Multi-agent systems

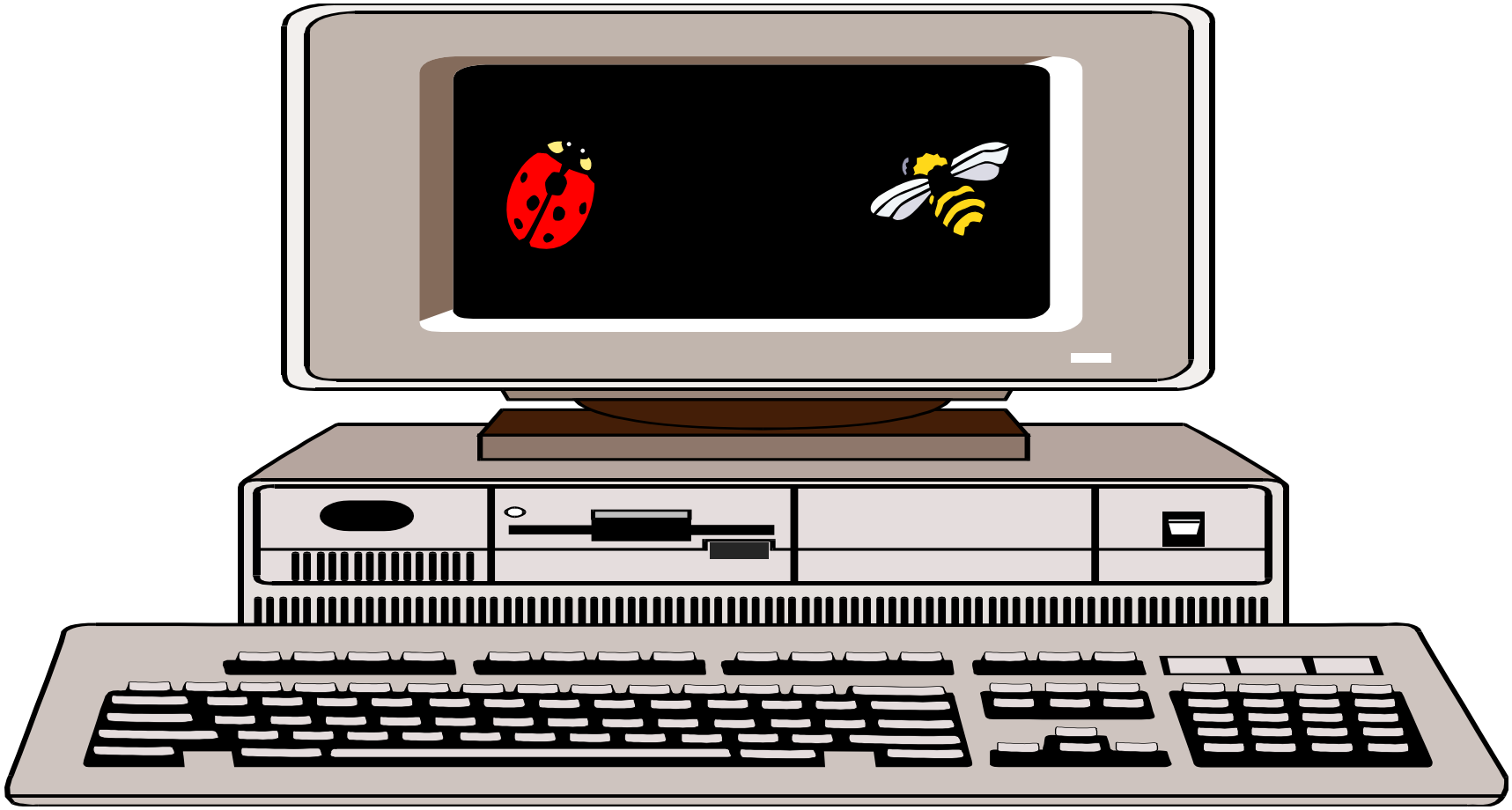
Andrej Lúčny

KAI FMFI UK

lucny@fmph.uniba.sk

