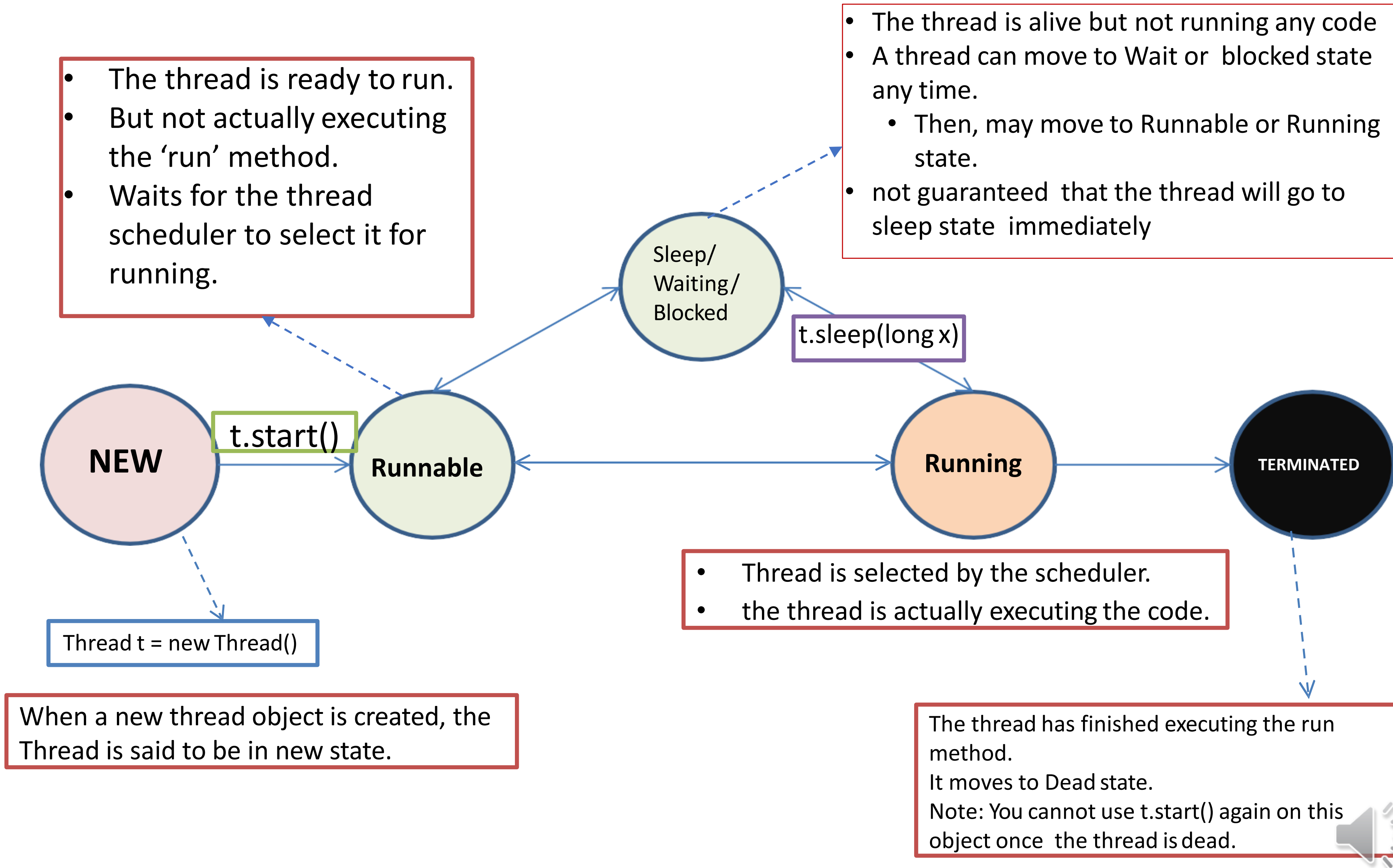
PGT307: Threads

Mohamed Elshaikh

Thread – States of a thread



Commonly Used Methods in threads

Sleep( )

Join( )

Commonly Used Methods in thread

Priorities

yield( )

Commonly Used Methods in threads

Sleep( )

tell the current thread **to pause** for certain time.

