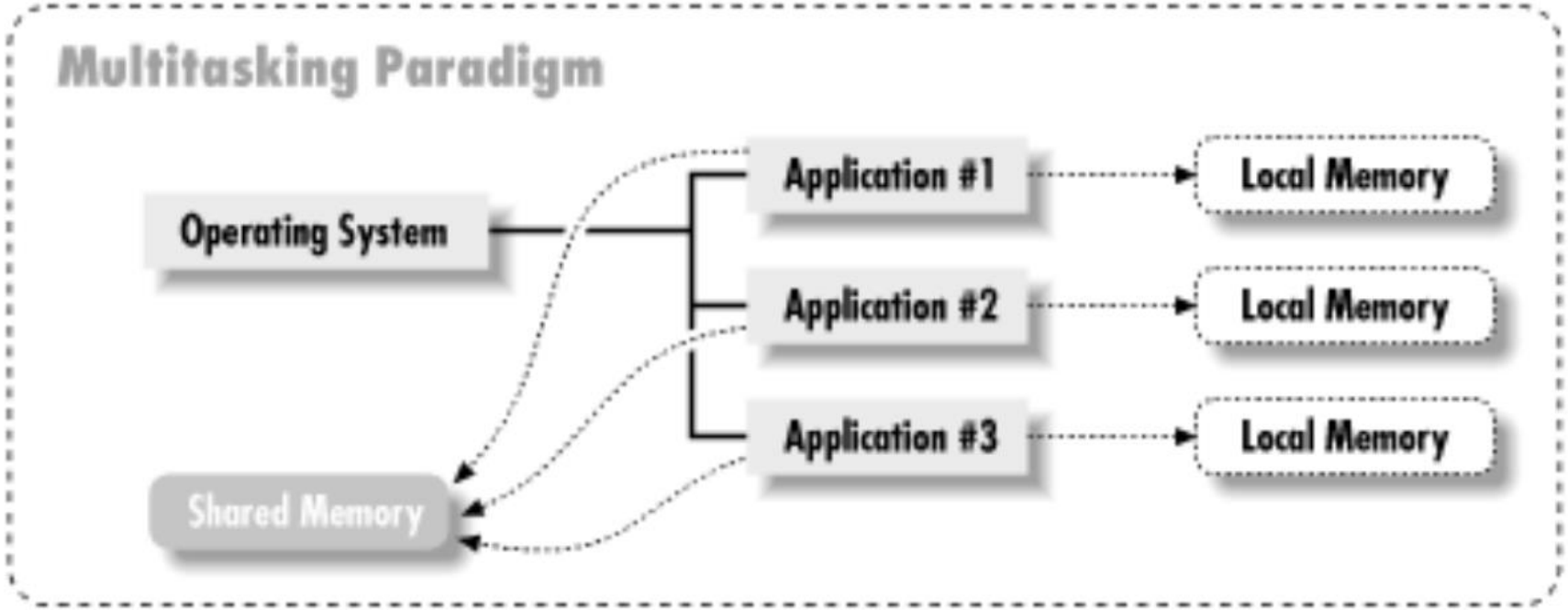

# Java Threading & Concurrency

Оперативни системи 2014
Аудиториски вежби

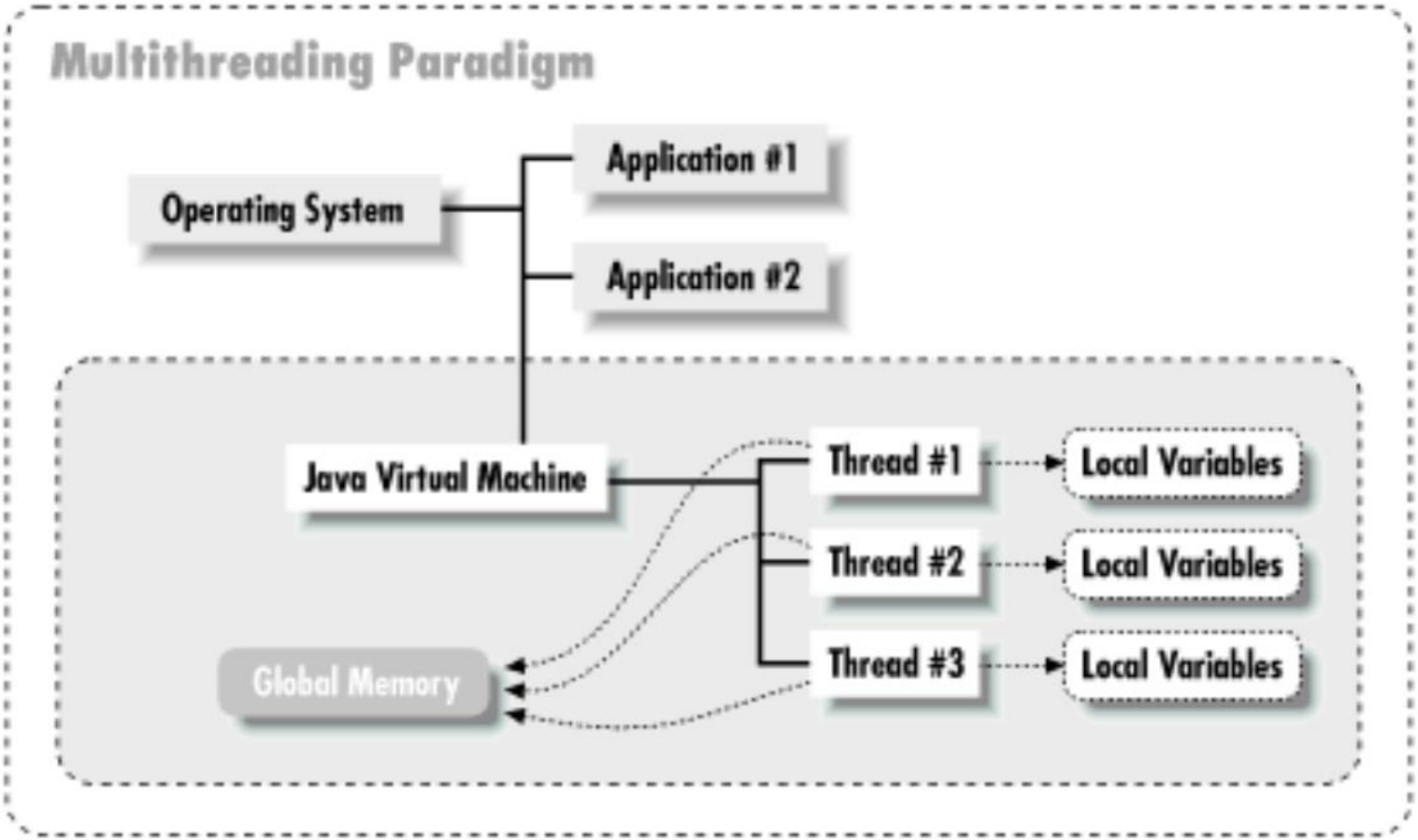
# Concurrency

- Concurrency means simultaneous execution of several tasks.
- Context:
  - Multiple applications;
  - Multiple processes within application;
  - Multiple threads within process;
- Main issues:
  - Mutual exclusion;
  - (Condition) Synchronization;
  - Deadlock;

