Concurrency

T · · Systems

Saint Petersburg, 2019

Content

1 Hardware basics

2 Software basics

3 Java thread management

4 Concurrency general concept

5 Java concurrency implementation

6 Conclusion

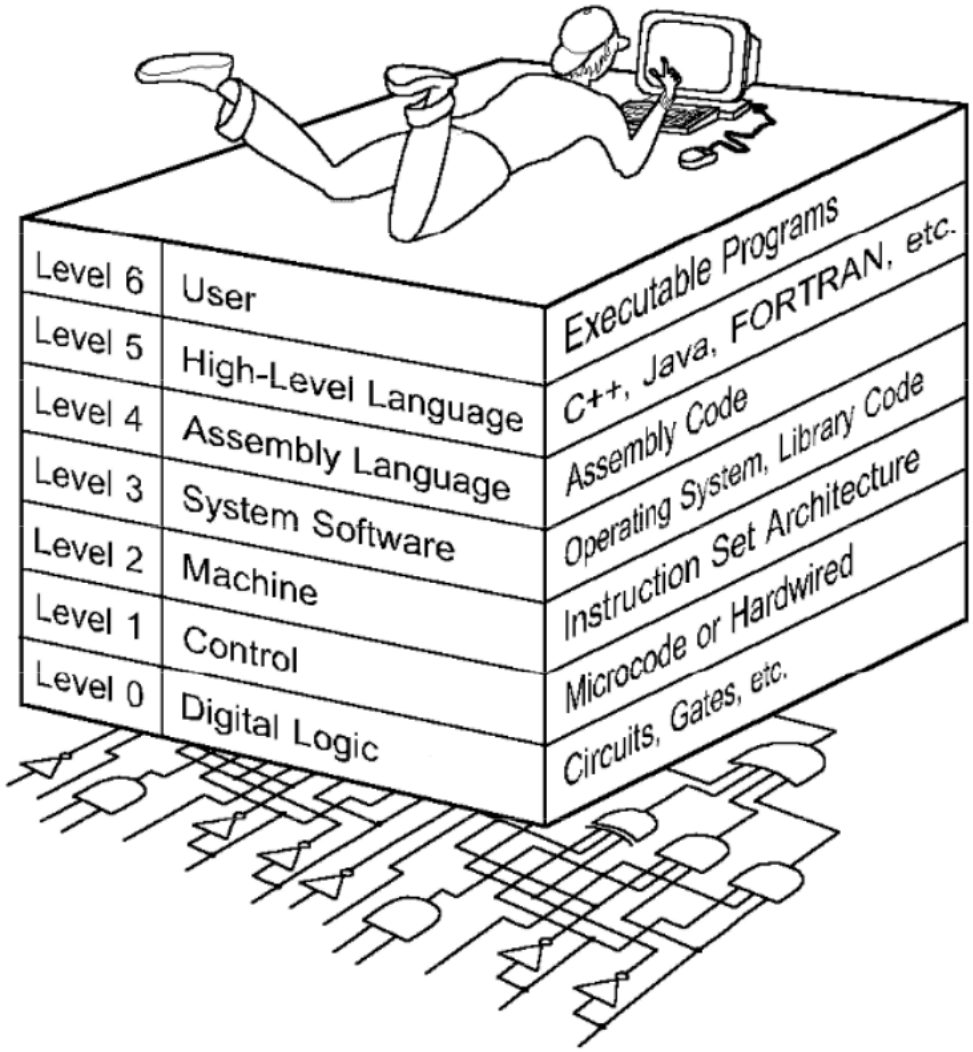
T · · Systems

Definition

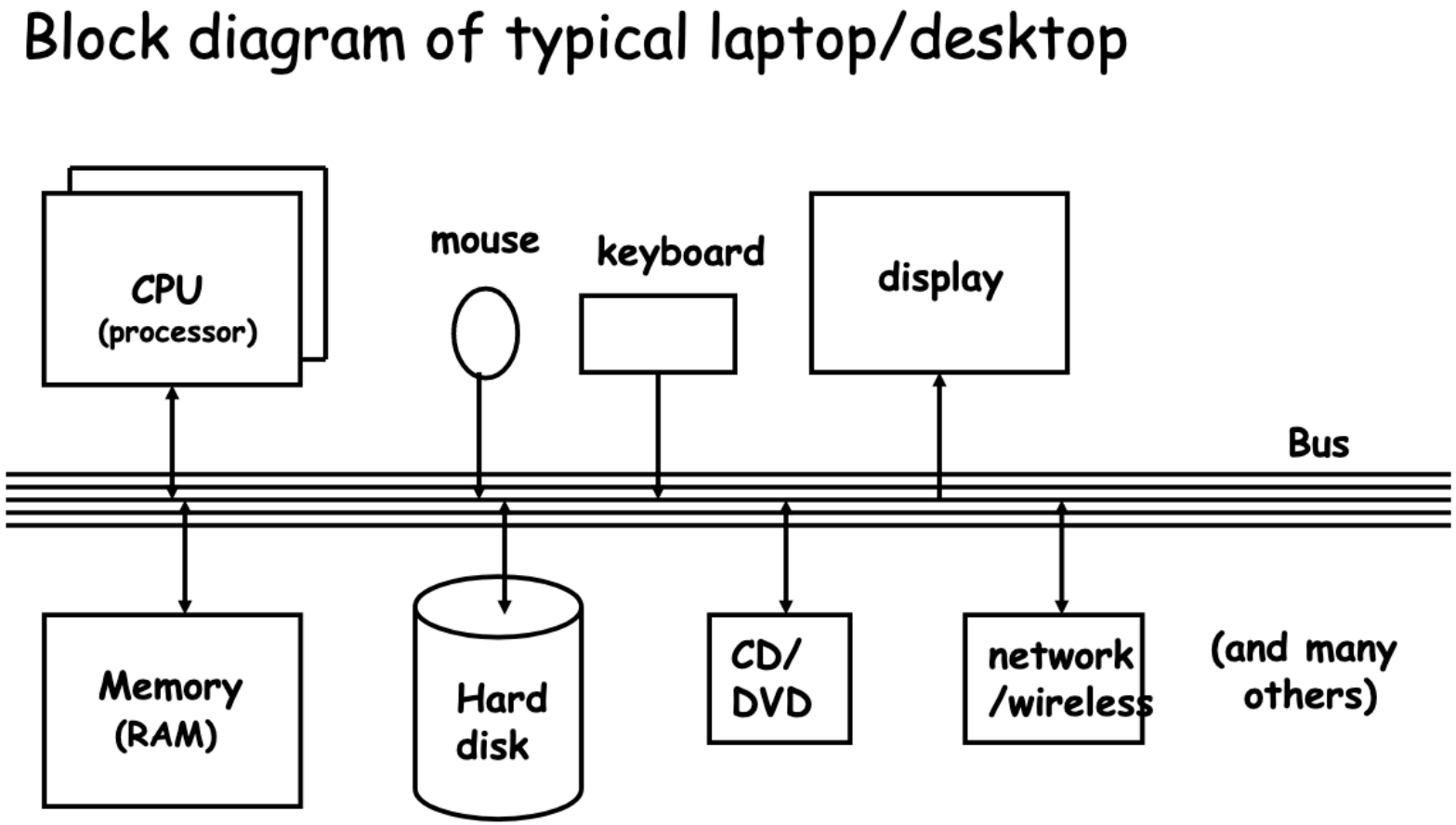
Concurrent computing is a form of computing in which several computations are executed during overlapping time periods—concurrently—instead of sequentially (one completing before the next starts)

Hardware basics

The World we are living in



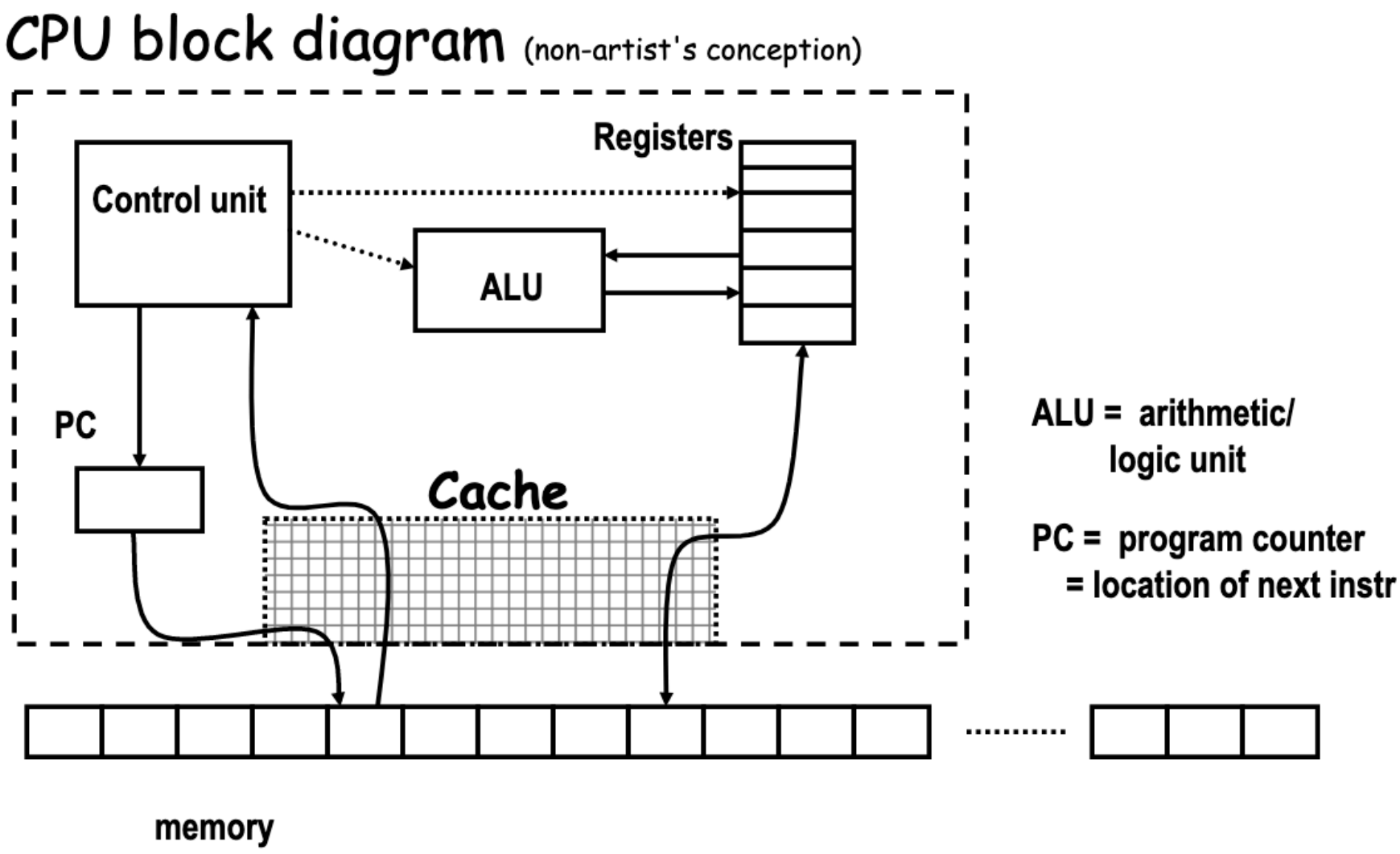
Computer architecture



CPU

- **can perform a small set of basic operations ("instructions")**
  - arithmetic: add, subtract, multiply, divide, …
  - memory access:
    fetch information from memory, store results back into memory
  - decision making: compare numbers, letters, …
    decide what to do next depending on result of previous computations
  - control the rest of the machine
    tell memory to send data to display; tell disk to read data from network; …
- **operates by performing sequences of simple operations *very* fast**
- **instructions to be performed are stored in the same memory as the data is**
  - instructions are encoded as numbers: e,g., Add = 1, Subtract = 2, …
- **CPU is a general-purpose device: putting different instructions into the memory makes it do a different task**
  - this is what happens when you run different programs

CPU



T · · Systems

How fast is fast?

- **CPU uses an internal "clock" (like a heartbeat) to step through instructions**
- **900 MHz, 2.5 GHz, etc., is the number of clock ticks per second**
  - 1 Hertz = 1 tick per second; abbreviated 1 Hz
  - mega = million
  - giga = billion
  - 1 MHz = 1 megaHertz = 1 million ticks per second
  - 1 GHz = 1 gigaHertz = 1 billion ticks per second = 1000 MHz
- **one instruction (like adding two numbers) might take one, two or several ticks, depending on design of the CPU**
  - might even complete more than one instruction in one tick
- **very rough approximations:**
  - PC/Mac processors execute about 2-3 billion instructions/sec
  - cellphone processors execute about 1-2 billion instructions/sec

Computer architecture

- **what instructions does the CPU provide?**
  - CPU design involves complicated tradeoffs among functionality, speed, complexity, programmability, power consumption, …
  - Intel and ARM are unrelated, totally incompatible
    - Intel: lot more instructions, many of which do complex operations
      - e.g., add two memory locations and store result in a third
    - ARM: fewer instructions that do simpler things, but faster
      - e.g., load, add, store to achieve same result
- **how is the CPU connected to the RAM and rest of machine?**
  - memory is the real bottleneck; RAM is slow (25-50 nsec to fetch)
    - modern computers use a hierarchy of memories (caches) so that frequently used information is accessible to CPU without going to memory
- **what tricks do designers play to make it go faster?**
  - overlap fetch, decode, and execute so several instructions are in various stages of completion (pipeline)
  - do several instructions in parallel
  - do instructions out of order to avoid waiting
  - multiple "cores" (CPUs) in one package to compute in parallel
- **speed comparisons are hard, not very meaningful**

CPU caching

Caching: making things seem faster than they are

- **cache: a small very fast memory for recently-used information**
  - loads a block of info around the requested info
- **CPU looks in the cache first, before looking in main memory**
  - separate caches for instructions and data
- **CPU chip usually includes multiple levels of cache**
  - faster caches are smaller
- **caching works because recently-used info is more likely to be used again soon**
  - therefore more likely to be in the cache already
- **cache usually loads nearby information at the same time**
  - nearby information is more likely to be used soon
  - therefore more likely to be in the cache when needed
- **this kind of caching is invisible to users**
  - except that machine runs faster than it would without caching

Bootstrapping: how does it all get started

- **CPU begins executing at specific memory location when turned on**
  - location is defined by the hardware: part of the machine's design
  - often in ROM (read-only memory) so not volatile but changeable

- **"bootstrap" instructions placed there read more instructions**
  - CPU tries to read first block from disk as bootstrap to copy more of the operating system
  - if that fails, tries to read bootstrap from somewhere else
    e.g., CD-ROM, USB, network, ...

