THREADED PROGRAMMING

GIOVANNA LEHMANN MIOTTO

ISOTDAQ SCHOOL 2012

OUTLINE

- **Multi-tasking, concurrency**
- **A bit of history**
- **What is a thread**
- **Essentials in concurrent programming (Java)**
  - Safety, liveness, performance
  - Memory model
- **Conclusions**
- **References**

MULTI-TASKING

- **We are used to it in every day life**
  - Watch TV while eating pop-corns and caressing your cat
  - Send an e-mail, launch a job on the printer and do something else while waiting for the answer and the printed document
  - …
- **On a Computer**
  - While a task waits on input, another one performs calculations on some data and a third one outputs messages

MULTI-TASKING

- **There are two different types of multi-tasking**
  - Interleaved usage of one resource by different tasks
    - I can use the same hand to caress my cat and eat pop-corn, but in reality I'll use the hand in a sequential way to either caress the cat or put the pop-corn in my mouth
  - Parallel usage of the same type of resource by different tasks
    - Using two hands the action of caressing the cat and putting pop-corn in my mouth can happen AT THE SAME TIME
    - both hands are capable of doing both tasks

EVOLUTION OF COMPUTER MULTI-TASKING

Multi-tasking history starts with the introduction of operating systems -> task scheduler

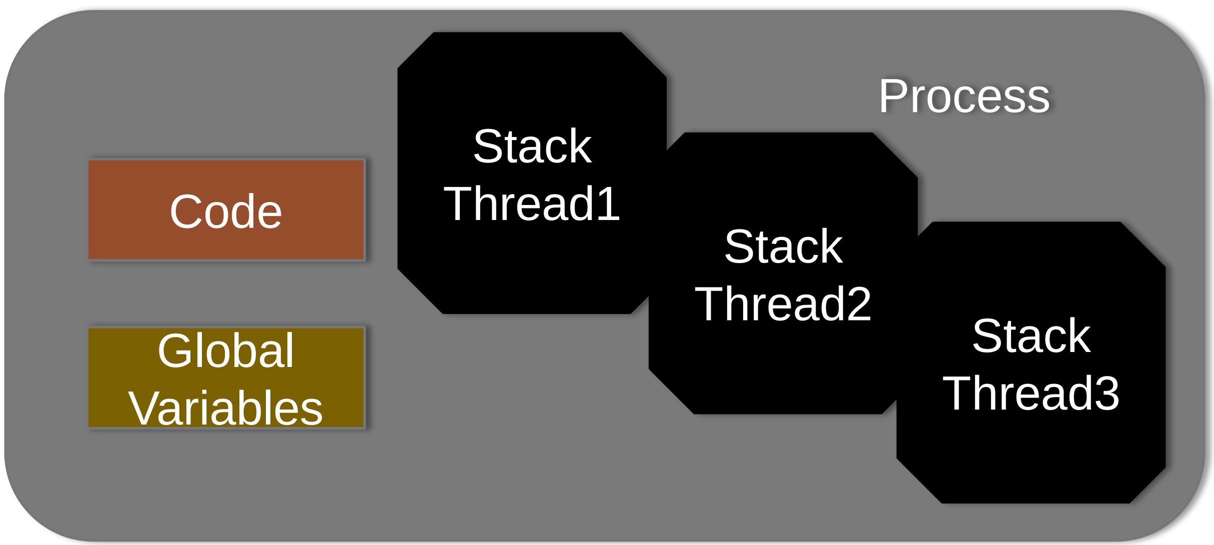
- **<u>Multiprogramming</u>**
  Program runs until it reaches instruction waiting for a peripheral, then its context is stored away and another program starts
- **<u>Cooperative multitasking</u>**
  Programs voluntarily cede time to one another
- **<u>Preemptive multitasking</u>**
  Operating system guarantees that each program gets time for execution + handling of interrupts (I/O)

FROM PROCESSES TO THREADS

- **Multitasking improved throughput of computers**
- **Developers started developing applications as sets of cooperating processes (e.g. one gets the input data, another performs calculations on the data, a third writes out the results) -> need for sharing data**
- **Threads born from the idea that most efficient way of sharing data was to share entire memory space**

WHAT IS A THREAD

- **Smallest unit of processing that can be scheduled by an operating system**
- **Threads are contained inside processes**
- **Threads of a process share memory and other resources**