# Concurrency: Multitasking Paradigm



# Concurrency: Multithreading Paradigm



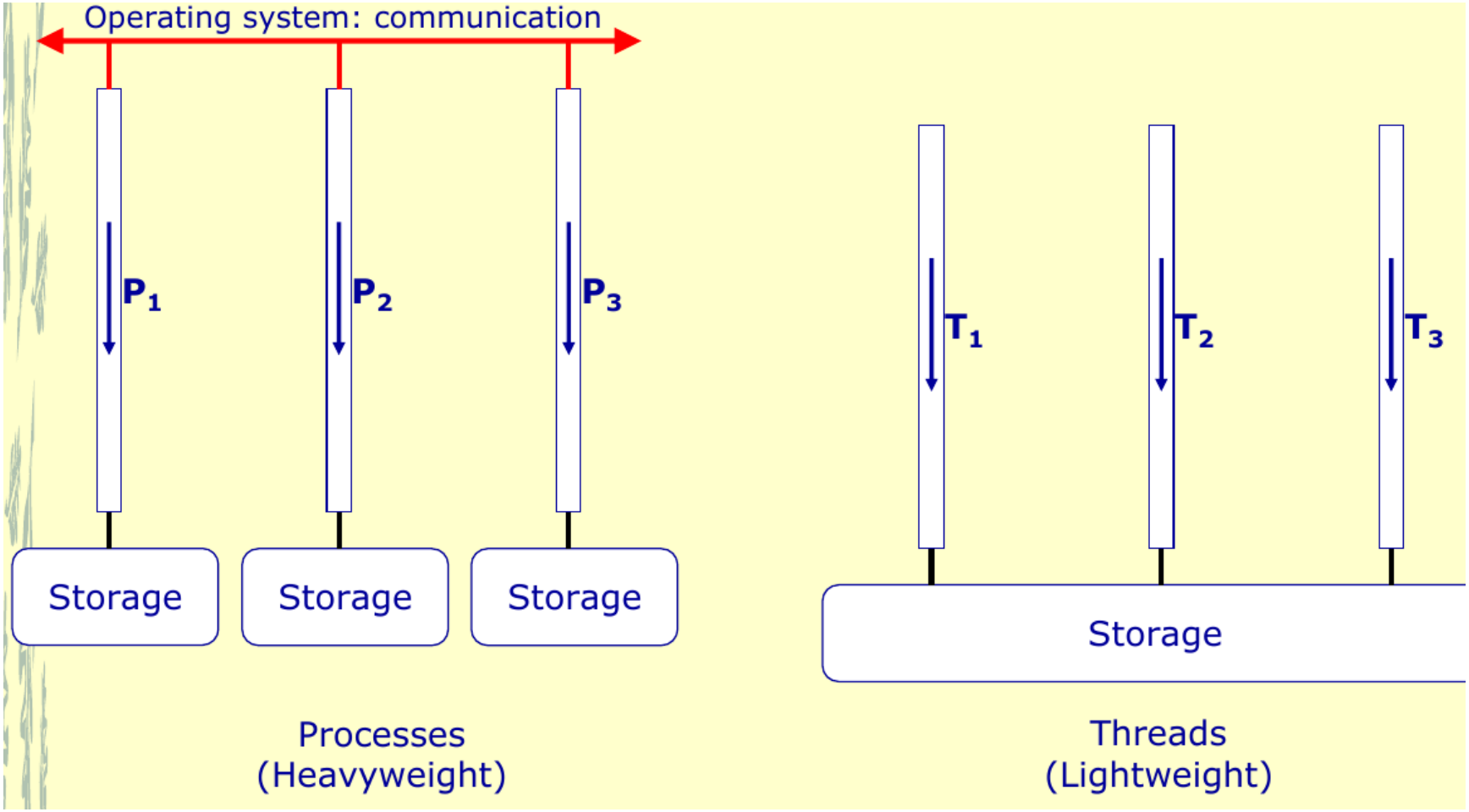
# Main Issues

- Mutual exclusion
  - Exclusive access to non-shareable, but shared resources.
    - typical example: printer;
  - “Locks” associated with resources.
  - Usually short duration of an operation.
- Condition synchronization
  - Usually related to “consumable” resources.
    - typical example: the producer-consumer example;
  - May imply a long (and repeated) wait.
  - Requires structures for management of waiting tasks.

