency] - [Cunccerency] - ...\src\com\marinova\udemy\App.java -...
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Cunccerency src com marinova udemy App.java
App
1: Project
Project
Cunccerency D:\JavaP
.idea
out
src
com.marinova.
App.java
Cunccerency.iml
External Libraries
App.java
App main()
1 package com.marinova.udemy;
2 class Runner1 implements Runnable {
3
4 @Override
5 public void run(){
6 for(int i=0;i<10;++i)
7 System.out.println("Runner1: "+i);
8 }
9 }
10 class Runner2 implements Runnable {
11
12 @Override
13 public void run() {
14 for(int i=0;i<10;++i)
15 System.out.println("Runner2: "+i);
16 }
17 }
18
19 public class App {
20 public static void main(String[] args) {
21
22 Thread t1 = new Thread((new Runner1()));
23 Thread t2 = new Thread((new Runner2()));
24
25 t1.start();
26 t2.start();
27 }
28 }
29
Database
Maven Projects
Ant Build
7: Structure

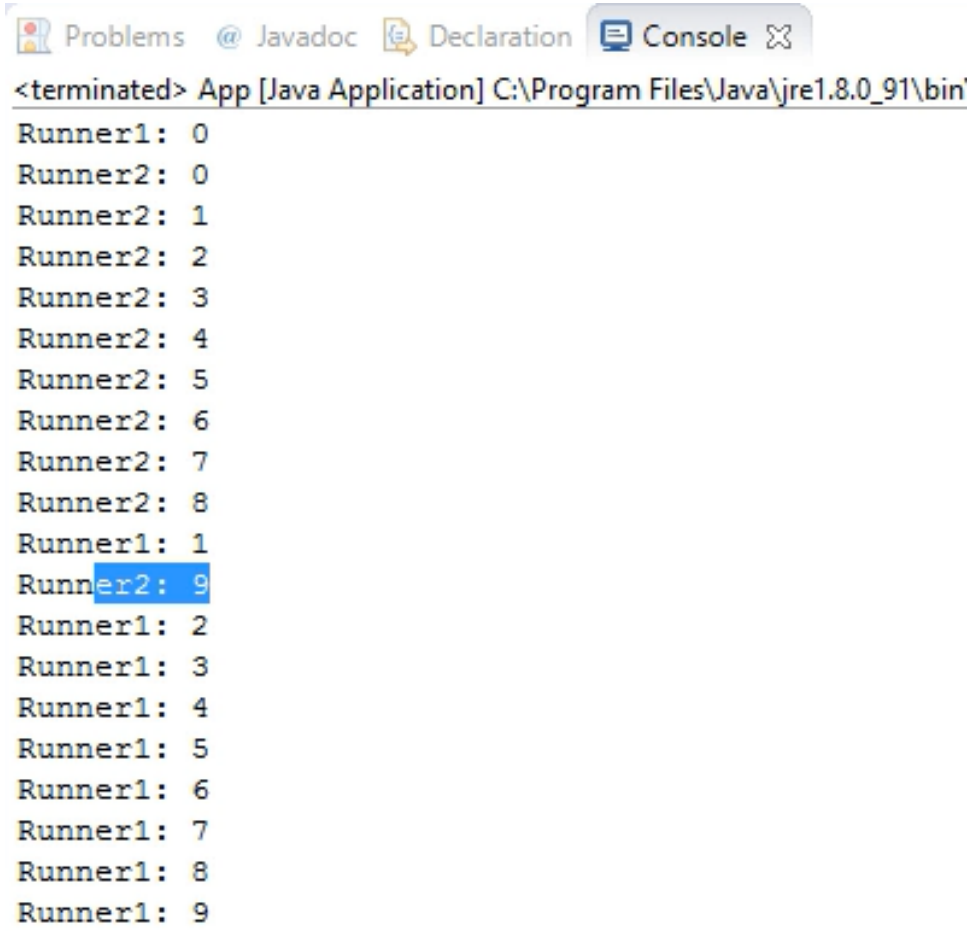
Резултат от multithreading изпълнение:



```java
package com.marinova.udemy;
class Runner1 extends Thread {

    @Override
    public void run(){
        for(int i=0;i<10;++i)
            System.out.println("Runner1: "+i);
    }
}
class Runner2 extends Thread {

    @Override
    public void run() {
        for(int i=0;i<10;++i)
            System.out.println("Runner2: "+i);
    }
}

public class App {
    public static void main(String[] args) {

    /*    Thread t1 = new Thread((new Runner1()));
        Thread t2 = new Thread((new Runner2()));*/
    Runner1 t1 = new Runner1();
    Runner2 t2 = new Runner2();
       t1.start();
       t2.start();
    }
}
```

- Друг вариант – наследяване на класа Thread

Резултат от изпълнението:

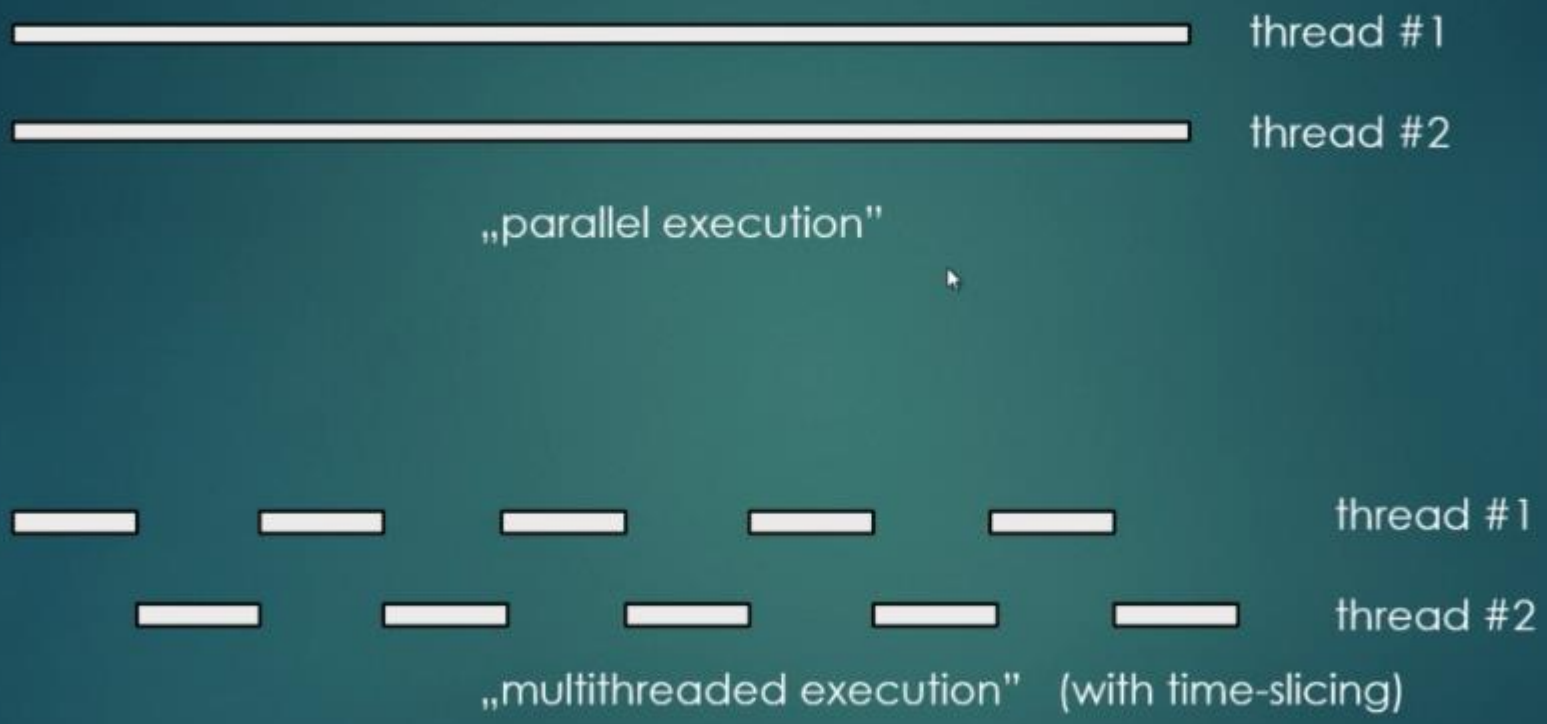


- Допълнителна задача: Стартирайте програмата, така че броя на итерациите да е 100. Разгледайте каква е последователността на изпълнение на t1 и t2.

```java
package com.marinova.udemy;
class Runner1 extends Thread {

    @Override
    public void run(){
        for(int i=0;i<100;++i){
            System.out.println("Runner1: "+i);
            try {
                Thread.sleep( millis: 100);
            }catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
class Runner2 extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 100; ++i) {
            System.out.println("Runner2: " + i);
            try {
                Thread.sleep( millis: 100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java" ...
Runner2: 0
Runner1: 0
Runner1: 1
Runner2: 1
Runner2: 2
Runner1: 2
Runner2: 3
Runner1: 3
Runner1: 4
Runner2: 4
Runner2: 5
Runner1: 5
Runner1: 6
Runner2: 6
Runner1: 7
Runner2: 7
Runner1: 8
Runner2: 8
Runner1: 9
Runner2: 9
Runner2: 10
Runner1: 10
Runner1: 11
Runner2: 11
Runner2: 12
Runner1: 12
Runner2: 13
Runner1: 13
Runner1: 14
Runner2: 14
Runner2: 15
Runner1: 15
Runner2: 16
```

Parallel *versus* multithreading



Runnable vs Thread

1. Create Thread using Runnable Interface vs Thread class

1.1. Runnable interface

```
public class DemoRunnable implements Runnable {
    public void run() {
        //Code
    }
}

//start new thread with a "new Thread(new demoRunnable()).start()" call
```

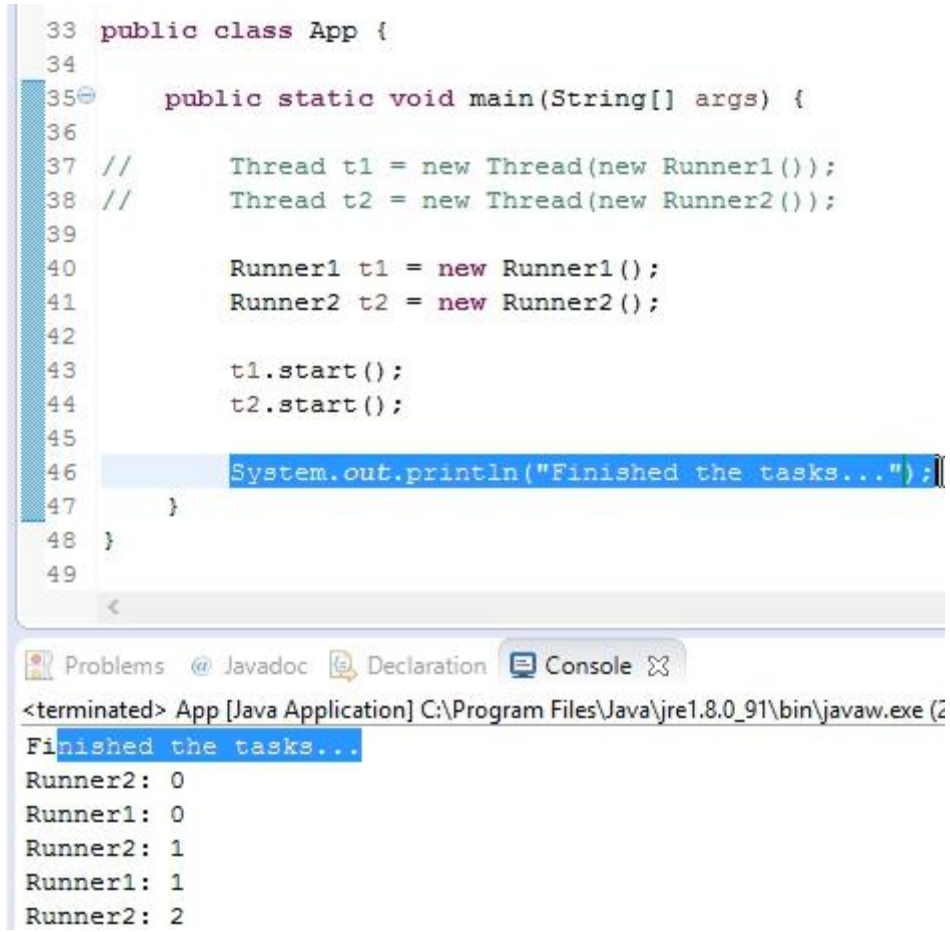
Runnable vs Thread – cont.

1.2. Thread class

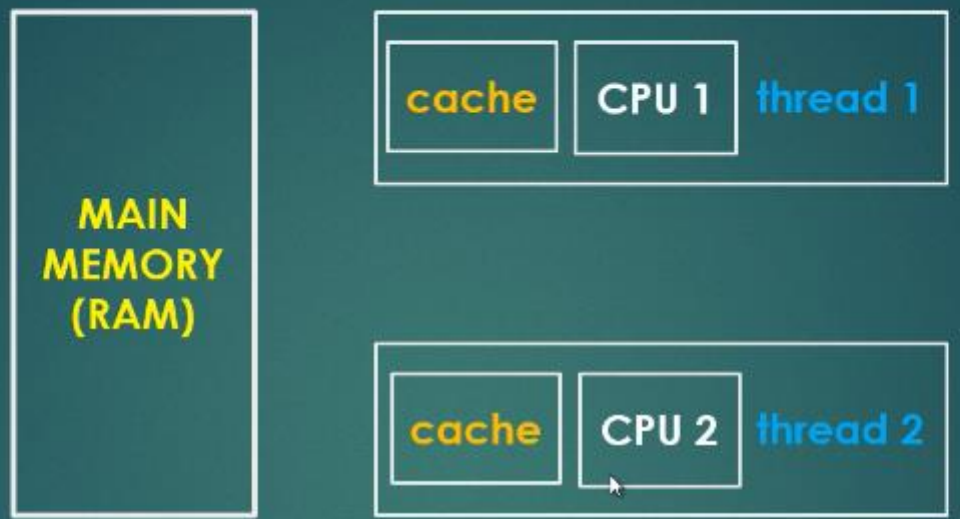
```
public class DemoThread extends Thread {
    public DemoThread() {
        super("DemoThread");
    }
    public void run() {
        //Code
    }
}
//start new thread with a "new demoThread().start()" call
```

Runnable vs Thread – cont.

