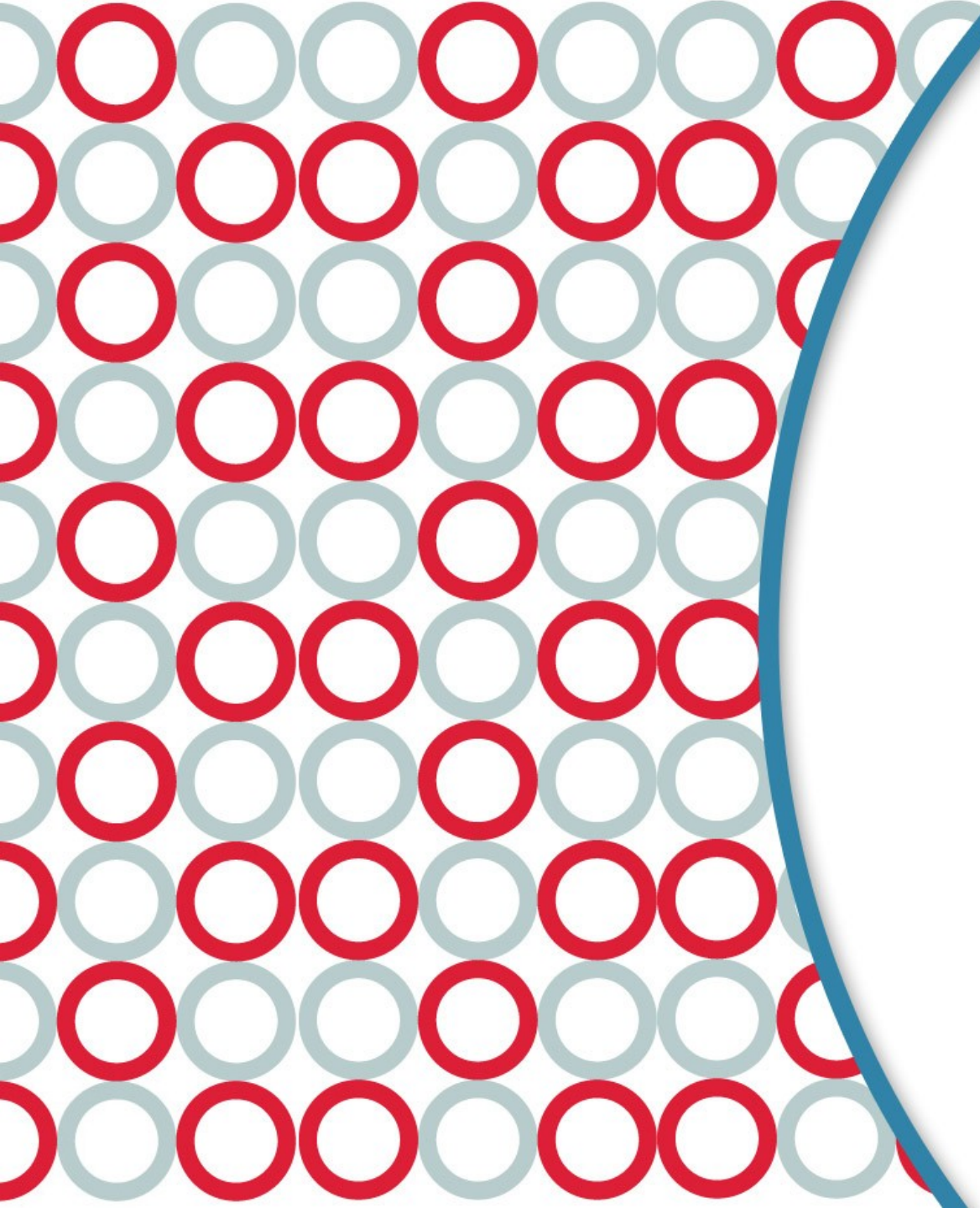


May 2019, IPT Course
Java Web Debelopment

Threads and Concurrency

Trayan Iliev

tiliev@iproduct.org
http://iproduct.org

About me

Trayan Iliev

– CEO of IPT – Intellectual Products & Technologies

– Oracle® certified programmer 15+ Y

– end-to-end reactive fullstack apps with Java, ES6/7, TypeScript, Angular, React and Vue.js

– 12+ years IT trainer

– Voxxed Days, jPrime, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Organizer RoboLearn hackathons and IoT enthusiast (http://robolearn.org)

Where to Find the Code?

Java Web Development projects and examples are available @ GitHub:

https://github.com/iproduct/course-java-web-development

Agenda for This Session

- Defining, Instantiating, and Starting Threads
- Synchronizing Code
- Thread Problems
- Immutable classes
- java.util.concurrent constructs

Concurrent Programming – Main Concepts I

- Program = data declarations + assignment and control-flow statements expressed in a programming language
- Concurrent Program = Program as collection of interacting computational processes that may be executed in parallel
- Concurrent programs can be executed sequentially on a single processor by interleaving the execution steps of each computational process, or executed in parallel by assigning each computational process to one of a set of processors that may be close or distributed across a network
- Parallel – when executions of several programs overlap in time by running them on separate processors
- Concurrent – potential parallelism, executions MAY overlap

Computer Systems, Processes, Threads, Tasks

- Computer systems (distributed computing) – we may map each logical unit of execution to a different computer (uniprocessor, multiprocessor, or cluster), providing unbounded autonomy and independence
- Processes – operating-system logical abstraction allowing computer system to support many units of execution. Each process typically represents a separate running program; for example, an executing JVM, guarantee some degree of independence, lack of interference, and security
- Threads – various constructs – further trade-offs in autonomy for lower overhead:Sharing, Scheduling, Communication
- Tasks – lightweight executable frameworks (e.g. Executors)

Concurrent Programming – Main Concepts II

- Multitasking – method where multiple tasks, also known as processes, share common processing resources such as a CPU (provided by kernel of contemporary operating systems)
- Cooperative multitasking/time-sharing – multitasking systems in which applications voluntarily give up control to one another
- Preemptive multitasking/time-sharing – multitasking allowing the computer system to reliably guarantee each process a regular "slice" of operating time, based on its priority
- Process – separate program run by OS with own address space – in Java: ***java.lang.ProcessBuilder***
- Thread – "light-weight" processes able to share resources in the scope of the OS process – e.g. memory, files etc.

Concurrent Programming – Advantages, Disadvantages and Typical Use Cases I

- Motivations:
  - Resource utilization – let other processes to run when waiting for input or output operations
  - Simplicity and Convenience – when coding programs that model different concurrent activities
  - Fairness – multiple users and programs may have equal claims for machine resources
  - Multiple Processors exploitation – real parallelism
  - Simplified Handling of Asynchronous Events
  - Responsive GUI Applications
  - Active Objects, Simulation, Embedded Systems, etc.