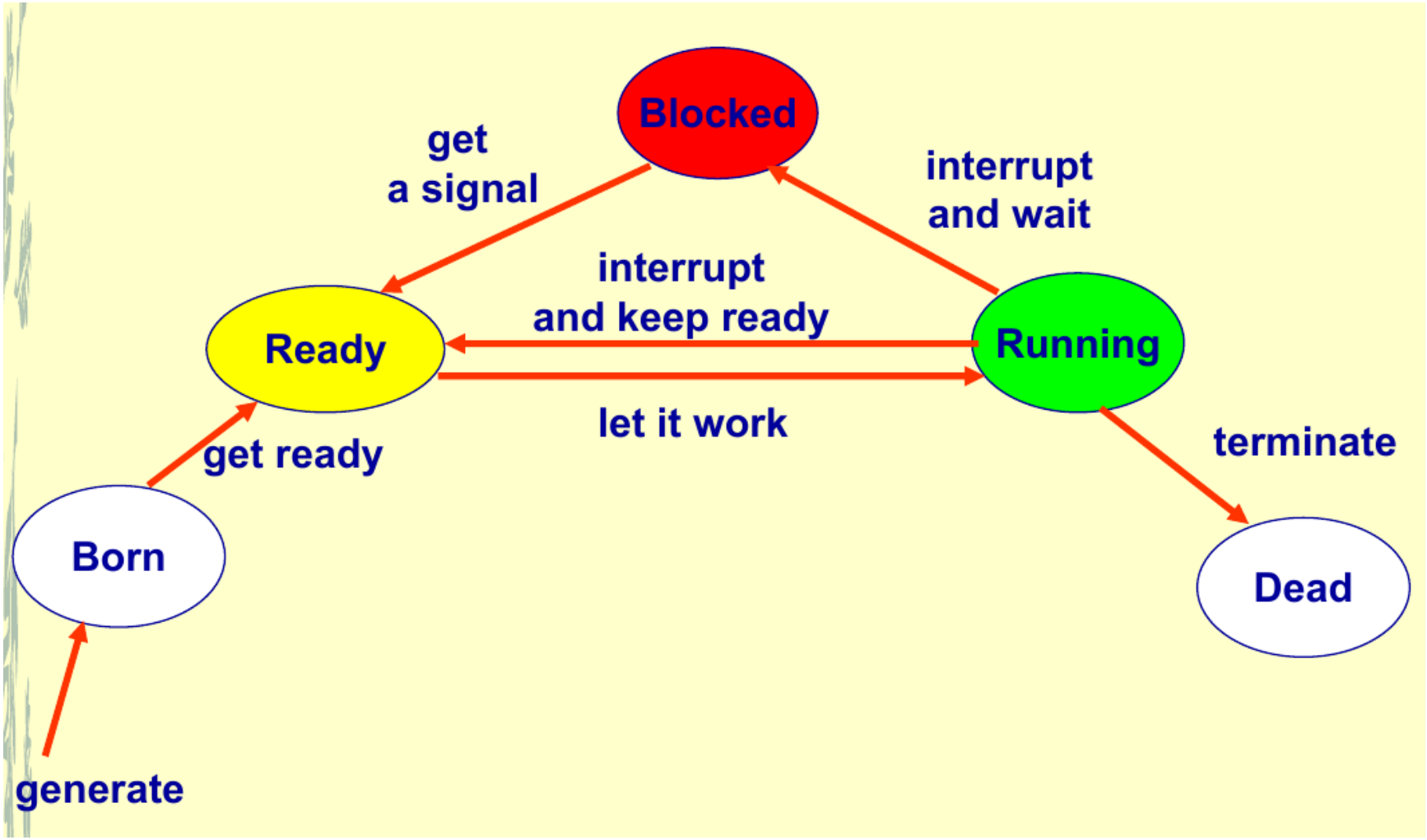
# Deadlock

- A set of processes is deadlock if each process in the set is waiting for an event, that only another process in the set can cause.
- Because all the processes are waiting, none of them will ever cause any of the events that could wake up any other member of the set, and all the processes continue to wait forever.
- Practical example: the dining philosophers table.