Sleep method accepts time in **milliseconds**

Can throw **InterruptedException**.

Specified sleep **time is not guaranteed**.

When complete, **moves to Runnable or Running** state.

- Example code:

```
try {
    Thread.sleep(1000*10); //Will sleep for Ten seconds
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Commonly Used Methods in threads

Priorities

Priority values:
- 1 (lowest priority)
- 10 (highest priority).

default thread priority:
- normal (value 5)

The thread with highest priority will be done first.
- But no guarantee that it will run immediately.

Current running thread has highest priority compared to threads that are waiting.

Thread scheduler decides which thread .

t.setPriority() - method set the priorities on a thread object.

The Priority should be set BEFORE the threads start() method is invoked.

Commonly Used Methods in threads

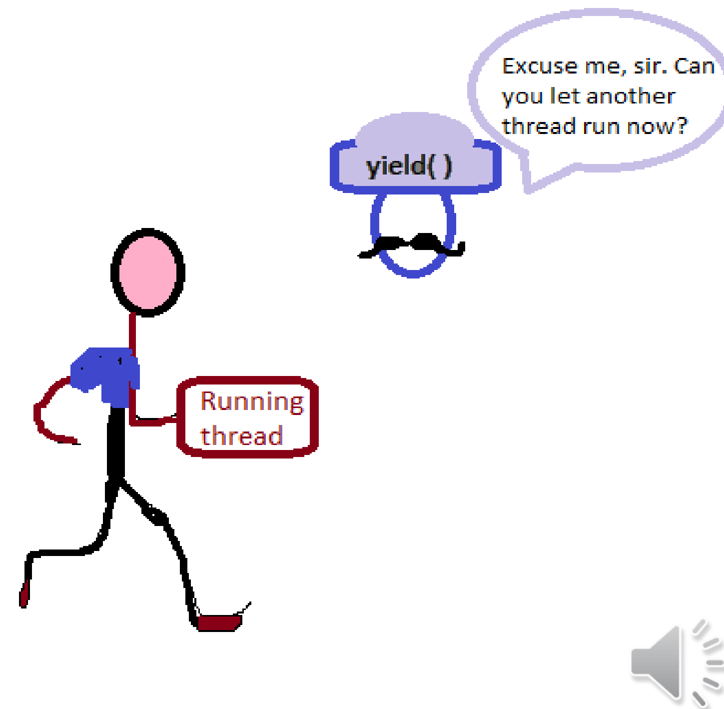
tells the *currently running thread* to give way to other threads with equal priority in the thread pool.

No guarantee that the currently executing thread will go to runnable state.

DOES NOT make the thread go to wait or blocking state.

- At the most, will change thread to move **from running to runnable state**.

yield( )



Commonly Used Methods in threads

Join( )

tells the currently running thread to **pause and wait until another thread** completes.

Guaranteed :

the currently executing thread will stop its execution

- will run after the thread (on which join is invoked) is complete.

Usually used if a thread needs to wait for a result from another thread.

Sample code:

```
double[ ] array = new double [200];
for (int n = 1; n < array.length; n++) {
      array[n] = Math.random(); }

AThread t = new AThread(array);
t.start();
try{
      t.join();
      System.out.println (“Minimum: “ + array[0];
      } catch (InterruptedException e) {
    }
```

The current thread pauses and let Athread runs until complete. (→ t.join();)

After Athread finishes, the main thread runs to print minimum value (→ System.out.println)

Commonly Used Methods in threads

Join( )

Thread Main

Sample code:

```
double[ ] array = new double [200];
for (int n = 1; n < array.length; n++) {
      array[n] = Math.random(); }