Software basics

What an operating system does

- **manages CPUs, schedules and coordinates running programs**
  - switches CPU among programs that are actually computing
  - suspends programs that are waiting for something (e.g., disk, network)
  - keeps individual programs from hogging resources
- **manages memory (RAM)**
  - loads programs in memory so they can run
  - swaps them to disk and back if there isn't enough RAM (virtual memory)
  - keeps separate programs from interfering with each other
  - and with the operating system itself (protection)
- **manages and coordinates input/output to devices**
  - disks, display, keyboard, mouse, network, ...
  - keeps separate uses of shared devices from interfering with each other
  - provides uniform interface to disparate devices
- **manages files on disk (file system)**
  - provides hierarchy of directories and files for storing information

How does an operating system work?

- **loaded into RAM & started when machine is turned on ("boot")**
  - so it starts out being in charge / running on the bare hardware
- **gives control in turn to each program that is ready to run**
- **responds to external events / devices / …**
  - does actions, relays events to programs, …
- **programs (applications) request services by "making a system call"**
  - execute a particular instruction that transfers control to specific part of operating system
  - parameters say what task to do
- **OS does operation, returns control (and result) to application**

How to run programs?

To run programs, the operating system must

- **fetch program to be run (usually from disk)**
- **load it into RAM**
  - maybe only part, with more loaded as it runs (dynamic libraries)
- **transfer control to it**
- **provide services to it while it runs**
  - reading and writing info on disk
  - communications with other devices
- **regain control and recover resources when program is finished**
- **protect itself from errant program behavior**
- **share memory & other resources among multiple programs running "at the same time"**
  - manage memory, disks, network, ...
  - protect programs from each other
  - manage allocation of CPUs among multiple activities

What is a Program?

Program is a file containing:

- executable code (machine instructions)
- data (information manipulated by these instructions)

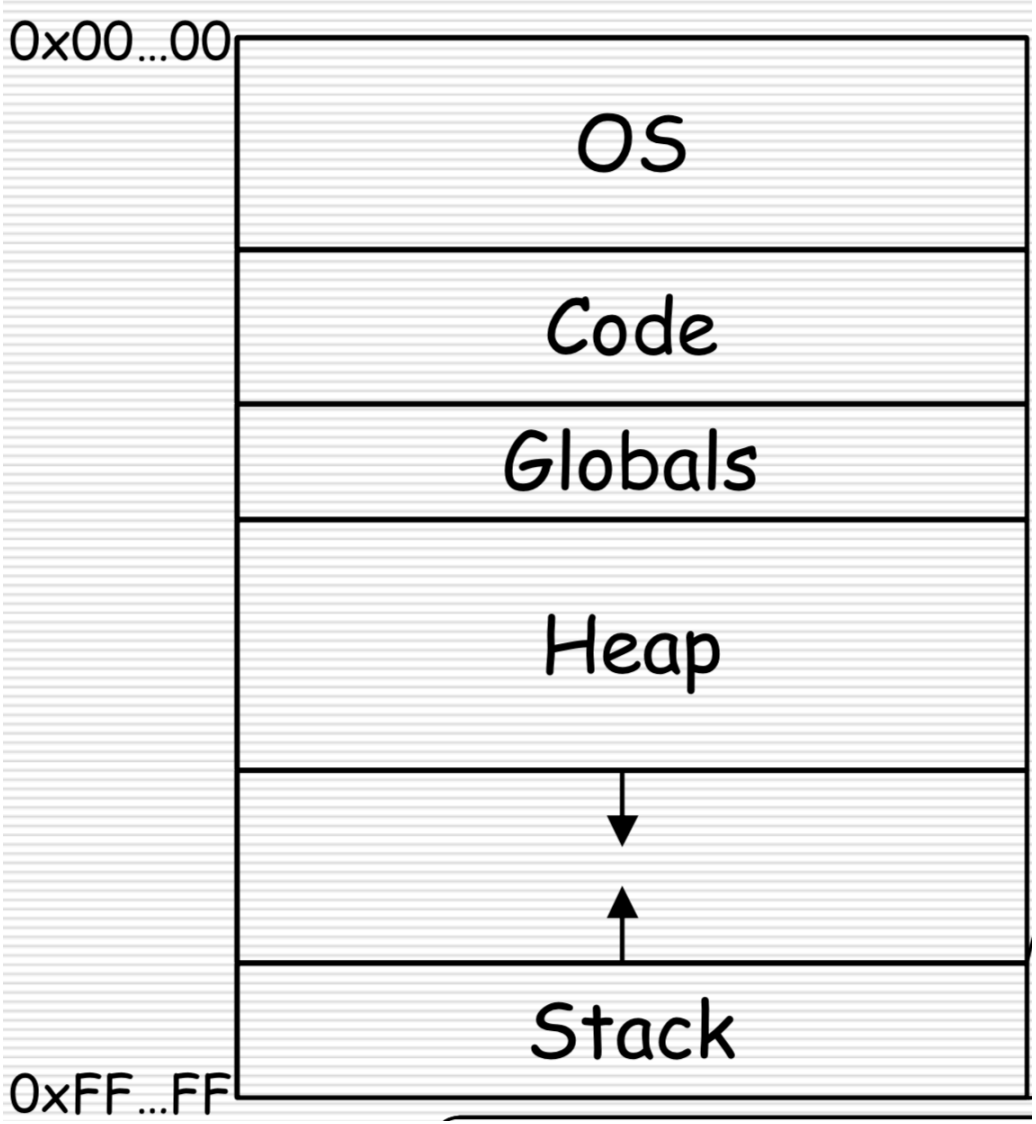
that together describe a computation

- Resides on disk
- Obtained via compilation & linking

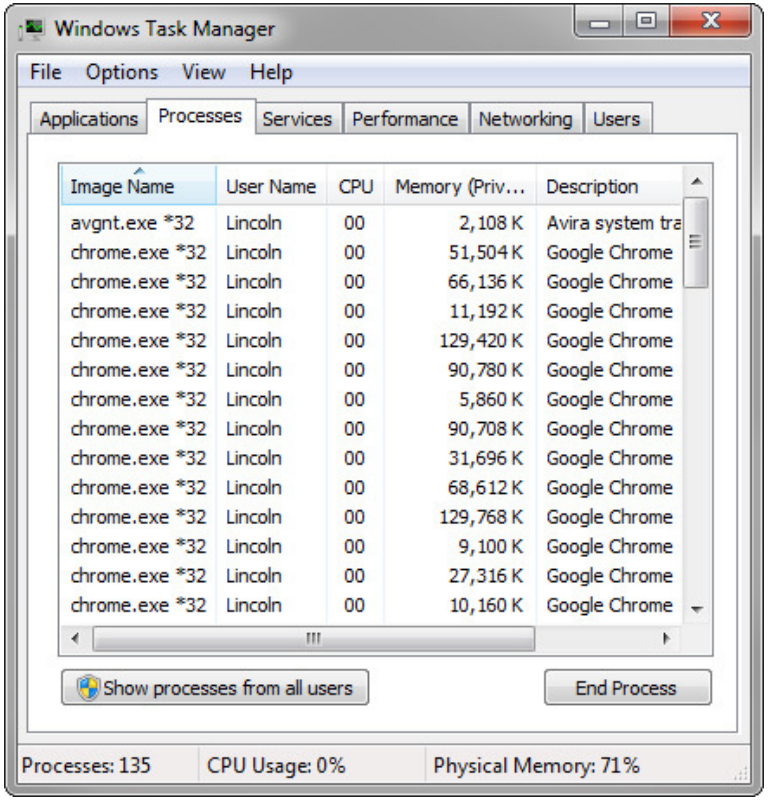
Process

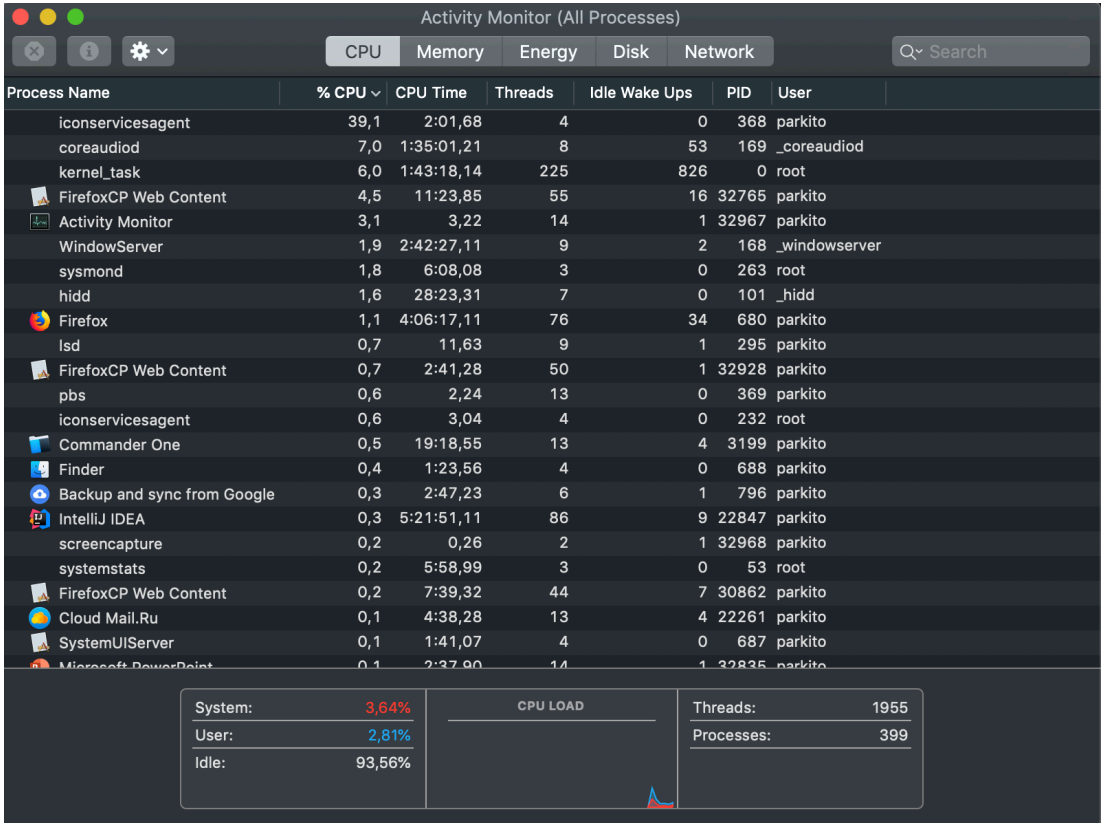
- A process is a name given to a program instance that has been loaded into memory and managed by the operating system
- Process address space is generally organized into *code*, *data (static/global)*, *heap*, and *stack* segments
- Every process in execution works with:
  - Registers: PC, Working Registers
  - Call stack (Stack Pointer Reference)

Process

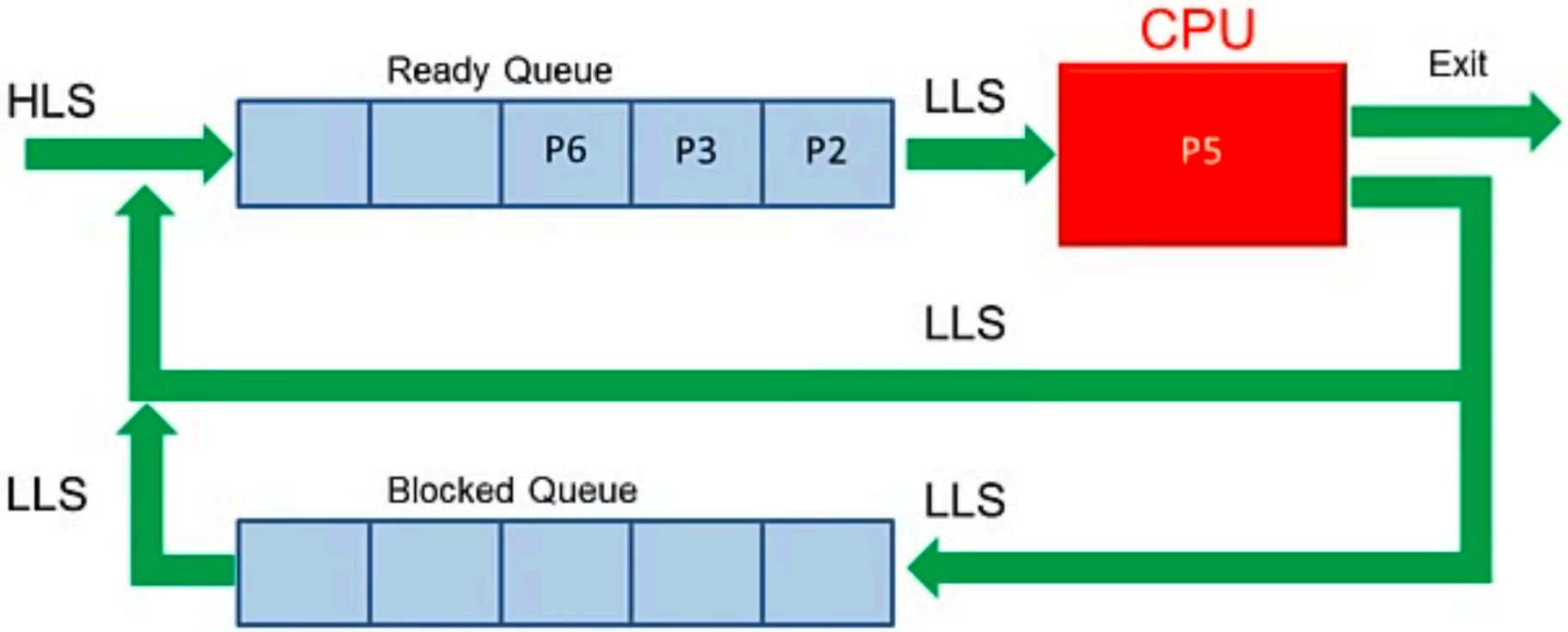


Processes


Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Image Name | User Name | CPU | Memory (Priv... | Description
avgnt.exe *32 | Lincoln | 00 | 2,108 K | Avira system tra
chrome.exe *32 | Lincoln | 00 | 51,504 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 66,136 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 11,192 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 129,420 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 90,780 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 5,860 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 90,708 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 31,696 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 68,612 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 129,768 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 9,100 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 27,316 K | Google Chrome
chrome.exe *32 | Lincoln | 00 | 10,160 K | Google Chrome
Show processes from all users
End Process
Processes: 135 CPU Usage: 0% Physical Memory: 71%