2. Difference between Runnable vs Thread


33 public class App {
34
35 public static void main(String[] args) {
36
37 // Thread t1 = new Thread(new Runner1());
38 // Thread t2 = new Thread(new Runner2());
39
40 Runner1 t1 = new Runner1();
41 Runner2 t2 = new Runner2();
42
43 t1.start();
44 t2.start();
45
46 System.out.println("Finished the tasks...");
47 }
48 }
49
Problems
Javadoc
Declaration
Console
<terminated> App [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (2
Finished the tasks...
Runner2: 0
Runner1: 0
Runner2: 1
Runner1: 1
Runner2: 2

Volatile

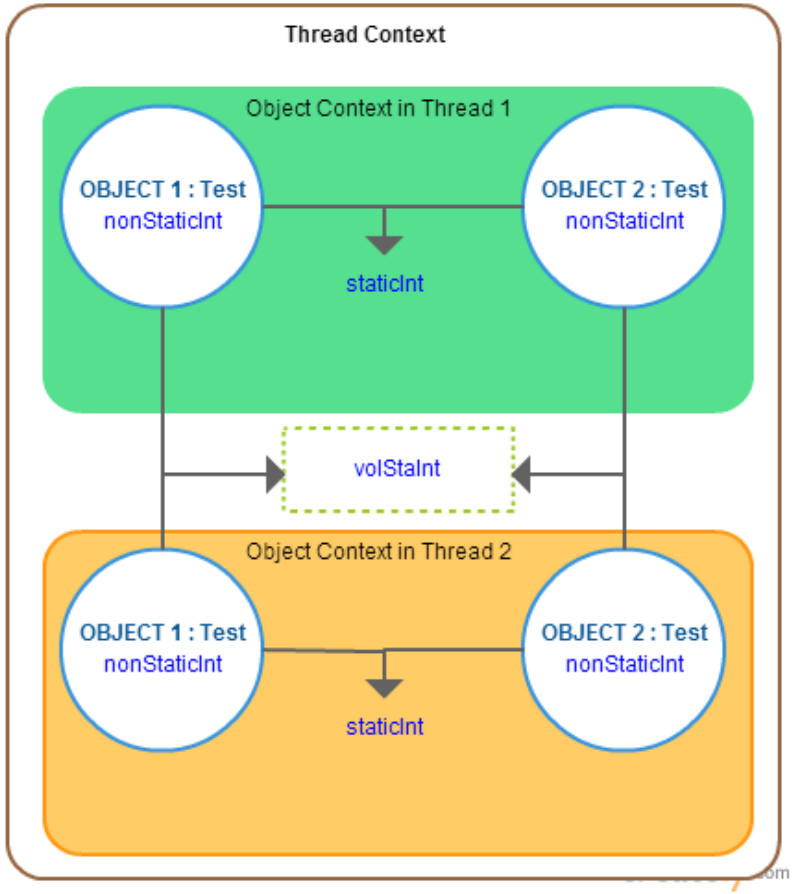


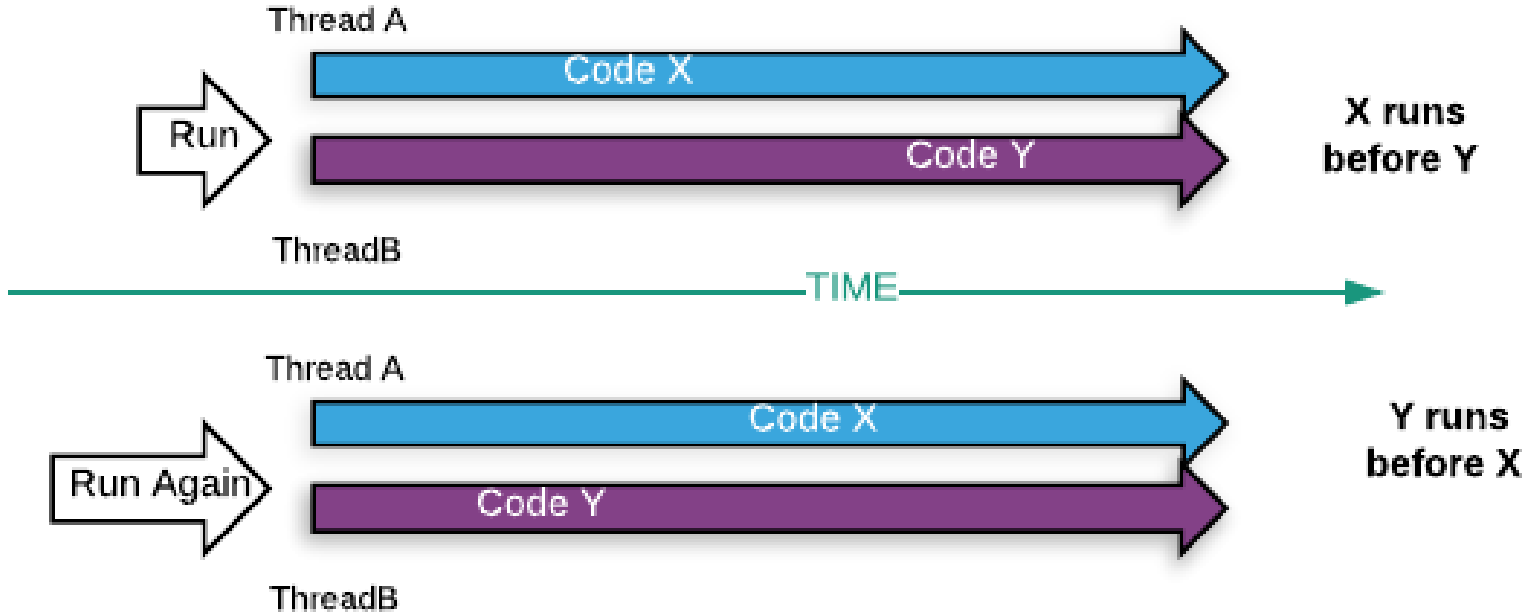
Every read of a volatile variable will be read from the **RAM** so from the main memory
// not from cache, usually variables are cached for performance reasons

Caches are faster → do not use ***volatile*** keyword if not necessary
(+ it prevents instruction reordering which is a performance boost technique)

Volatile Vs Static in JAVA

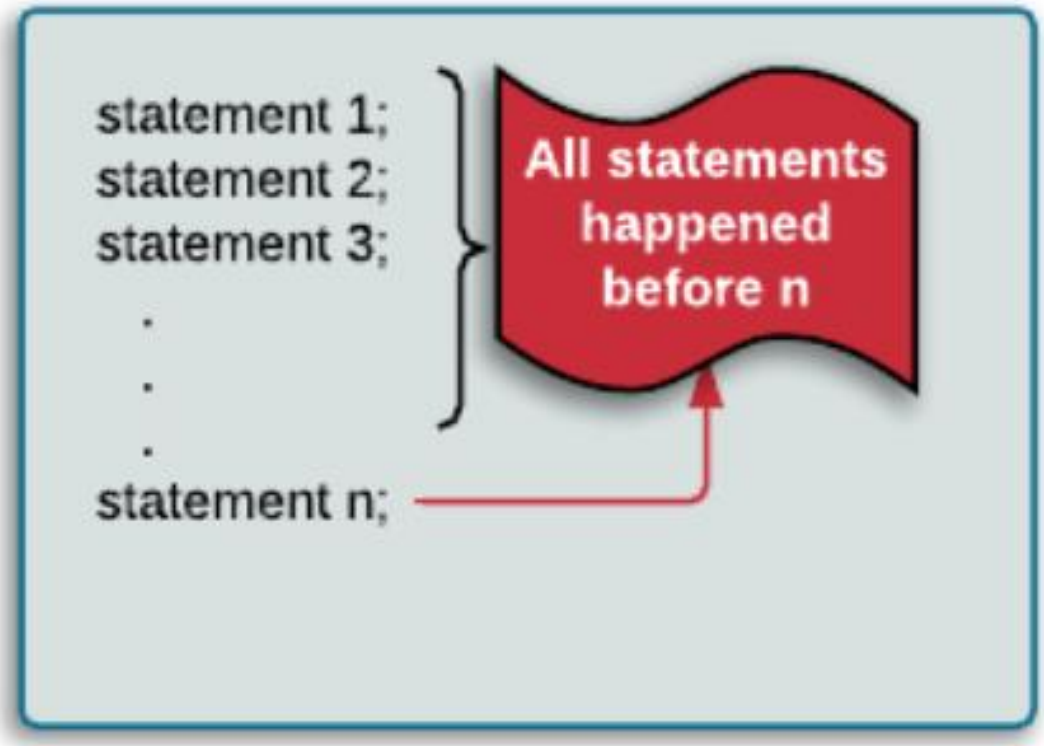
```
Class Test {
  int nonStaticInt;
  static int staticInt;
  volatile static int volStaInt;
}
```



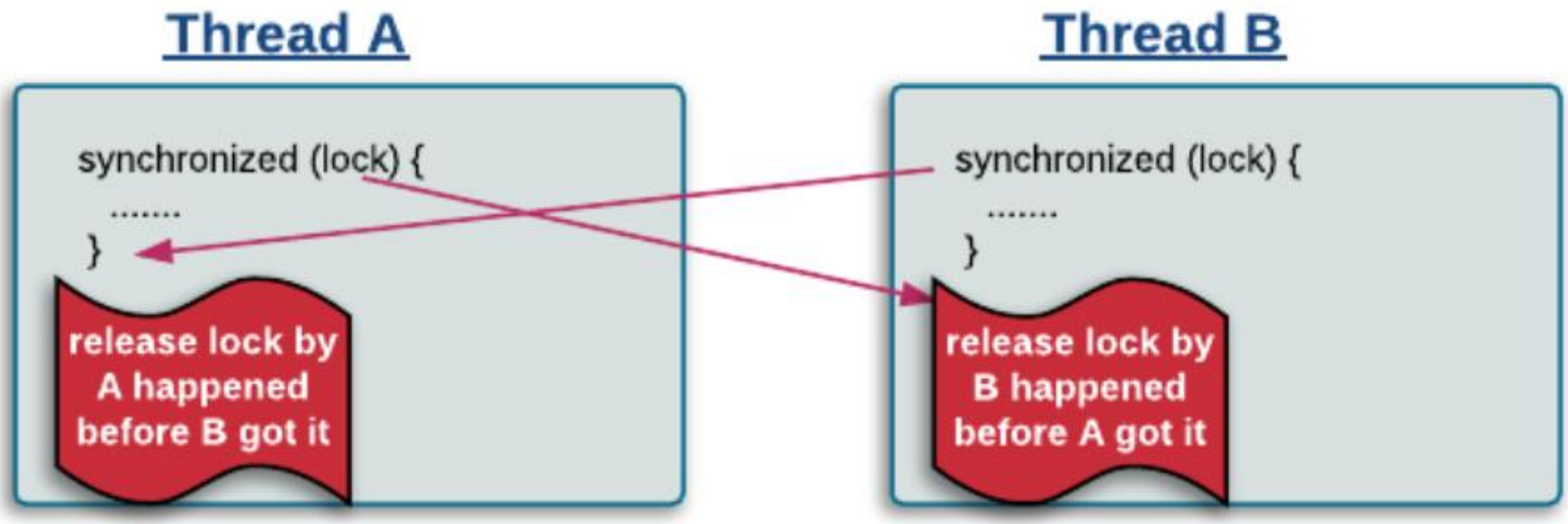


If X should always run before Y then X should have "happens before" relation with Y.