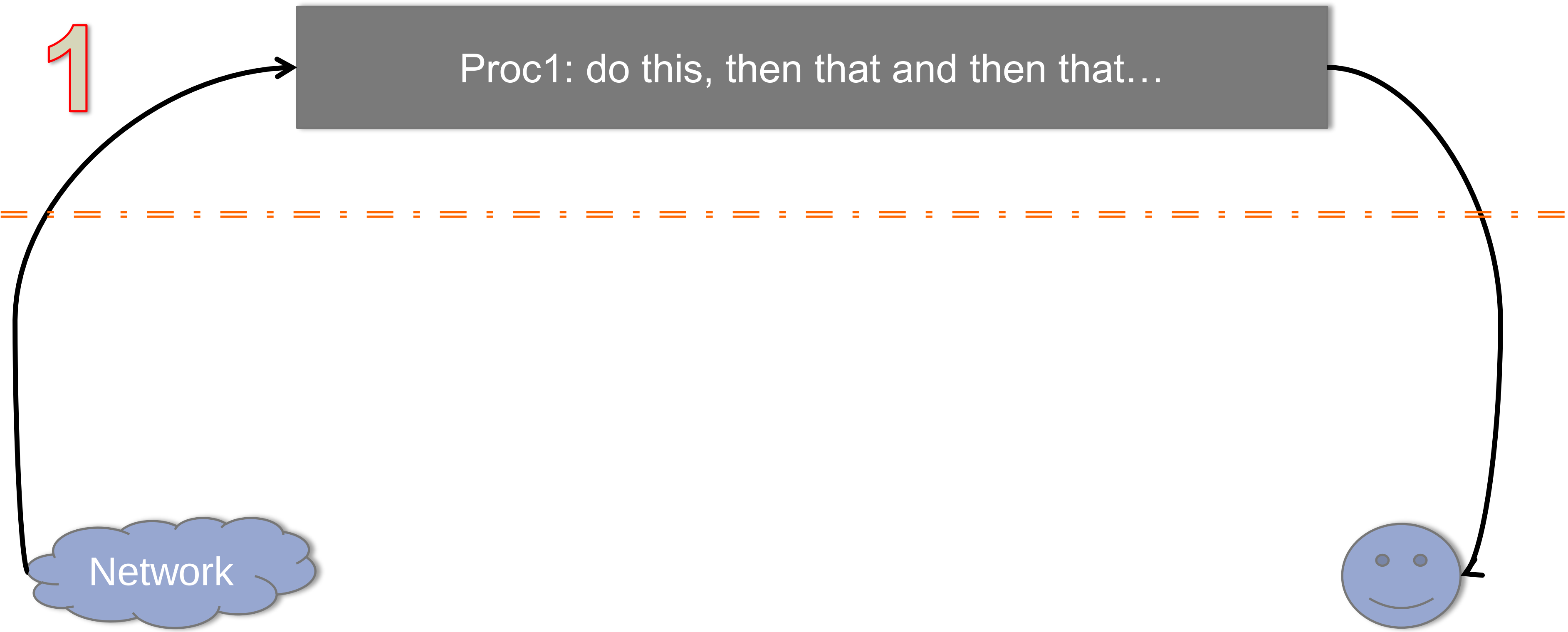


PROCESS VS THREAD

Each *process* provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the *primary thread*, but can create additional threads from any of its threads.

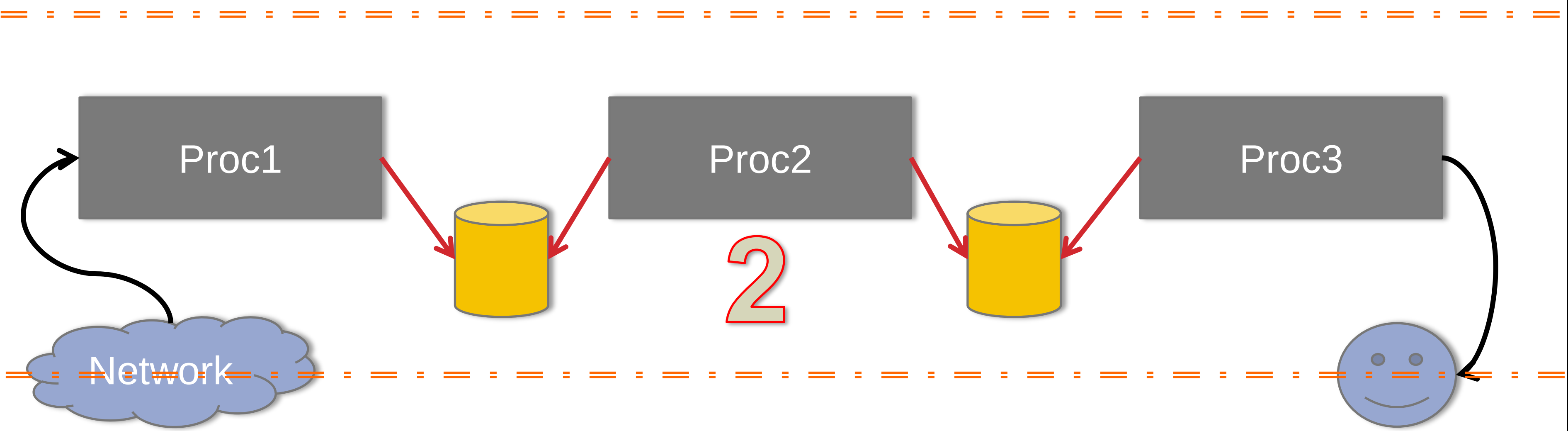
A *thread* is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The *thread context* includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