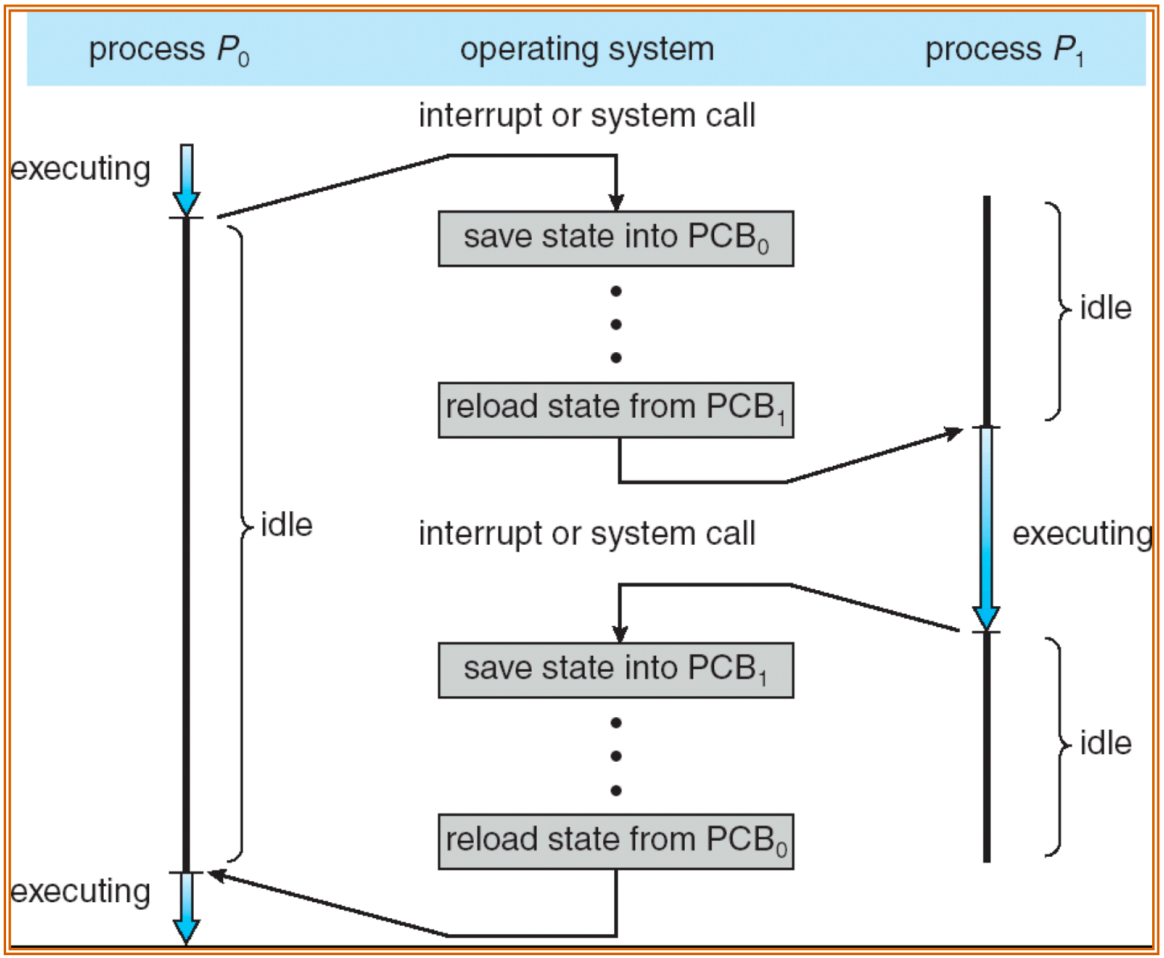

Activity Monitor (All Processes)
CPU Memory Energy Disk Network
Search
Process Name | % CPU | CPU Time | Threads | Idle Wake Ups | PID | User
iconservicesagent 39,1 2:01,68 4 0 368 parkito
coreaudiod 7,0 1:35:01,21 8 53 169 _coreaudiod
kernel_task 6,0 1:43:18,14 225 826 0 root
FirefoxCP Web Content 4,5 11:23,85 55 16 32765 parkito
Activity Monitor 3,1 3,22 14 1 32967 parkito
WindowServer 1,9 2:42:27,11 9 2 168 _windowserver
sysmond 1,8 6:08,08 3 0 263 root
hidd 1,6 28:23,31 7 0 101 _hidd
Firefox 1,1 4:06:17,11 76 34 680 parkito
lsd 0,7 11,63 9 1 295 parkito
FirefoxCP Web Content 0,7 2:41,28 50 1 32928 parkito
pbs 0,6 2,24 13 0 369 parkito
iconservicesagent 0,6 3,04 4 0 232 root
Commander One 0,5 19:18,55 13 4 3199 parkito
Finder 0,4 1:23,56 4 0 688 parkito
Backup and sync from Google 0,3 2:47,23 6 1 796 parkito
IntelliJ IDEA 0,3 5:21:51,11 86 9 22847 parkito
screencapture 0,2 0,26 2 1 32968 parkito
systemstats 0,2 5:58,99 3 0 53 root
FirefoxCP Web Content 0,2 7:39,32 44 7 30862 parkito
Cloud Mail.Ru 0,1 4:38,28 13 4 22261 parkito
SystemUIServer 0,1 1:41,07 4 0 687 parkito
System: 3,64%
User: 2,81%
Idle: 93,56%
CPU LOAD
Threads: 1955
Processes: 399

Process scheduler



T · · Systems

CPU Switch From Process to Process



Context switch steps:

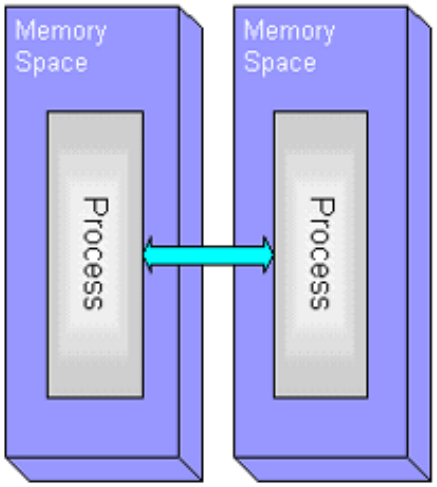
1. Save current process to PCB
2. Decide which process to run
3. Reload of new process from PCB

Context switch should be fast, because it is overhead.

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is *overhead*; the system does no useful work while switching
- Time dependent on hardware support
  - Hardware designers try to support routine context-switch actions like saving/restoring all CPU registers by one pair of machine instructions

Inter-process communications



① Write it to a file

② Send it over the local network

(with a HTTP request or something)

③ Use a pipe!

'program1 | program2'

cool thing: you get buffering automatically

cool thing: you can easily switch to having the 2 programs on different machines.

④ shared memory

processes can share memory, not just threads on the same process!

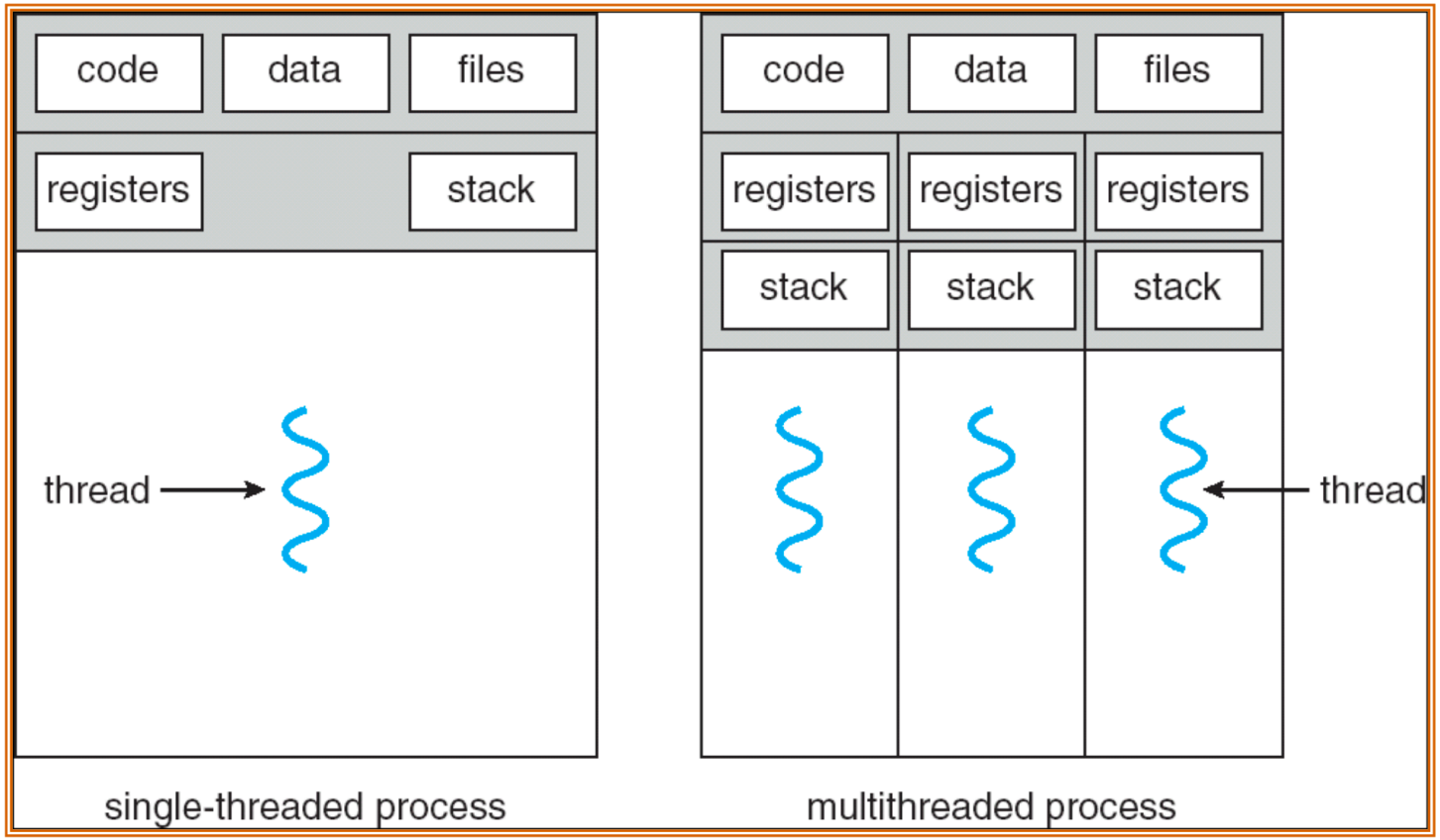
④ Unix domain sockets

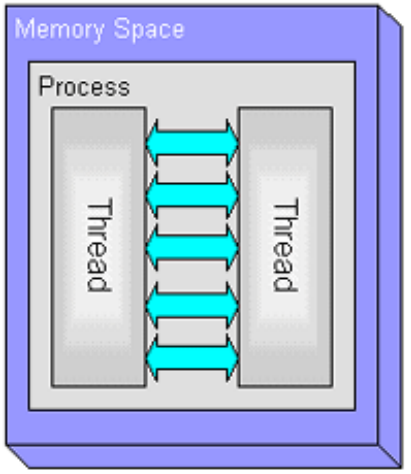
Another way to send a stream of data.

T · · Systems

Threads

A **thread** is a single sequential flow of control in a program.



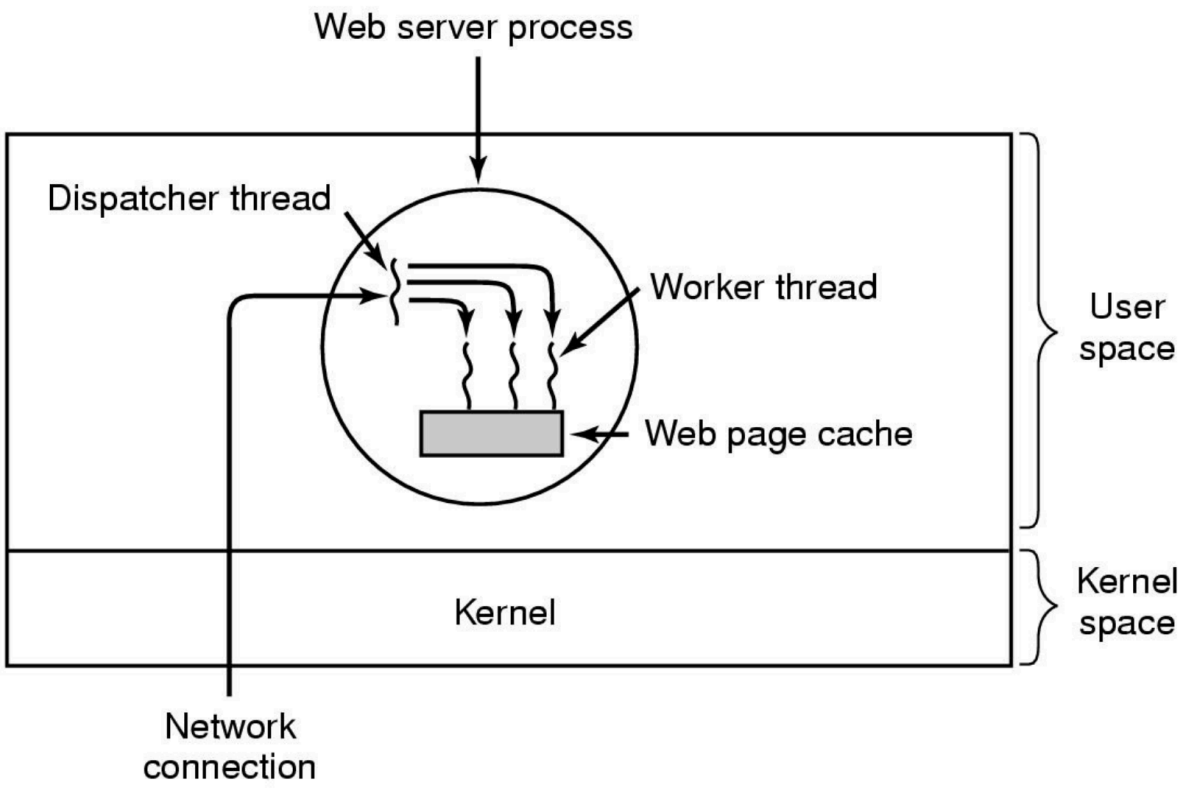


T · · Systems

Threads

- Threads have their own identity (thread ID), and can function independently.
- They share the address space within the process, and reap the benefits of avoiding any IPC (Inter-Process Communication) channel (shared memory, pipes and so on) to communicate.
- Threads of a process can directly communicate with each other
  - for example, independent threads can access/update a global variable.
- This model eliminates the potential IPC overhead that the kernel would have had to incur. As threads are in the same address space, a thread context switch is inexpensive and fast.