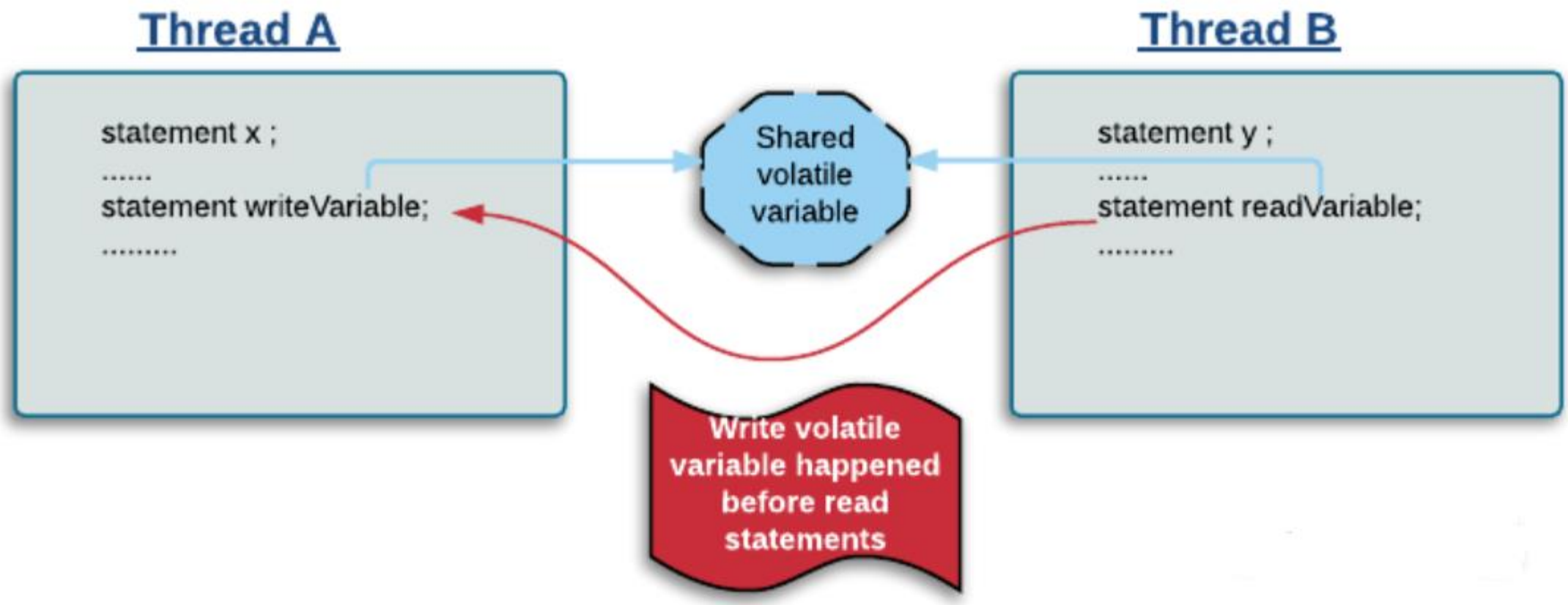
Single Thread rule



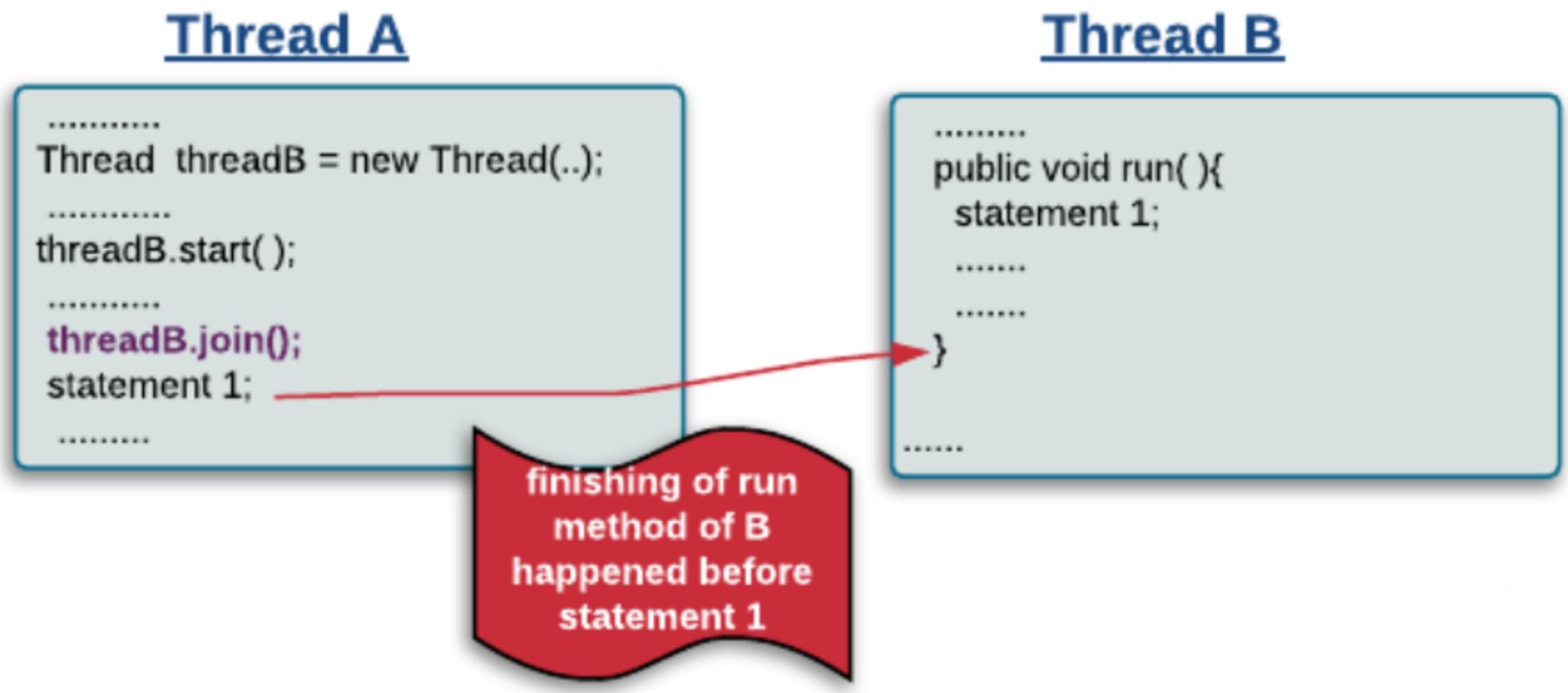
Monitor Lock rule



Volatile Variable Rule



Thread Join rule



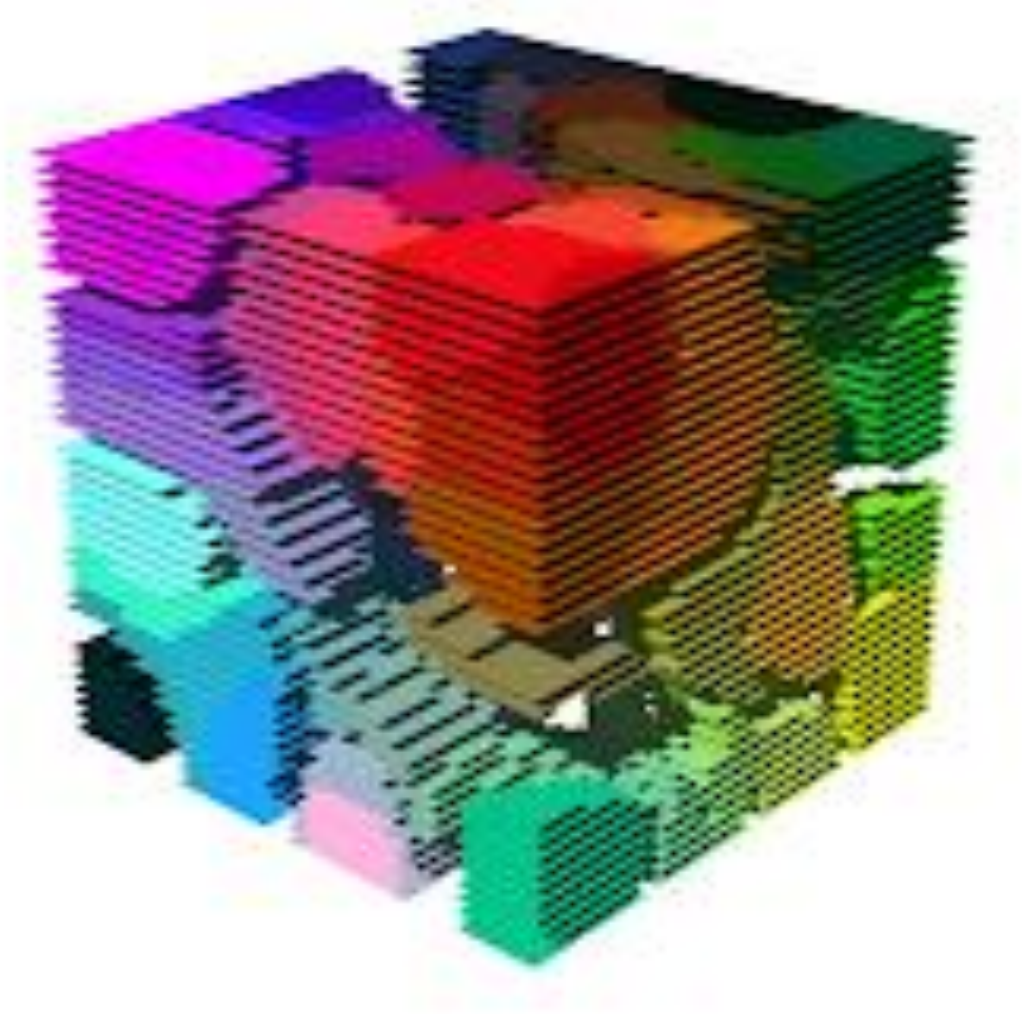
Conditions for correct use of volatile

- You can use volatile variables instead of locks only under a restricted set of circumstances. Both of the following criteria must be met for volatile variables to provide the desired thread-safety:
- Writes to the variable do not depend on its current value.
- The variable does not participate in invariants with other variables.

Listing 1. Non-thread-safe number range class

```
1  @NotThreadSafe
2  public class NumberRange {
3      private int lower, upper;
4
5      public int getLower() { return lower; }
6      public int getUpper() { return upper; }
7
8      public void setLower(int value) {
9          if (value > upper)
10             throw new IllegalArgumentException(...);
11         lower = value;
12     }
13
14     public void setUpper(int value) {
15         if (value < lower)
16             throw new IllegalArgumentException(...);
17         upper = value;
18     }
19 }
```

Performance considerations



Разликата м/у deadlock vs livelock

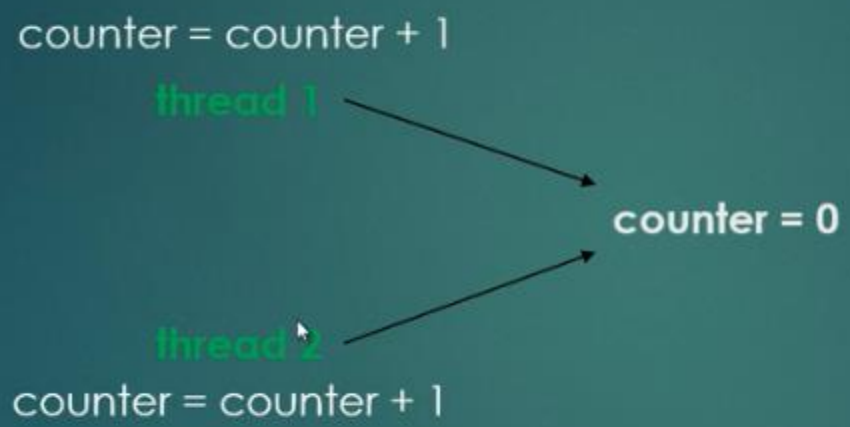
- Deadlock(„мъртва хватка“) е ситуация, в която две или повече действия чакат друго да завърши изпълнението си, но това не се случва
- Databases ->“Мъртва хватка“ се случва когато 2 процеса искате да се подновят данни от 2 реда, но в обратна последователност. Напр. Процес А подновява ред 1 и тогава ред 2, като в същото време процес Б подновява ред 2 и след това ред 1!!!
- ОС -> „мъртва хватка“ е ситуация, която се случва, когато един процес или нишка влезе в състояние “waiting”, за да изчака необходимия му ресурс да се освободи от друг процес, но този процес от друга страна също е в състояние “waiting”, защото изчаква друг процес

livelock

- Една нишка,често действа като отговор на действието на друга нишка
- Ако друга нишка действа като отговор на действието на предходна нишка -> livelock!!!!
- Livelocked threads – това да нишки, които са не могат да изпълнят поради някоя причина. Такива нишки не са блокирани -> те са по скоро са в състояние “busy”
- Подобно на ситуацията когато двама човека се опитват да минат през коридор: А се премества на ляво за да пропусне В, докато В се премества на дясно, за да пусне А. Те се блокират един друг с тяхното действие.

Синхронизиране на нишки

Volatile


counter = counter + 1
thread 1
counter = 0
thread 2
counter = counter + 1

Volatile

Counter remained 1 instead of 2

~ we should make sure the threads are going to wait for each other to finish the given task on the variables !!!

Как работи *multithreading*?

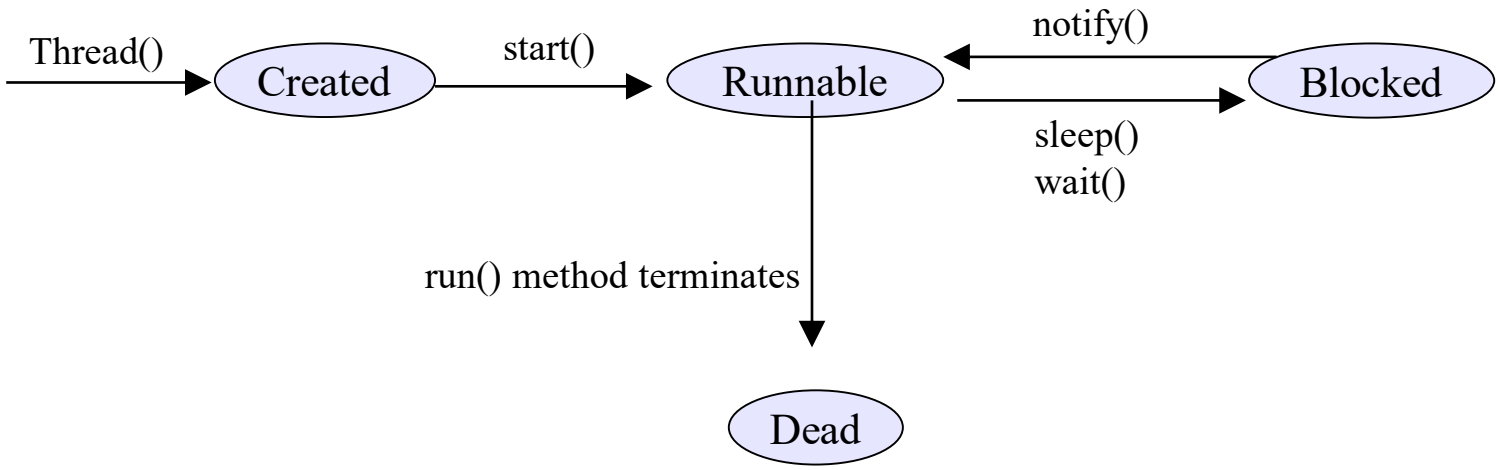
- Всяка нишка има свое собствен "context"
  - Всеки context съдържа виртуални регистри и отделен стек
- Така наречения "scheduler" взима решение коя нишка да се изпълнява в даден момент.
  - За виртуалната памет може да има отделен scheduler
  - Много ОСи директно поддътжат multithreading, виртуалната памет може да използва системния диспечър(scheduler) за поддръжка на нишките
- Диспечърът(scheduler) поддържа списък с готови нишки (които са записани в run queue), както и списък от нишки в състояние waiting (wait queue)
- Всяка нишка има приоритет. Диспечърът избира за изпълнение тези нишки с най-висок от run queue
  - Забележете: програмистът не може да знае колко са нишките, готови за изпълнение и обикновенно, броят на нишките, както и тяхното изпълнение ще различно върху разлини платформи.