SW DESIGN EVOLUTION



- **Difficult to handle "de-randomization"….**
- **Code filled with loops and checks**

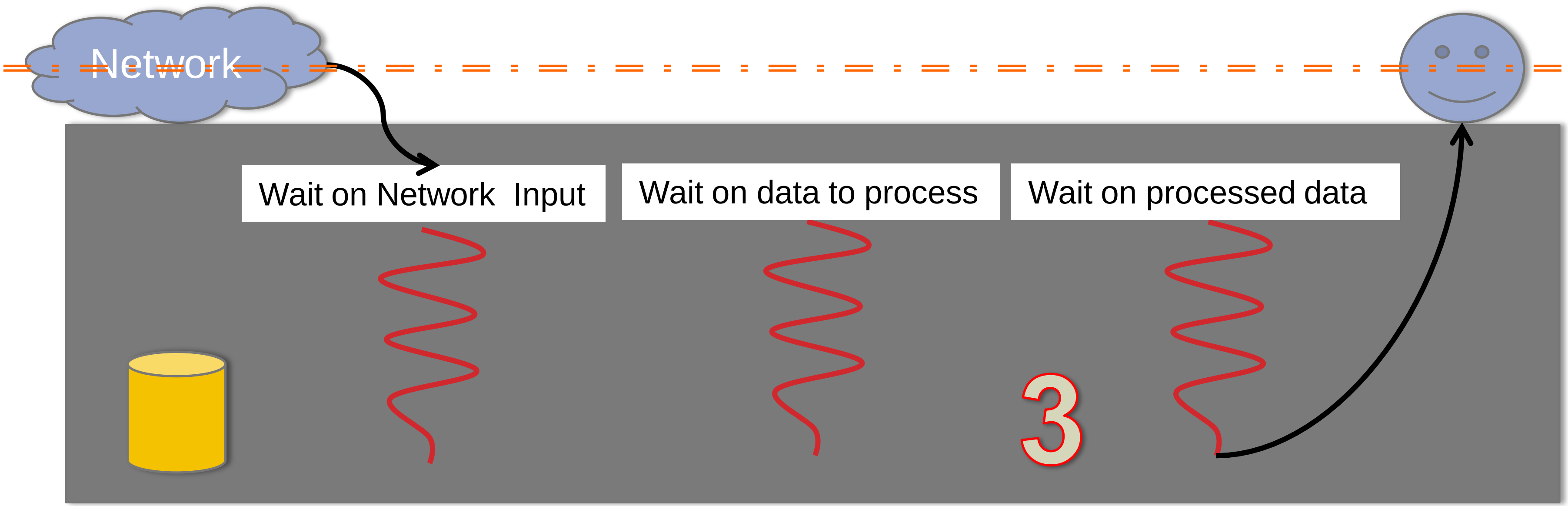
SW DESIGN EVOLUTION



- **Code better structured, cleaner**
- **Many data copies**

SW DESIGN EVOLUTION

- **Preserve code simplicity**
- **Optimize usage of resources (memory!)**



MULTI-THREADING

There are two different environments for multi-threading

- **Single processor: Interleaved usage of one resource by different tasks**
    - Actions "seem" to happen in parallel but they are actually sequential
    - Multithreading allows to design simpler, modular code with efficient use of resources
- **Multiple processors/cores: Parallel usage of the same type of resource by different tasks**
    - Threads are truly running AT THE SAME TIME
    - Multithreading allows the usage of multiple CPUs by one process!