http://www.agentspace.org/mas

MAS in VM



Virtual machine

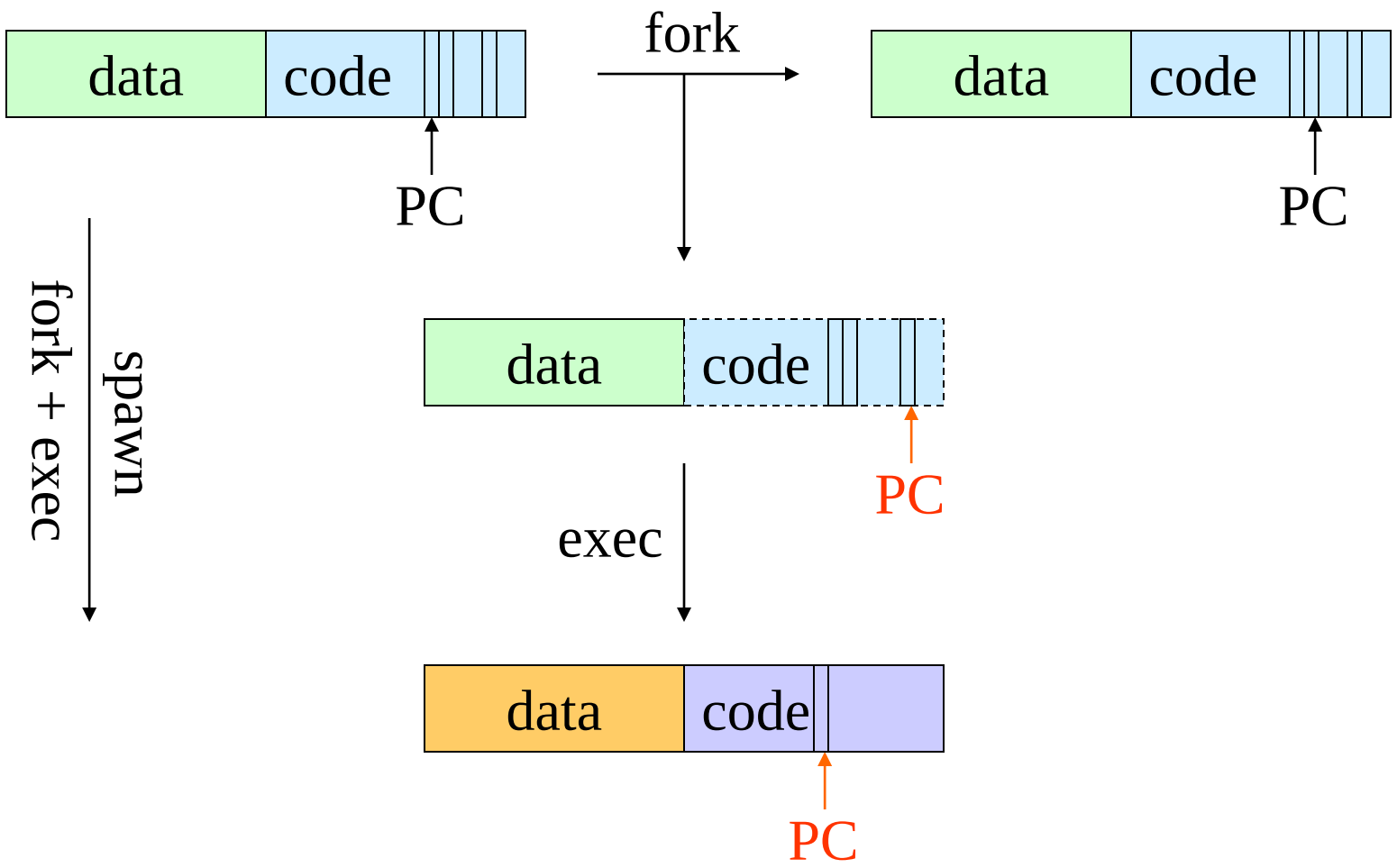
Software framework which:

- overrides differences in operating system and hardware
- enables to run program compiled to so called bytecode, i.e. which does not call native operations of processor but provides appropriate interpretation
- provides to these programs multi-process, multi-thread or multi-object environment
- implements basic set of functions, so called primitives

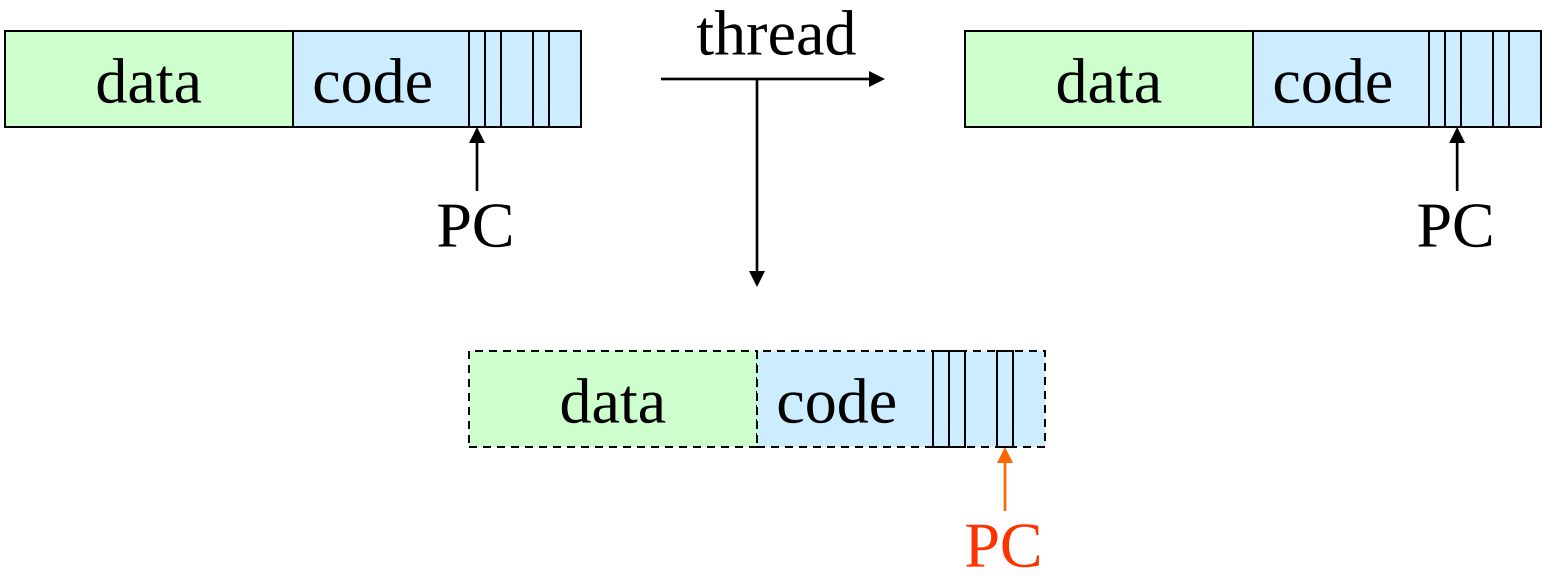
Processes, threads and objects

- Process is launched program which has own code, PC (program counter) and data
- Thread is launched program which has own code and PC but shares data with other threads
- object is „launched“ program which has own data and code but no PC