Concurrent Programming – Advantages, Disadvantages and Typical Use Cases II

- Risks and Challenges:
  - Safety – need to synchronize the execution of different processes and to enable them to communicate - example:

```
public class UnsafeSequence {
    private int value;
    public int getNext() {
        return value++;
    }
}
```

  - Liveness – the process/thread should do something useful (trivial case for safe process - “dummy” that does nothing)
  - Performance – e.g. contexts switching overhead

Concurrent Execution as Interleaving of Atomic Statements I

- Atomic Statement – executed to completion without interleaving statements from another processes/threads

Integer n ← 0	
p	q
p1: n ← n + 1	q1: n ← n + 1

Process **p**	Process **q**	**n**
p1: n ← n + 1	q1: n ← n + 1	0
(end)	q1: n ← n + 1	1
(end)	(end)	2

Process **p**	Process **q**	**n**
p1: n ← n + 1	q1: n ← n + 1	0
p1: n ← n + 1	(end)	1
(end)	(end)	2

Source: Principles of Concurrent and Distributed Programming (Second edition), Ben-Ari, M., Addison-Wesley, 2006

Concurrent Execution as Interleaving of Atomic Statements II

Limited Critical Reference (LCR)

Integer n ← 0	
p	q
p1: temp ← n p1: n ← temp + 1	q1: temp ← n q2: n ← temp + 1

Process **p**	Process **q**	**n**	**p.temp**	**q.temp**
p1: temp ← n	q1: temp ← n	0	?	?
p2: n ← temp + 1	**q1: temp ← n**	0	0	?
p2: n ← temp + 1	q2: n ← temp + 1	0	0	0
(end)	**q2: n ← temp + 1**	1		0
(end)	(end)	1		

Source: Principles of Concurrent and Distributed Programming (Second edition), Ben-Ari, M., Addison-Wesley, 2006

Concurrent Programming – Advantages, Disadvantages and Typical Use Cases II

- Risks and Challenges:
  - Safety – need to synchronize the execution of different processes and to enable them to communicate - example:

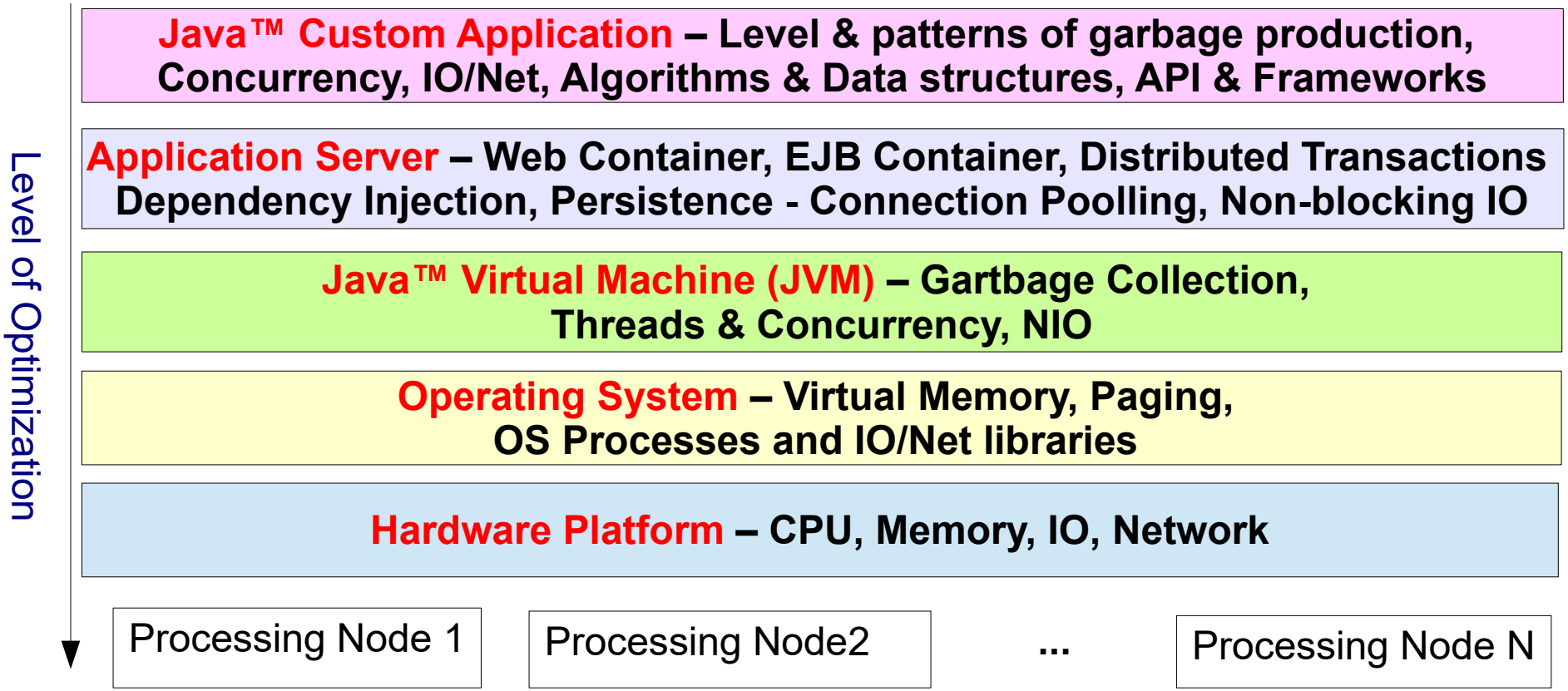
```
public class SafeSequence {
    Private int value;
    public synchronized int getNext() {
        return value++;
    }
}
```

  - Liveness – the process/thread should do something useful (trivial case for safe process - “dummy” that does nothing)
  - Performance – e.g. contexts switching overhead

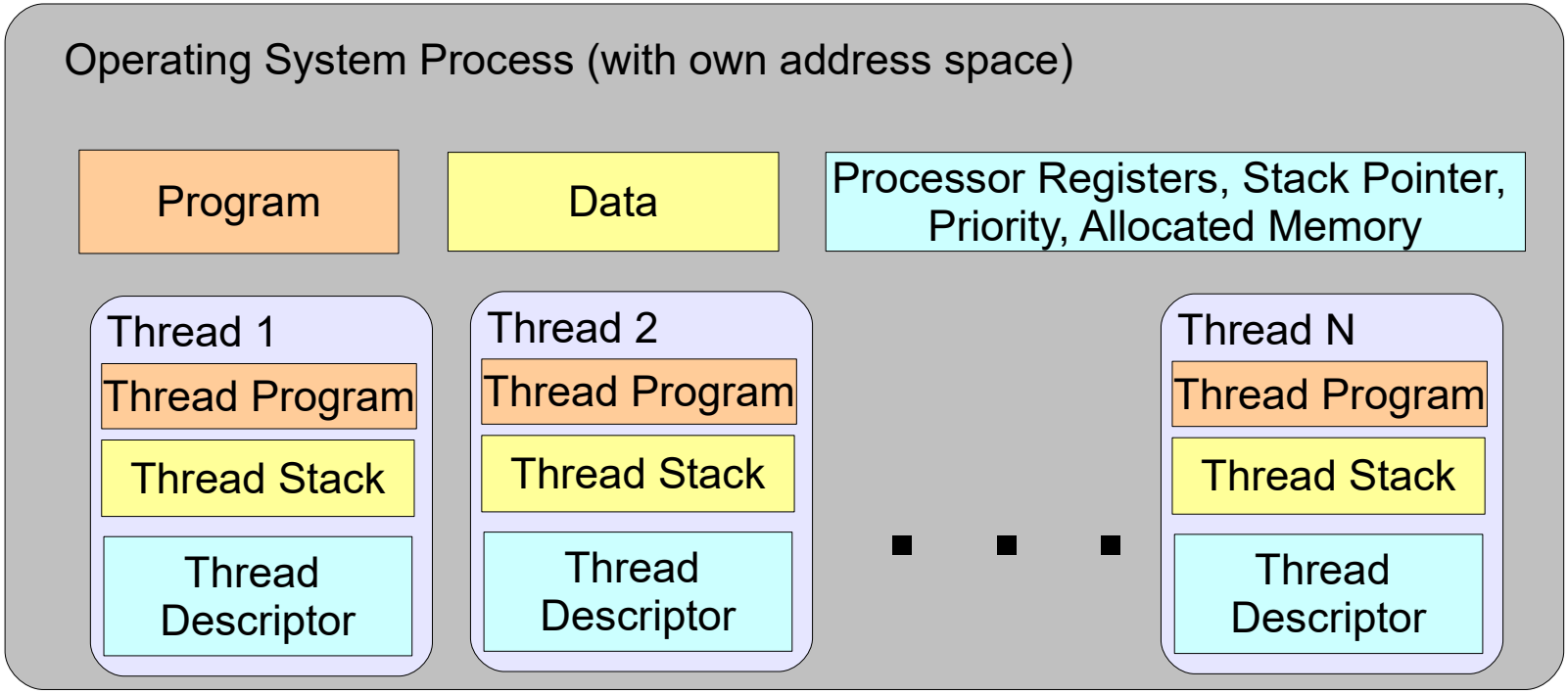
Concurrency as Abstract Parallelism. Multitasking