Thread Usage



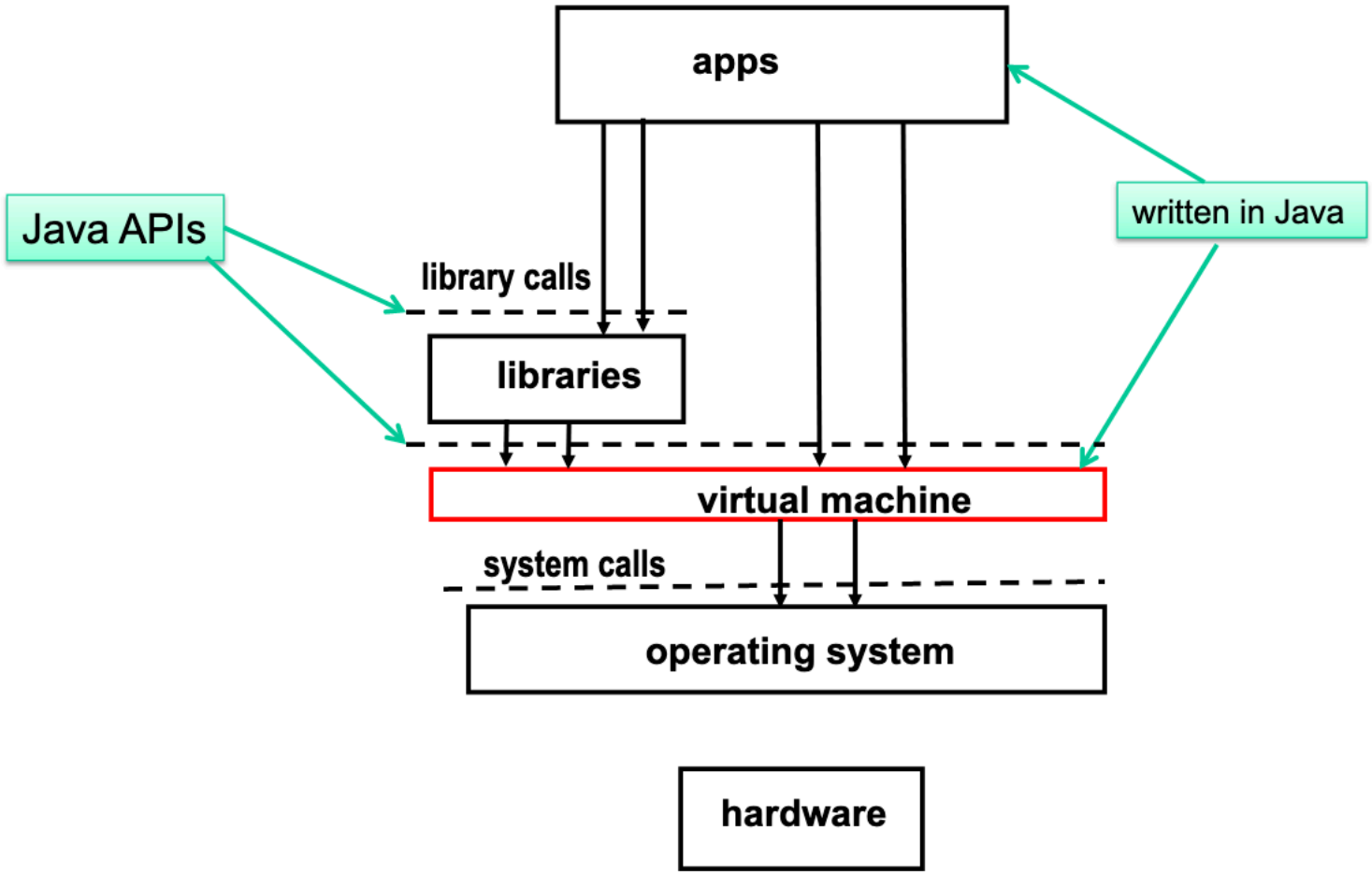
Threads

PROCESS	THREAD
An instance of a computer program that is being executed	A component of a process which is the smallest execution unit
Heavyweight	Lightweight
Switching requires interacting with the operating system	Switching does now require interacting with the operating system
Each has its own memory space	Use the memory of the process they belong to
Requires more resources	Requires minimum resources
Difficult to create a process	Easier to create
Inter-process communication is slow because each process has a different memory address	Inter-thread communication is fast because the threads share the same memory address of the process they belong to
In a multi-processing environment, each process executes independently	A thread can read, write or modify data of another thread

T · · Systems

Java basics

JVM execution architecture



T · · Systems

Thread creation

Threads can be declared in one of two ways:

(1) by extending the `Thread` class

(2) by implementing the `Runnable` interface

Thread creation

```
public class MyThread extends Thread {
    public void run() {
        // the thread body
    }

    // other methods and fields
}
```

Creating and starting a thread:

```
new MyThread().start();
```

Thread creation

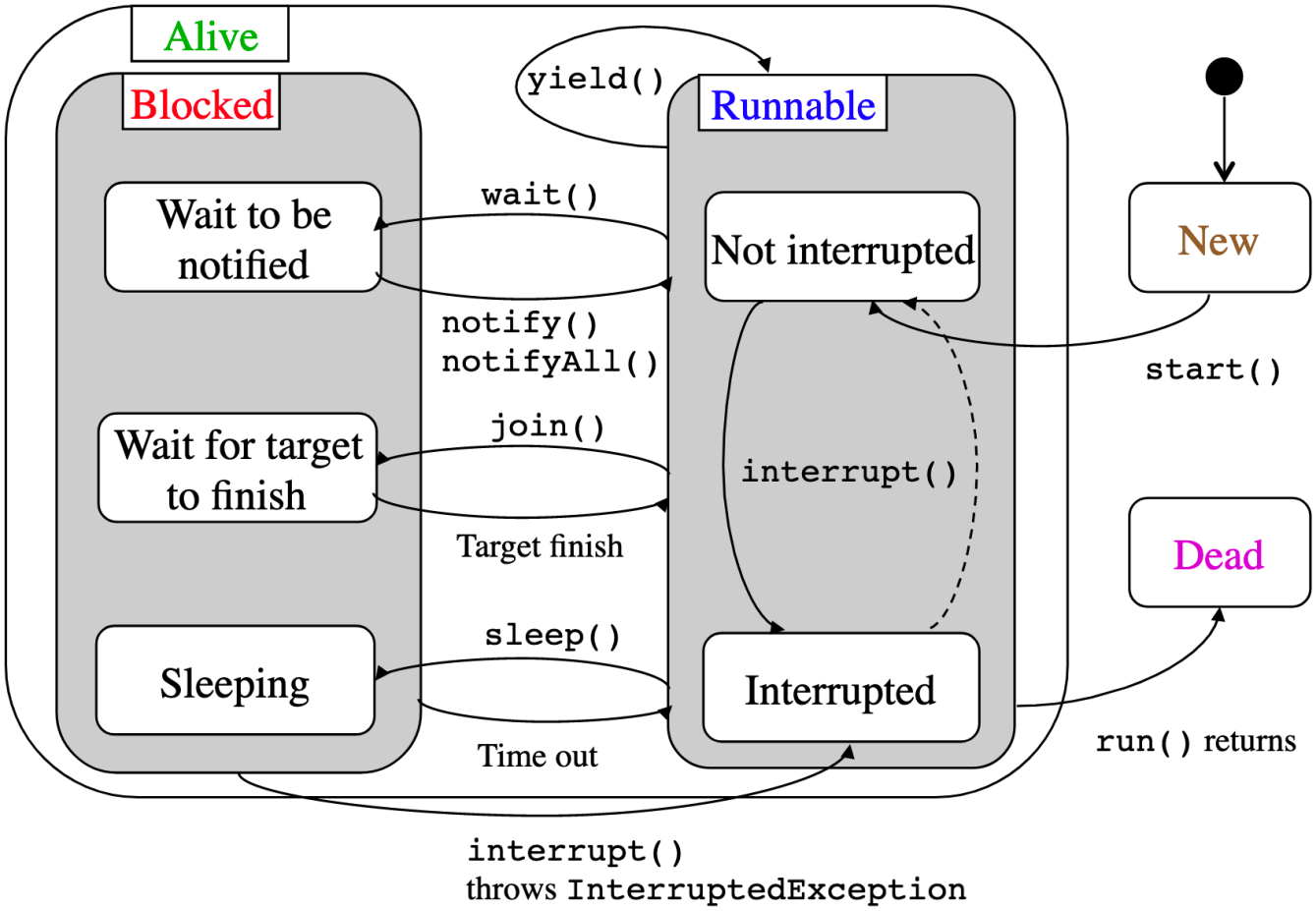
```
public class MyRunnable implements Runnable

{
    public void run() {
        // the thread body
    }

    // other methods and fields
}
```

Creating and starting a thread:

```
new Thread(new MyRunnable()).start();
```

Example 1,2,3

Thread life cycle



Race condition

When multiple processes are accessing shared data without **access control** the final result depends on the execution order creating what we call **race conditions.**

- A serious problem for any concurrent system using shared variables!

```
public class Account {
    // ...
    public boolean withdraw(long amount) {
        if (amount <= balance) {
            long newBalance = balance - amount;
            balance = newBalance;
            return true;
        }
        return false;
    }

    private long balance;
}
```

T · · Systems

Race condition

Suppose that two threads A and B have access to a shared variable "Balance":

Thread A:	Thread B:
Balance = Balance - 100	Balance = Balance - 200

Further, assume that Thread A and Thread B are executing concurrently in a time-shared, multi-programmed system.

Race condition

- The statement " Balance = Balance – 100" is implemented by several machine level instructions such as:
  - A1. LOAD R1, BALANCE // load Balance from memory into Register 1 (R1)
  - A2. SUB R1, 100 // Subtract 100 from R1
  - A3. STORE BALANCE, R1 // Store R1's contents back to the memory location of Balance.
- Similarly, "Balance = Balance – 200" can be implemented by the following:
  - B1. LOAD R1, BALANCE
  - B2. SUB R1, 200
  - B3. STORE BALANCE, R1

T · · Systems

Race condition

Observe: In a *time-shared* or multi-processing system the *exact instruction execution order* cannot be predicted!

Scenario 1:

A1. LOAD R1, BALANCE

A2. SUB R1, 100

A3. STORE BALANCE, R1

Context Switch!

B1. LOAD R1, BALANCE

B2. SUB R1, 200

B3. STORE BALANCE, R1

Balance is effectively decreased by 300!

Scenario 2:

A1. LOAD R1, BALANCE

A2. SUB R1, 100

Context Switch!

B1. LOAD R1, BALANCE

B2. SUB R1, 200

B3. STORE BALANCE, R1

Context Switch!

A3. STORE BALANCE, R1

Balance is effectively decreased by 100!

Critical region (section)

A **critical region** is a section of program code that should be executed by only one thread at a time.

Java provides a *synchronization* mechanism to ensure that, while a thread is executing statements in a critical region, no other thread can execute statements in the same critical region at the same time.

Synchronization may be applied to methods or a block of of statements.

T Systems

Lock (monitor)

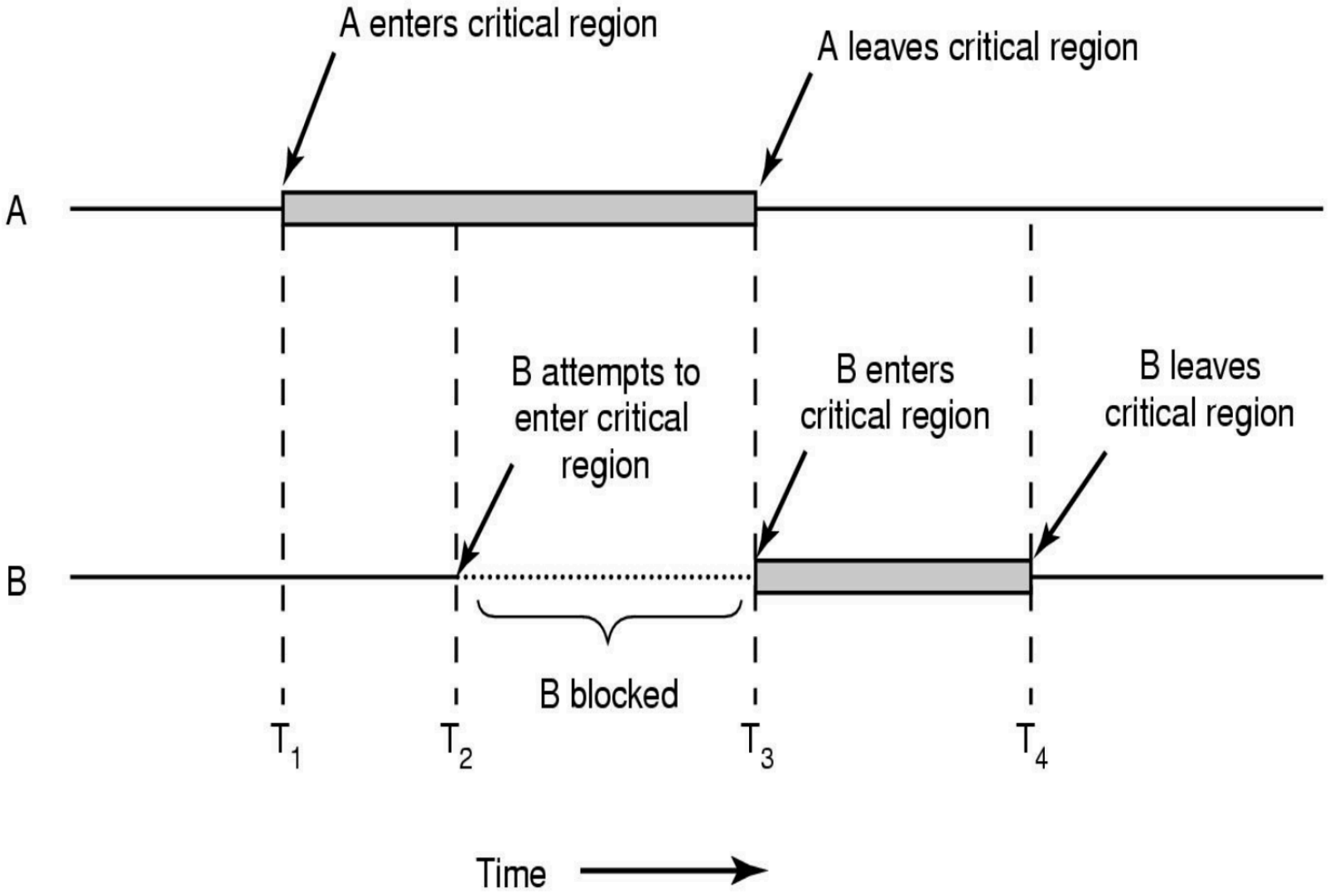
The synchronization mechanism is implemented by associating each object with a **lock**.