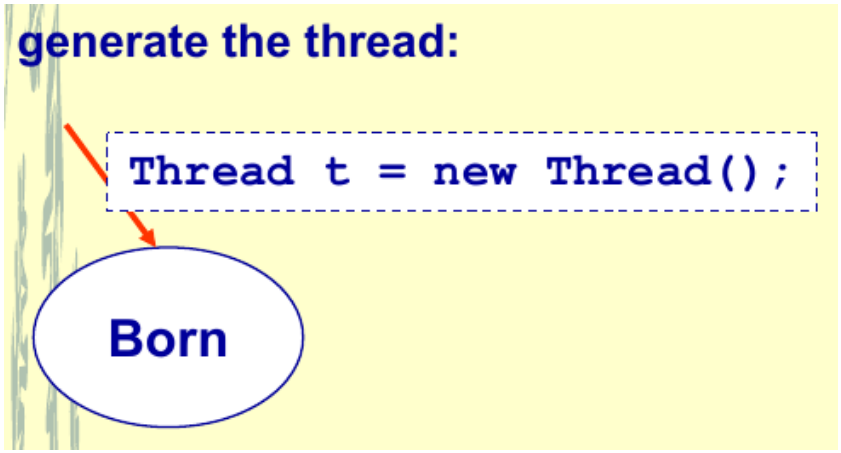
# Process vs. Thread

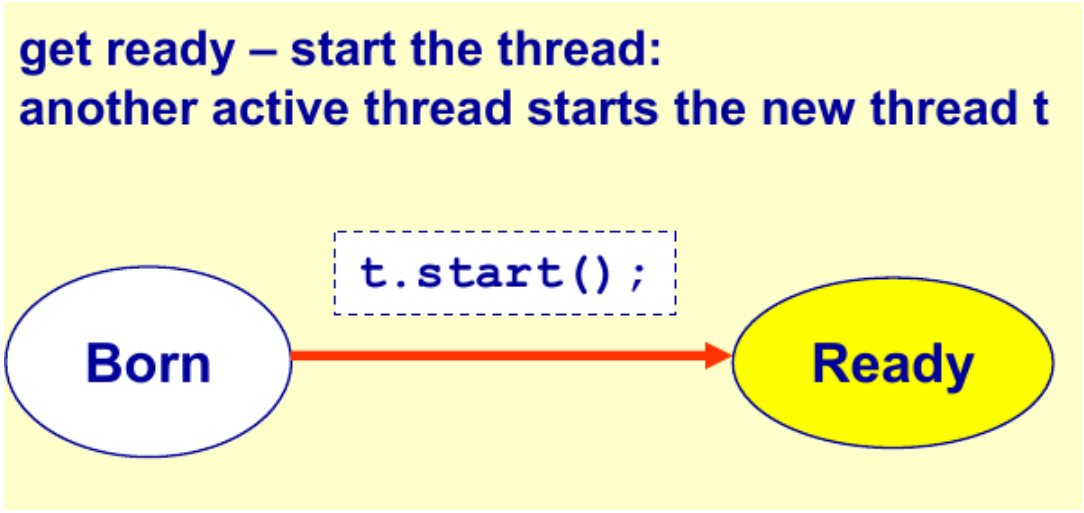


# Thread Lifecycle

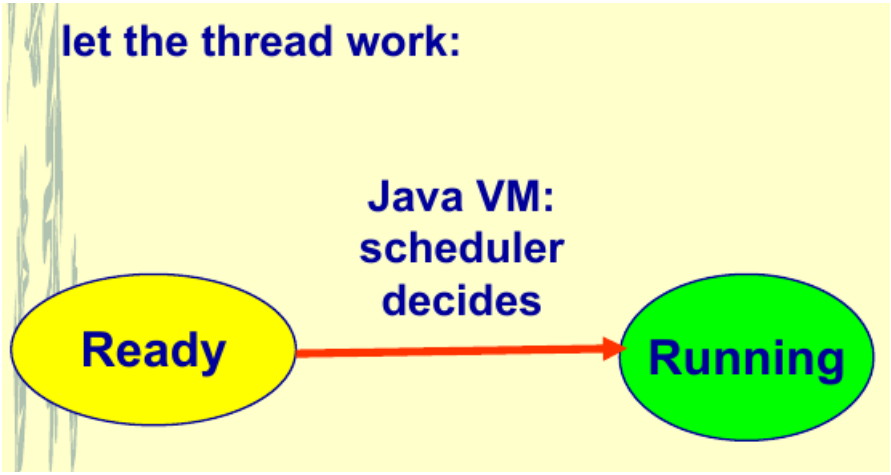


# Thread Lifecycle in Detail



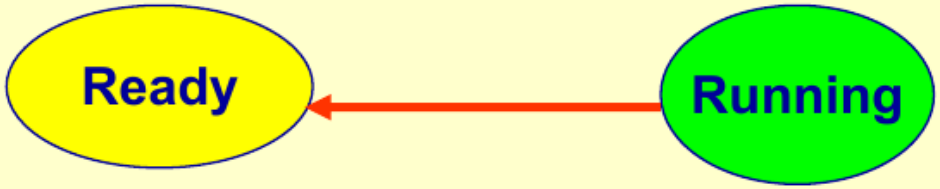



# Thread Lifecycle in Detail




interrupt the thread
and keep it ready:

- Time slice ends
- Process with higher priority appears
- Process releases the processor: t.yield()



# Thread Lifecycle in Detail

**Interrupt and wait:**

- Process goes to sleep: t.sleep()
- Process waits for the end of another thread u: u.join()
- Wait for the unlock of another object o: o.wait()


**Get a signal and get ready:**

- Time interval of sleep ends
- Another thread wakens t: t.interrupt()
- A thread whose end was waited for by the blocked thread „dies“
- Object o becomes unlocked

# Java Threads Model

- Multiple threads within a process (JVM).
- Exclusion synchronization built into the language (each object has a lock).
- Condition synchronization based on monitors.
- JVMs may exploit multiple processors.
- There is a default thread in each Java program which runs the ‘main’ of the program.
- New threads can be created and started.
- Thread is an object itself in Java.
- Hierarchies of thread classes are possible.

# Java Threads: Thread Class

- Java provides the abstract class java.lang.Thread whose method run() is intended to contain the thread's main logic.
- A class T can be derived from Thread and override the run() method. Instances of T are threads and can be started by calling the start() method of class Thread.
- The start() method calls run() method – you **should not** run the method run() by yourself.
- A thread terminates when its run() method terminates.

```
package ex1;

public class Main {

  public static void main(String[] args) {
    T obj = new T();
    obj.start();
    //obj runs in parallel
    //with the main thread
  }

}

class T extends Thread {
  public void run() {
    // thread's main logic
  }
}
```