- Encapsulation is an abstraction that separates the publicly visible interface of a software component (object, module, system) from its hidden implementation
- Concurrency is an abstraction that is designed to make it possible to reason about the dynamic behaviour of programs
- *Definition:* Concurrent Program consists of a finite set of (sequential) processes. The processes are defined using a finite set of atomic statements. The execution of a concurrent program proceeds by executing a sequence of the atomic statements obtained by arbitrarily interleaving the atomic statements from the processes. A computation is an execution sequence that can occur as a result of the interleaving. Computations are sometimes called scenarios.

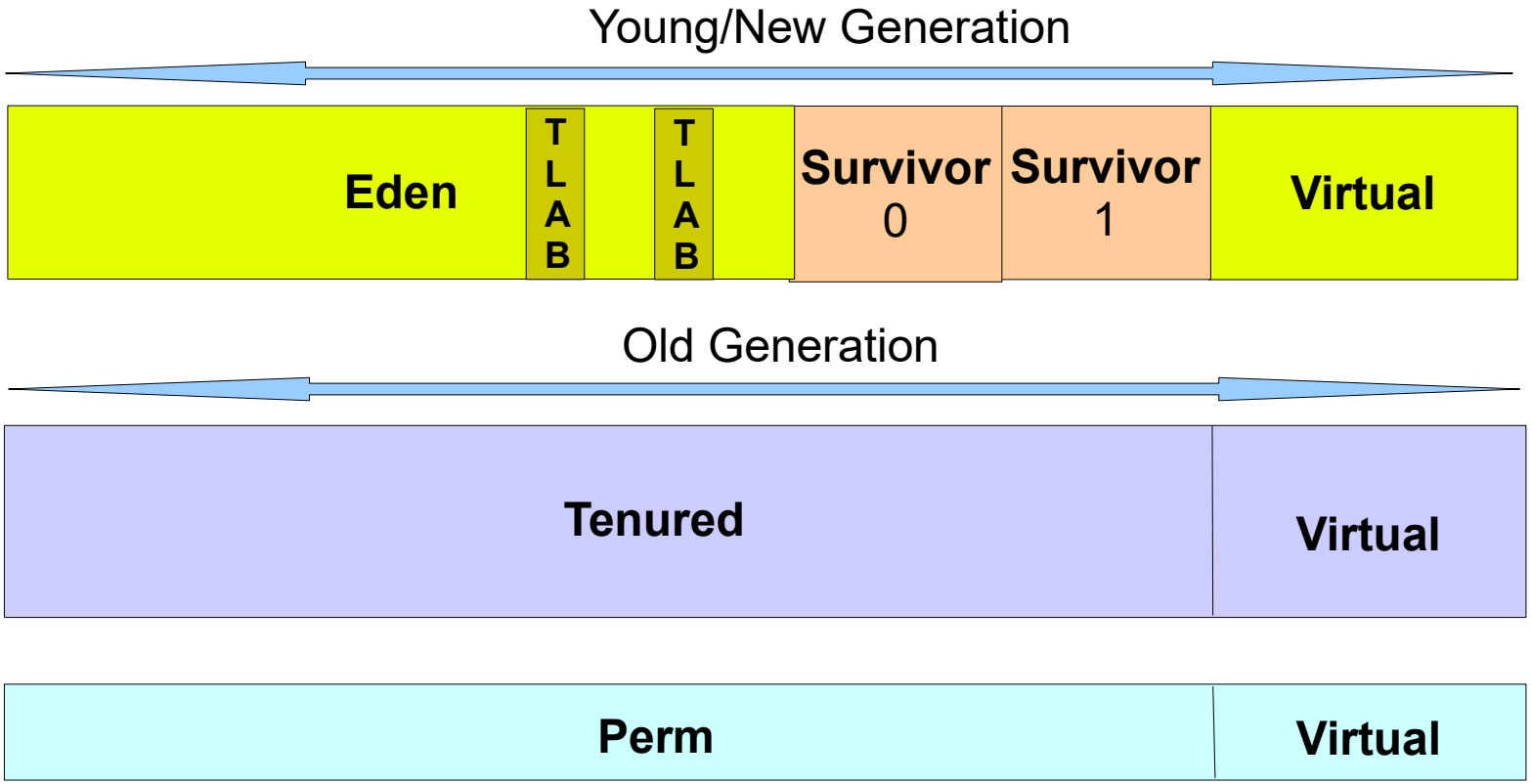
Application Stack



Processes of OS and Threads



Threads Memory Allocation



Concurrency and OO Programming. Object and Activity Centric Abstractions I

- Objects ↔ Concurrency – linked from the early days of programming languages (e.g. Simula – the first OO language & among the first concurrent languages, Ada – brings concurrent programming out from the world of specialized, low-level languages and systems)
- Object Models – Sequential, Active Objects (Actors), Mixed
- Object-centric view – a system is a collection of interconnected objects
- Activity-centric view – a system is a collection of possibly concurrent activities
- Safety ↔ Liveness Correctness Properties (Duality)
- Reusability ↔ Performace QoS Concerns

Concurrency and OO Programming. Object and Activity Centric Abstractions II

- A class is thread safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code
- Type Safety (can be chacked by compiler) ↔ Multithread Safety (cannot be easily checked automatically)
- Two ways ensuring thread safety:
  - Formal Methods – Labelled Transition System (LTS), Temporal Logics (Linear / Branching Temporal Logic)
  - Careful Engineering Practices – guarantee object consistency (invariants): C1 Method A C2: C1,C2 - consistent