A thread must obtain *exclusive possession* of a lock before entering a critical region.

- For a synchronized instance method, the lock associated with the receiving object `this` is used.

- For a synchronized block, the lock associated with the result of the expression `exp` is used.

Lock (monitor)



T · · Systems

Example 4

Synchronization

- Coordinate processes all competing to access shared data

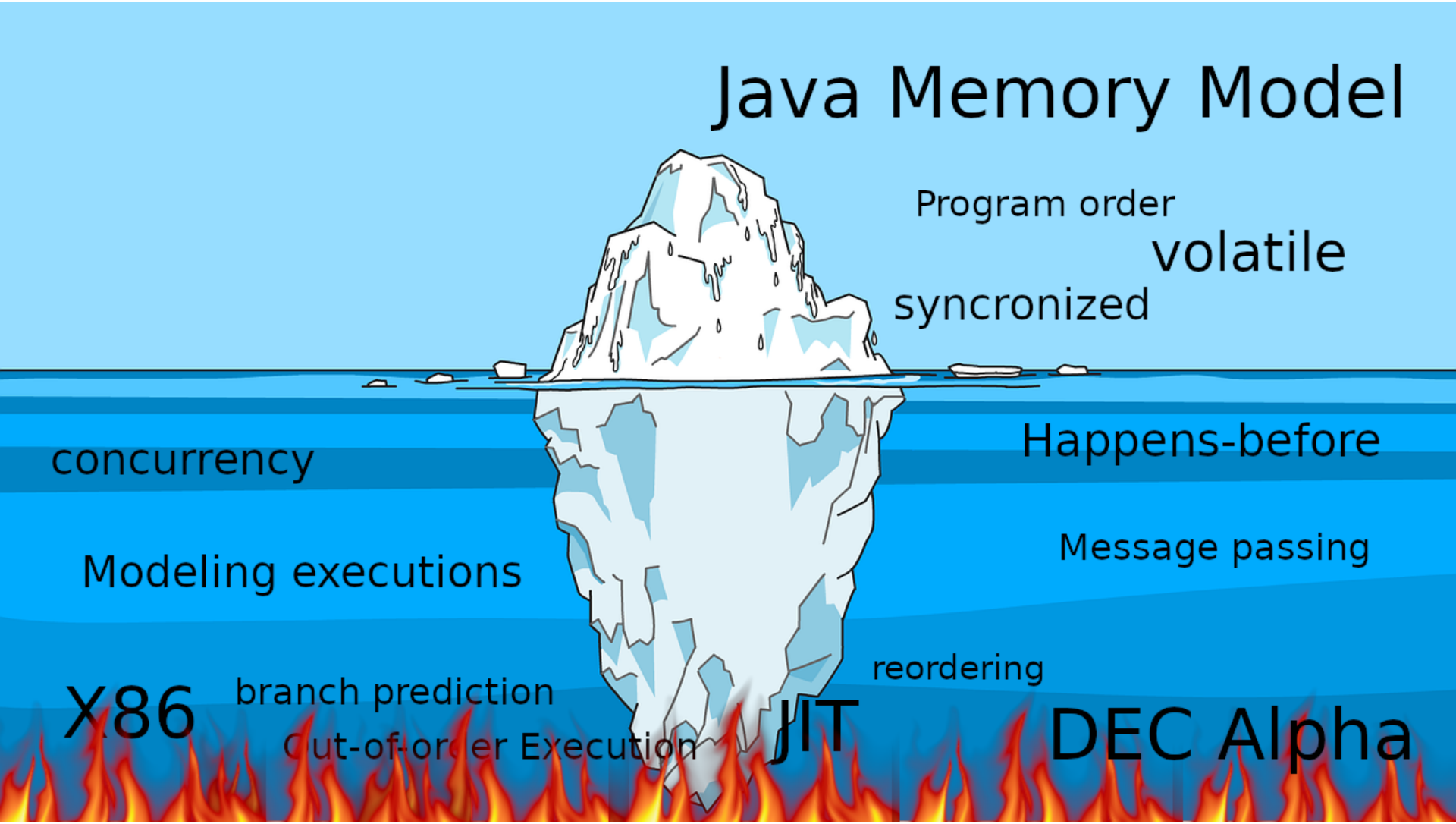
- Each process has a code segment, called critical section (critical region), in which the shared data is accessed.

- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in that critical section.

- The execution of the critical sections by the processes is **mutually exclusive.**

- **Critical section should be Short and with Bounded Waiting**

JMM


Java Memory Model
Program order
volatile
syncronized
concurrency
Happens-before
Modeling executions
Message passing
reordering
X86
branch prediction
Out-of-order Execution
JIT
DEC Alpha

WIKIPEDIA
The Free Encyclopedia

Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

Java memory model

From Wikipedia, the free encyclopedia

The **Java memory model** describes how threads in the Java programming language interact through memory. Together with the description of single-threaded execution of code, the memory model provides the semantics of the Java programming language.

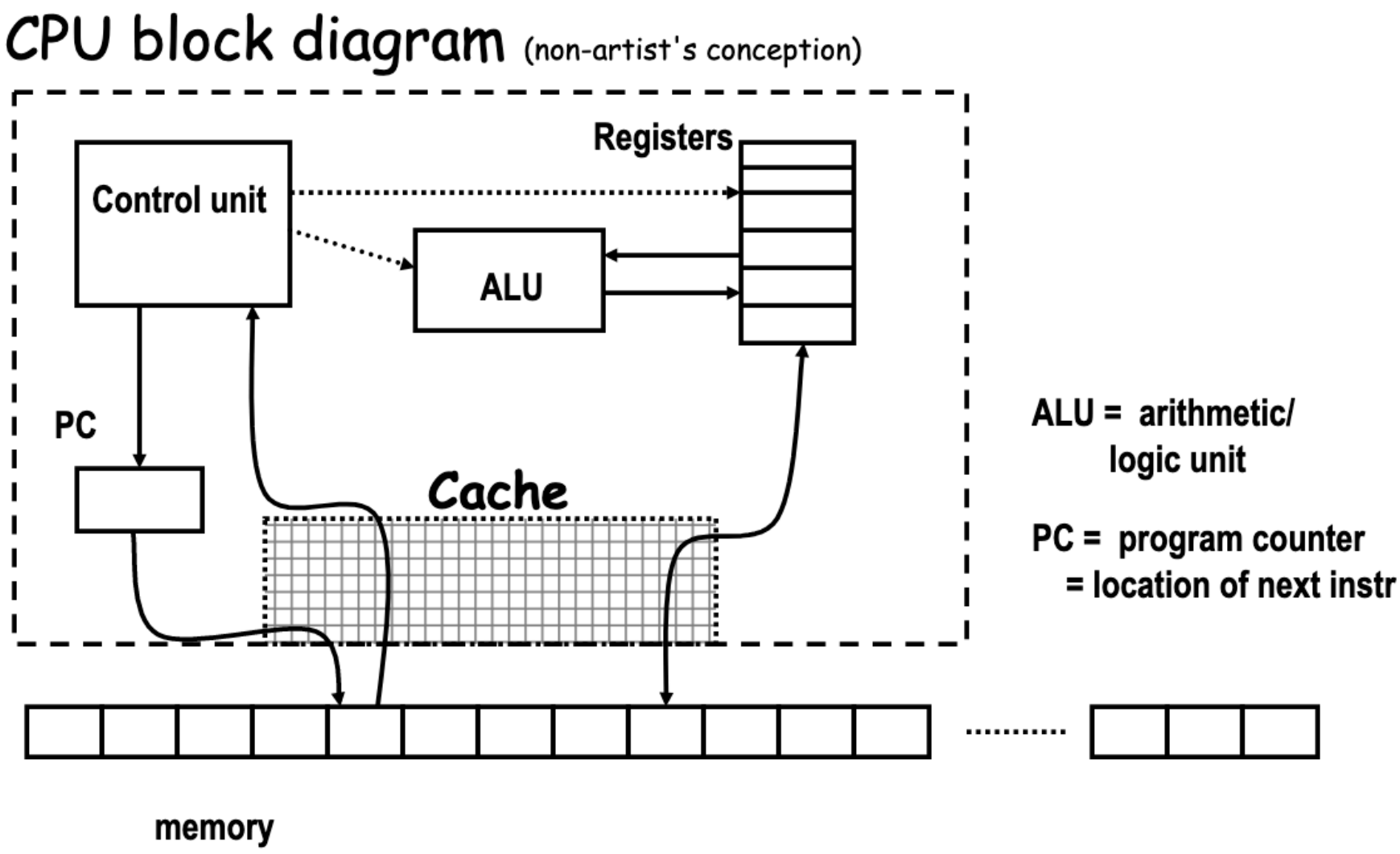
Main page
Contents
Featured content

Model properties

The Java Memory Model specifies when values must be transferred between main memory and per-thread *working memory*:

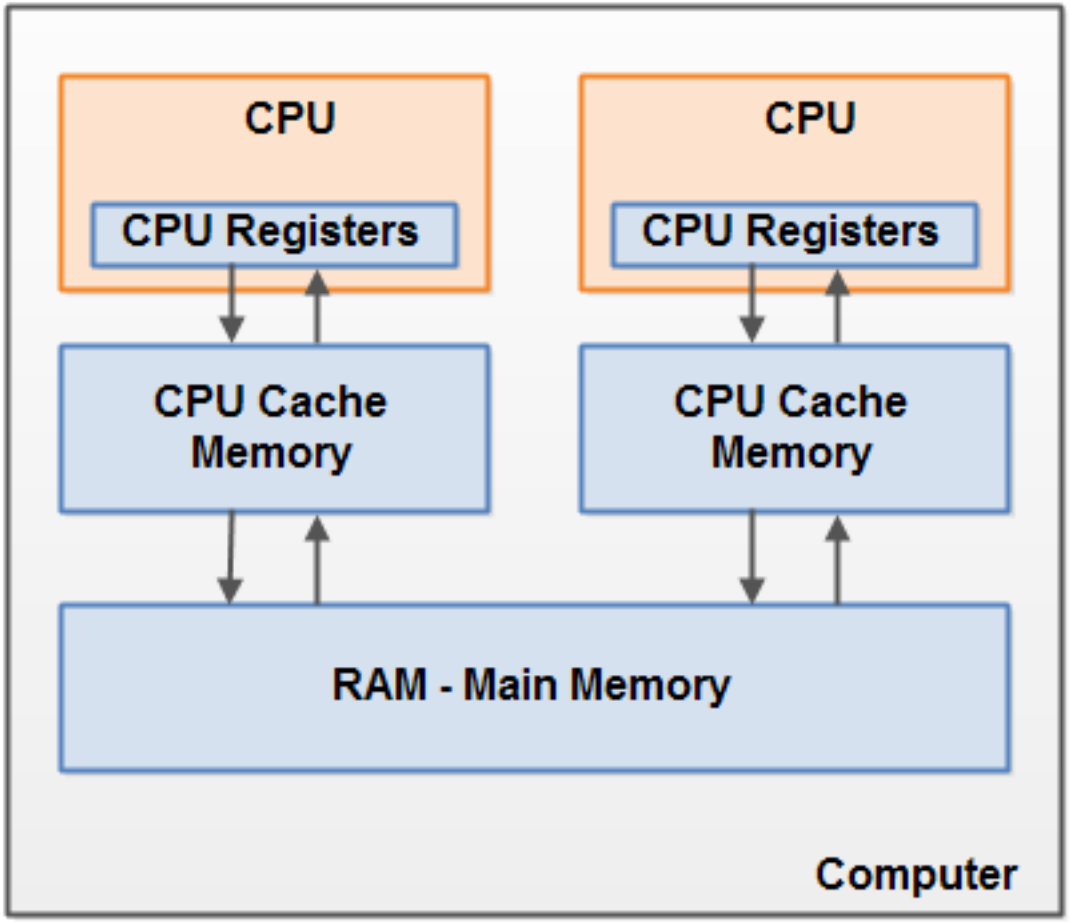
- **atomicity**: which instructions must have indivisible effects
- **visibility**: under what conditions are the effects of one thread visible to another; and
- **ordering**: under what conditions the effects of operations can appear out of order to any given thread.