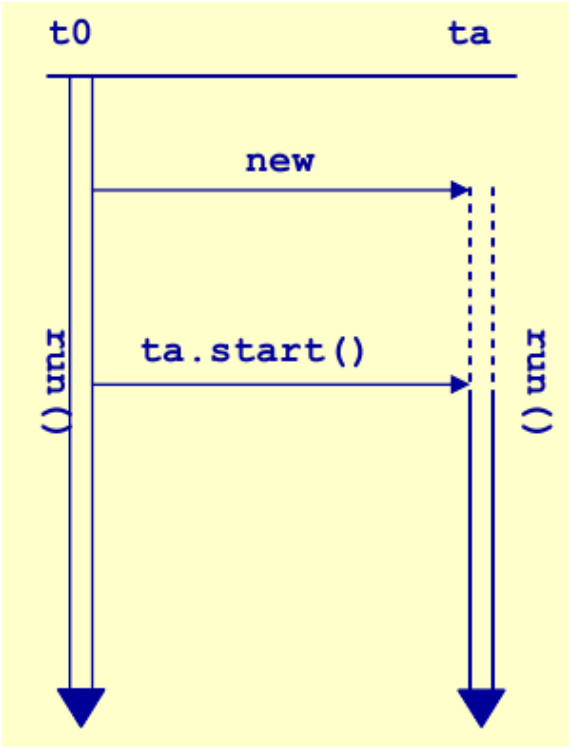
# Java Threads: Runnable Interface

- Due to single inheritance, a new mechanism is required when T must inherit another class.
- The interface Runnable contains the run() method – Thread actually implements Runnable.
- A class T2 can then inherit from another class and implement Runnable.
- An instance of T2 is given as argument to a constructor of Thread, to create a Thread object.

```
public class Main2 {

public static void main(String[] args) {
  Runnable obj = new T2();
  Thread tobj = new Thread(obj);
  tobj.start();
    // tobj runs in parallel
    // with the main thread
  }
}

class Base {}

class T2 extends Base implements
Runnable {
  public void run() {
    // thread's main logic
  }
}
```

# General Java Thread Lifecycle

- Generate: a running thread (`t0`) generates a new thread:
  - `ta = new ThreadA1();`
- Start: the running thread starts the new one:
  - `ta.start();`
- Work: the JVM scheduler starts run().
- Die: run() method ends its work.



# Thread Execution Example

```
public class ThreadBasicTest {
    public static void main(String[] args) {
        Thread ta = new ThreadA1();
        Thread tb = new ThreadB1();
        ta.start();
        tb.start();
        System.out.println("Main done");
    }
}

class ThreadA1 extends Thread {
    public void run() {
        for (int i = 1; i <= 20; i++) {
            System.out.println("A: " + i);
        }
        System.out.println("A done");
    }
}

class ThreadB1 extends Thread {
    public void run() {
        for (int i = -1; i >= -20; i--) {
            System.out.println("\t\tB: " + i);
        }
        System.out.println("B done");
    }
}
```

```
Main done
A: 1
A: 2
A: 3
A: 4
A: 5
A: 6
A: 7
A: 8
        B: -1
        B: -2
        …
        B: -14
        B: -15
        B: -16
        B: -17
        B: -18
        B: -19
        B: -20
        B done
A: 9
A: 10
…
A: 17
A: 18
A: 19
A: 20
A done
```

```
Main done
        B: -1
A: 1
A: 2
        B: -2
        B: -3
        B: -4
A: 3
        B: -5
… (наизменично)
        B: -14
A: 13
        B: -15
A: 14
        B: -16
A: 15
        B: -17
A: 16
        B: -18
A: 17
        B: -19
A: 18
        B: -20
A: 19
        B done
A: 20
A done
```

# Ending Thread Execution

- An alive thread continues to be alive until:
  - run() returns normally;
  - run() returns abruptly;
  - destroy() is invoked on thread;
  - program terminates;
- When a thread's run() terminates, the thread does not hold locks anymore.
- destroy() is drastic, does not release locks and some JVMs do not implement it.
- Interruption is advisory.

# Interruption Threads

- A thread can interrupt another thread with the interrupt() method. The interrupted thread uses interrupted() to test and clear the interrupted state.
- Example:
  - Thread 1 has:

```
thread2.interrupt();
```

  - Thread 2 has:

```
while (!interrupted()) {
   // normal execution
}
```

  - Thread 2 is cancellable.

# Threads Working Together

- In a multithreaded program, we almost never know which thread will finish first (or when).
- Waiting for a thread to complete is done with the join() method, from the class Thread.

```
public class UseJoin {
    public static void main(String[] args) {
        Count c = new Count();
        c.start();
        try {
            c.join();
            //what would happen without this line?
            System.out.println(
                    "Result = " + c.getResult());
        } catch (InterruptedException e) {}
    }
}

class Count extends Thread {
    private long result;
    public void run() {
        result = count();
    }
    public long getResult() {
        return result;
    }
    public long count() {
        long r = 0;
        for (r = 0; r < 1000000; r++);
        return r;
    }
}
```

# Risks of Threads

- Safety Hazards (Correctness)
  - In the absence of sufficient synchronization, the ordering of operations in multiple threads is unpredictable and sometimes surprising.
  - Illustrates a common concurrency hazard called a race condition.
- Liveness Hazards (Deadlock)
  - A liveness failure occurs when an activity gets into a state such that it is permanently unable to make forward progress.
    - infinite loop;
  - If thread A is waiting for a resource that thread B holds exclusively, and B never releases it, A will wait forever.