Ensuring Consistency / Thread Safety

- Exclusion techniques – guarantee atomicity of public actions, preventing race conditions producing storage conflicts:
  - Read/Write Conflicts – e.g. one thread reads the value of a field, while another tries to write to it
  - Write/Write Conflicts – e. g. two threads writing to the same field
- Default Rules:
  - Always lock during updates to object fields
  - Always lock during access of possibly updated object fields
  - Never lock when invoking methods on other objects

Three Ways to Achieve Thread Safety

- Exclusion – use synchronization whenever access state variable
  - Stateless objects do not need synchronization
- Immutability – make state variables immutable
  - Use immutable data types and value containers – java.lang.Integer, java.math.BigDecinal, java.lang.String, etc.
  - Use final fields & Do not allow fields to be accessed until construction is complete
- Confinement – don't share the state variables across threads: within Methods, Threads, Objects, Groups of Threads

Critical Section Problem I

- Definition: Each of N processes is executing in a infinite loop a sequence of statements that can be divided into two subsequences: critical section and non-critical section
- Correctness Specification:
  - Mutual Exclusion – statements from critical sections must not be interleaved
  - Freedom from Deadlock – if one or more processes are trying to enter their critical sections – one of them must succeed
  - Freedom from Starvation – if any process tries to enter its critical section it must succeed **eventually** (after finite number of steps)

Processes and Threads in Java

- Process – separate program run by OS with own address space – in Java: ***java.lang.ProcessBuilder***
- Thread – "light-weight" processes able to share resources in the scope of the OS process – e.g. memory
  - Constructor: public Thread(ThreadGroup group, Runnable target, String name)
  - Interface Runnable defines method void **run()** - using Runnble instead of overriding Thread.run() separates our synchronised methods from these of the Thread class
  - ThreadGroup – not very useful (allows to specify security permissions, limit maximum priority of threads, or catch exceptions)

Thread Creation, Running, Yielding, Sleeping, Joining, Changing Priority, Daemon Threads

- Starting Runnable in new Thread – run() and start():

  new Thread(myRynnable).start();

- Main methods of **Thread** class:
  - isAlive() - true if it has started but not terminated
  - setPriority() - between MIN_PRIORITY=1 & MAX_PRIORITY=10
  - join() - suspends the caller until the target thread completes
  - sleep(long msecs) – suspends the thread for msecs ms
  - yield() - provides hint to JVM to run other thread if available
- Daemon threads - setDaemon(true)

Advantages using JSR 166: Concurrency Utilities

- Easier and faster development using already proven and tested concurency constructs and classes in java.util.concurent package and its sub-packages
- Better productivity and flexibility choosing alternatives
- Locking idioms that simplify many concurrent applications
- High-level API for launching and managing tasks in pooled threads using Executors
- Easier to manage large collections of data reducing the need for synchronization
- Better code maintainability in the future

JSR 166: Concurrency Utilities

- Framework classes for implementation of fine-grained concurrency – ***Executor framework*** (package ***java.util.concurrent***) standardized submission, (scheduled) execution, and control of asynchronous tasks based on policies, by pooling and reusing threads
- Highly-efficient concurrent collections and maps: ***BlockingQueue, ConcurrentMap, ConcurrentNavigableMap***
- Atomic valiables reducing the need for client-side synchronization for development of effeicient parallel algorithms, counters, and generators - ***java.util.concurrent.atomic***
- Synchronizers: *semaphores, mutexes, barriers, latches, exchangers*
- Locks: more efficient and flexible resource locking compared to synchronized – ***java.util.concurrent.locks***
- Nanosecond task timing precision

Executors Framework I

- Executor interfaces defined in java.util.concurrent package:
  - Executor – basic interface supporting execution of new tasks
  - ExecutorService – subinterface of Executor adding lifecycle management features, for the executor and individual tasks submitted
  - ScheduledExecutorService – subinterface of ExecutorService adding support for scheduled and/or periodic execution of tasks

Executors Framework II

- Types of ExecutorService:
  - CachedThreadPool – threads created as needed
  - FixedThredPool – fixed number of threads
  - SingleThredPool – singleton thread – no synch needed
  - ScheduledThreadPool – provides ability for sheduling and periodical execution of tasks
- Returning asynchronous results – Callable & Future
- Canceling tasks and executors – methods: shutdown(), shutdownNow(), awaitTermination()

Returning Async Result Using Callable & Future I

```
Callable<List<ImageData>> task =
        new Callable<List<ImageData>>() {
            public List<ImageData> call() {
                List<ImageData> result = new ArrayList<ImageData>();
                for (ImageInfo imageInfo : imageInfos)
                    result.add(imageInfo.downloadImage());
                return result;
            }
        };
Future<List<ImageData>> future = executor.submit(task);
```

Returning Async Result Using Callable & Future II

```
        try {
            List<ImageData> imageData = future.get();
            for (ImageData data : imageData)
                renderImage(data);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            future.cancel(true);
        } catch (ExecutionException e) {
            throw Throwable(e.getCause());
        }
}
```

Canceling ExecutorSevice and Tasks

```
void shutdownAndAwaitTermination(ExecutorService executor) {
   executor.shutdown();   // Disable tasks submission
   try {
      if (!executor.awaitTermination(120, TimeUnit.SECONDS)) {
      executor.shutdownNow();   // Cancel tasks currently submitted
         if (!pool.awaitTermination(120, TimeUnit.SECONDS))
         System.err.println("Pool did not finish successfully");
      }
   } catch (InterruptedException ie) {
     pool.shutdownNow();      Thread.currentThread().interrupt();
   }
}
```

Canceling ExecutorSevice and Tasks

```
void shutdownAndAwaitTermination(ExecutorService executor) {
   executor.shutdown();   // Disable tasks submission
   try {
      if (!executor.awaitTermination(120, TimeUnit.SECONDS)) {
         executor.shutdownNow();   // Cancel tasks  submitted
         if (!executor.awaitTermination(120, TimeUnit.SECONDS))
            System.err.println("Executor did not finish successfully");
      }
   } catch (InterruptedException ie) {
      executor.shutdownNow();
      Thread.currentThread().interrupt();
 } }
```

Alternatives for Thread Construction

- Inner class extending **Thread**
- Anonymous inner class extending **Thread**
- Class implementing interface **Runnable**
- Anonymous inner class implementing interface **Runnable**
- Starting a thread in separate method

Sharing resources between multiple threads

- Problems accessing resources from multiple threads - Interference
- Ornamental garden problem (Alan Burns & Geoff Davies, 1993):

Sharing resources between multiple threads

- Synchronization mechanisms
  - synchronized methods
  - critical sections (synchronized blocks)
  - semaphores – methods: acquire(), release()
  - monitors – class Object methods: wait(), notify, notifyAll()
  - channels – PipedInputStream, PipedOutputStream,PipedReader, PipedWriter
- Synchronization Objects – Monitors – represented in Java as class with synchronized methods / blocks
  - Condition synchronization – block thread until certain