CPU



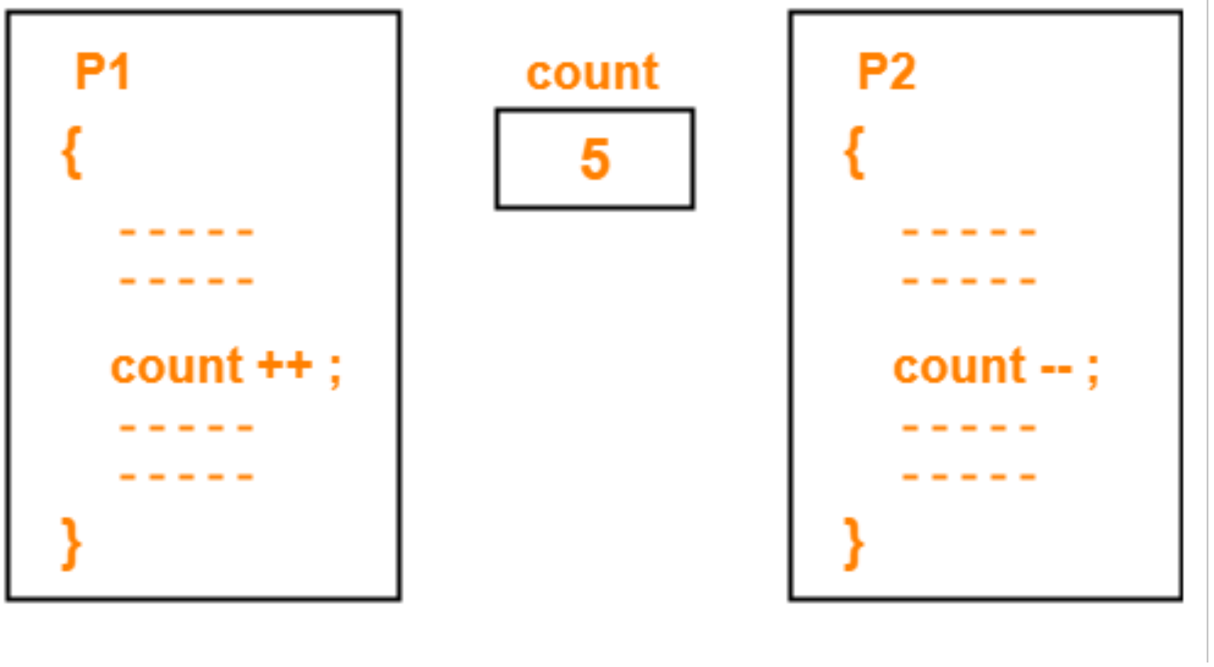
T · · Systems

CPU

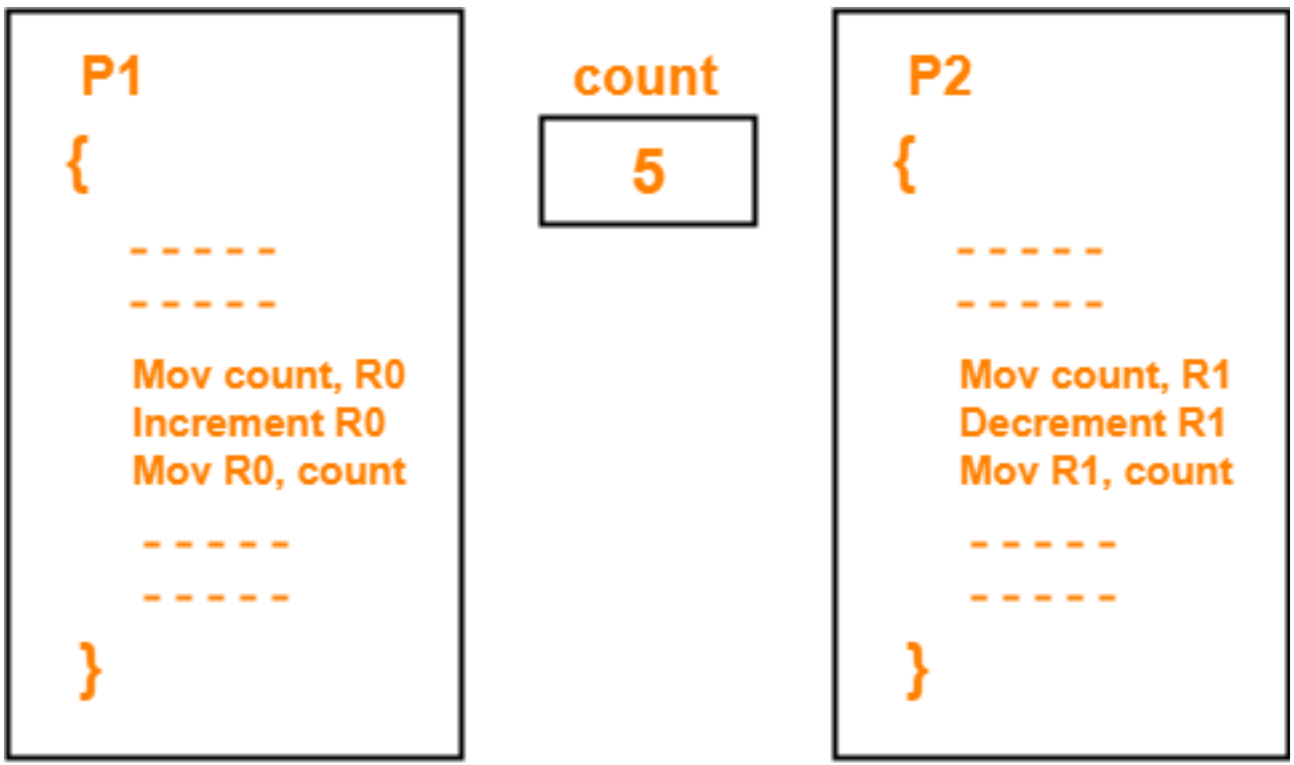


T · · Systems

Atomicity

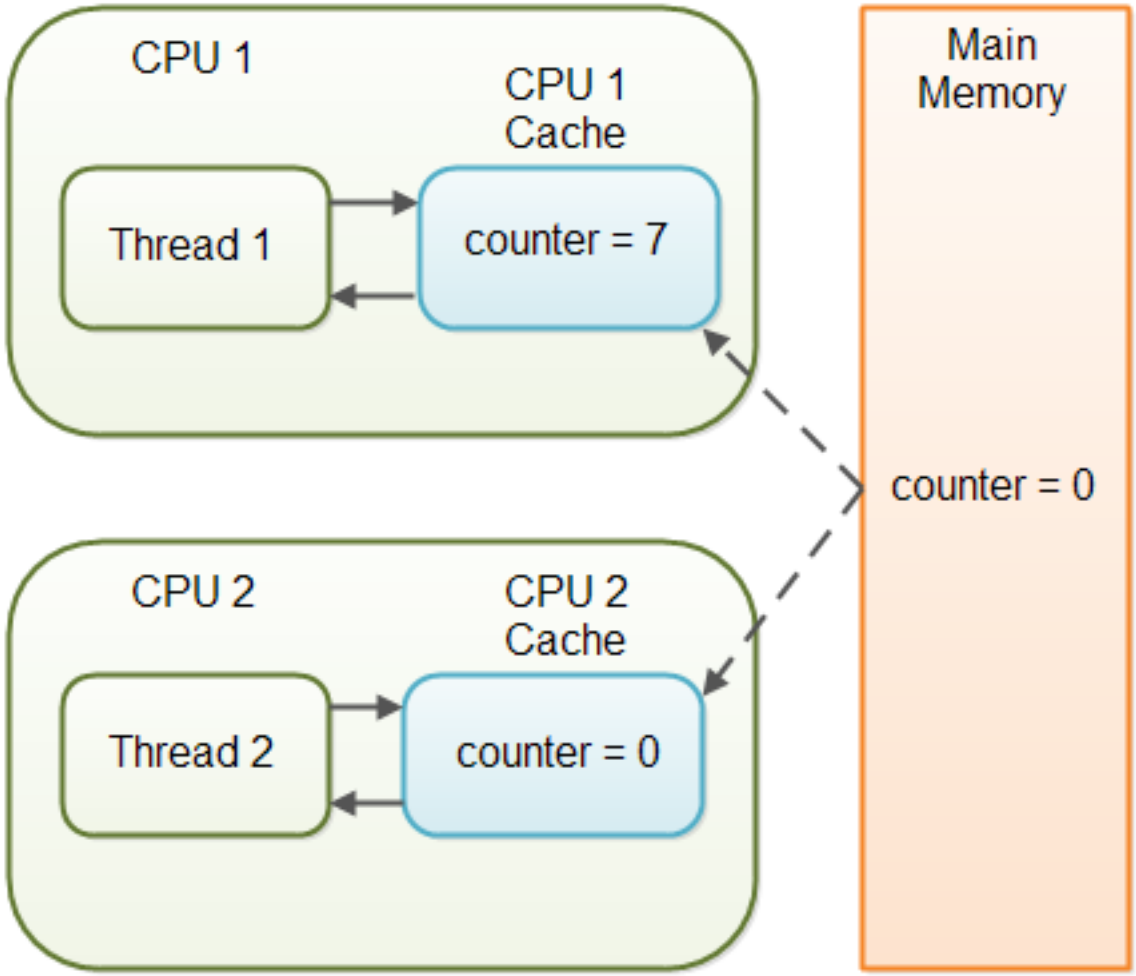


Atomicity



P1
{
- - - - -
- - - - -
Mov count, R0
Increment R0
Mov R0, count
- - - - -
- - - - -
}
count
5
P2
{
- - - - -
- - - - -
Mov count, R1
Decrement R1
Mov R1, count
- - - - -
- - - - -
}

Visibility



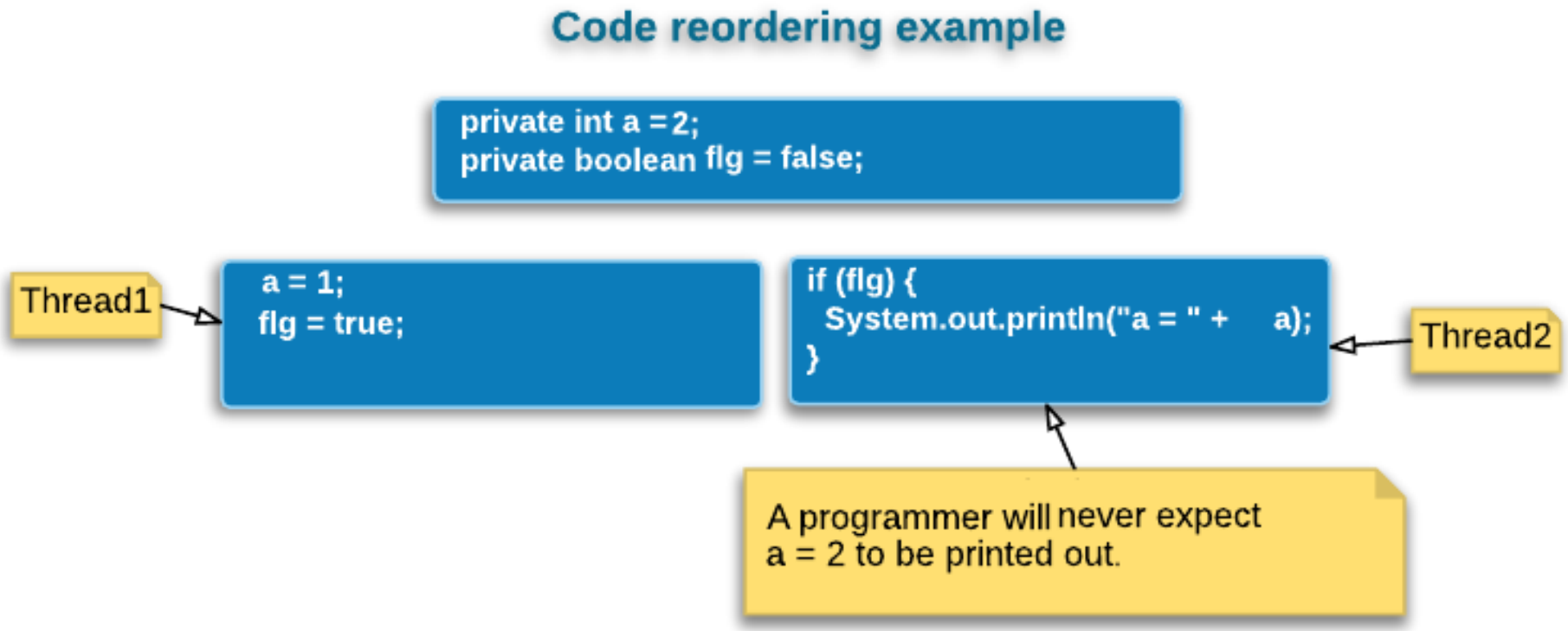
T · · Systems

Reordering

```
public class ReorderExample {
  private int a = 2;
  private boolean flg = false;

  public void method1() {
      a = 1;
      flg = true;
  }

  public void method2() {
      if (flg) {
          //2 might be printed out on some JVM/machines
          System.out.println("a = " + a);
      }
  }

  public static void main(String[] args) {
      for (int i = 0; i < 100; i++) {
          ReorderExample reorderExample = new ReorderExample();
          Thread thread1 = new Thread(() -> {
              reorderExample.method1();
          });
          Thread thread2 = new Thread(() -> {
              reorderExample.method2();
          });
          thread1.start();
          thread2.start();
      }
  }
}
```

T · · Systems

Reordering


Code reordering example
private int a = 2;
private boolean flg = false;
Thread1
a = 1;
flg = true;
if (flg) {
System.out.println("a = " + a);
}
Thread2
A programmer will never expect
a = 2 to be printed out.

Model properties

- **atomicity**: reads and writes to memory cells corresponding to fields of any type *except* `long` or `double` are guaranteed to be atomic
- **visibility**: changes to fields made by one thread are not guaranteed to be visible to other threads
- **ordering**: from the point of view of other threads, instructions may appear to be executed out of order

Volatile

If a field is declared `volatile`, a thread must reconcile its working copy of the field with the master copy every time it accesses the variable.

- reads and writes to a `volatile` field are guaranteed to be atomic (even for `longs` and `doubles`);
- new values are immediately propagated to other threads; and
- from the point of view of other threads, the relative ordering of operations on `volatile` fields are preserved.

However the ordering and visibility effects surround only the single read or write to the `volatile` field itself, e.g, '++' on a `volatile` field is not atomic.

Volatile

- Reads & writes go directly to memory
  - Caching disabled
- Volatile longs & doubles are atomic
- Volatiles reads/writes cannot be reordered
- Reads/writes become acquire/release pairs

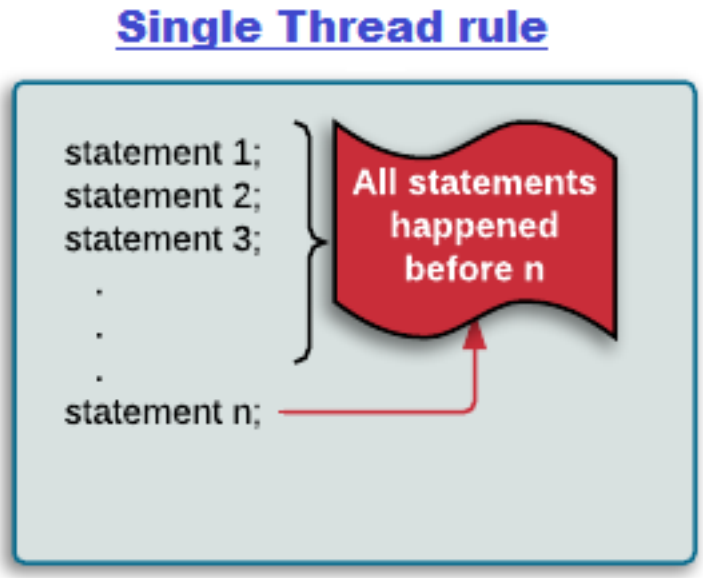
Happens before

Happens-before relationship is a guarantee that action performed by one thread is visible to another action in different thread.

Transitivity: If A happens-before B, and B happens-before C, then A happens-before C.