# Risks of Threads

- Performance Hazards
  - Poor service time, responsiveness, throughput, resource consumption, or scalability
  - Degree of runtime overhead
    - Context switches
      - Saving and restoring execution context, loss of locality, and CPU time spent scheduling threads instead of running them
    - When threads share data
      - Synchronization mechanisms that can inhibit compiler optimizations, flush or invalidate memory caches, and create synchronization traffic on the shared memory bus

# Atomicity

- Execution of operation is atomic if either all of the operations occur or none of them occur.
- Atomicity ensures serializability.
  - A concurrent execution is serializable if the execution is guaranteed to correspond to some serial execution of those threads;
  - Ensures predictable result of the operations;

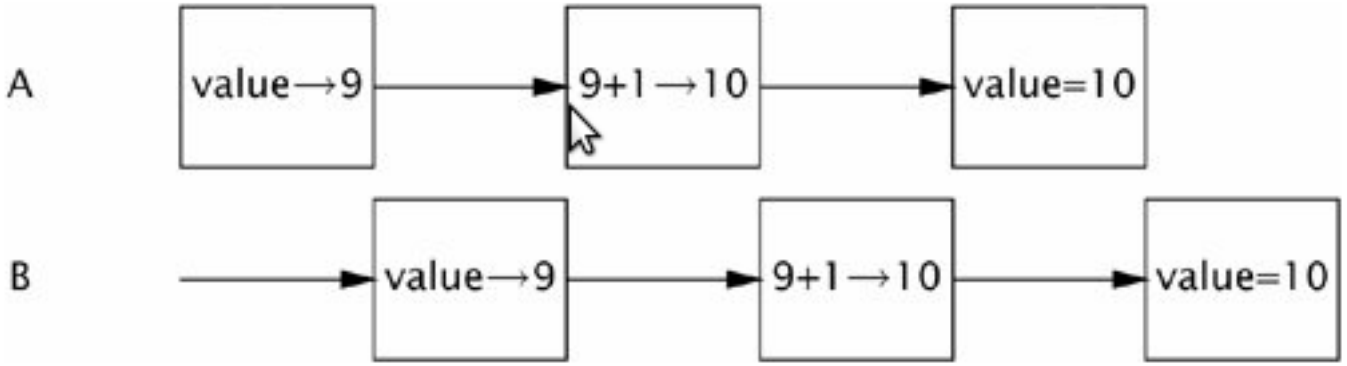
# Atomicity

- Example: var ++
  - Operations:
    - Read the value of the var variable from its memory location into the processors registers;
    - Increment the value;
    - Write the incremented value into the memory location of the variable;
  - More than one processor cycle!
    - The process / thread can be switched after each of the previous operations;
    - Therefore, the ++ operation is not atomic, i.e. it has an unpredictable result in a multithreaded environment;

# Race Condition

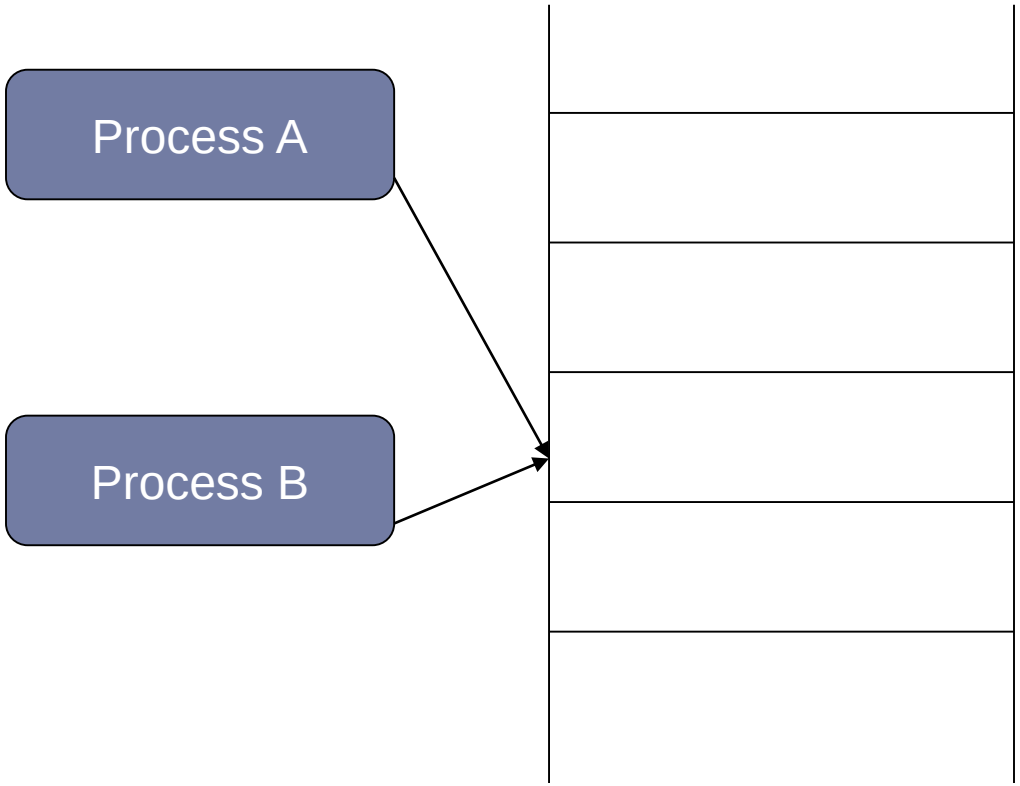
- A race condition occurs when more than one thread is performing a series of actions on shared resources and several possible outcomes can exist based on the order of the actions from each thread are performed.

```
@NotThreadSafe
public class UnsafeSequence {
    private int value;

    /** Returns a unique value. */
    public int getNext() {
        return value++;
    }
}
```