Sharing resources between multiple threads

- Synchronization mechanisms
  - synchronized methods
  - critical sections (synchronized blocks)
  - semaphores – methods: acquire(), release()
  - monitors – class Object methods: wait(), notify, notifyAll()
  - channels – PipedInputStream, PipedOutputStream,PipedReader, PipedWriter
- Synchronization Objects – Monitors – represented in Java as class with synchronized methods / blocks
  - Condition synchronization – allows to block thread until certain conditions holds – using: wait(), notify, notifyAll()

Atomic Operations & Volatile Attributes I

- volatile keyword doesn't guarantee any type of locking (++ operation is not atomic in Java)
- If changed frequently the expense of preventing compiler optimizations, and frequent flushing the value to other threads can result in slower execution than synchronous locking of the whole method
- volatile can be used when:
  - The field need not obey any invariants with respect to others.
  - Writes to the field do not depend on its current value.
  - No thread ever writes an illegal value with respect to intended semantics.
  - The actions of readers do not depend on values of other non-volatile fields

Atomic Operations & Volatile Attributes II

- Atomicity – access and updates to fields of any type except long and double are guaranteed to be atomic
- Access even to long and double fields will be atomic if these fields are declared volatile
- Atomicity per se does not guarantee we have most recent values of the fields and is of little importance
- volatile keyword guarantees that changes to field made in one thread will be visible to other threads immediately (synchronized has the same memory effect of acquiring/flushing changes to other processes)

Atomic Lock-Free Thread-Safe Programming Using Atomic Variables - I

- java.util.concurent.atomic – набор от класове, които поддържат thread-safe програмиране без заключване (lock-free) чрез използване на една единствена променлива
- Класовете разширяват концепцията за volatile стойности и масиви до такива, които предоставят операция за атомарно условно обновяване:

```
boolean compareAndSet(expectedValue, updateValue);
```

```
class SerialNumberGenerator{
  private final AtomicLong serialNumber = new AtomicLong(0);
  public long next() {
    return serialNumber.getAndIncrement();
  }}
```

Atomic Lock-Free Thread-Safe Programming Using Atomic Variables - II

- AtomicBoolean
- AtomicInteger
- AtomicLong
- AtomicReference
- AtomicMarkableReference
- AtomicStampedReference
- AtomicIntegerArray
- AtomicLongArray
- AtomicReferenceArray
- AtomicIntegerFieldUpdater
- AtomicLongFieldUpdater
- AtomicReferenceFieldUpdater

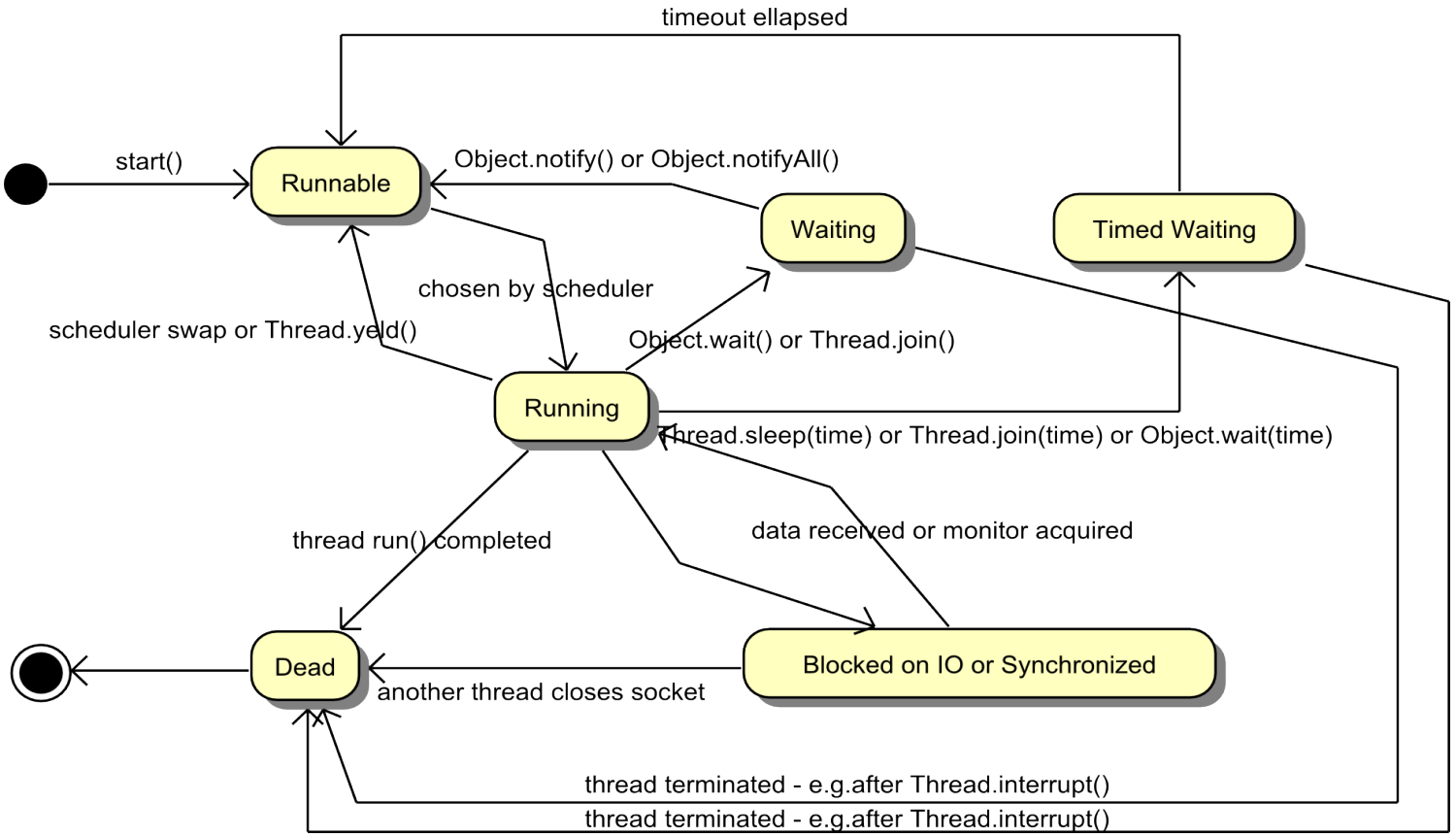
Interrupting Threads

- Interrupting Thread (not a normal way for thread to finish – use boolean flag instead) – t.interrupt();
- Methods throwing InterruptedException:
  - Object.wait()
  - Thread.join()
  - Thread.sleep(long msecs)
- IOStream and blocking methods don't throw InterruptedException
- Thread.interrupted() - returns and clears the interruption status

Thread States. Changing Thread State – wait(), notify(), notifyAll() I

- Thread states
  - New – momentary state after thread is being created but before finishing initialization
  - Runnable – thread is ready to be executed when scheduled by the JVM/OS mechanisms
  - Running – thread is currently executed by CPU
  - Blocked – the thread is waiting some resource to become available or some condition to become true
  - Dead – thread is no longer schedulable or runnable
- Common reasons for blocking: I/O operations, synchronized methods/blocks, sleep() and wait()