AThread t = new AThread(array);
t.start();
try{
      t.join();
      System.out.println (“Minimum: “ + array[0];
      } catch (InterruptedException e) {
    }
```

t.join();

Synchronization

- There can be instances when two or more threads access a common resource (data/file).
- In order to maintain consistency, it is imperative that the resource is made **available to only one thread at a time**.
- If multiple threads require an access to an object, synchronization helps in maintaining consistency.
- Synchronization is to prevent data corruption by allowing only one thread to perform an operation on an object at a time.
    - For simple synchronization Java provides the `synchronized` keyword

Synchronization

- Two ways to implement thread synchronization:

1. synchronized method
   - Using `synchronized` keyword with **method** definition
2. synchronized statements
   - Using `synchronized` keyword with any **block of code**

Example 1 – synchronizing methods

Syntax:

```
<modifier> synchronized <return type> methodName(){
      //method body
}
```

Example:

```
public class SynchronizedCounter implements Runnable {
    private int count = 0;
    public void run(){
        this.increment(5);
    }
    public synchronized void increment(int a) { //this is a method
        count += a;
        System.out.println("Current: " + count);
    }
}
```

The synchronized keyword on a method means that if this is already locked anywhere (on this method or elsewhere) by another thread, we need to wait till this is unlocked before entering the method

Example 2 – synchronizing blocks

Syntax:

```
synchronized(this){
        //statement for body block
}
```

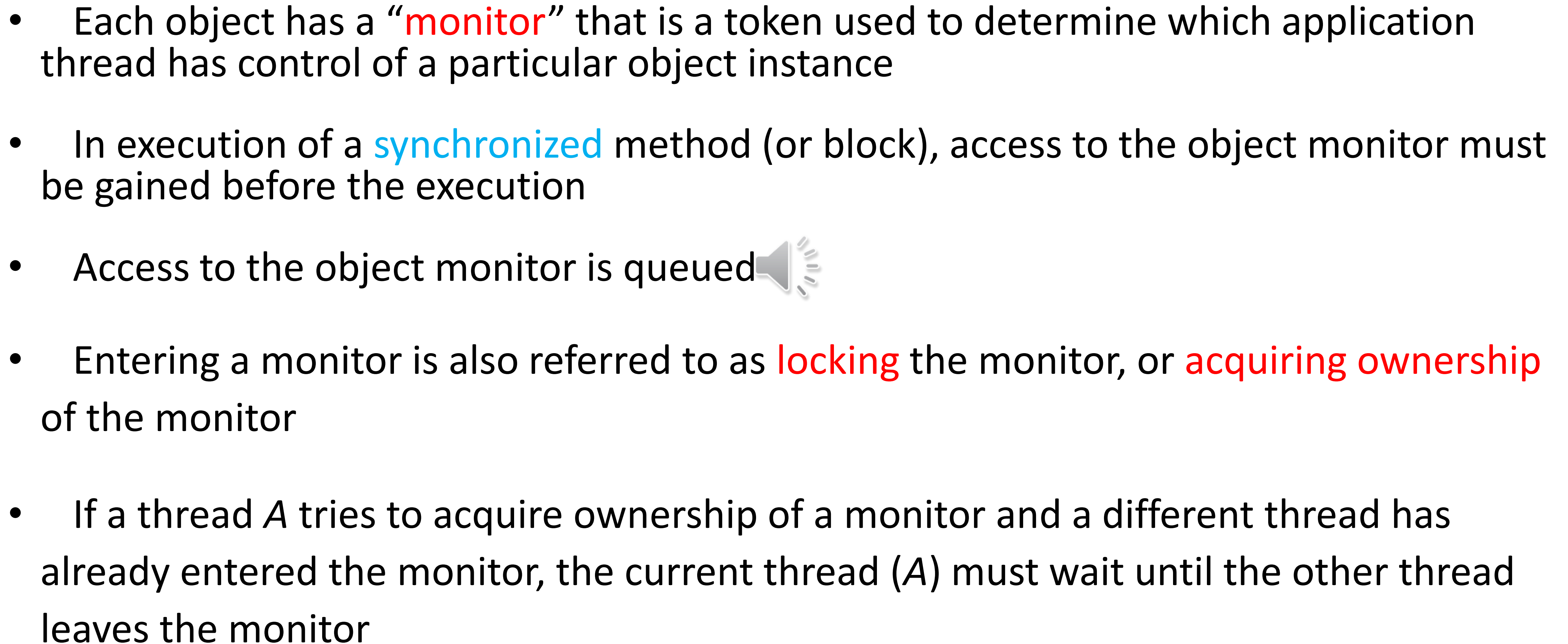
Example:

```
public void addName(String name) { //normal method
     synchronized(this) { //block of synchronized statements
         lastName = name;
         nameCount++;
     }
  nameList.add(name);
}
```

When synchronizing a block, key for the locking should be supplied (usually would be `this`)

The advantage of not synchronizing the entire method is **efficiency.**

Monitors



- Each object has a "monitor" that is a token used to determine which application thread has control of a particular object instance
- In execution of a synchronized method (or block), access to the object monitor must be gained before the execution
- Access to the object monitor is queued
- Entering a monitor is also referred to as locking the monitor, or acquiring ownership of the monitor
- If a thread *A* tries to acquire ownership of a monitor and a different thread has already entered the monitor, the current thread (*A*) must wait until the other thread leaves the monitor

Deadlock

- Synchronization can lead to another possible problem: deadlock.
- Deadlock occurs when two threads need exclusive access to the same set of resources and each thread holds the lock on a different subset of those resources.
- If neither thread is willing to give up the resources it has, both threads come to an indefinite halt.

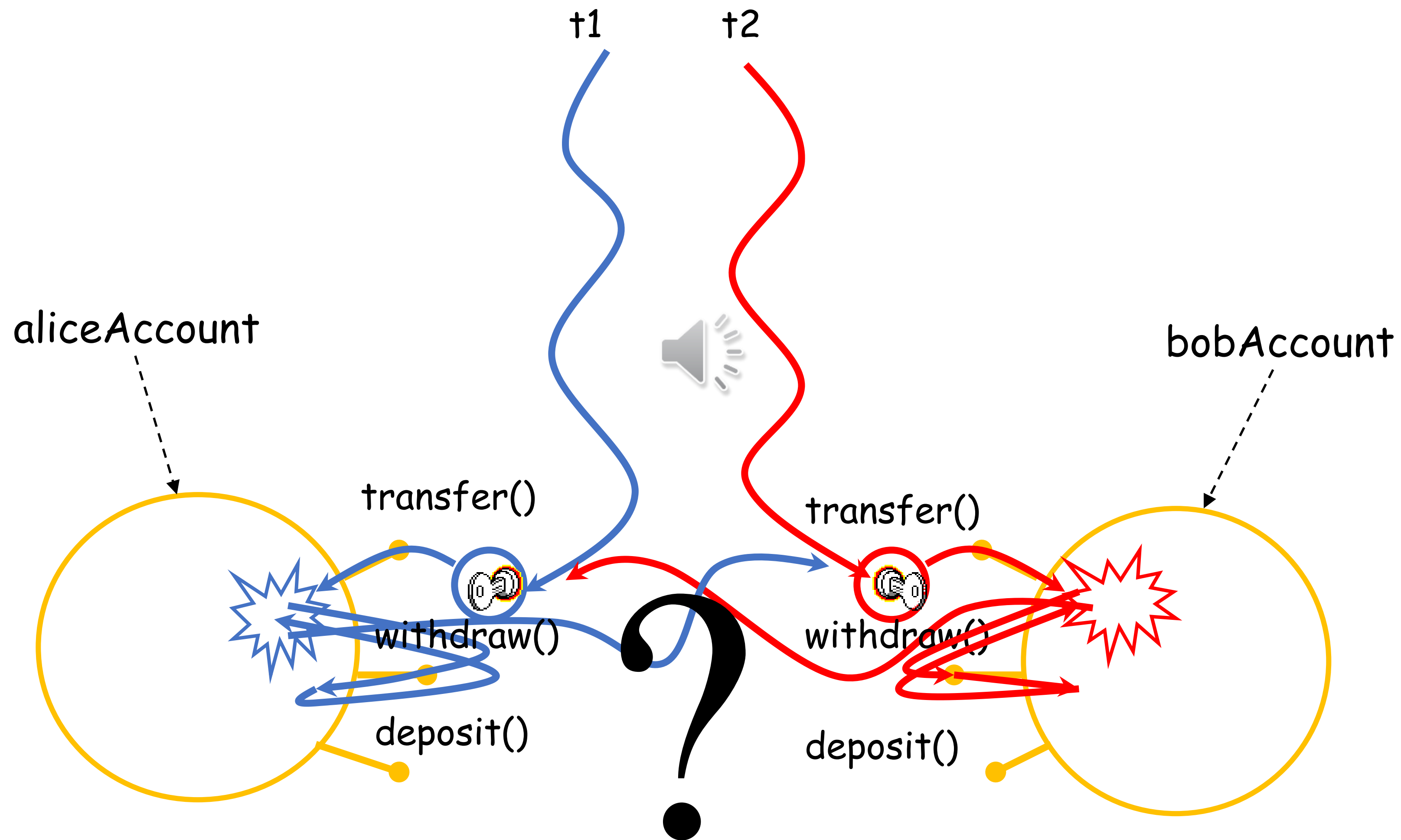
Deadlock Example