PROS AND CONS OF THREADS

+

- **Exploiting multiple processors**
- **Simplicity of modeling**
- **Simplified handling of asynchronous commands**
- **More responsive user interfaces**

-

- **Safety hazards**
- **Liveness hazards**
- **Performance hazards**

"Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that non-determinism." -- 'The Problem with Threads, Edward A. Lee, UC Berkeley, 2006

THREAD SAFETY

- **Managing access to state and, in particular, to shared, mutable state.**
- **State: any data that can affect externally visible behavior of an object**
- **Shared: variable that can be accessed by several threads**
- **Mutable: value of a variable can change over time**

- **Writing thread safe code is about protecting data from uncontrolled concurrent access**

THREAD SAFETY

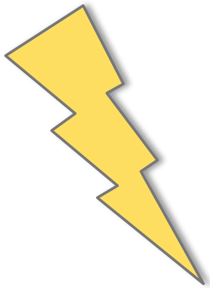
If different threads access the same mutable state variable without appropriate synchronization <u>your program is broken</u>.

To fix it:

- **Don't share state variables across threads**
- **Make state variables immutable**
- **Use synchronization whenever accessing the variable**

It is far easier to design a class to be thread-safe than to retrofit it for thread safety later.

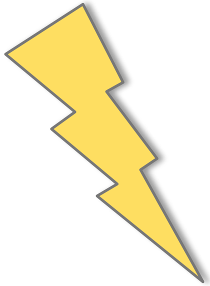
EXAMPLE 1

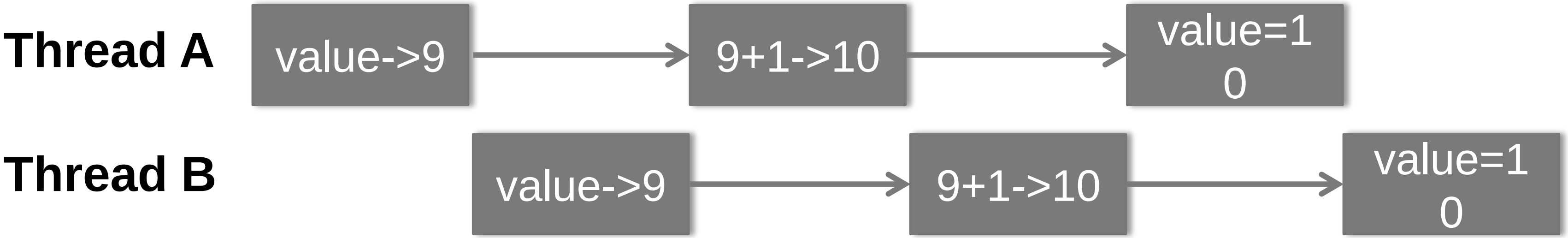


```
public class UnsafeSequence {
        private int value;
        /** Returns a unique value **/
        public int getNext() {
                return value++;
        }
}
```

EXAMPLE 1

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





We have missed one counter increase!!!

EXAMPLE 1: ISSUES

Compound actions, such as "check-then-act" or "read-modify-write" must be atomic in order to be thread safe.

Operations A and B are atomic if:

- **For a thread running A, if B is being executed by another thread, either all of B has executed or none of it.**

An atomic operation is atomic with respect to all operations, including itself, that operate on the same state.

See another example in the backup slides…

EXAMPLE 1 CONT

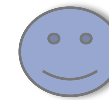
```
public class Sequence {
        private final AtomicLong value=
                                new AtomicLong(0);
        /** Returns a unique value **/
        public int getNext() {
                return value.incrementAndGet();
        }
}
```

For single variables atomic variable classes are provided.

Attention: using a thread safe state variable makes a class thread safe only if there is a single state variable!!