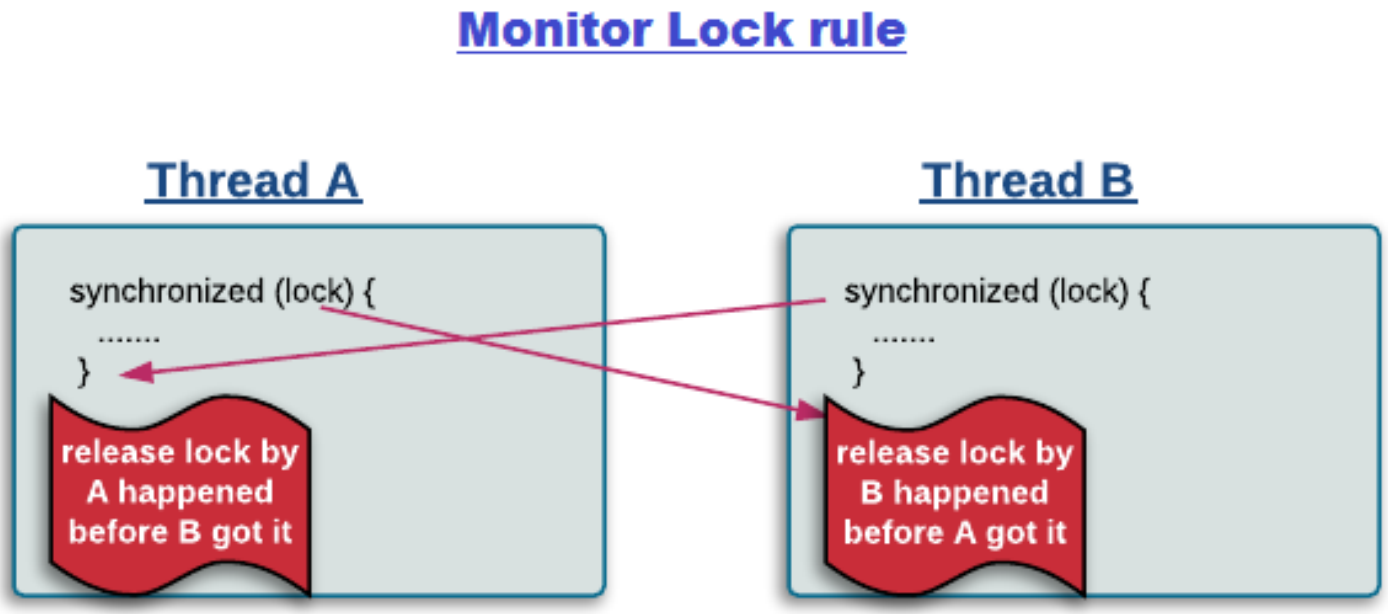
Single thread rule

Single thread rule: Each action in a single thread happens-before every action in that thread that comes later in the program order.



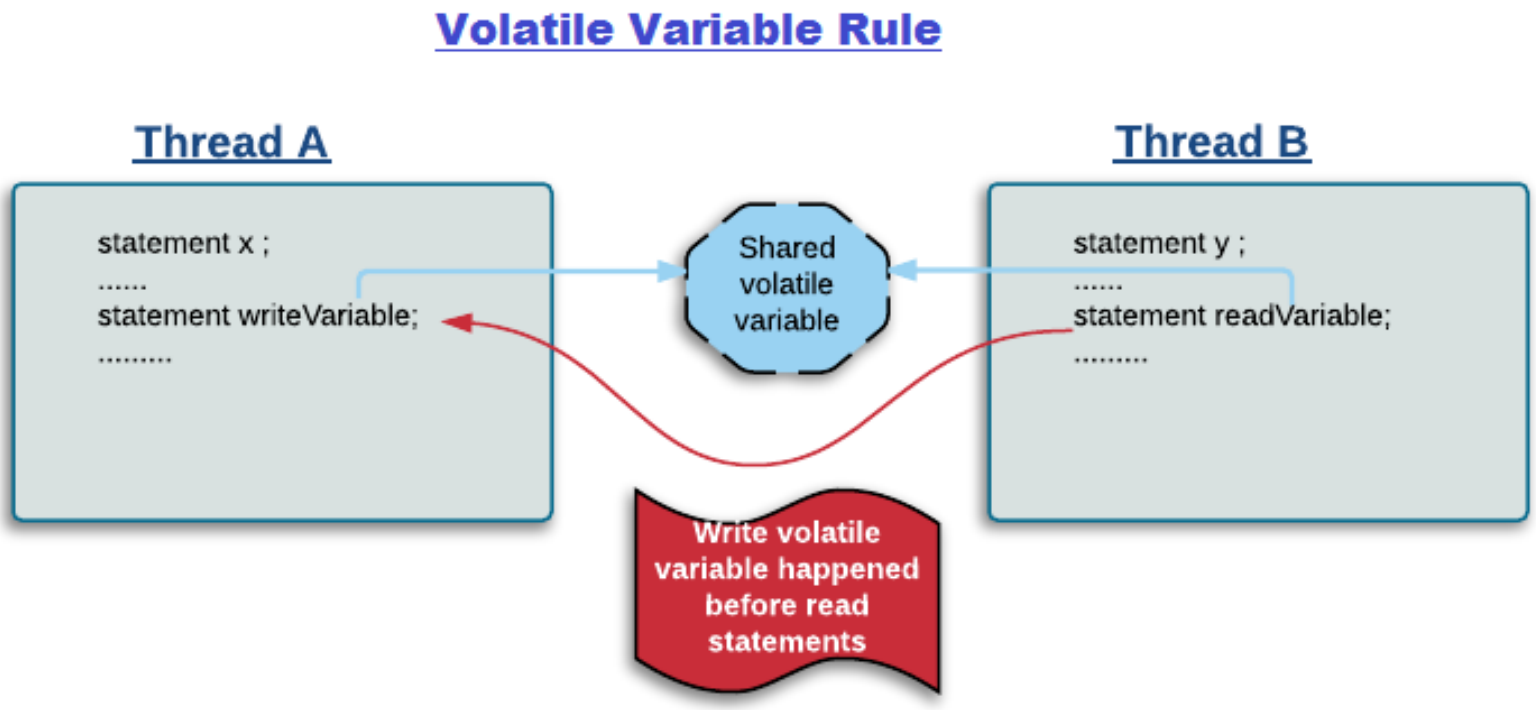
Monitor rule

Monitor lock rule: An unlock on a monitor lock (exiting synchronized method/block) happens-before every subsequent acquiring on the same monitor lock.



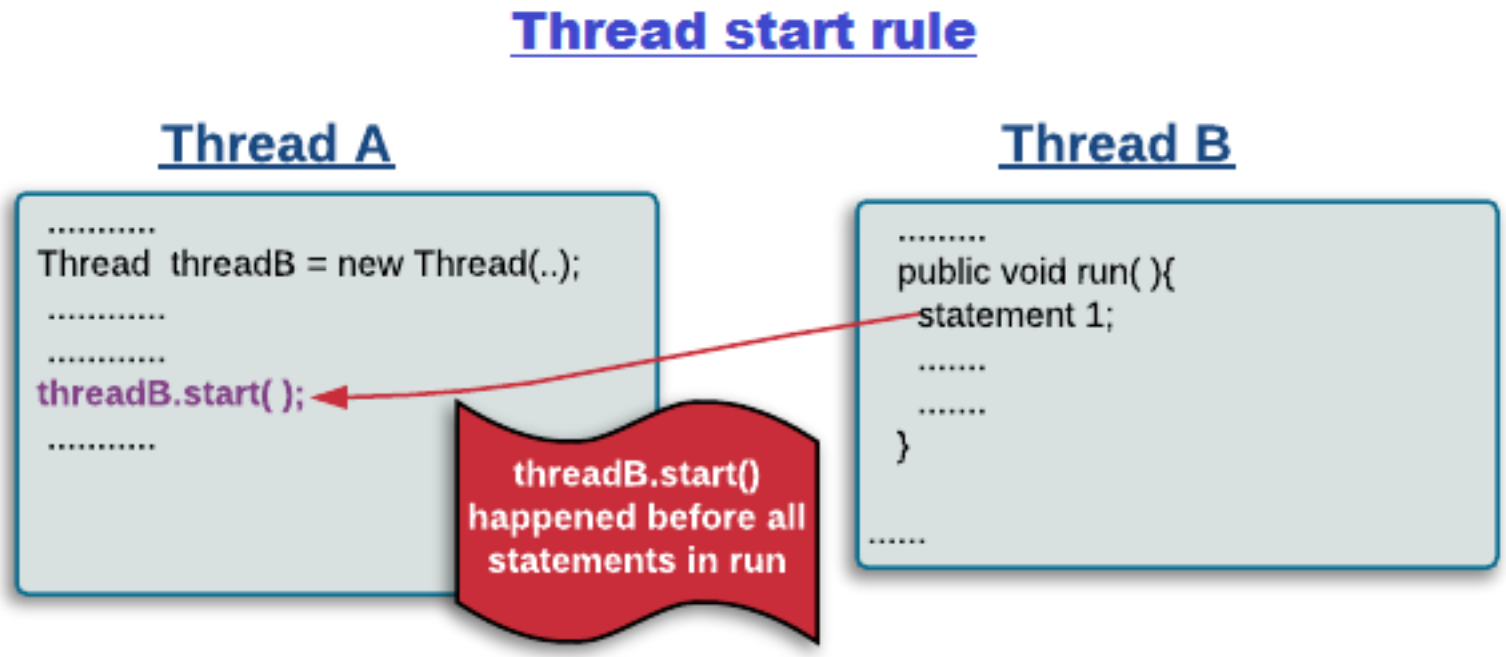
Volatile rule

Volatile variable rule: A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors (synchronized block around reads and writes), but without actually acquiring monitors/locks.



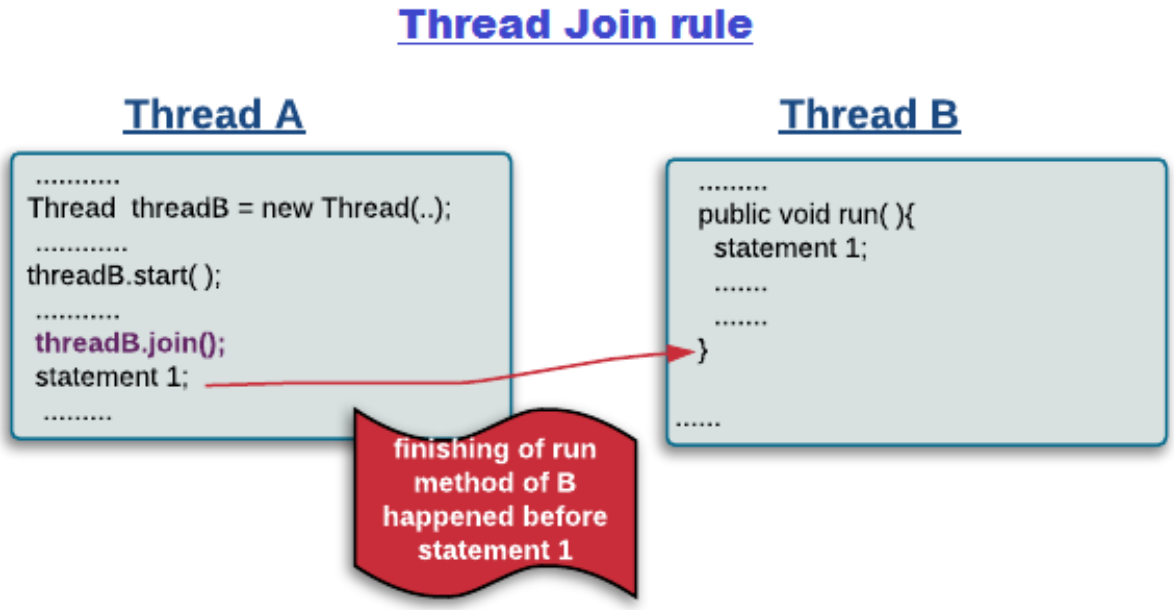
Thread start rule

Thread start rule: A call to Thread.start() on a thread happens-before every action in the started thread. Say thread A spawns a new thread B by calling thread A.start(). All actions performed in thread B's run method will see thread A's calling thread A.start() method and before that (only in thread A) happened before them.



Thread join rule

Thread join rule: All actions in a thread happen-before any other thread successfully returns from a join on that thread. Say thread A spawns a new thread B by calling threadA.start() then calls threadA.join(). Thread A will wait at join() call until thread B's run method finishes. After join method returns, all subsequent actions in thread A will see all actions performed in thread B's run method happened before them.



thread safety = atomicity + visibility

- **thread safe**: Can be used in a multithreaded environment.
  - Many of the Java library classes are *not* thread safe!
  - In other words, if two threads access the same object, things break.

- Examples:
  - AWT and Swing are not thread safe; if two threads are modifying a GUI simultaneously, they may put the GUI into an invalid state.
  - `ArrayList` and other collections from `java.util` are not thread safe; two threads changing the same list at once may break it.
  - `StringBuilder` is not thread safe.

- Counterexamples:
  - The `Random` class chooses numbers in a thread-safe way.

Example 5,6,7

Thread signaling

Guarded suspension can be implemented using the the `wait()`, `notify()` and `notifyAll()` methods of the `Object` class.

The `wait` method should be invoked when a thread is temporarily unable to continue and we want other threads to proceed.

The `notify()` and `notifyAll()` methods should be invoked when we want a thread to notify other threads that they may proceed.

Wait

- Must only be invoked in a synchronized method or a synchronized block
- The thread is suspended and waits for a `notify`-signal
- Releases the lock associated with the receiving object
- The awakened thread must reobtain the lock before it can resume at the point immediately following the invocation of the `wait()` method.
- The variants `wait(long millis)` and `wait(long millis, int nanos)` makes it possible to specify a maximum wait time.

Notify

- Must only be invoked in a synchronized method or a synchronized block
- Wakes up one of the suspended threads waiting on the given object.
- `notifyAll()` wakes up all threads waiting on a given object. At most one of them proceeds.

Thread signaling issue

Issue: Spurious Wakeups.

- Due to spurious wakeups, JVM is permitted to remove threads from wait sets without explicit instructions.which causes extra logic around calls to wait:

```
while (!<some condition>) {
    try {
        obj.wait();
    }
    catch(InterruptedException ie) { }
}
```

- Where `<some condition>` is set by notifying thread.

Example 8

Starvation

Some threads don't make progress.

Happens if there always exist some threads with a higher priority, or if a thread with the same priority never releases the processor.

Avoid "busy waiting", such as

```
while (x < 2)
    ;  // nothing
```

Dormancy

A waiting thread is never awakened.

Happens if a waiting thread is not notified.

When in doubt, use the `notifyAll()` method.

Deadlock

Two or more threads are blocked, each waiting for resources held by another thread.