```java
public class BankAccount {
  //data members
  private int accountNumber;
  private float balance;

  public synchronized void deposit(float amount) {
        balance += amount;
    }

    public synchronized void withdraw(float amount) {
        balance -= amount;
    }

  public synchronized void transfer(float amount, BankAccount target) {
        withdraw(amount);
        target.deposit(amount);
    }
}
```

```java
public class MoneyTransfer implements Runnable {

    private BankAccount from, to;
    private float amount;

    public MoneyTransfer(BankAccount from, BankAccount to,
                                            float amount)  { //constructor
        this.from = from;
        this.to = to;
        this.amount = amount;
    }

    @Override
    public void run() {
        from.transfer(amount,to);
    }
}
```

```
BankAccount aliceAccount = new BankAccount();
BankAccount bobAccount = new BankAccount();
...

// At one place
Runnable transaction1 =
    new MoneyTransfer(aliceAccount, bobAccount, 1200);
Thread t1 = new Thread(transaction1);
t1.start();

// At another place
Runnable transaction2 =
    new MoneyTransfer(bobAccount, aliceAccount, 700);
Thread t2 = new Thread(transaction2);
t2.start();
```

Deadlocks Situation



Preventing Deadlock

- The most important technique for preventing deadlock is to **avoid unnecessary synchronization**.
- If there's an alternative approach for ensuring thread safety, such as making objects immutable or keeping a local copy of an object, use it.
- Synchronization should be a last resort for ensuring thread safety.
- If you do need to synchronize, keep the synchronized blocks small and try not to synchronize on more than one object at a time.

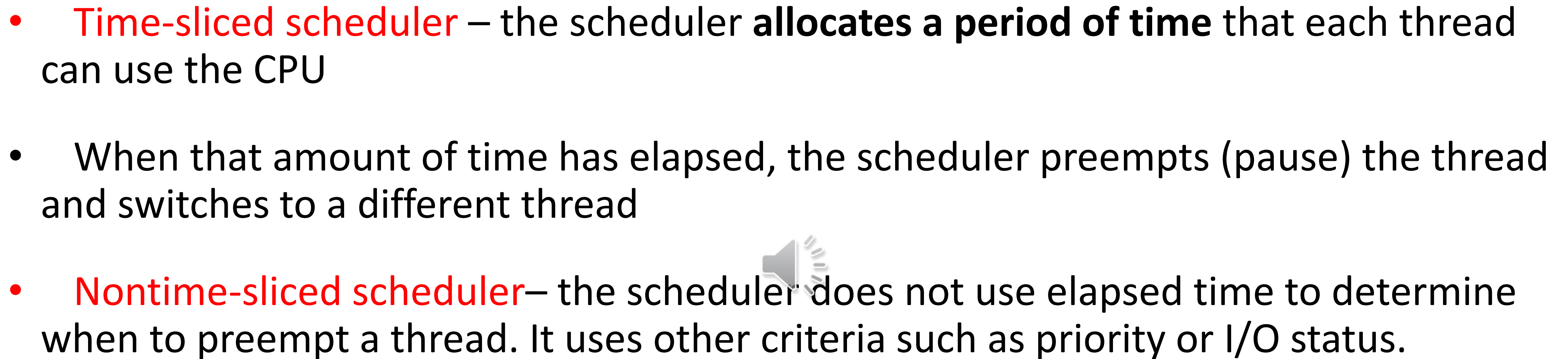
Scheduling

- Thread scheduling is the mechanism used to determine how runnable threads are allocated CPU time
- A thread-scheduling mechanism is either preemptive or cooperative