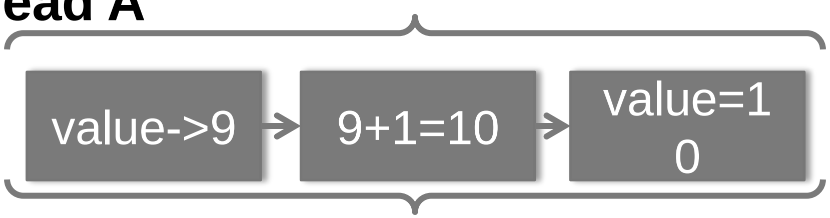
EXAMPLE 1 CONT

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

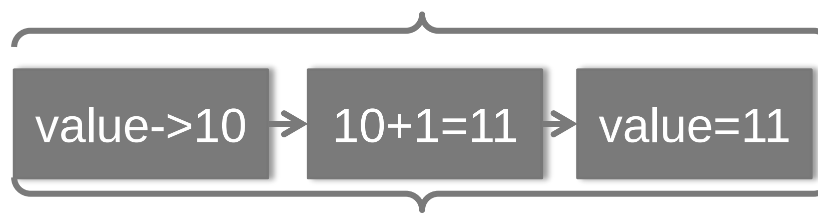


= acquire intrinsic lock from the beginning to the end of the method

Thread A

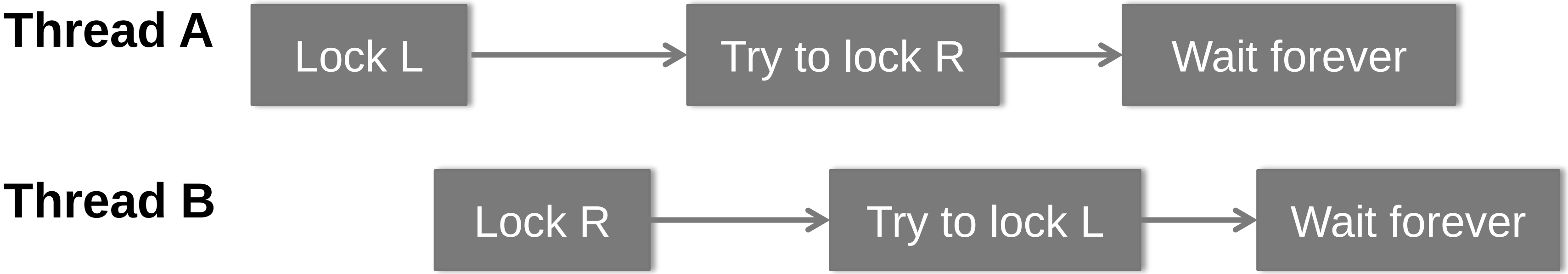


Thread B



LIVENESS

- **Thread safety requires access to state to be synchronized**
- **Abuse of locks can lead to deadlocks**
  - The story of the dining philosophers…



LIVENESS CONT

- **Resources deadlocks**
  - Example: you need access to 2 DBs to perform a task. One thread opens a connection to DB1 and another opens a connection to DB2. No thread can complete its task since they cannot acquire the resources they need.
- **Starvation**
  - Thread perpetually denied access to resources it needs, e.g. CPU
  - Example: bad choice of thread priorities, non-terminating constructs while holding a lock, …
- **Livelock**
  - A thread is not blocked but continues doing an operation that fails

PERFORMANCE

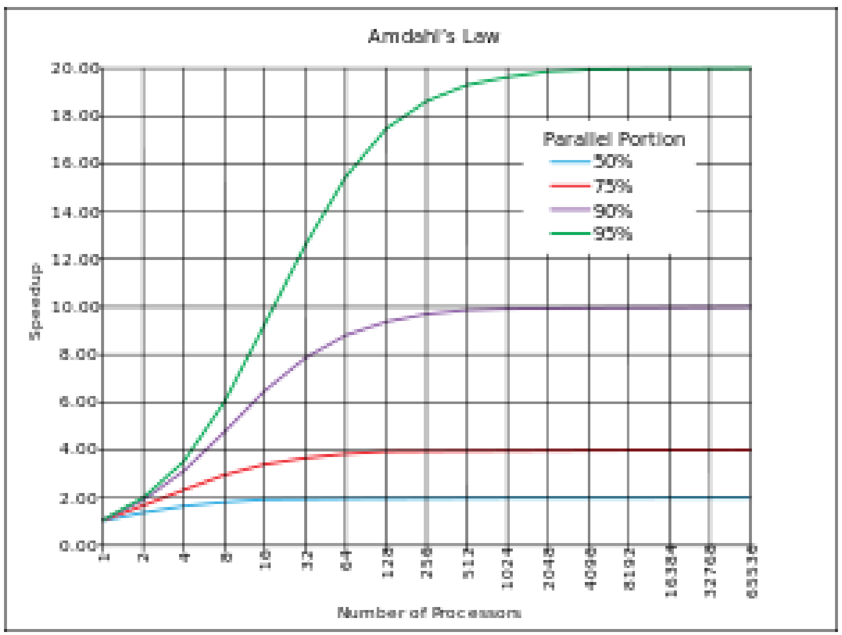
- **Moving to multithreaded programs you pay**
  - Synchronization
  - Scheduling (context switching)

- **Performance is not only speed**
  - Consider complexity of code for maintenance & testing

- **Performance is always a tradeoff: first make it right, then speed it up, if necessary!**

AMDAHL'S LAW

Multithreading can only help improving performance if problem can be decomposed in parts that can be executed in parallel!