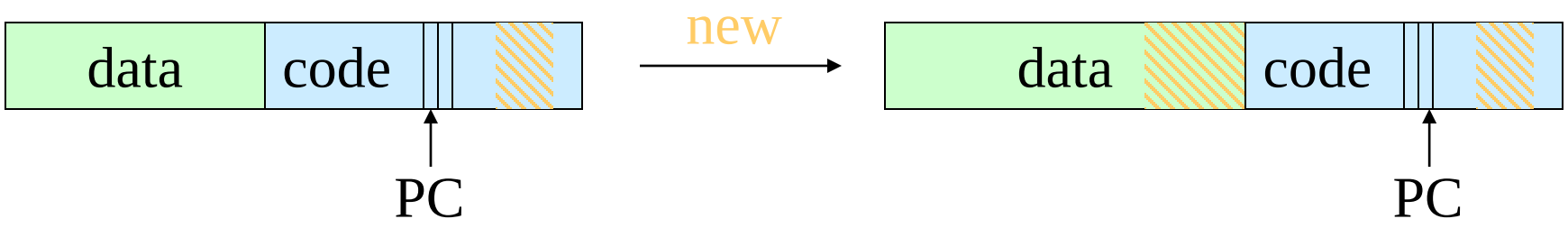
Process creation



Thread creation



Object creation



OOP

- VM have been designed for OOP (see Smalltalk)
- We transfer real world entities into virtual world in computer by description of objects
- Relations among these objects are specified by message passing

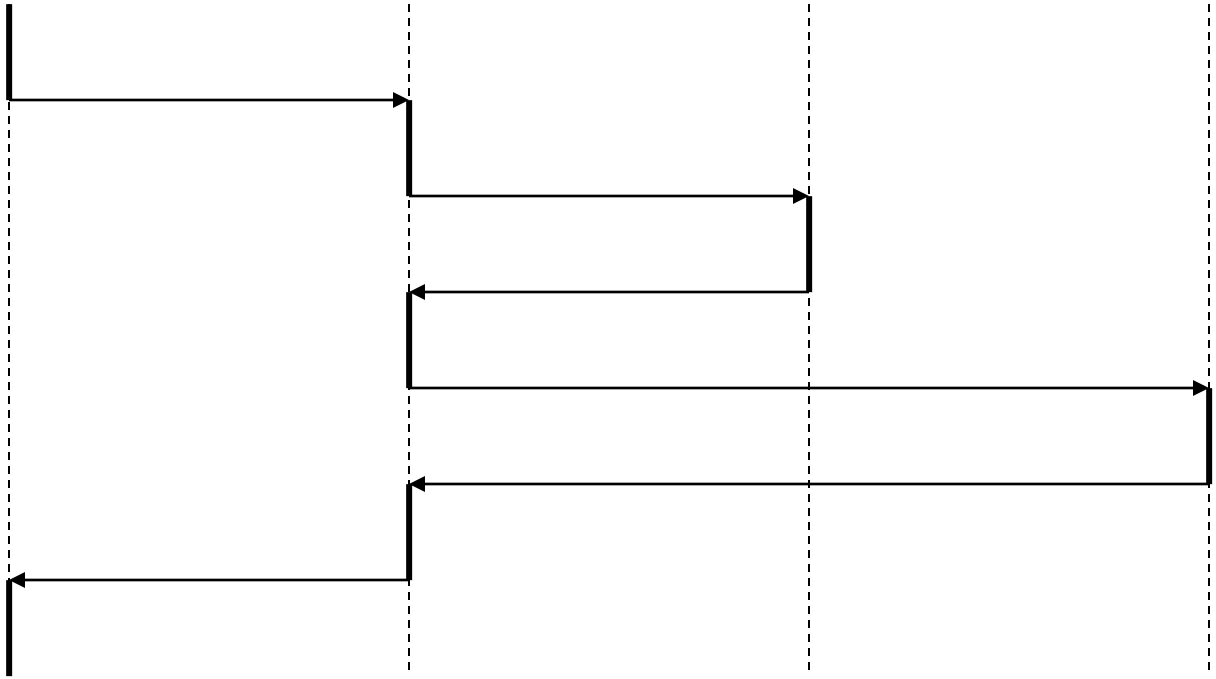
```
object.message(arguments)
```

Models of computation for OOP

- Standard model – class-instance model
- object is created as instance of certain class
- there is inheritance among classes
- message passing is synchronous (blocked until we get response)
- message passing is equivalent to calling of method with particular arguments

Control in the standard model

- Control is given over by calling methods of other objects got back with returned value