- Preemptive scheduling – the thread scheduler preempts (pauses) a running thread to allow different threads to execute
- Cooperative scheduling – the scheduler never interrupts a running thread
- A cooperative thread scheduler waits for the running thread to pause itself before handing off control of the CPU to a different thread.

Scheduling – Starvation

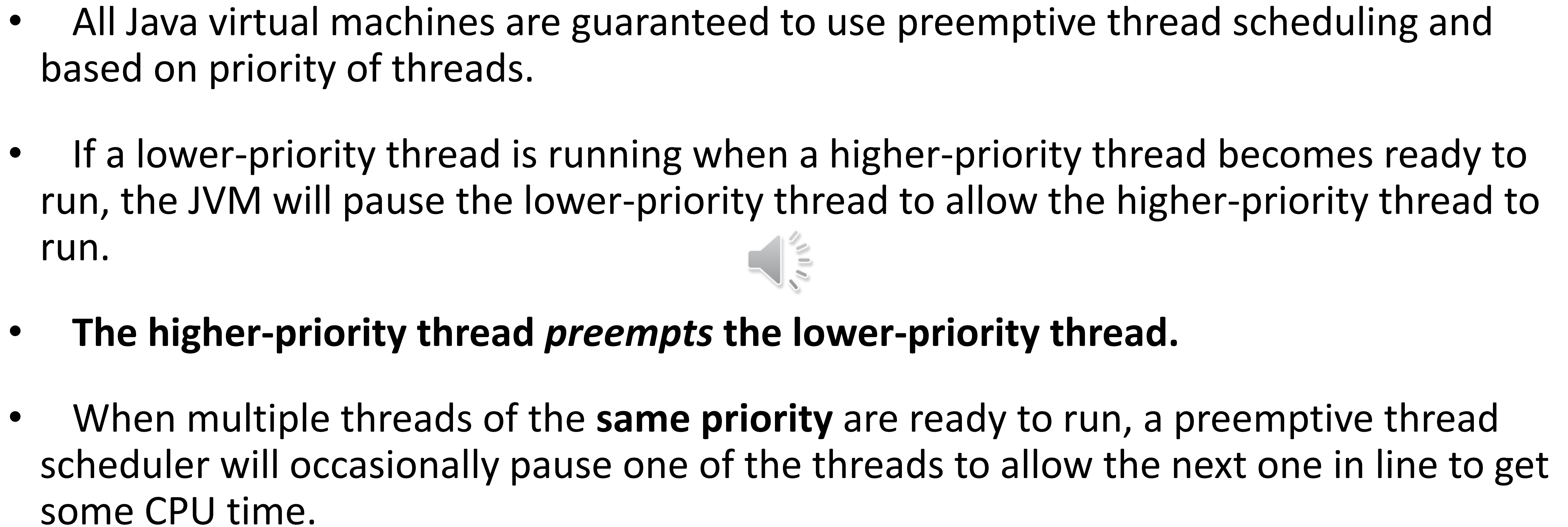
- A cooperative scheduler may cause starvation (runnable threads, ready to be executed, wait to be executed in the CPU a lot of time, maybe even forever)
- Sometimes, starvation is also called a livelock

Time-Sliced Scheduling



- Time-sliced scheduler – the scheduler **allocates a period of time** that each thread can use the CPU
- When that amount of time has elapsed, the scheduler preempts (pause) the thread and switches to a different thread
- Nontime-sliced scheduler– the scheduler does not use elapsed time to determine when to preempt a thread. It uses other criteria such as priority or I/O status.

Java Scheduling



Waiting on an object

- The `wait()` method is part of the `java.lang.Object` interface
- Allows two threads to cooperate based on a single shared lock object
- It requires a lock on the object's monitor to execute
- It must be called from a synchronized method, or from a synchronized segment of code.
- wait() causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object
- Upon call for wait(), the thread **releases ownership of this monitor and waits until another thread notifies the waiting threads of the object**

The wait( ) method

- `wait()` is also similar to `yield()`
- Both take the current thread off the execution stack and force it to be rescheduled
- However, wait() is not automatically put back into the scheduler queue
- `notify()` must be called in order to get a thread back into the scheduler's queue

Example – wait() and notify()