The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20× as shown in the diagram, no matter how many processors are used.

$$speedup \text{ £ } \frac{1}{S + \frac{(1 - S)}{N_{proc}}}$$

THREAD SAFETY, LIVENESS & PERFORMANCE

- **For safety…**
  - Temptation of putting locks everywhere

- **Size of synchronized code blocks**
  - Tradeoff between safety and liveness/performance
  - Do NOT prematurely sacrifice simplicity (risk of compromizing safety) for the sake of performance

- **Avoid holding locks during lengthy computations or operations at risk of not completing quickly (e.g. network or console I/O)**

AND THIS WAS THE INTUITIVE BIT...

SHARING OBJECTS

- **Up to now we discussed how to AVOID concurrent access of data**
- **Now we'll see how to share objects so they can be safely accessed by multiple threads!**
- **A new element enters the game: visibility**
  - Ensure that when a thread modifies the state of an object other threads can actually see the changes

VISIBILITY – ONE THREAD

- **Single threaded environment:**

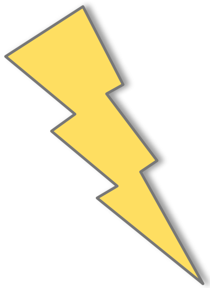
```
int number = 1;
boolean ready = true;
if(ready) {
        System.out.println(number);
}
```

- **The printed value will always be = 1**

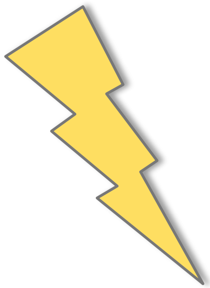
VISIBILITY - THREADS

```
public class NoVisibility {
		private static boolean ready;
		private static int number;

		private static class ReaderThread extends Thread {
				public void run() {
						while (!ready)
								Thread.yield();
						System.out.println(number);
				}
		}
		public static void main(String[] args) {
				new ReaderThread().start();
				number = 42;
				ready = true;
		}
}
```



VISIBILITY - THREADS



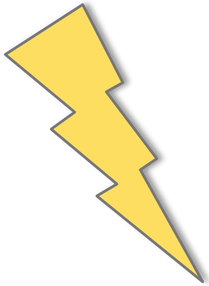
```
public class NoVisibility {
        private static boolean ready;
        private static int number;

        private static class ReaderThread extends Thread {
                public void run() {
                        while (!ready)
                                Thread.yield();
                        System.out.println(number);
                }
        }
        public static void main(String[] args) {
                new ReaderThread().start;
                number = 42;
                ready = true;
        }
}
```

NoVisibility **could return 0, because** ready **might become visible before** number**!**

NoVisibility **could loop forever, because** ready **might never become visible to the thread!**

EXAMPLE 2



```
public class MutableInteger {
        private int value;
        public int get() {return value;}
        public synchronized void set(int val)
                {this.value=val;}
}
```

If a thread uses set and another uses get, the getter might read stale values.

Synchronizing the set method is not sufficient!

REORDERING

- **No guarantee that operations happen in the order specified in the code (as long as the re-ordering is not visible from that thread).**
- **Another thread might always see things happening in a different order or not at all.**