Thread States Diagram



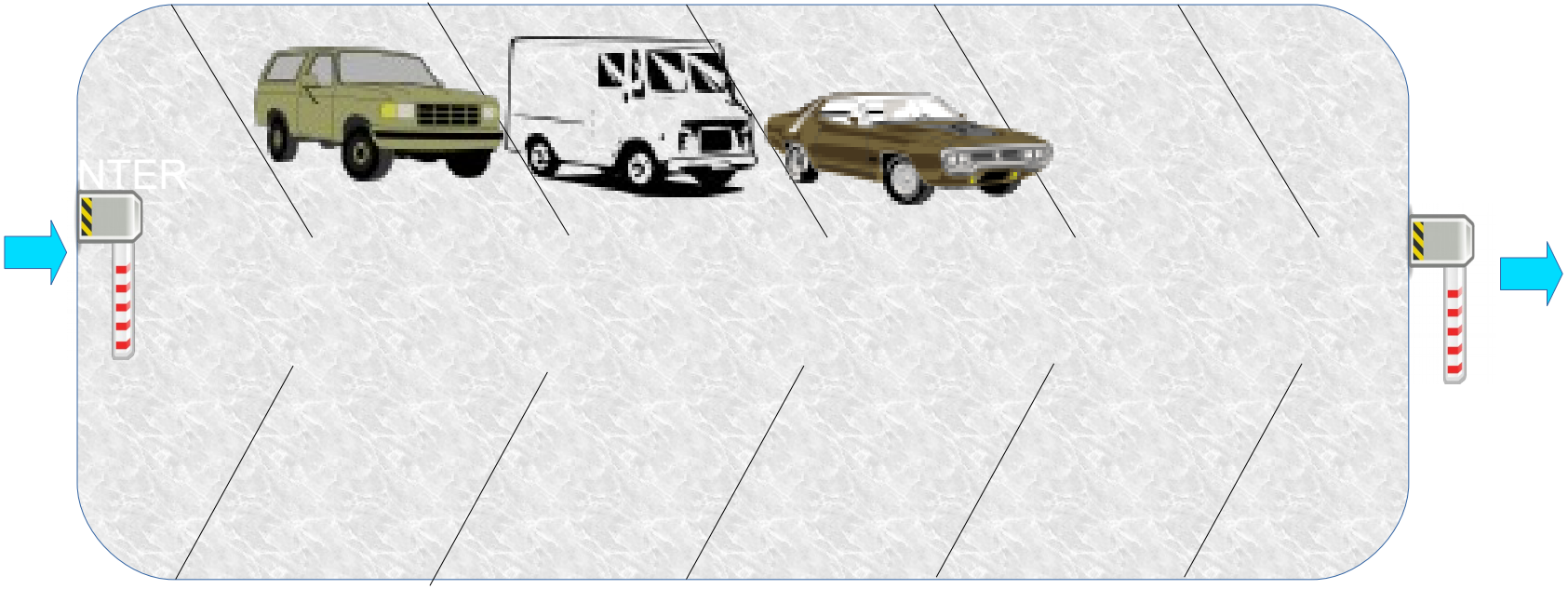
Thread states and changing thread state – wait(), notify(), notifyAll() II

- Correct/incorrect ways for blocking unblocking
  - Correct: wait() - notify() - free resource locks before blocking threads –
  - wait() should be always guarded by:
    while (!condition) object.wait();
  - wait() and notify() / notifyAll() should be called in synchronized block for the same object
  - Incorrect: stop(), suspend(), resume() - don't free acquired resource locks => Deadlock prone - deprecated
- Ways to stop the thread correctly
  - using volatile boolean flag for finishing computation
  - using method interrupt()

Thread states and changing thread state – wait(), notify(), notifyAll() III

```
TaskA:
synchronized(sharedMonitor) {
    <setup condition for TaskB>
    sharedMonitor.notify();
}
TaskB:
synchronized(sharedMonitor) {
    while(!condition)
        sharedMonitor.wait();
}
```

Car Park Controller Problem



Using Locks and Conditions: ReentrantLock, ReentrantReadWriteLock, Condition I

- java.util.concurent.lock – providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors
- Permit much greater flexibility in the use of locks and conditions, at the expense of more awkward syntax
- Locks can back out of an attempt to acquire a lock:
  - tryLock() - backs out if the lock is not available immediately or before a timeout expires
  - lockInterruptibly() - backs out if another thread sends an interrupt before the lock is acquired
- newCondition() - creates Conditions on which to wait/signal (objects should possess the lock, analogous of wait()/notify()

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html

Using Locks and Conditions: ReentrantLock, ReentrantReadWriteLock, Condition II

- Typical usage pattern:

```
class MyClass {
    private final ReentrantLock lock = new
ReentrantLock();
    public void doSomething() {
        lock.lock();  // block until condition succeeds
        try {
            // method body operations
        } finally {
            lock.unlock();
        }
    }
}
```

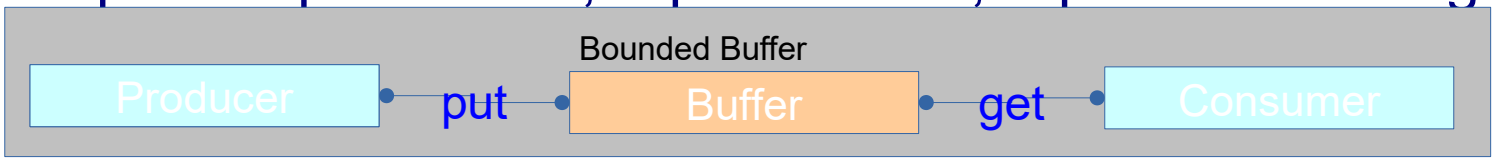
Using Locks and Conditions: ReentrantLock, ReentrantReadWriteLock, Condition III

- ReadWriteLock – provides a pair of associated locks, one for reading operations (shareable) and another for writing operations (exclusive)
- Allowing better concurrency levels in accessing shared data than that using normal mutual exclusion lock
- Example

.....

Producer-Consumer Problem I

- Problem for synchronizing the work of one or more Producers with one or more Consumers of some type of items/messages
- There are different solutions to the problem:
  - using wait(), notify(), notifyAll()
  - using Locks and Conditions (java.util.concurent.lock)
  - using Bounded Buffers – BlockingQueues / Dequeues
  - using Semaphores
  - using Channels (Pipes) –PipedInputStream, PipedOutputStream, PipedReader, PipedWriterUsing



Monitor Invariants

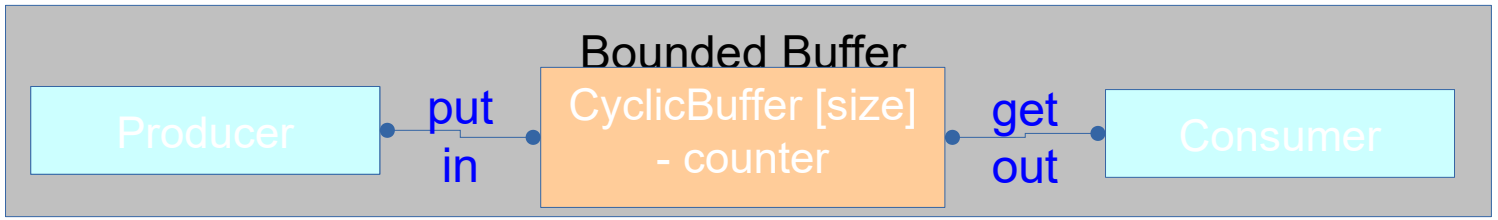
- Car Park Controller: 0 <= empty spaces <= N
- Semaphore Invariant: 0 <= value
- Bounded Buffer Invariant:

  0 <= count <= buffer size &&

  0 <= in <= buffer size &&

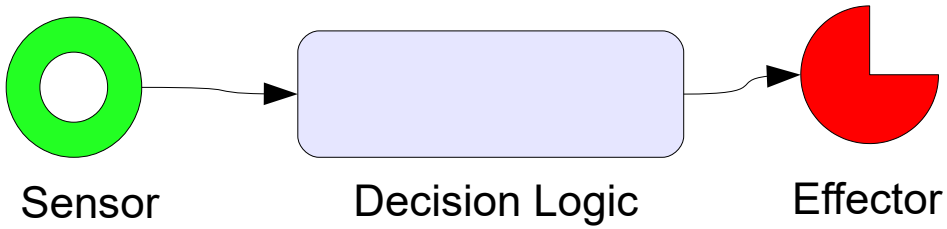
  0 <= out <= buffer size &&

  in == (out + count) % buffer size



Flow networks [Doug Lea, 1993]

- “A flow network is a collection of objects that all pass oneway messages transferring information and/or objects to each other along paths from sources to sinks. Flow patterns may occur in any kind of system or subsystem supporting one or more series of connected steps or stages, in which each stage plays the role of a producer and/or consumer”. Categories:
  - *Control systems*
  - *Assembly systems*
  - *Dataflow systems*
  - *Workflow systems*
  - *Event systems*



Source:Concurrent Programming in Java: Design Principles and Patterns, (second edition) by Doug Lea , published November 1, 1999 by Addison-Wesley – http://gee.cs.oswego.edu/dl/cpj/index.html

Concurrent Programming Tools in java.unitl.concurrent

- BlockingQueues
- BlockingDeques
- PriorityBlockingQueue
- DelayQueue
- SynchronousQueue
- CountDownLatch
- CyclicBarrier
- Exchanger
- Semaphore

Deadlock and Livelock

- Definition: If some processes are trying to enter their critical sections, then one of them must eventually succeed. Otherwise we have deadlock condition where some processes are blocked each other forever.
- Deadlock is a situation wherein two or more competing actions are each waiting for the other to finish, and thus neither ever does. It is often seen in a paradox like the "chicken or the egg
- A situation in which a thread waits to be notified of a condition but, on waking, finds that another thread has inverted the condition again. The first thread is forced to wait again. When this happens indefinitely, the thread is in Livelock

Necessary and Sufficient Conditions for Deadlock (Coffman, Elphick & Shoshani, 1971)

- Mutual Exclusion: At least one resource must be non-shareable. Only one process can use the resource at any given instant of time.
- Hold and Wait or Resource Holding: A process is currently holding at least one resource and requesting additional resources which are being held by other processes.
- No Preemption: The operating system must not de-allocate resources once they have been allocated; they must be released by the holding process voluntarily.
- Circular Wait: A process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, P = {P1, P2, ..., PN}, such that P1 is waiting for a resource held by P2, P2 is waiting for a resource held by P3 and so on till PN is waiting for a resource held by P1

Source: Wikipedia, available at: http://en.wikipedia.org/wiki/Deadlock

Techniques – Deadlock Detection [Wikipedia]

- Under deadlock detection, deadlocks are allowed to occur. Then the state of the system is examined to detect that a deadlock has occurred and subsequently it is corrected. An algorithm is employed that tracks resource allocation and process states, it rolls back and restarts one or more of the processes in order to remove the detected deadlock. Model checking - a finite state-model .progress analysis - finds all possible terminal sets in the model = deadlock.
- After a deadlock is determined, it can be corrected by using one of the following methods:
  - Process Termination: One or more process involved in the deadlock may be aborted. Partial computations will be lost.
  - Resource Preemption: Resource allocated to various processes may be successively preempted and allocated to other processes until deadlock is broken.

Source: Wikipedia, available at: http://en.wikipedia.org/wiki/Deadlock

Techniques – Deadlock Prevention I [Wikipedia]

- Removing the mutual exclusion condition means that no process will have exclusive access to a resource - non-blocking synchronization algorithms.

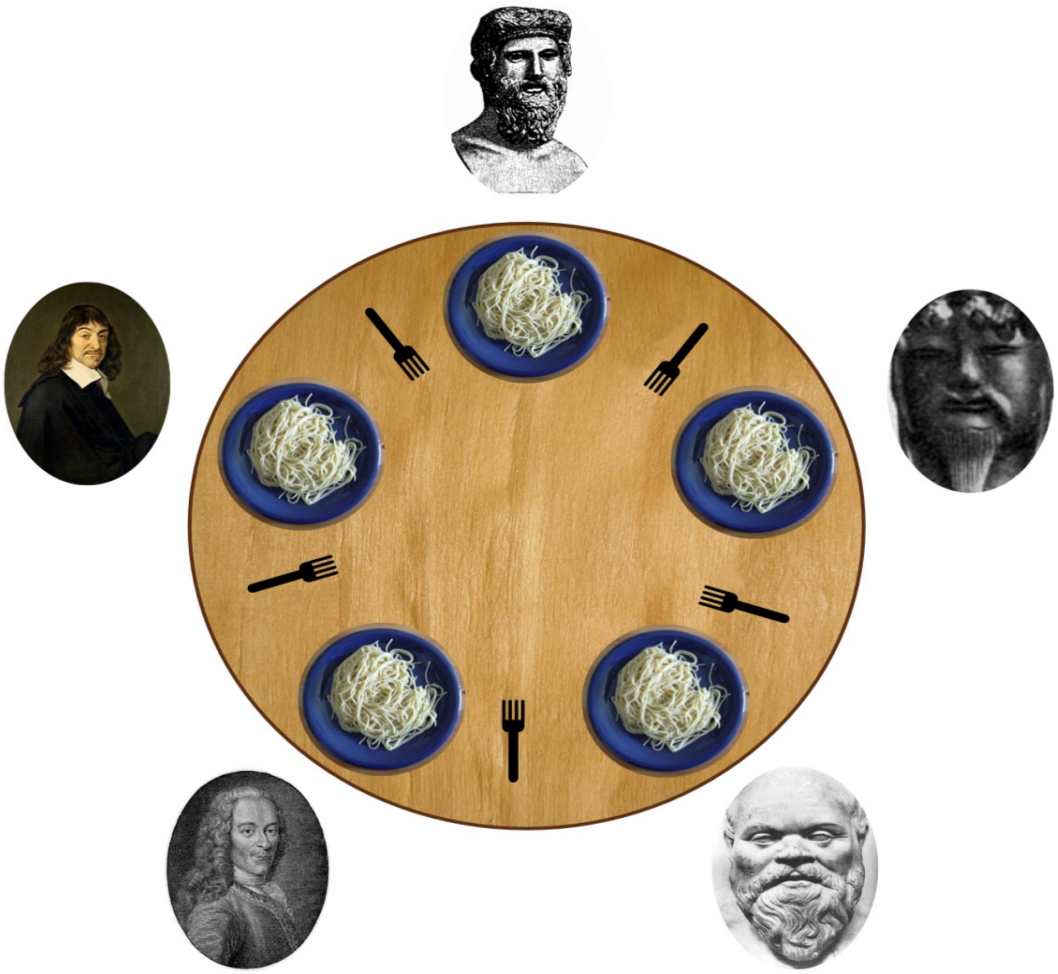
- The hold and wait or resource holding conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations). They must release all their currently held resources before requesting all the resources they will need from scratch. A process requiring a popular resource may have to wait indefinitely as such a resource may always be allocated to some process, resulting in resource starvation. These algorithms, such as serializing tokens, are known as the all-or-none algorithms.