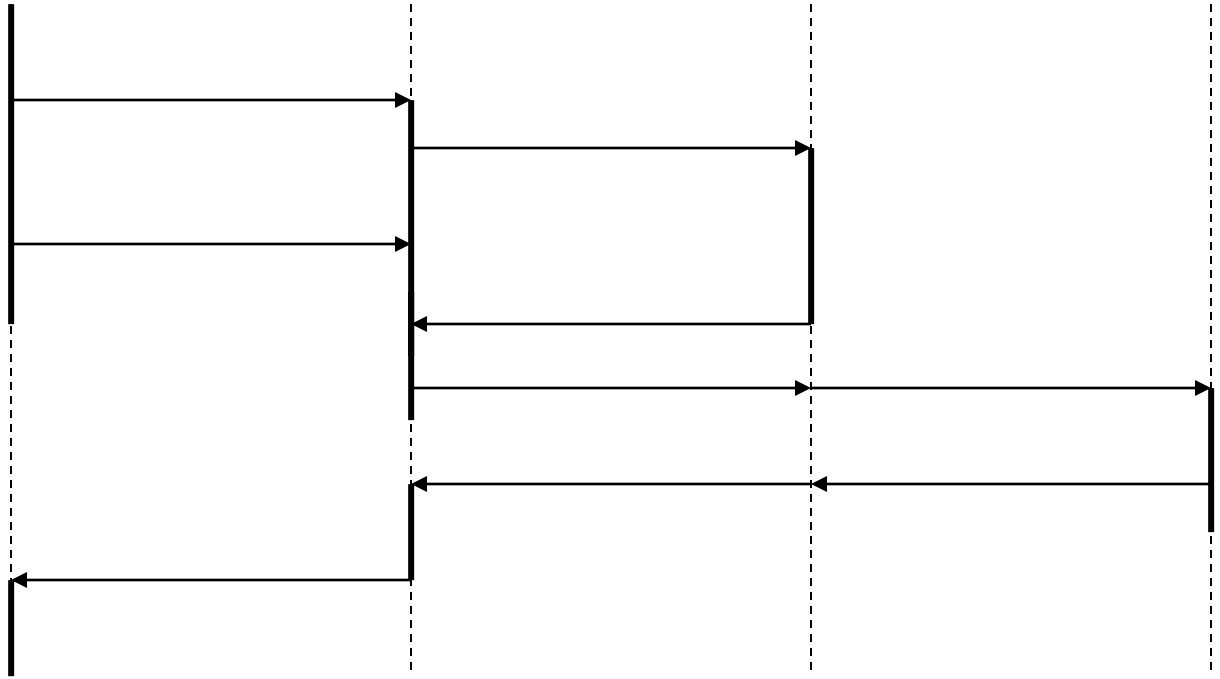


Models of computation for OOP

- Non-standard – actor model
- object (aktor) is created as modified copy of another object
- there is delegation among objects
- message passing is asynchronous (no blocking)
- message passing is equivalent to parallel calling of methods with particular arguments

Control in actor model

- Control is not passed to, we have more PCs, which fairly alternate



Implementation of actors in the standard model

- Actor model can be implemented in the standard model by establishing of MessageQueue
- The only PC we have is used for pop of message from top of the queue and performing of the corresponding method which cause pushing further messages into the queue

Actors and agents

- Actor is similar to agent which employs just direct communication
- Actor has no means of indirect communication
- (Unlike agent, actor is rather message-driven since its message is tied with call of corresponding method for its processing. Agent processes messages rather in a general handler.)