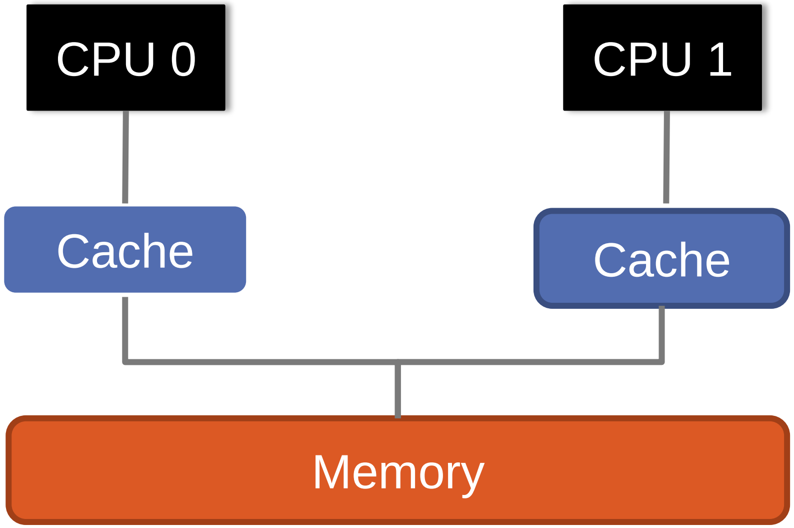
Why does reordering happen?

How do I tell my program to behave as I want?

MODERN COMPUTER ARCHITECTURES

- **CPUs much faster than memory systems**
  - ➔ Caches accessible in a few clock cycles
  - ➔ Cache coherency protocols to prevent inconsistent or lost data

- **If CPU 0 wants to write to a variable it has to invalidate all caches for it and wait for an acknowledge from all other CPUs before writing**

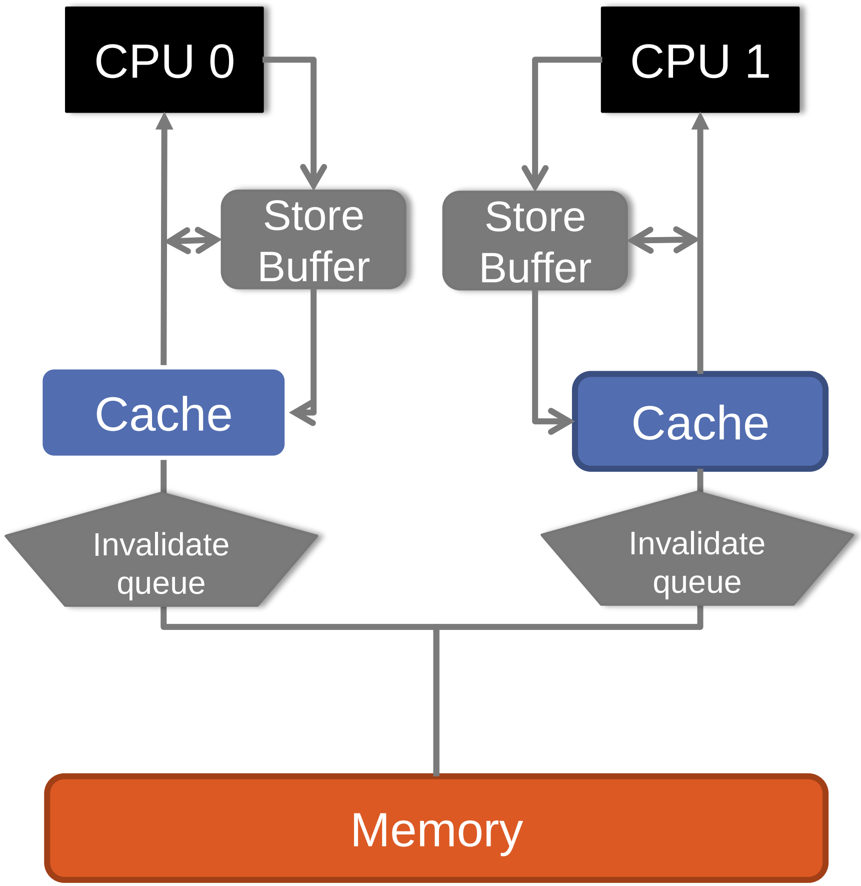


This architecture is not very efficient….

MODERN COMPUTER ARCHITECTURES

- **“Store buffers” and “invalidate queues”**
  - When CPU 0 wants to write it puts the variable in store buffer without waiting for cache invalidation of all other CPUs
  - When CPU 0 wants to write, CPU 1 can immediately send an invalidate acknowledge by putting the invalidate message in the invalidate queue.