Реализация на нишки в Java

- Някои езици за програмиране директно поддържат нишки
  - Поддържат add-on thread режим
  - Режимът add on thread в повечето случаи е сложен за използване
- В JVM има отделни нишки от ОС
  - В JVM се изпозва garbage collection
- Нишките се представят от класа Thread.
  - Обект от класа има в себе си състояние на нишка
  - Осигурява методи като interrupt, start, sleep, yield, wait
- Методът main се изпълнява от главната нишка.
  - Когато приложението изискава повече нишки, то те трябва допълнително да се създат.

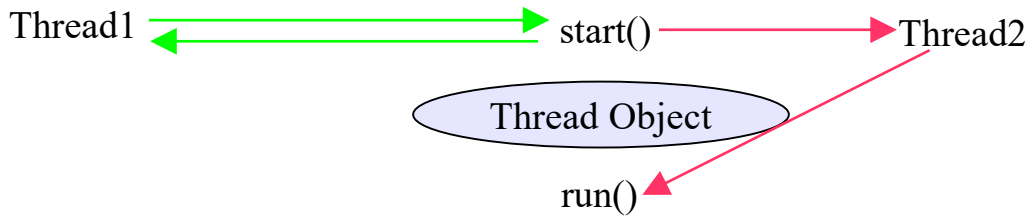
Състояния на нишките

- Нишките могат да бъдат в едно от 4-те състояния
  - Created, Running, Blocked, и Dead
- Състоянието на нишките се променя в зависимост от:
  - Методите за управление на нишките :start, sleep, yield, wait, notify
  - Спиране на изпълнението на програмата



Как се стартираThread?

- Извиква се метода run()
  - run() се изпълнва, когато метода start() се извиква от нишката
- Нишката ще се терминира ако метода run() е терминиран
  - За да не се терминира нишката, метода run() трябва да не се прекрати
  - Методът run() често има в себе си безкраен цикъл, за да не се спре нишката
- Една нишка извиква друга, когато тя извика метода start на другата нишка.
  - Последователността от събития може да обърка тези, които се по-запознати с едно-нишковото програмиране.

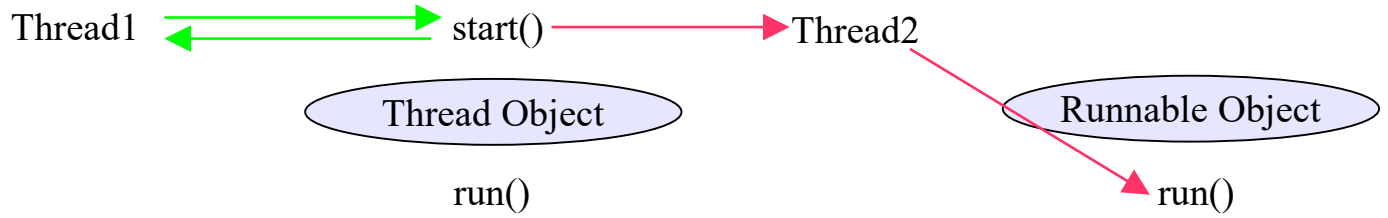


Creating your own Threads

- The obvious way to create your own threads is to subclass the Thread class and then override the run() method
  - This is the easiest way to do it
  - It is not the recommended way to do it.
- Because threads are usually associated with a task, the object which provides the run method is usually a subclass of some other class
  - If it inherits from another class, it cannot inherit from Thread.
- The solution is provided by an interface called Runnable.
  - Runnable defines one method - public void run()
- One of the Thread classes constructor takes a reference to a Runnable object
  - When the thread is started, it invokes the run method in the runnable object instead of its own run method.

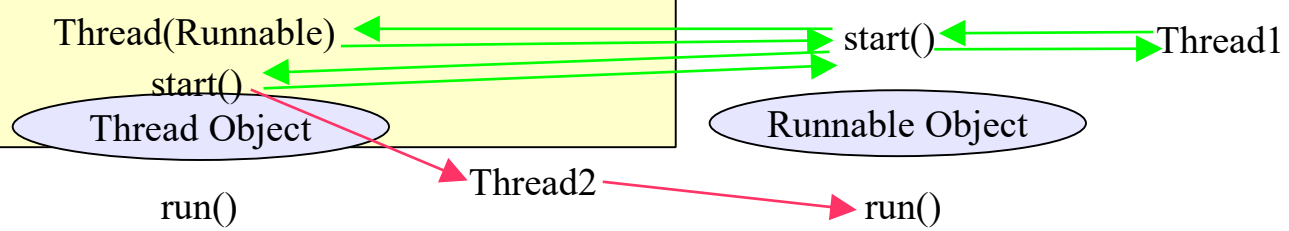
Using Runnable

- In the example below, when the Thread object is instantiated, it is passed a reference to a "Runnable" object
  - The Runnable object must implement a method called "run"
- When the thread object receives a start message, it checks to see if it has a reference to a Runnable object:
  - If it does, it runs the "run" method of that object
  - If not, it runs its own "run" method



Example Code

```
public class Test implements Runnable
{
  private Thread theThread;

  public void start()
  {
        if (theThread == null)
        {
              theThread = new Thread(this);
              theThread.start();
        }
  }

  public void run()
  {
        // This method runs in its
        // own thread
  }
```



Properly Terminating Threads

- In Java 1.1, the Thread class had a stop() method
  - One thread could terminate another by invoking its stop() method.
  - However, using stop() could lead to deadlocks
  - The stop() method is now deprecated. DO NOT use the stop method to terminate a thread
- The correct way to stop a thread is to have the run method terminate
  - Add a boolean variable which indicates whether the thread should continue or not
  - Provide a set method for that variable which can be invoked by another thread

Terminating Thread Example

```
public class Test implements Runnable
{
  private Thread theThread;
  private boolean stopThread = false;

  public void start()
  {
        if (theThread == null)
        {
                theThread = new Thread(this);
                theThread.start();
        }
  }

  public void setStopThread(boolean aValue)
  {
        stopThread = aValue;
  }

  public void run()
  {
        while(true)
        {
                if (stopThread)
                        break;
                //  ....
```

Creating Multiple Threads

- The previous example illustrates a Runnable class which creates its own thread when the start method is invoked.
- If one wished to create multiple threads, one could simple create multiple instances of the Runnable class and send each object a start message
  - Each instance would create its own thread object
- Is the a maximum number of threads which can be created?
  - There is no defined maximum in Java.
  - If the VM is delegating threads to the OS, then this is platform dependent.
  - A good rule of thumb for maximum thread count is to allow 2Mb of ram for each thread
    - Although threads share the same memory space, this can be a reasonable estimate of how many threads your machine can handle.

Thread Priorities

- Every thread is assigned a priority (between 1 and 10)
  - The default is 5
  - The higher the number, the higher the priority
  - Can be set with setPriority(int aPriority)
- The standard mode of operation is that the scheduler executes threads with higher priorities first.
  - This simple scheduling algorithm can cause problems. Specifically, one high priority thread can become a "CPU hog".
  - A thread using vast amounts of CPU can share CPU time with other threads by invoking the yield() method on itself.
- Most OSes do not employ a scheduling algorithm as simple as this one
  - Most modern OSes have thread aging
    - The more CPU a thread receives, the lower its priority becomes
    - The more a thread waits for the CPU, the higher its priority becomes
  - Because of thread aging, the effect of setting a thread's priority is dependent on the platform

Yield() and Sleep()

- Sometimes a thread can determine that it has nothing to do
  - Sometimes the system can determine this. ie. waiting for I/O
- When a thread has nothing to do, it should not use CPU
  - This is called a busy-wait.
  - Threads in busy-wait are busy using up the CPU doing nothing.
    - Often, threads in busy-wait are continually checking a flag to see if there is anything to do.
- It is worthwhile to run a CPU monitor program on your desktop
  - You can see that a thread is in busy-wait when the CPU monitor goes up (usually to 100%), but the application doesn't seem to be doing anything.
- Threads in busy-wait should be moved from the Run queue to the Wait queue so that they do not hog the CPU
  - Use yield() or sleep(time)
  - Yield simply tells the scheduler to schedule another thread
  - Sleep guarantees that this thread will remain in the wait queue for the specified number of milliseconds.

Concurrent Access to Data

- Those familiar with databases will understand that concurrent access to data can lead to data integrity problems
  - Specifically, if two sources attempt to update the same data at the same time, the result of the data can be undefined.
  - The outcome is determined by how the scheduler schedules the two sources.
    - Since the schedulers activities cannot be predicted, the outcome cannot be predicted
- Databases deal with this mechanism through "locking"
  - If a source is going to update a table or record, it can lock the table or record until such time that the data has been successfully updated.
  - While locked, all access is blocked except to the source which holds the lock.
- Java has the equivalent mechanism. It is called synchronization
  - Java has a keyword called synchronized

Synchronization

- In Java, every object has a lock
  - To obtain the lock, you must synchronize with the object
- The simplest way to use synchronization is by declaring one or more methods to be synchronized
  - When a synchronized method is invoked, the calling thread attempts to obtain the lock on the object.
    - if it cannot obtain the lock, the thread goes to sleep until the lock becomes available
  - Once the lock is obtained, no other thread can obtain the lock until it is released. ie, the synchronized method terminates
  - When a thread is within a synchronized method, it knows that no other synchronized method can be invoked by any other thread
    - Therefore, it is within synchronized methods that critical data is updated

Providing Thread Safe Access to Data

- If an object contains data which may be updated from multiple thread sources, the object should be implemented in a thread-safe manner
  - All access to critical data should only be provided through synchronized methods (or synchronized blocks).
  - In this way, we are guaranteed that the data will be updated by only one thread at a time.

```
public class SavingsAccount
{
  private float balance;

  public synchronized void withdraw(float anAmount)
  {
        if ((anAmount>0.0) && (anAmount<=balance))
                balance = balance - anAmount;
  }

  public synchronized void deposit(float anAmount)
  {
        if (anAmount>0.0)
                balance = balance + anAmount;
  }
```

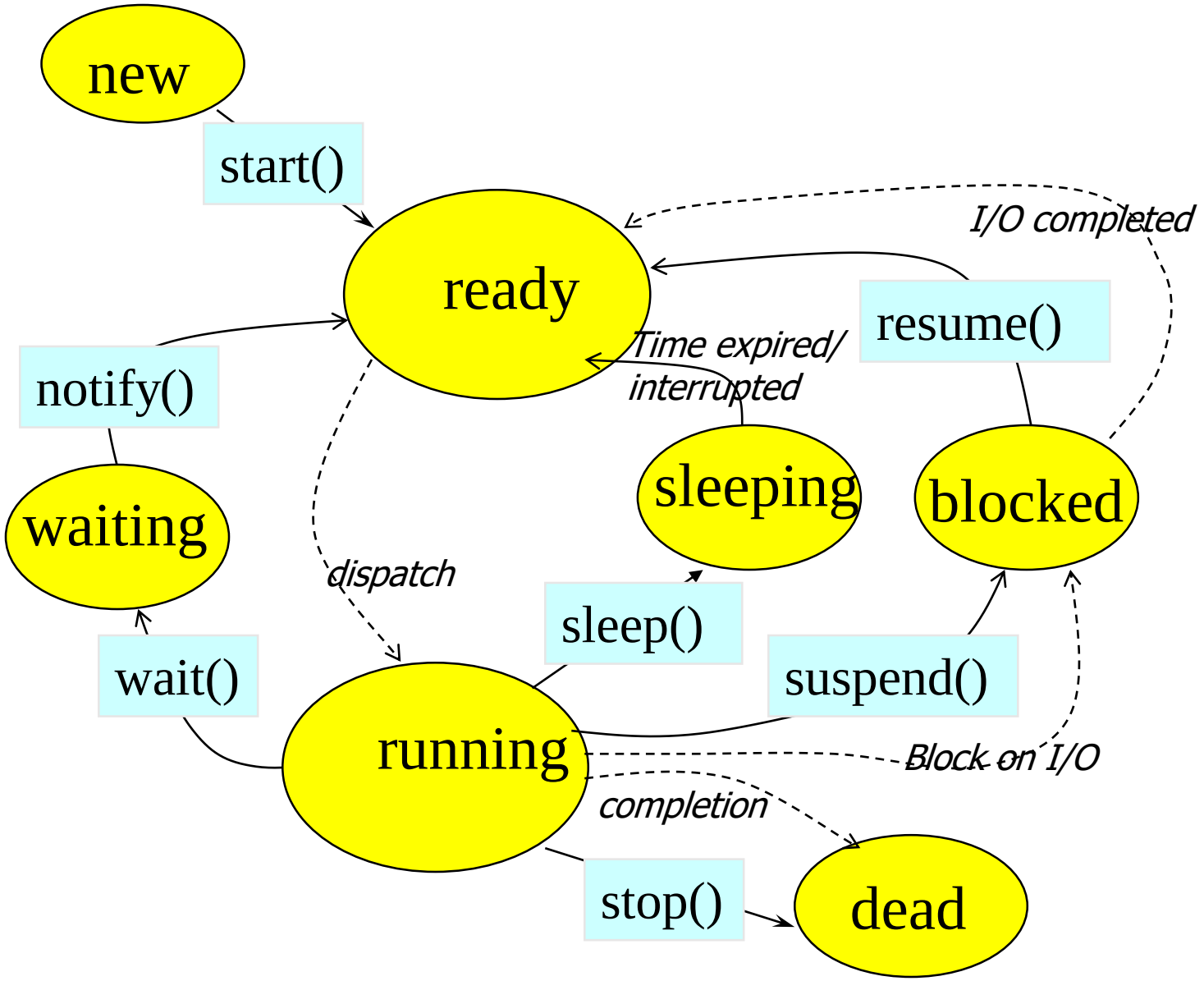
Thread Safety Performance Issues

- However, there is an overhead associated with synchronization
  - Many threads may be waiting to gain access to one of the object's synchronized methods
  - The object remains locked as long as a thread is within a synchronized method.
  - Ideally, the method should be kept as short as possible.
- Another solution is to provide synchronization on a block of code instead of the entire method
  - In this case, the object's lock is only held for the time that the thread is within the block.
  - The intent is that we only lock the region of code which requires access to the critical data. Any other code within the method can occur without the lock.
  - In high load situations where multiple threads are attempting to access critical data, this is by far a much better implementation.