Source: Wikipedia, available at: http://en.wikipedia.org/wiki/Deadlock

Techniques – Deadlock Prevention II [Wikipedia]

- The no preemption condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a priority algorithm. Preemption of a "locked out" resource generally implies a rollback, and is to be avoided, since it is very costly in overhead. Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control.
- The final condition is the circular wait condition – e.g. disabling interrupts during critical sections and using a hierarchy to determine a partial ordering of resources. If no obvious hierarchy exists, even the memory address of resources has been used to determine ordering and resources are requested in the increasing order of the enumeration. The Dijkstra's solution can also be used.

Problem for Dining Philosophers



Source: Wikipedia, available at:
http://upload.wikimedia.org/wikipedia/commons/6/6a/Dining_philosophers.png

References I

1. Java Concurrency in Practice by Brian Goetz, with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. Addison-Wesley, 2006 – http://jcip.net/

2. Concurrency: State Models & Java Programs (2nd Edition), by Jeff Magee and Jeff Kramer, Wiley, 2006 – http://www.doc.ic.ac.uk/~jnm/book/

3. Object Oriented System Development, by Dennis de Champeaux, Doug Lea, and Penelope Faure, originally published in 1993 by Addison-Wesley – http://gee.cs.oswego.edu/dl/oosdw3/index.html

4. Concurrent Programming in Java: Design Principles and Patterns, (second edition) by Doug Lea, published November 1, 1999 by Addison-Wesley – http://gee.cs.oswego.edu/dl/cpj/index.html

References II

5. Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs, Richard H. Carver, Kuo-Chung Tai, ISBN: 0471725048, Wiley, 2005 – http://www.amazon.com/Modern-Multithreading-Implementing-Debugging-Multithreaded/dp/0471725048

6. Thinking in Java. 3-rd ed., Eckel, B., Prentice Hall, 2002 – http://www.mindview.net/Books/TIJ

7. Thinking in Java. 4-th ed., Eckel, B., Prentice Hall, 2006 – http://mindview.net/Books/TIJ4

8. The Java™ Language Specification (third edition), James Gosling, Bill Joy, Guy Steele and Gilad Bracha, Addison-Wesley, 2005 – http://java.sun.com/docs/books/jls/

References III

9. Oracle®/ Sun Microsystems Java™ Technologies webpage – http://java.sun.com/

10. Effective Java Second Edition, Bloch, J., Sun Microsystems, 2008

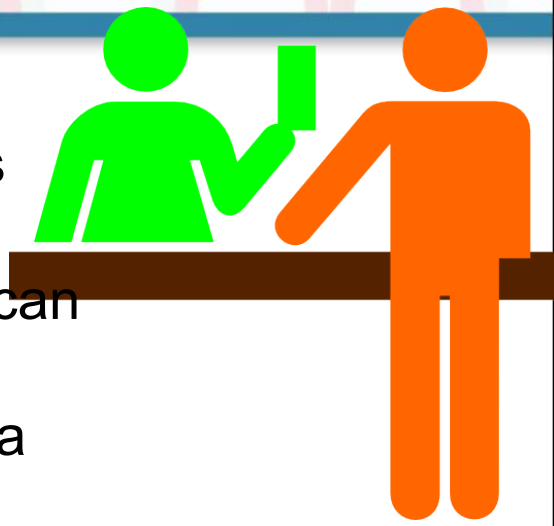
11. Principles of Concurrent and Distributed Programming (Second edition), Ben-Ari, M., Addison-Wesley, 2006

12. Verification of Sequential and Concurrent Programs, 3rd Edition by Apt, K., de Boer, F., Olderog. E., Texts in Computer Science. Springer-Verlag, 2009

13. Tutorial - Concurrency Lesson – http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html

14. VisualVM - http://visualvm.java.net/

Team Mission I: Ticket Booking



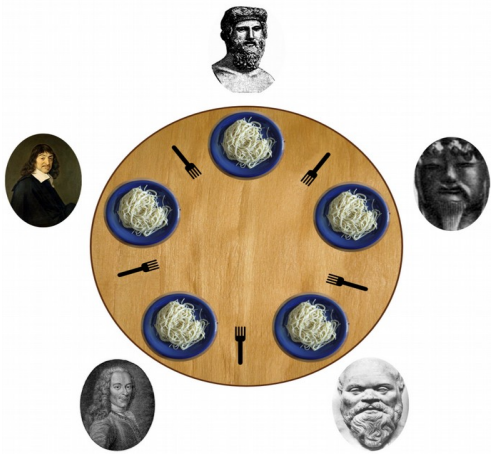
A central computer connected to remote terminals via communication links is used to automate seat reservations for a concert hall. A booking clerk can display the current state of reservations on the terminal screen. To book a seat, a client chooses a free seat and the clerk enters the number of the chosen seat at the terminal and issues a ticket. A system is required which avoids double-booking of the same seat while allowing clients free choice of the available seats. Write a Java program that implements the seat reservation system that does not permit double-bookings. (Hint: It is only necessary to model a few terminals and a few seats. Remember, a seat can appear to be free although it is booked or being booked by another clerk.)

Team Mission II: Roller Coaster

A roller-coaster control system only permits its car to depart when it is full. Passengers arriving at the departure platform are registered with the roller-coaster controller by a turnstile. The controller signals the car to depart when there are enough passengers on the platform to fill the car to its maximum capacity of M passengers. The car goes around the roller-coaster track and then waits for another M passengers. A maximum of M passengers may occupy the platform. The roller coaster consists of three processes/monitors: TURNSTILE, CONTROL and CAR. TURNSTILE and CONTROL interact by the shared action passenger indicating an arrival and CONTROL and CAR interact by the shared action depart signaling car departure.

Source: Concurrency: State Models & Java Programs (2nd Edition), by Jeff Magee and Jeff Kramer, Wiley, 2006 – http://www.doc.ic.ac.uk/~jnm/book/

Team Mission III: Dining Philosophers



- Provide deadlock free solution to the dining philosophers problem modelling forks as semaphores. You can apply one of the following strategies:
  - Introduce an asymmetry by programming the first four philosophers to take the left fork first, and the fifth philosopher to take the right fork first
  - Provide BUTTLER process permitting only four philosophers to sit at the table at the same time
  - Provide ordering between forks and allow to take the lower number fork first
  - Use coin to to decide which fork to take first

Source: Wikipedia, available at: http://upload.wikimedia.org/wikipedia/commons/6/6a/Dining_philosophers.png

Thank's for Your Attention!

Trayan Iliev

CEO of IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD

https://plus.google.com/+IproductOrg