# Race Condition

- Example: Printer Spooler



# Critical Regions and Mutual Exclusion

- A critical region is a part of the code from a process / thread which accesses shared variables, shared files, or other shared memory objects.
- With multiple processes / threads running in the system, we need to provide access to the critical region to only one of the processes / threads, at any given time.
- We call this mutual exclusion.

# Mutual Exclusion: Race Condition Solution

- Mutex
  - Provides atomicity of one or more instructions;
  - In Java, the java.util.concurrent.locks.Lock implementations act as a mutex by the use of the following methods:
    - lock()
      - Acquires the lock.
      - If the lock is not available, then the current thread becomes disabled for thread scheduling purposes, and lies dormant until the lock has been acquired.
    - unlock()
      - Releases the lock.

# Mutual Exclusion: Monitor

- The use of mutex is error prone, and relies on the programmer discipline.
- Monitor
  - Every object in Java contains a ‘monitor’
    - used to provide mutual exclusion access to critical sections of code;
  - Synchronized
    - marking a method or code block as `synchronized`;
  - Only one thread at a time is allowed to execute any critical section of code for a particular monitor.

# Synchronized Example

```
@ThreadSafe
public class SafeSequence {
    private int value;

    /** Returns a unique value. */
    public synchronized int getNext() {
        return value++;
    }
}
```

```
@ThreadSafe
public class SafeSequence {
    private int value;

    /** Returns a unique value. */
    public int getNext() {
            synchronized(this) {
                return value++;
            }
    }
}
```

# Synchronized Use-Case

```
SafeSequence a=new SafeSequence();
SafeSequence b=new SafeSequence();
```

- Case 1: Thread 1 calls `a.getNext()` and Thread 2 calls `b.getNext()`
  - The both threads use different instances, which are synchronized with different monitors;
  - There is no critical region in this case, and the execution of the threads will be simultaneous (parallel execution);
- Case 2: Thread 1 and Thread 2 both call `a.getNext()`
  - The both threads are waiting on a method synchronized by a same monitor (the instance a), so the methods will be invoked one after the other;
  - There will be mutual exclusion in the critical region;
  - However, the ordering of the execution can't be predicted;

# Static Synchronized Example

```
@ThreadSafe
public class SafeSequence {
    private static int value;

    /** Returns a unique value. */
    public static synchronized int getNext() {
        return value++;
    }
}
```

```
@ThreadSafe
public class SafeSequence {
    private static int value;

    /** Returns a unique value. */
    public static int getNext() {
            synchronized(SafeSequence.class) {
                return value++;
            }
    }
}
```

# Static Synchronized Use-Case

```
SafeSequence a=new SafeSequence();
SafeSequence b=new SafeSequence():
```

- Case 1: Thread 1 calls `a.getNext()` and Thread 2 calls `b.getNext()`
  - The method `getNext()` is static, which means that it is same for all instances, i.e. all instances will delegate to the call:

    `SafeSequence.getNext()`
  - The both threads are waiting on a method synchronized by a same monitor (the `SafeSequence.class` monitor), so the methods will be invoked one after the other in the separate threads;
  - There will be mutual exclusion in the critical region;
  - However, the ordering of the execution can't be predicted;

# Synchronization

- Dependencies among the operations in different threads
  - One thread should wait for the result of the other thread;
- Semaphores
  - The java `java.util.concurrent.Semaphore` class.
  - Conceptually, a semaphore maintains a set of permits.
  - Each Semaphore.acquire() call blocks if necessary, until a permit is available, and then takes it.
  - Each Semaphore.release() call adds a permit, potentially releasing a blocking acquirer.
  - Often used to restrict the number of threads than can access some (physical or logical) resource.
    - Therefore, a mutex is sometimes referred to as a binary semaphore.

# Semaphore Example (not complete)

- Semaphore initialization

```
Semaphore empty = new Semaphore(1);
```

- Thread 1 – Producer

```
empty.acquire();
putItems(buffer);
```

- Thread 2 – Consumer

```
if(noMoreItems()) {
  empty.release();
}
Item = getItem(buffer);
```

# Deadlock

- Mutual exclusion
  - Resource can be assigned to at most one thread;
- Hold and wait
  - Threads both hold some resource and request other resource;
- Circular wait
  - A cycle exists in which each thread waits for a resource that is assigned to another thread;

# Прашања?