Actors and agents

- There are no other structures in the actor system than actors.
- While agents just enrich system with various structures
- This hybrid nature of MAS enables programmer to select the most appropriate structures from case to case, what is more flexible

Implementation of MAS on VM

- Usually we employ standard model
- Limited operation in one-thread VM. It is difficult to combine quick and slow processes which would need to be interrupted by preemptive scheduling
- It is better to attach own thread to each agent. Then its course can be preemptively interrupted.
- VM is more suitable for MAS real application lower latency of process scheduling it has

Implementation of MAS on VM

- Direct communication via MessageQueue (one global or one per each agent)
- Indirect communication via Space

JVM

- JVM is one of VM
- JVM provides multi-object environment
- JVM provides multi-thread environment
- JVM does not provides multi-process environment

Threads

class Thread

```
final
class A extends Thread {
  public A () {
    start();
  }
  public void run () {
    for (;;) { … };
  }
}
```

interface Runnable

```
class A implements Runnable {
  public A () {
    new Thread(this).start();
  }
  public void run () {
    for (;;) { … };
  }
}
```

Threads synchronization

- each object in JVM has so called monitor, which enables to synchronize threads which manipulate it
- Such manipulation locks the monitor and stops all other threads which would also try to manipulate the object.
- Critical area of the manipulation is defined as here:

```
synchronized (obj) { …
}
```

Threads synchronization

- When a thread aim to move its PC in a critical area, it block until it the monitor of the corresponding object is locked for the thread
- The PC is blocked if the monitor is locked for other thread
- The PC gets into the criticial area and the monitor is locked for the thread otherwise

Threads synchronization

- Synchronized methods

```
… method (…) {
  synchronized (this) {
    …
  }
}
```

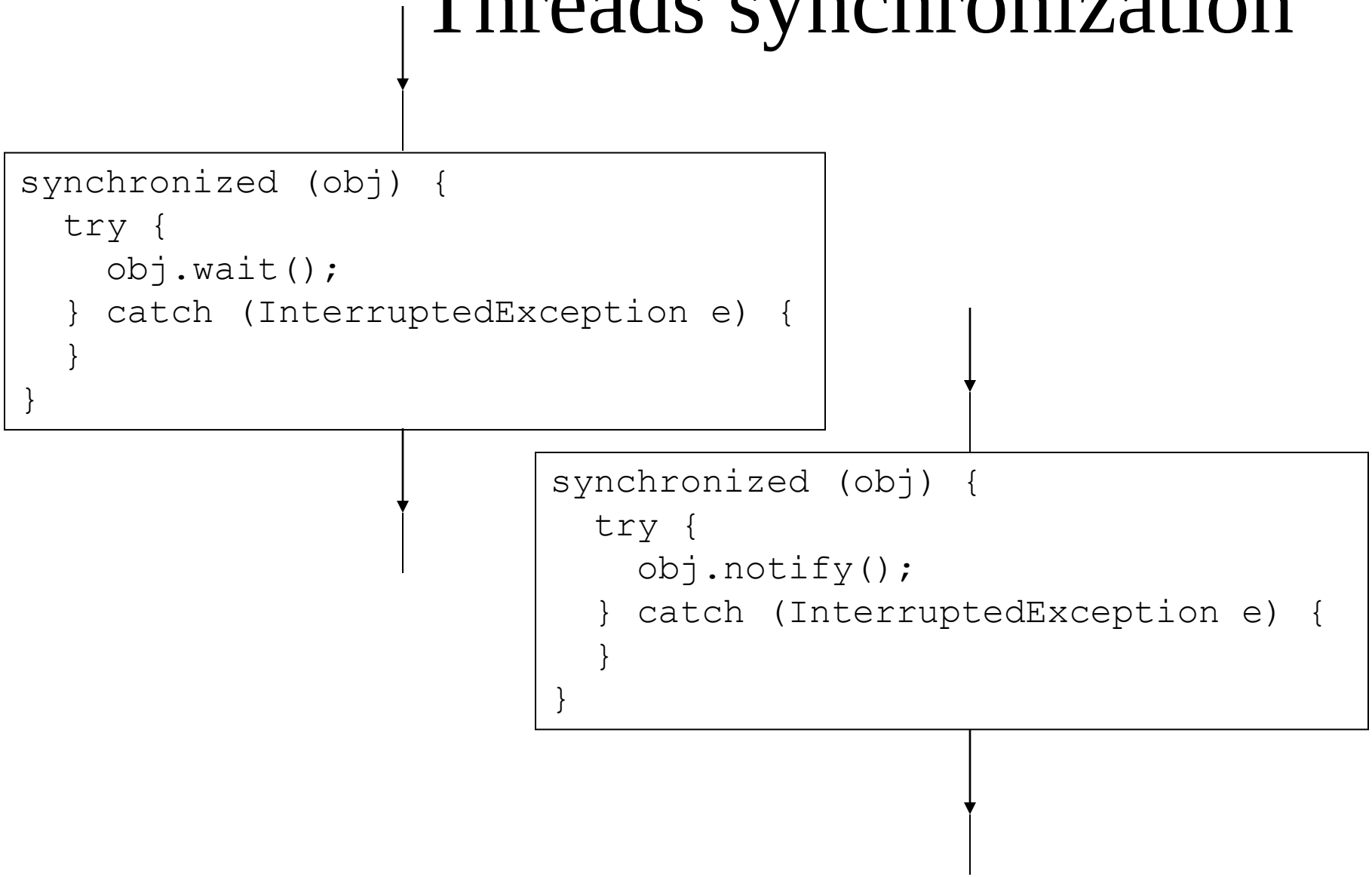
```
synchronized … method (…) {
  …
}
```