- **This only works if instructions can be given that ensure that operations occur in the right order**



MEMORY BARRIERS

- **A memory barrier (fence instruction) is an instruction to enforce an ordering constraint on memory operations issued before and after the barrier instruction**
- **Normally, developers using high level languages don't use the low level memory barriers directly but use the synchronization operations offered by the language**
  - Doing otherwise would make code completely non-portable!
- **Languages which support multi-threading must thus provide a memory model**

MEMORY MODEL

- **A memory (consistency) model specifies the values that a shared variable read in a multithreaded program is allowed to return.**

- **A memory model describes**
  - how memory reads and writes may be executed by a processor relative to their program order, and
  - how writes by one processor may become visible to other processors

- **A memory model is an arbitration mechanism to determine how multiple threads access shared objects in memory.**

MEMORY MODEL

- **Two levels**
  - Compiler
  - Hardware

- **Three fundamental properties**
  - Atomicity: operations that are executed without interruption.
  - Ordering: a memory model determines what re-orderings are possible (relatively to program order).
  - Visibility: determines when other threads will see changes made by the current thread

MEMORY MODEL

A key role of the memory model is to define the tradeoff between

- **programmability (stronger guarantees for programmers)**
- **performance (greater flexibility for reordering program memory operations).**

EXAMPLE 2

```
public class SynchronizedInteger {
        private int value;
        public synchronized int get() {return value;}
        public synchronized void set(int val)
                {this.value=val;}
}
```

Locking can be used to guarantee that one thread sees the effects of another in a predictable manner.

Everything thread A did prior to a synchronized block is visible to thread B when it executes a synchronized block guarded by the same lock.

A WORD ON C++

- **Currently, multi-threaded C or C++ programs combine a single threaded programming language with a separate threads library (for UNIX pthreads)**
  - Strictly speaking not correct (http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf)
- **The new C++11 standard has a memory model**
  - The memory model for the hardware that will run the produced binary must not allow results that would be illegal for the C++ model applied to the original program.
  - The C++11 attempts to come up with something which will address all those issues while still being less constraining (and thus better performing) than Java's memory model.

CONCLUSIONS

- **Threaded programming allows**
  - Better factorized and simpler code implementation
  - Efficient use of resources
- **There are nevertheless some pitfalls**
  - Safety, liveness, performance
  - Visibility
- **It is very important to include thread support from the initial design of software**
  - Adding thread safety a posteriori can be a nightmare
- **The new C++11 standard evolved from the original C++; it incorporates a memory model and thus supports multi-threading at language level**

REFERENCES

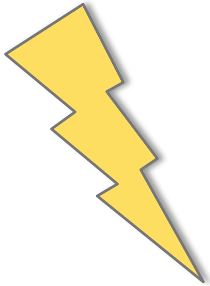
- **B Goetz, JAVA Concurrency in Practice, Addison Wesley ed.**
- **P. E. McKenney, Memory Barriers: a Hardware View for Software Hackers, http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2009.04.05a.pdf**
- **H-J Boehm, Threads Cannot be Implemented as a Library, HPL-2004-209, http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf**
- **Wikipedia:**
  - Parallel computing, Thread, Process (Computing), …

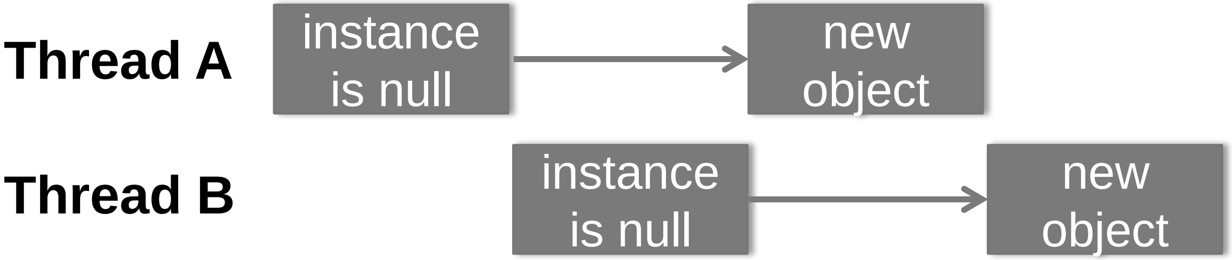
BACKUP

EXAMPLE 3

```
public class LazyInitRace{
        private ExpensiveObject instance=null;

        public ExpensiveObject getInstance() {
                if (instance == null)
                  instance = new ExpensiveObject();
                return instance;
        }
}
```





We end up with 2 different objects instead of 1!!!

EXAMPLE 3

```
public class LazyInit{
        private ExpensiveObject instance=null;

        public synchronized ExpensiveObject getInstance() {
                if (instance == null)
                  instance = new ExpensiveObject();
                return instance;
        }
}
```

Thread A

Thread B