Block Synchronization

```
public class SavingsAccount
{
  private float balance;

  public void withdraw(float anAmount)
  {
        if (anAmount<0.0)
              throw new IllegalArgumentException("Withdraw amount negative");
        synchronized(this)
        {
              if (anAmount<=balance)
                    balance = balance - anAmount;
        }
  }

  public void deposit(float anAmount)
  {
        if (anAmount<0.0)
              throw new IllegalArgumentException("Deposit amount negative");
        synchronized(this)
        {
              balance = balance + anAmount;
        }
  }
```

Life Cycle of Thread



A Program with Three Java Threads

- Write a program that creates 3 threads

Three threads example

```
class A extends Thread
{
    public void run()
     {
        for(int i=1;i<=5;i++)
         {
            System.out.println("\t From ThreadA: i= "+i);
         }
          System.out.println("Exit from A");
     }
}

class B extends Thread
{
    public void run()
     {
        for(int j=1;j<=5;j++)
         {
            System.out.println("\t From ThreadB: j= "+j);
         }
          System.out.println("Exit from B");
     }
}
```

```
class C extends Thread
{
    public void run()
     {
        for(int k=1;k<=5;k++)
         {
            System.out.println("\t From ThreadC: k= "+k);
         }

          System.out.println("Exit from C");
    }
}

class ThreadTest
{
     public static void main(String args[])
     {
            new A().start();
            new B().start();
            new C().start();
     }
}
```

Run 1

- [raj@mundroo] threads [1:76] java ThreadTest

  From ThreadA: i= 1

  From ThreadA: i= 2

  From ThreadA: i= 3

  From ThreadA: i= 4

  From ThreadA: i= 5

Exit from A

  From ThreadC: k= 1

  From ThreadC: k= 2

  From ThreadC: k= 3

  From ThreadC: k= 4

  From ThreadC: k= 5

Exit from C

  From ThreadB: j= 1

  From ThreadB: j= 2

  From ThreadB: j= 3

  From ThreadB: j= 4

  From ThreadB: j= 5

Exit from B

Run2

- [raj@mundroo] threads [1:77] java ThreadTest

  From ThreadA: i= 1

  From ThreadA: i= 2

  From ThreadA: i= 3

  From ThreadA: i= 4

  From ThreadA: i= 5

  From ThreadC: k= 1

  From ThreadC: k= 2

  From ThreadC: k= 3

  From ThreadC: k= 4

  From ThreadC: k= 5

Exit from C

  From ThreadB: j= 1

  From ThreadB: j= 2

  From ThreadB: j= 3

  From ThreadB: j= 4

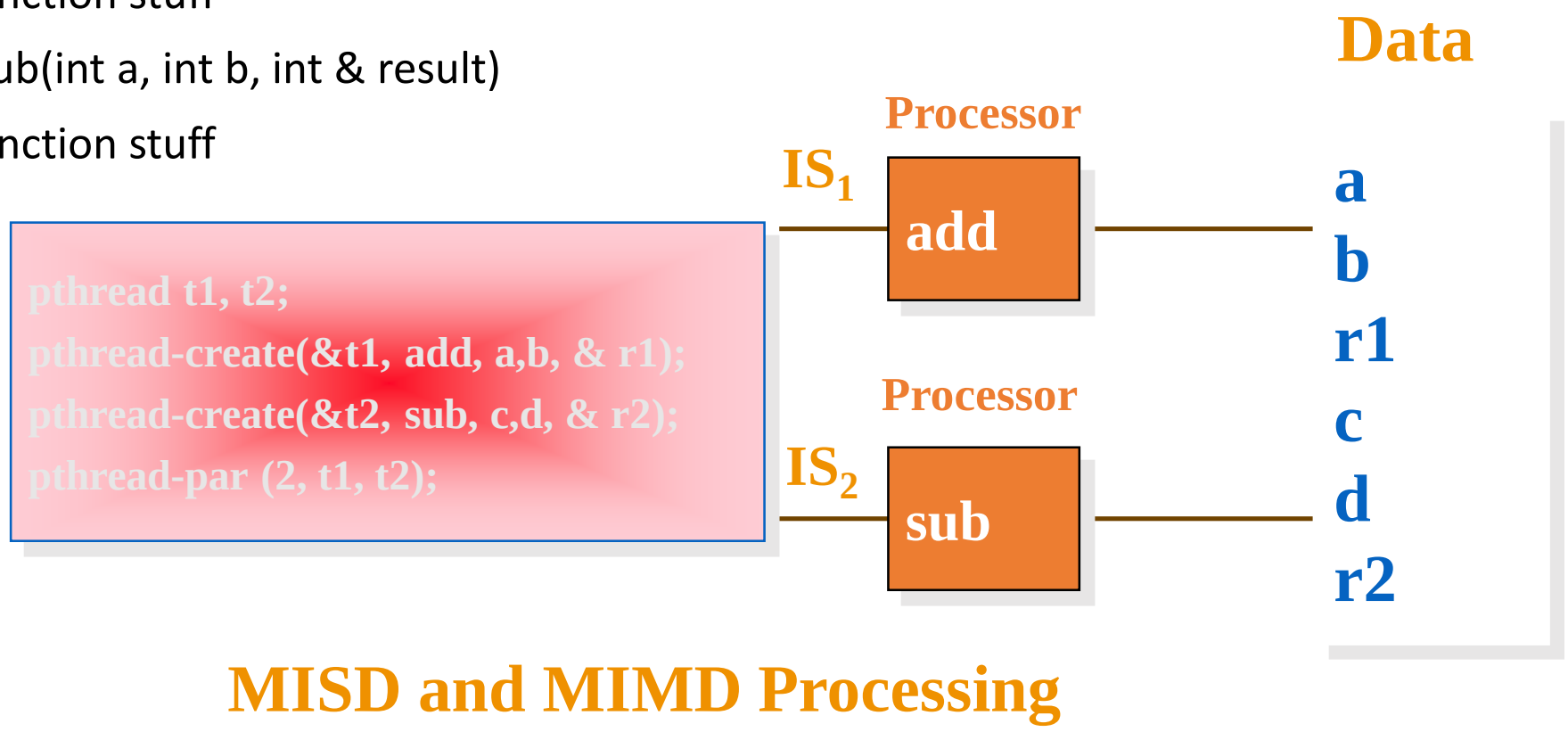
  From ThreadB: j= 5

Exit from B

Exit from A

Process Parallelism

- int add (int a, int b, int & result)
- // function stuff
- int sub(int a, int b, int & result)
- // function stuff

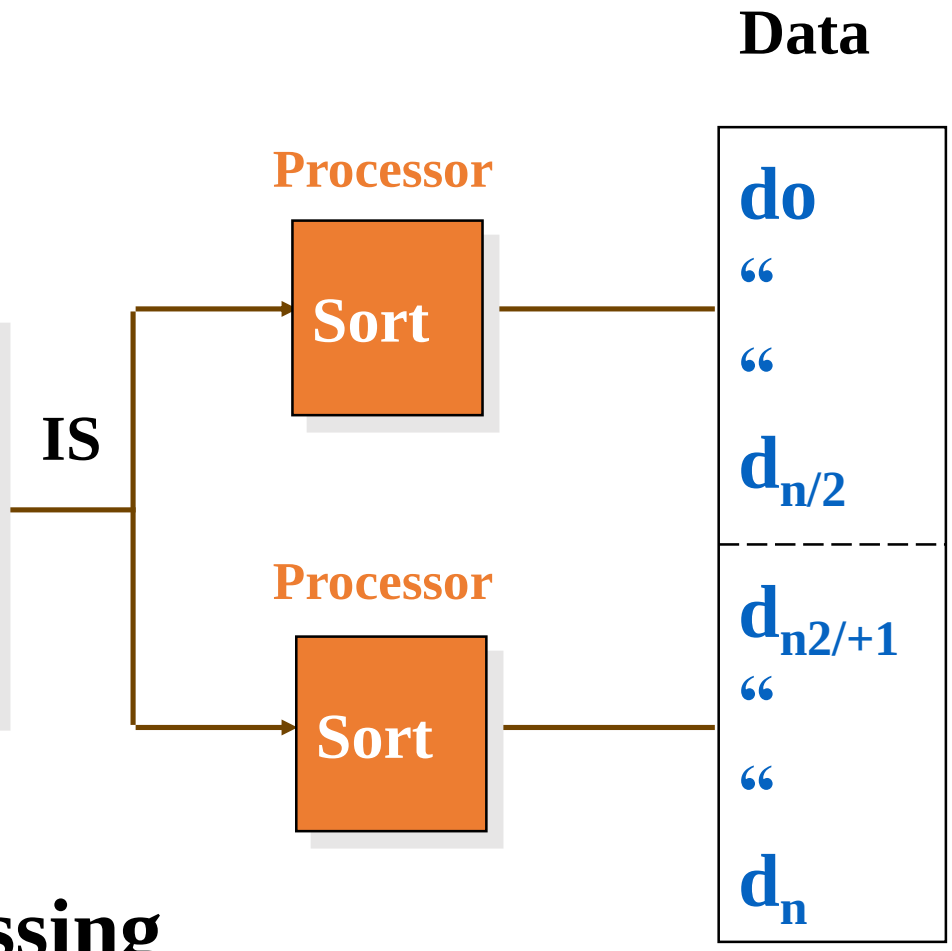


MISD and MIMD Processing

Data Parallelism

- sort(int *array, int count)
- //.....
- //.....

```
pthread-t, thread1, thread2;
“
“
pthread-create(& thread1, sort, array, N/2);
pthread-create(& thread2, sort, array, N/2);
pthread-par(2, thread1, thread2);
```



SIMD Processing

Thread Priority

- In Java, each thread is assigned priority, which affects the order in which it is scheduled for running. The threads so far had same default priority (NORM_PRIORITY) and they are served using FCFS policy.
  - Java allows users to change priority:
    - ThreadName.setPriority(intNumber)
      - MIN_PRIORITY = 1
      - NORM_PRIORITY=5
      - MAX_PRIORITY=10

Thread Priority Example

```
class A extends Thread
{
    public void run()
     {
        System.out.println("Thread A started");
        for(int i=1;i<=4;i++)
         {
            System.out.println("\t From ThreadA: i= "+i);
         }
          System.out.println("Exit from A");
     }
}
class B extends Thread
{
    public void run()
     {
        System.out.println("Thread B started");
        for(int j=1;j<=4;j++)
         {
            System.out.println("\t From ThreadB: j= "+j);
         }
          System.out.println("Exit from B");
     }
}
```