- Synchronized objects – objects of adaptor pattern, which encapsulate certain objects by synchronization of each method.

Threads synchronization

- Thread, which PC is in a critical area can unlock monitor (and block itself) by calling `wait()` on the object corresponding to the area.
- In this case just another thread can unblock it by calling `notify()` or `notifyAll()` on the same object
- while `notify()` unblock just random one of such blocked threads, `notifyAll()` unblock all of them

Threads synchronization



Space - singleton

- Space is implemented as object of singleton pattern
- Its constructor is `private`
- Instead of constructor we call static method, e.g. `getInstance()`, which return the only instance of such class (this instance is stored in static `private` attribute

```
public class Singleton {
  static Singleton instance =
    new Singleton();
  private Singleton () {
    ...
  }
  public static
    Singleton getInstance() {
      return instance;
  }
}
```

Space – time validity, priority

Time validity:

- The write operation stores not just value but also time validity into space
- The read operation check if the stored value has not expired yet, if so, the value is ignored as it has never been written.

Priority:

- The write operation stores also priority into space
- The write operation has no effect if it has lower priority than the data written in past until their time validity is not expired.

Agent

- Object with associated thread
- Object enrich by a certain standard mechanism of data exchange with other agents
- Object, which methods are not dedicated to be called from outside, they are called just from the thread

Timers

- Timing in Java is based on overloaded method `wait(),` where the optional argument specifies timeout in miliseconds
- Employing this method Java implements object Timer which has one thread which is mostly blocked and wait for launching of timered tasks at the proper time
- These tasks must be derived from class TimeredTask and override method `public void run()`

Triggers

- Agent can register for notification of some events
- Then agent sleeps by calling wait()
- The event appears
- Trigger call notify() and agent wakes up

Event propagation

- Agent modularity can be considered as ambition to remove specific interface at side of event listener

Event propagation – traditional

```
class Listener implements Event1Listener,
   Event2Listener, ..., EventNListener {
   ...
   obj1.addListener(this); ...
   obj2.addListener(this); ...
   objN.addListener(this);
   ...
  ... event1Performed(... arg1 ...) { ... };
  ... event2Performed(... arg2 ...) { ... };
  ...
  ... eventNPerformed(... argN ...) { ... };
}
```

Event propagation –“agent-based”

```
class Listener extends Agent {
  ...
  for (;;) {
    ... event = getEvent(...); // Receive
    switch (event.getType) {
      case ... : event1Performed(...); break;
      case ... : event2Performed(...); break;
      ...
      case ... : eventNPerformed(...); break;
    }
  };
}
```

Event propagation

- In the traditional case, the event is processed in thread in which it has been generated
- In the agent-based case, the event is processed in thread associated with agent, the thread in which the event has been generated just unblock the agent thread.
- i.e. each event listener has own thread for event processing.
- Thus it need to implement just one interface which enables the unblocking