```java
synchronized(lock) {
    while (!resourceAvailable()) {
        lock.wait();
    }
    consumeResource();
}
```

Wait until resource available

```java
produceResource();
synchronized(lock) {
    lock.notifyAll();
}
```

Now available, wake up all the threads waiting on the object

*notify method wakes up only one thread waiting on the object and that thread starts execution.
*So if there are multiple threads waiting for an object, this method will wake up only one of them.

Thread Pools and Executors

- Adding multiple threads to a program dramatically improves performance, especially for I/O-bound programs such as most network programs.
- However, threads are prone to cause **overhead**.
- Starting a thread and cleaning up after a thread that has died takes a noticeable amount of work from the virtual machine, especially if a program spawns hundreds of threads → this can **overload the garbage collector** or other parts of the VM and **hurt performance**
- Switching between running threads also carries overhead.

Thread Pools and Executors

- The **Executors** class in **`java.util.concurrent`** makes it quite easy to set up thread pools.

A **thread pool** is a group of **pre-instantiated, idle threads** which stand ready to be given work (instead of instantiating new **threads** for each task).

Preferred when **there is a large number of short tasks** to be done rather than a small number of long ones.

- Simply submit each task as a **`Runnable`** object to the pool.
- The method return a **`Future`** object representing the future state of the task. If you submitted a `Runnable`, the `Future` object return null once the task finished.

Executors Example

- ➔ Let say you want to gzip every file in the current directory using a `java.util.zip.GZIPOutputStream`
- ➔ This is a filter stream that compresses all the data it writes.
- ➔ This is an I/O-heavy operation because all the files have to be read and written.
- ➔ But, data compression is a very CPU-intensive operation, so you don't want too many threads running at once.
- ➔ Use thread pool → Each **client thread** will compress files while the **main program** will determine which files to compress

Executors Example – GZipRunnable class

```
import java.io.*;
import java.util.zip.*;

public class GZipRunnable implements Runnable {
  private final File input;

  public GZipRunnable(File input) {
  this.input = input;
  }

  @Override
  public void run() { // don't compress an already compressed file
  if (!input.getName().endsWith(".gz")) {
    File output = new File(input.getParent(), input.getName() + ".gz");
      if (!output.exists()) { // Don't overwrite an existing file
      try {
        InputStream in = new BufferedInputStream(new FileInputStream(input));
        OutputStream out = new BufferedOutputStream(
                          new GZIPOutputStream(
                          new FileOutputStream(output)));)
        int b;
        while ((b = in.read()) != -1) out.write(b);
        out.flush();
      } catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }
} }
```

Executors Example – main

```
import java.io.*;
import java.util.concurrent.*;
public class GZipAllFiles {
  public final static int THREAD_COUNT = 4;

  public static void main(String[] args) {
     ExecutorService pool = Executors.newFixedThreadPool(THREAD_COUNT);
   for (String filename : args) {
     File f = new File(filename);
     if (f.exists()) {
       if (f.isDirectory()) {
         File[] files = f.listFiles();
         for (int i = 0; i < files.length; i++) {
           if (!files[i].isDirectory()) { // don't recurse directories
             Runnable task = new GZipRunnable(files[i]);
             pool.submit(task);
           }
         }
       }else{
         Runnable task = new GZipRunnable(f);
         pool.submit(task);
       }
     }
   }
     pool.shutdown();
   }
}
```

constructs the pool with a fixed thread count of 4

iterates through all the files and directories listed on the command line

files in those directories is used to construct a GZipRunnable

submit to the pool for eventual processing by one of the 4 threads

notifies the pool that no further tasks will be added to its internal queue

THANK YOU