Thread Priority Example

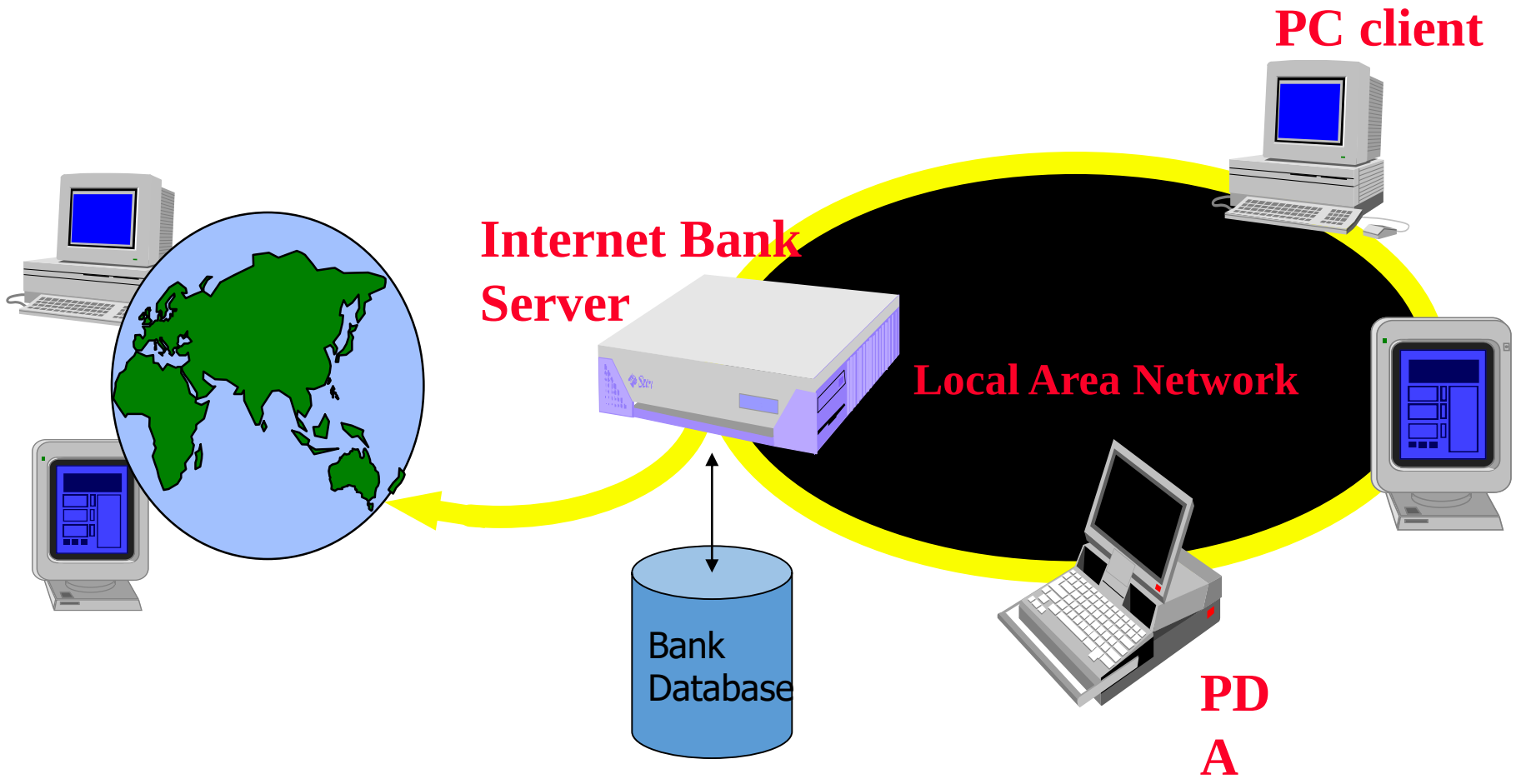
```
class C extends Thread
{
      public void run()
       {
           System.out.println("Thread C started");
           for(int k=1;k<=4;k++)
            {
                 System.out.println("\t From ThreadC: k= "+k);
            }
             System.out.println("Exit from C");
      }
}
```

```
class ThreadPriority
{
        public static void main(String args[])
         {
                  A threadA=new A();
                  B threadB=new B();
                  C threadC=new C();
                threadC.setPriority(Thread.MAX_PRIORITY);
                threadB.setPriority(threadA.getPriority()+1);
                threadA.setPriority(Thread.MIN_PRIORITY);
                System.out.println("Started Thread A");
                 threadA.start();
                System.out.println("Started Thread B");
                 threadB.start();
                System.out.println("Started Thread C");
                 threadC.start();
                 System.out.println("End of main thread");
        }
}
```

Accessing Shared Resources

- Applications Access to Shared Resources need to be coordinated.
  - Printer (two person jobs cannot be printed at the same time)
  - Simultaneous operations on your bank account.
  - Can the following operations be done at the same time on the same account?
    - Deposit()
    - Withdraw()
    - Enquire()

Online Bank: Serving Many Customers and Operations



Shared Resources

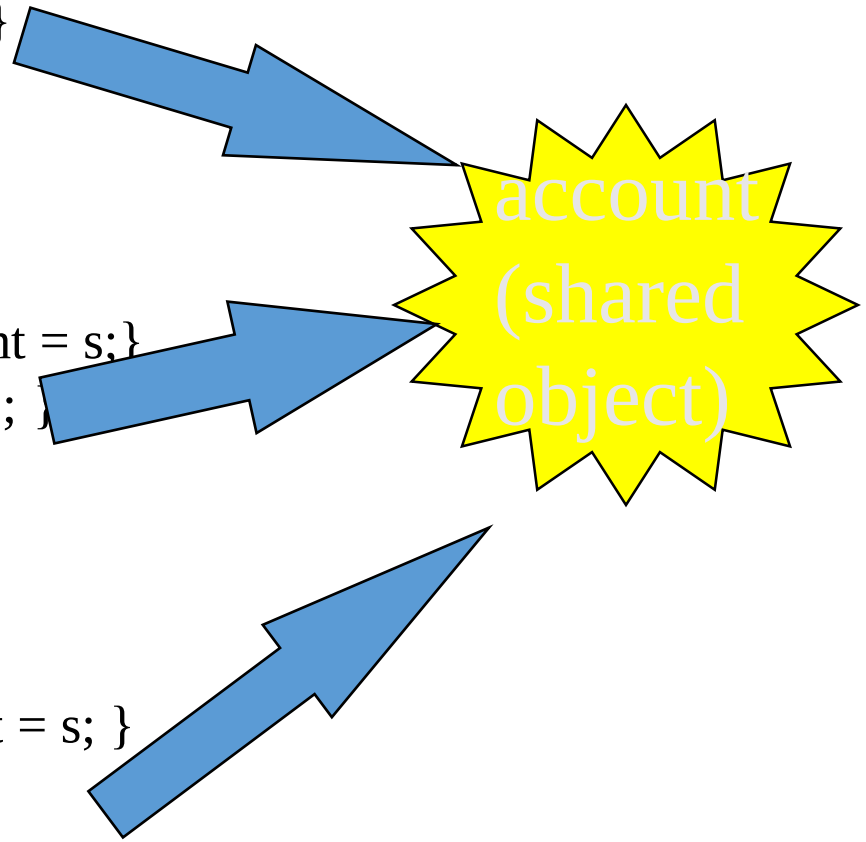
- If one thread tries to read the data and other thread tries to update the same data, it leads to inconsistent state.
- This can be prevented by synchronising access to the data.
- Use “Synchronized” method:
  - public synchronized void update()
  - {
    - …
  - }

the driver: 3rd Threads sharing the same object

```
class InternetBankingSystem {
        public static void main(String [] args  ) {
            Account accountObject = new Account ();
            Thread t1 = new Thread(new MyThread(accountObject));
            Thread t2 = new Thread(new YourThread(accountObject));
            Thread t3 = new Thread(new HerThread(accountObject));
            t1.start();
            t2.start();
            t3.start();
          // DO some other operation
       } // end main()
}
```

Shared account object between 3 threads

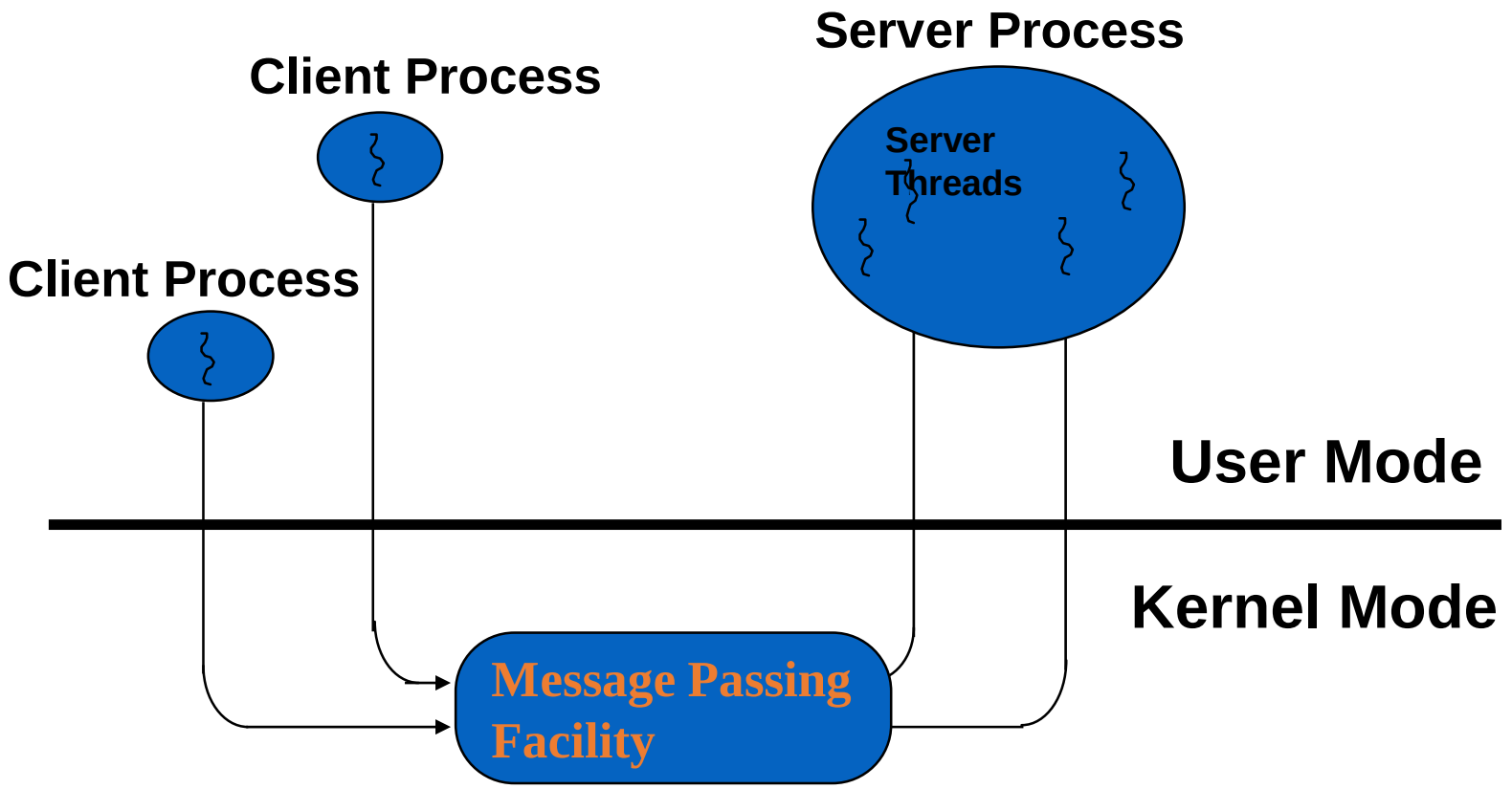
```
class MyThread implements Runnable  {
 Account account;
        public MyThread (Account s) {  account = s;}
        public void run() { account.deposit(); }
} // end class MyThread

class YourThread implements Runnable  {
 Account account;
        public YourThread (Account s) { account = s;}
        public void run() { account.withdraw(); }
} // end class YourThread

class HerThread implements Runnable  {
 Account account;
        public HerThread (Account s) { account = s; }
        public void run() {account.enquire(); }
} // end class HerThread
```



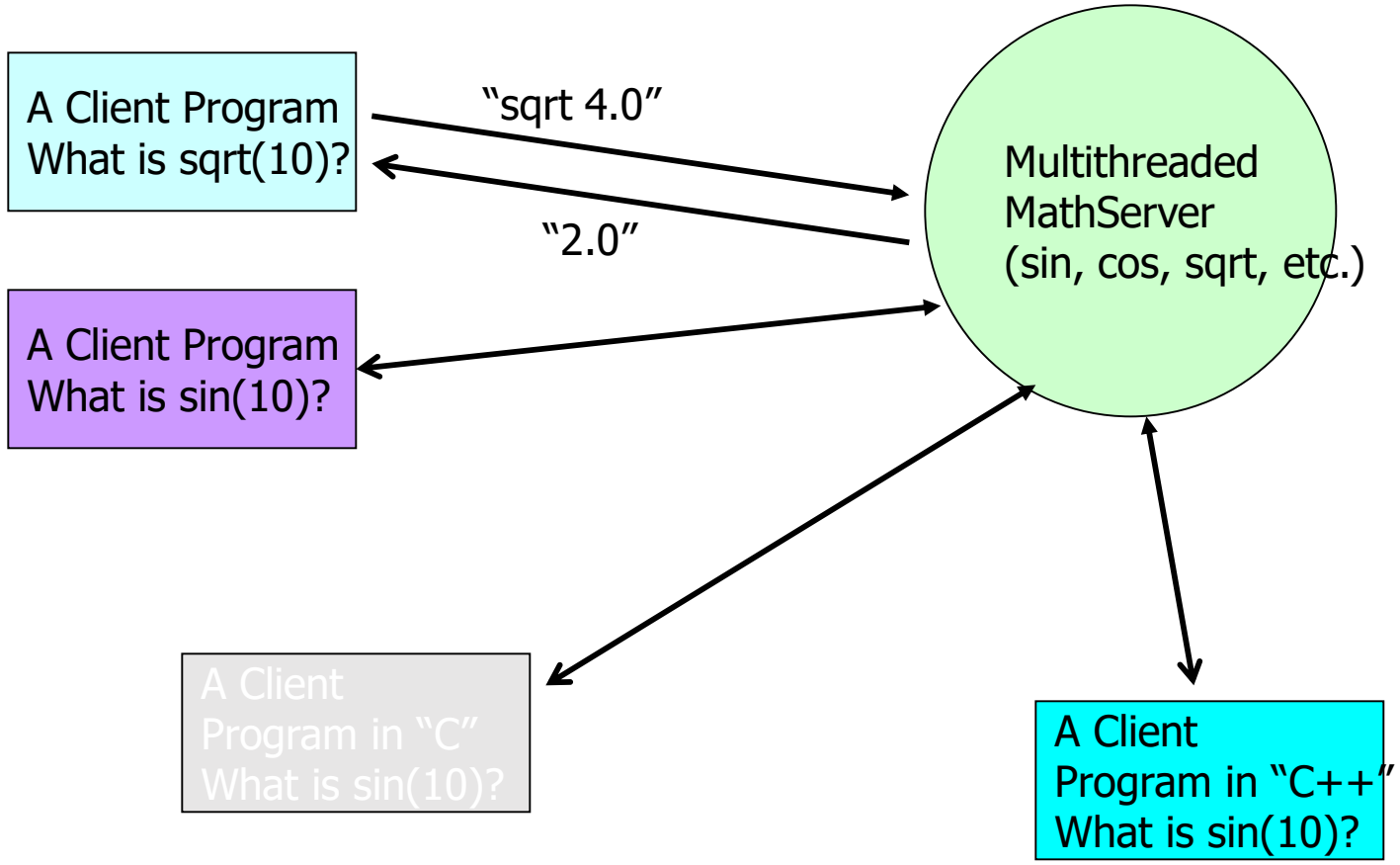
Monitor (shared object access): serializes operation on shared object

```
class Account {   // the 'monitor'
  int balance;

      // if 'synchronized' is removed, the outcome is unpredictable
       public synchronized void deposit( ) {
         // METHOD BODY : balance += deposit_amount;
        }

        public synchronized void withdraw( ) {
          // METHOD BODY: balance -= deposit_amount;
        }
        public synchronized void enquire( ) {
          // METHOD BODY: display balance.
        }
}
```

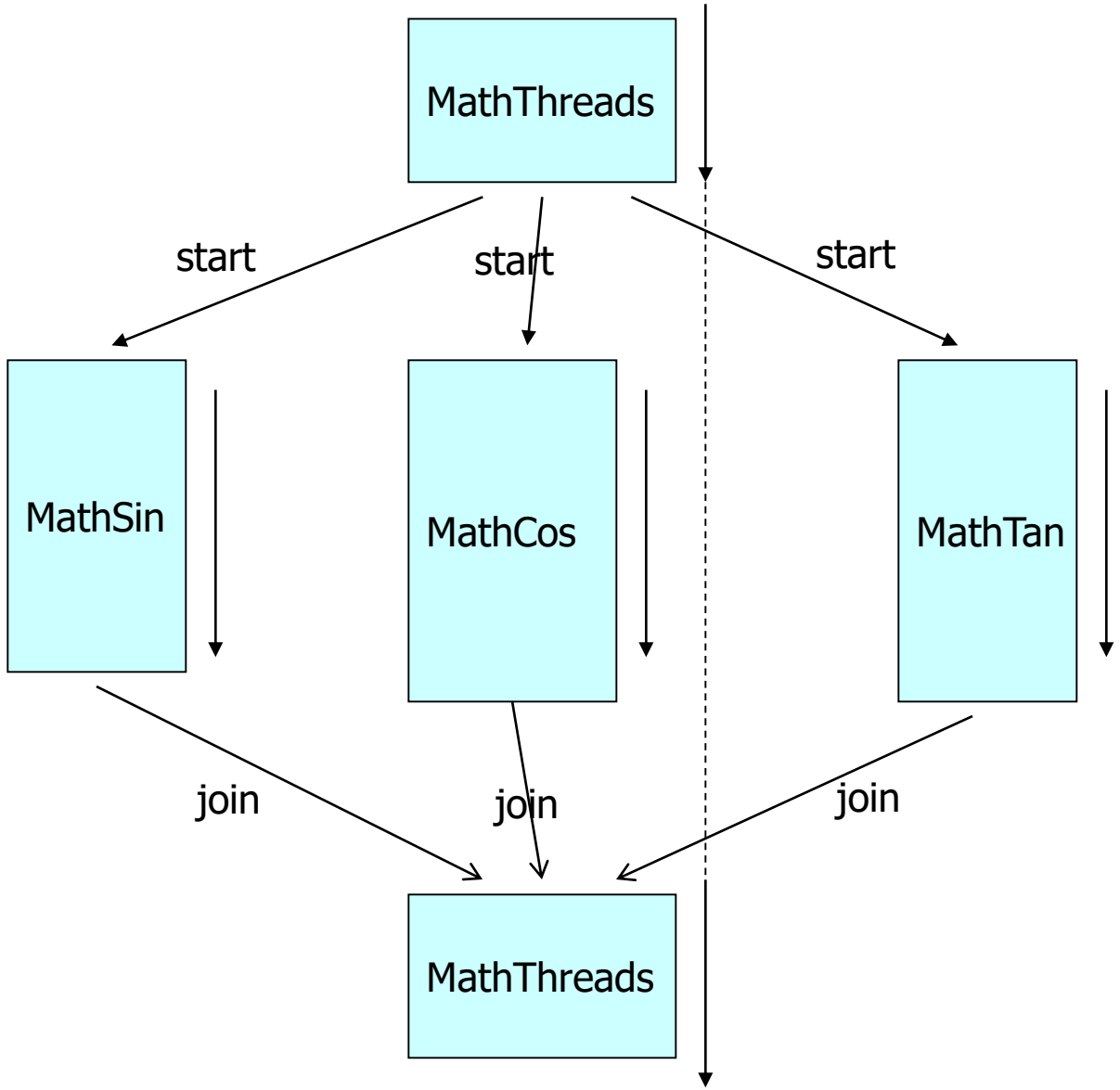
Multithreaded Server



Assignment 1: Multithreaded MathServer – Demonstrates the use of Sockets and Threads



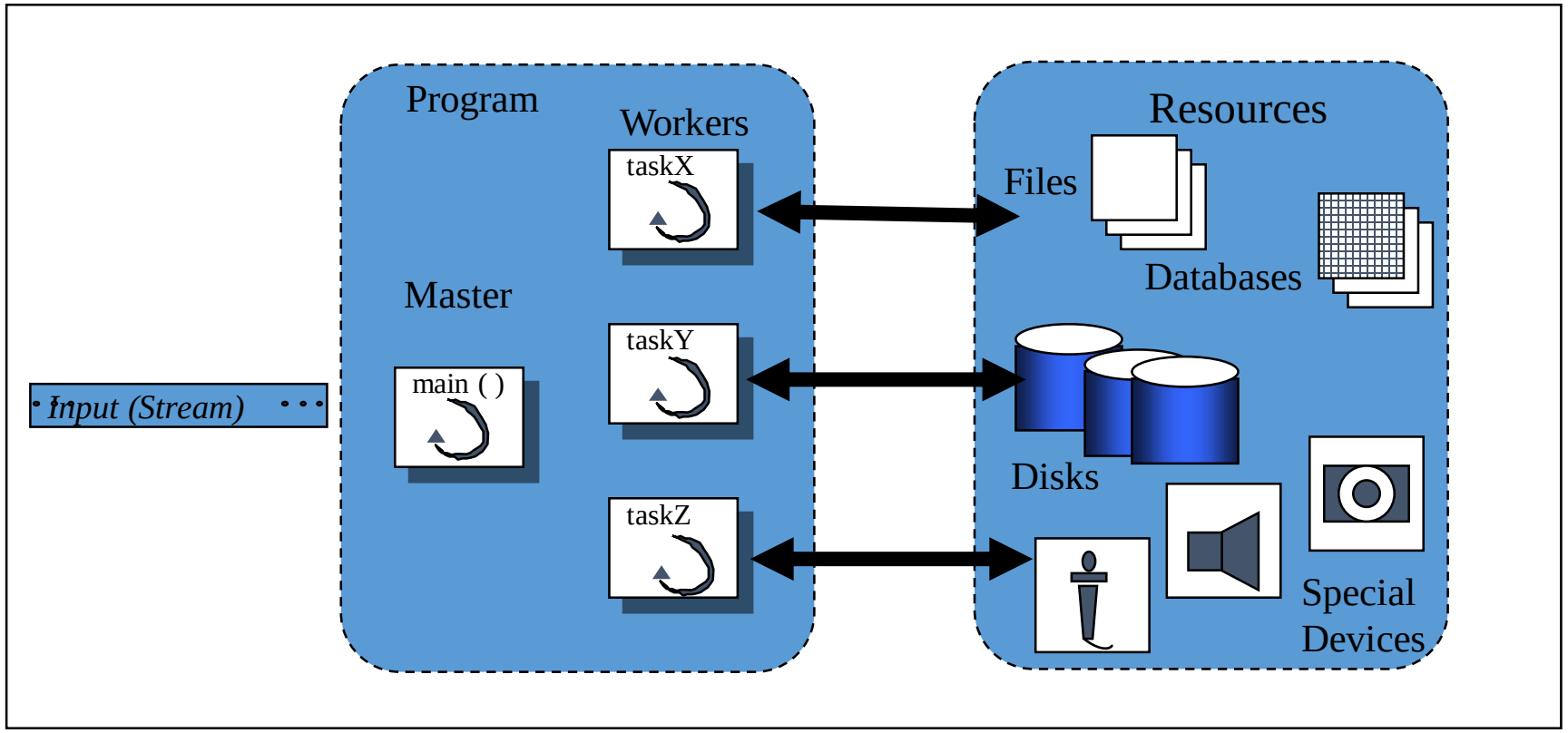
A Multithreaded Program



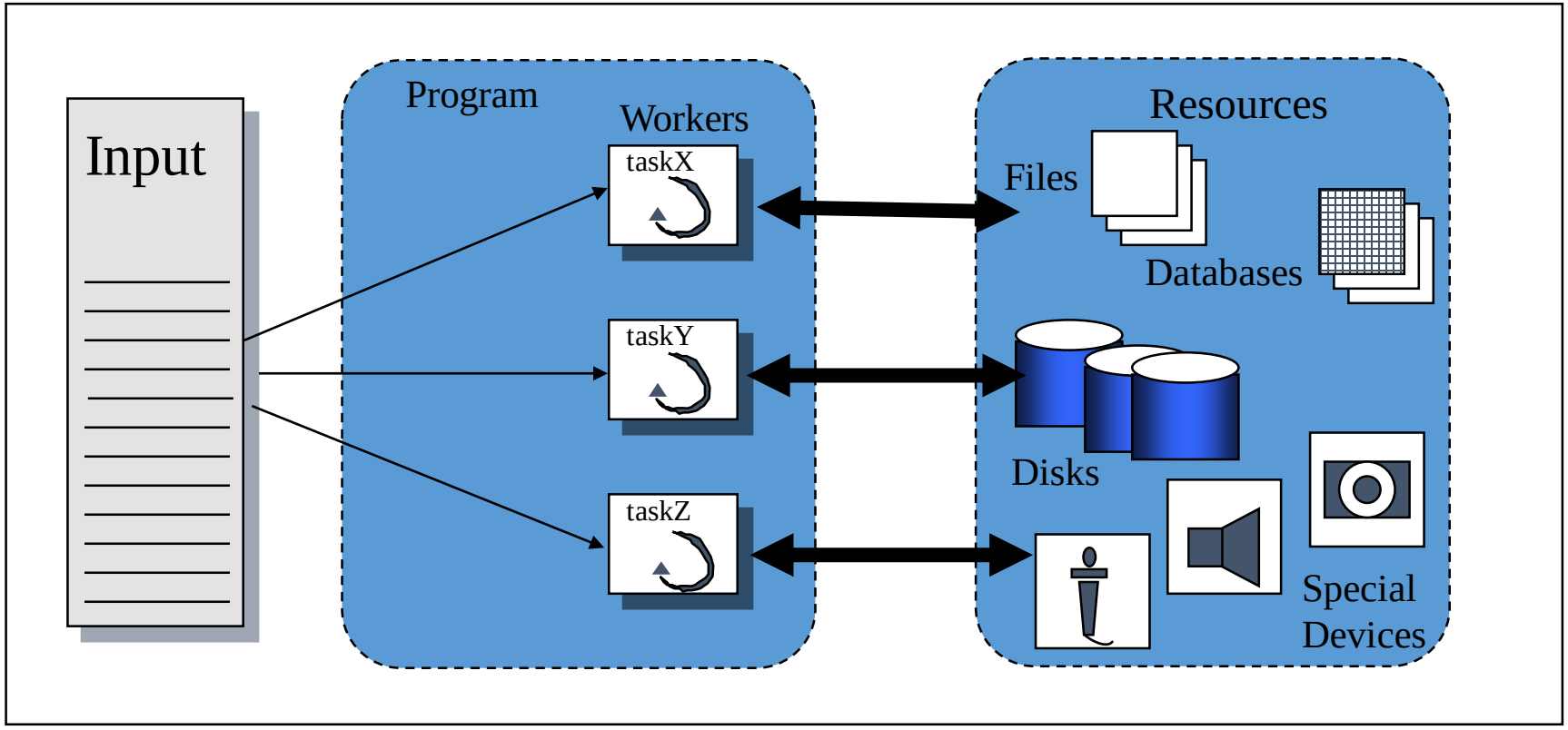
Thread concurrency/operation models

- The master/worker model
- The peer model
- A thread pipeline

The master/worker model

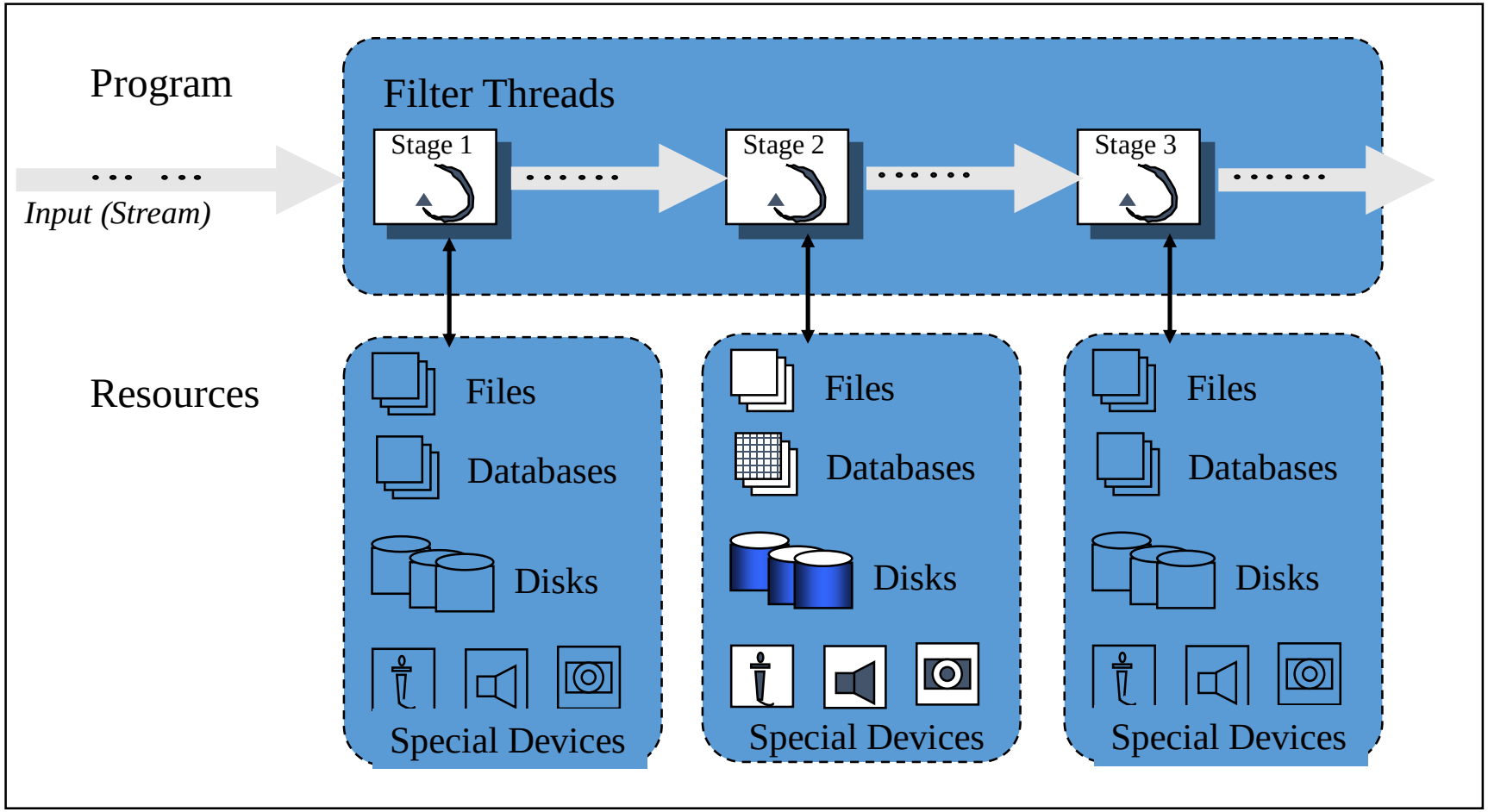


The peer model



A thread pipeline

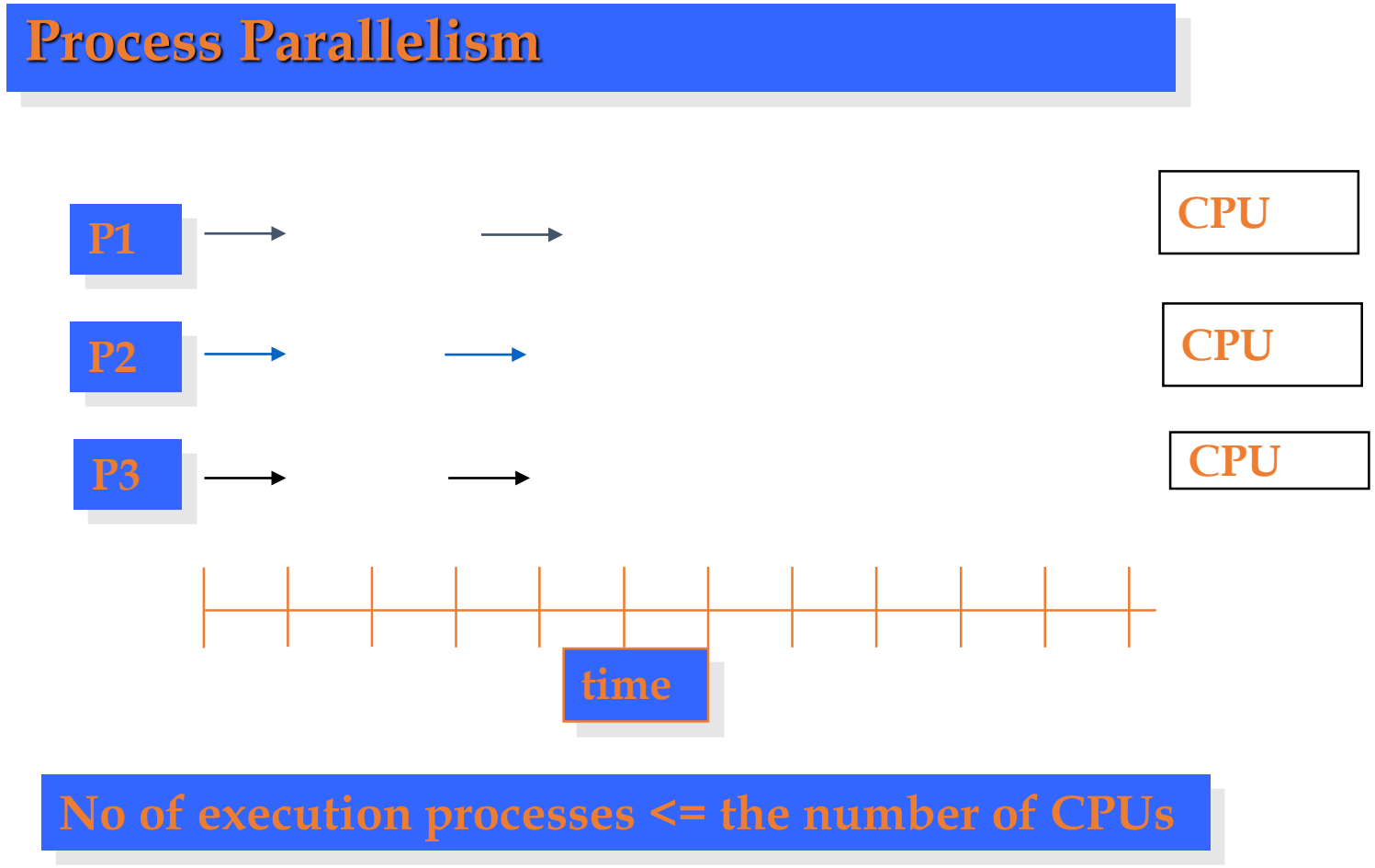
A thread pipeline



Multithreading and Multiprocessing Deployment issues

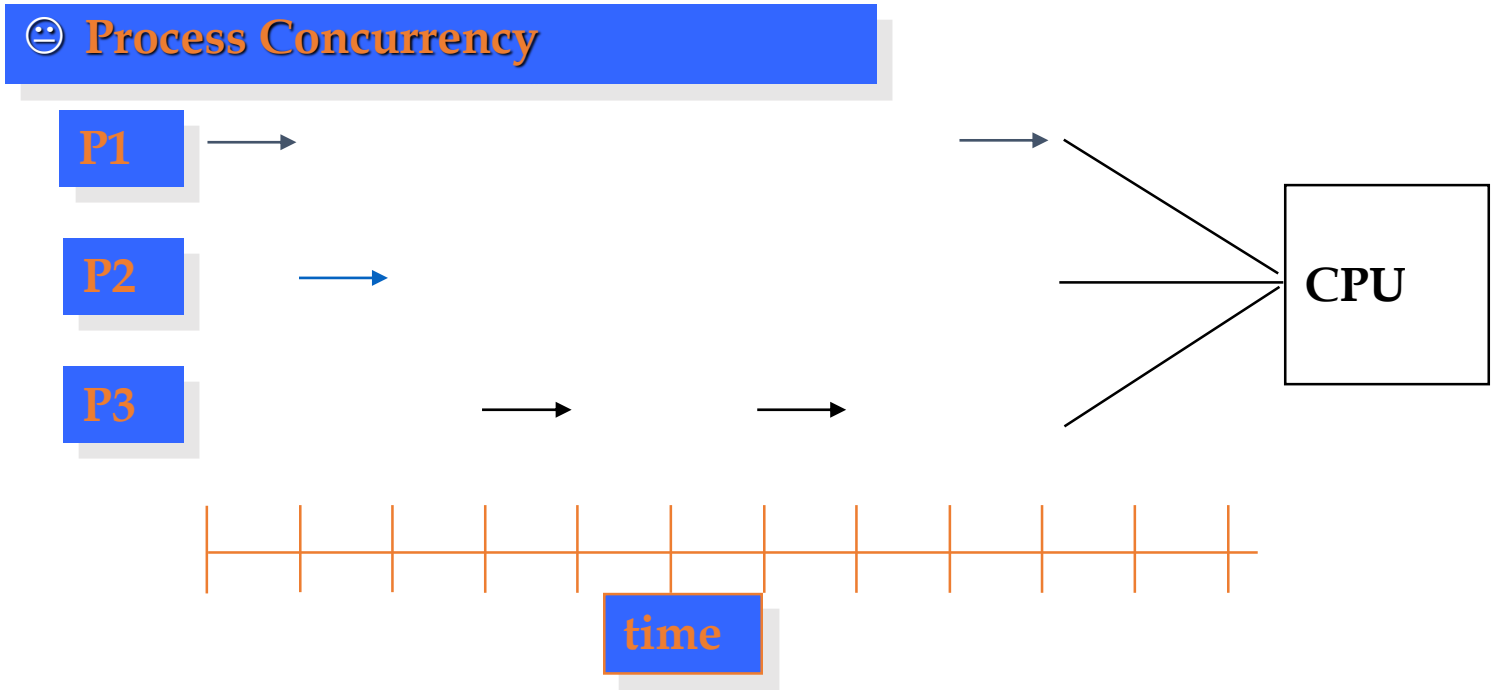
On Shared and distributed memory systems

Multithreading - Multiprocessors



Multithreading on Uni-processor

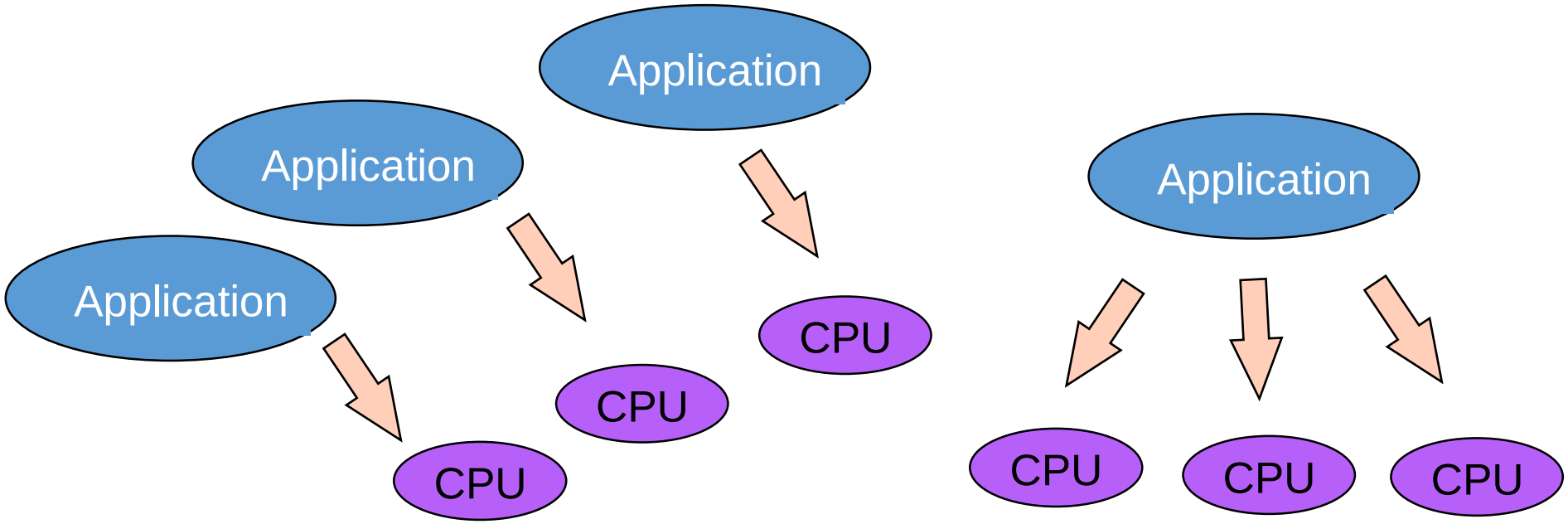
- Concurrency Vs Parallelism



Number of Simultaneous execution units > number of CPUs

Multi-Processing (clusters & grids) and Multi-Threaded Computing

Threaded Libraries, Multi-threaded I/O



Better Response Times in Multiple Application Environments

Higher Throughput for Parallelizeable Applications