It is usually caused by two or more threads competing for resources and each thread requiring exclusive possession of those resources simultaneously.

```
synchronized(a) {
    synchronized(b) {
        ...
    }
}
```

thread1

```
synchronized(b) {
    synchronized(a) {
        ...
    }
}
```

thread2

T Systems

Deadlock

```
public class DiskDrive {
    public synchronized InputStream openFile(String fileName) {
        ...
    }

    public synchronized void writeFile(String fileName,
                                       InputStream in) {
        ...
    }

    public synchronized void copy(DiskDrive destination,
                                  String fileName) {
        InputStream in = openFile(fileName);
        destination.writeFile(fileName, in);
    }
}
```

T · · Systems

Deadlock

thread1: `c.copy(d, file1)`

Invoke `c.copy(...)`

Obtains lock of `c`

Invoke `c.openFile(...)`

Invoke `d.writeFile(...)`

Unable to obtain lock of `d`

thread2: `d.copy(c, file2)`

Invoke `d.copy(...)`

Obtains lock of `d`

Invoke `d.openFile(...)`

Invoke `c.writeFile(...)`

Unable to obtain lock of `c`

Deadlock!

Deadlock

The runtime system is neither able to detect nor prevent deadlocks. It is the responsibility of the programmer to ensure that deadlock cannot occur.

An often applied technique is **resource ordering**:

```
synchronized(a) {
    synchronized(b) {
        ...
    }
}
```

thread1

```
synchronized(a) {
    synchronized(b) {
        ...
    }
}
```

thread2

Deadlock cannot occur

T · · Systems

Example 9

Daemons

- ✓ **Daemon thread** is a low priority thread (in context of JVM) that runs in background to perform tasks such as Garbage Collection (GC) etc.
- ✓ **Daemon thread** do not prevent the JVM from exiting (even if the **daemon thread** itself is running) when all the user threads (non-daemon threads) finish their execution. JVM terminates itself when all user threads (non-daemon threads) finish their execution, JVM does not care whether **Daemon thread** is running or not.
- ✓ If JVM finds running **daemon thread** (upon completion of user threads), it terminates the thread and after that shutdown itself.

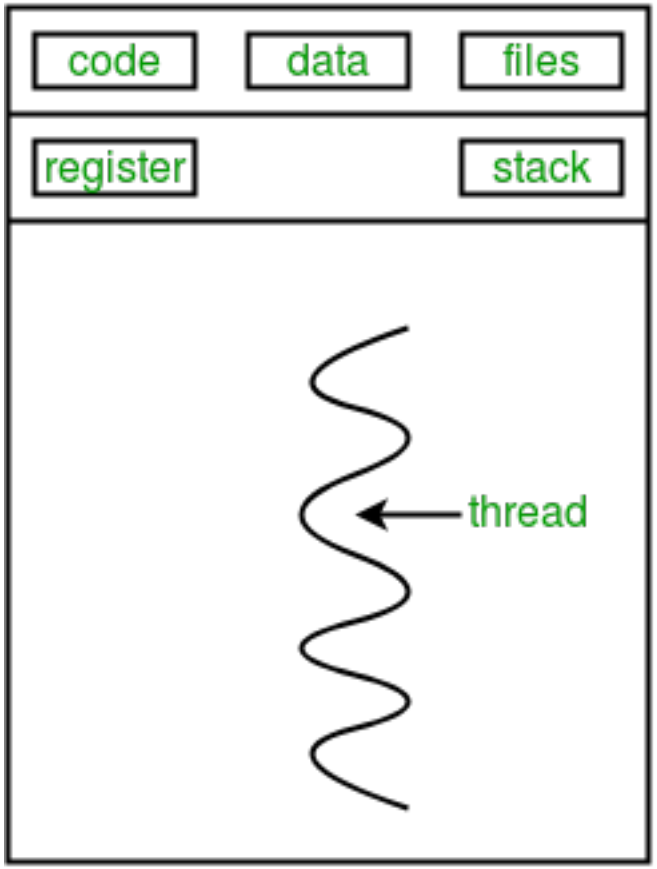
- ✓ A newly created thread inherits the daemon status of its parent. That's the reason all threads created inside main method (child threads of main thread) are non-daemon by default, because main thread is non-daemon. However you can make a user thread to Daemon by using **setDaemon() method** of thread class.

- ✓ When the JVM starts, it creates a thread called "Main". Your program will run on this thread, unless you create additional threads yourself. The first thing the "Main" thread does is to look for your static void main (String args[]) method and invoke it. That is the entry-point to your program. If you create additional threads in the main method those threads would be the child threads of main thread.

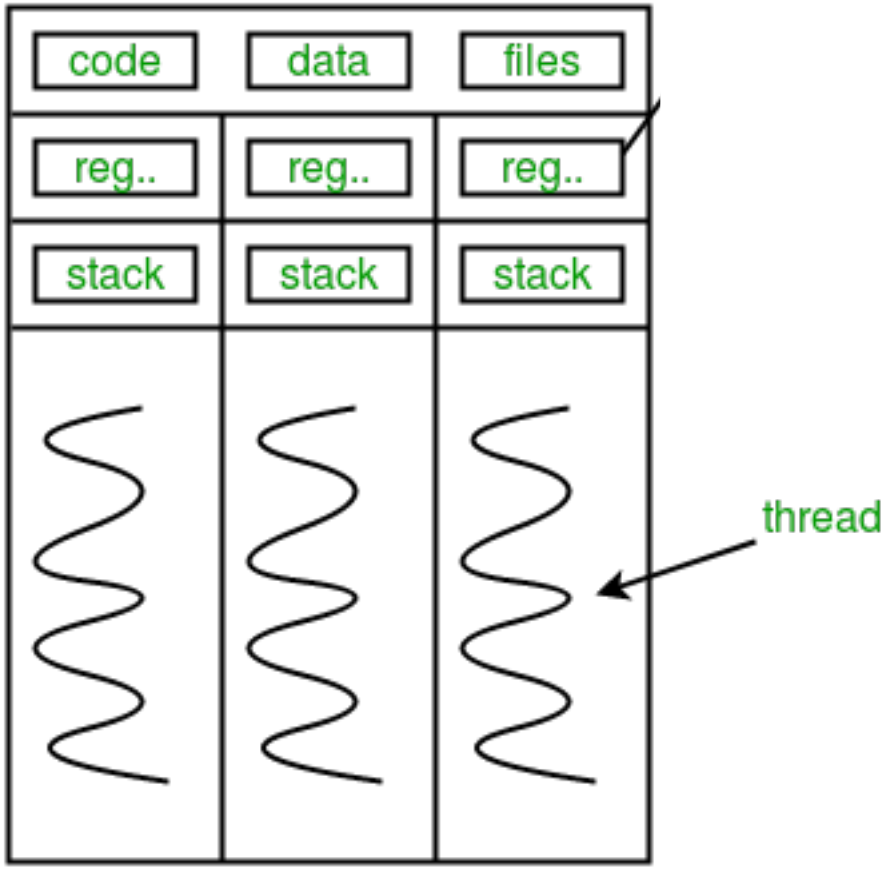
The main difference between Daemon thread and user threads is that the JVM does not wait for Daemon thread before exiting while it waits for user threads, it does not exit until unless all the user threads finish their execution.

Conclusion

Sequential vs concurrent

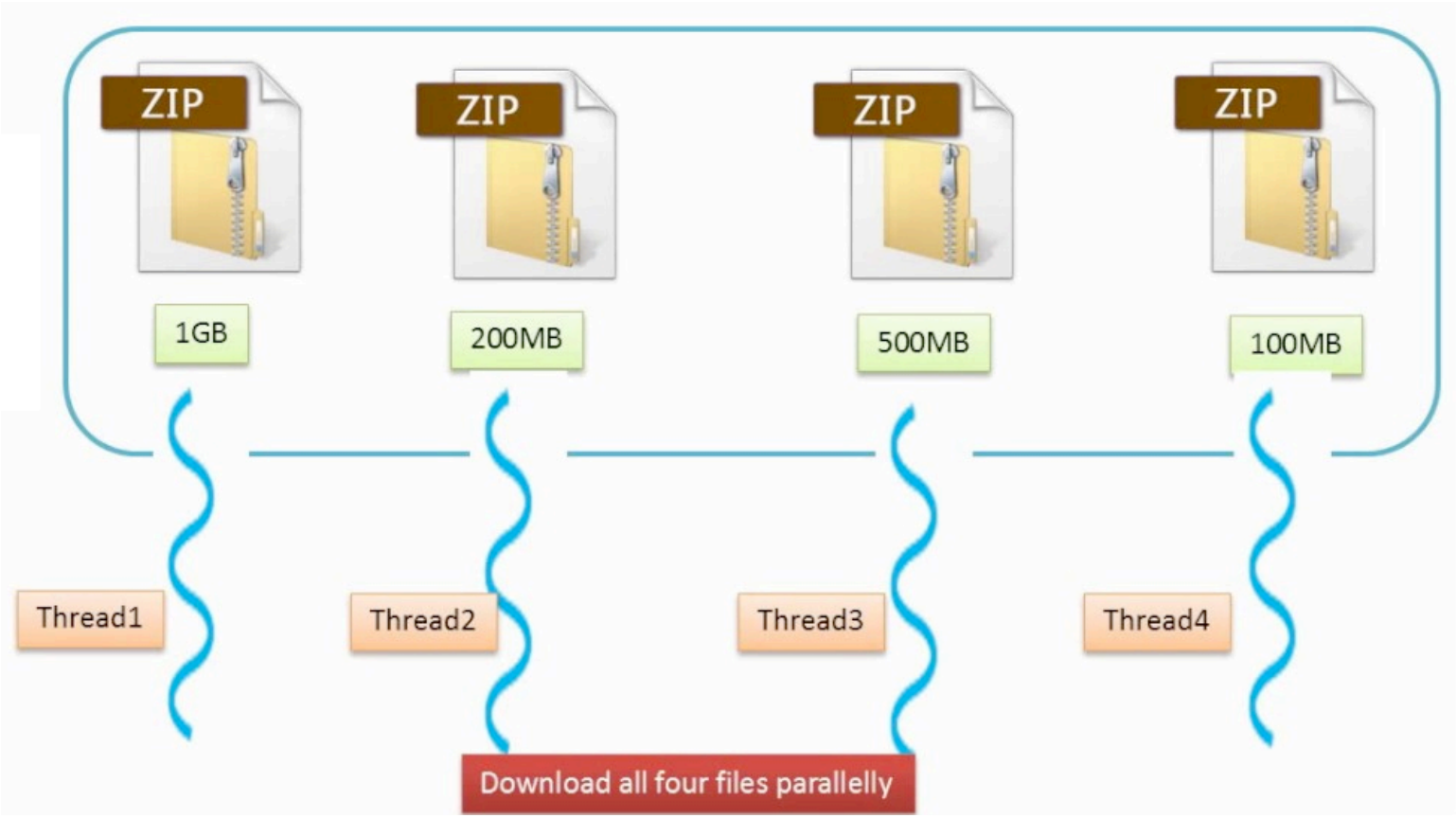


single-threaded process



multithreaded process

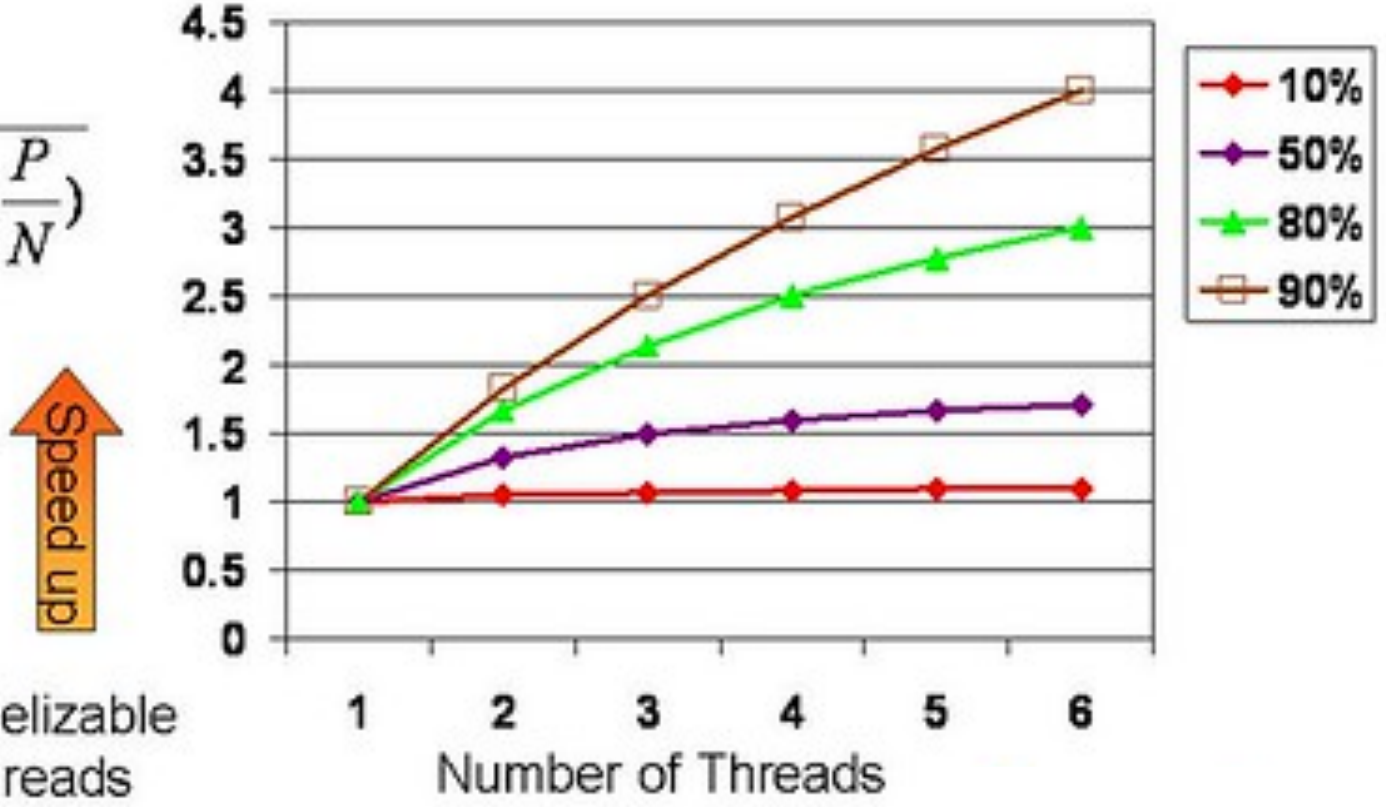
Performance



Amdahl's law

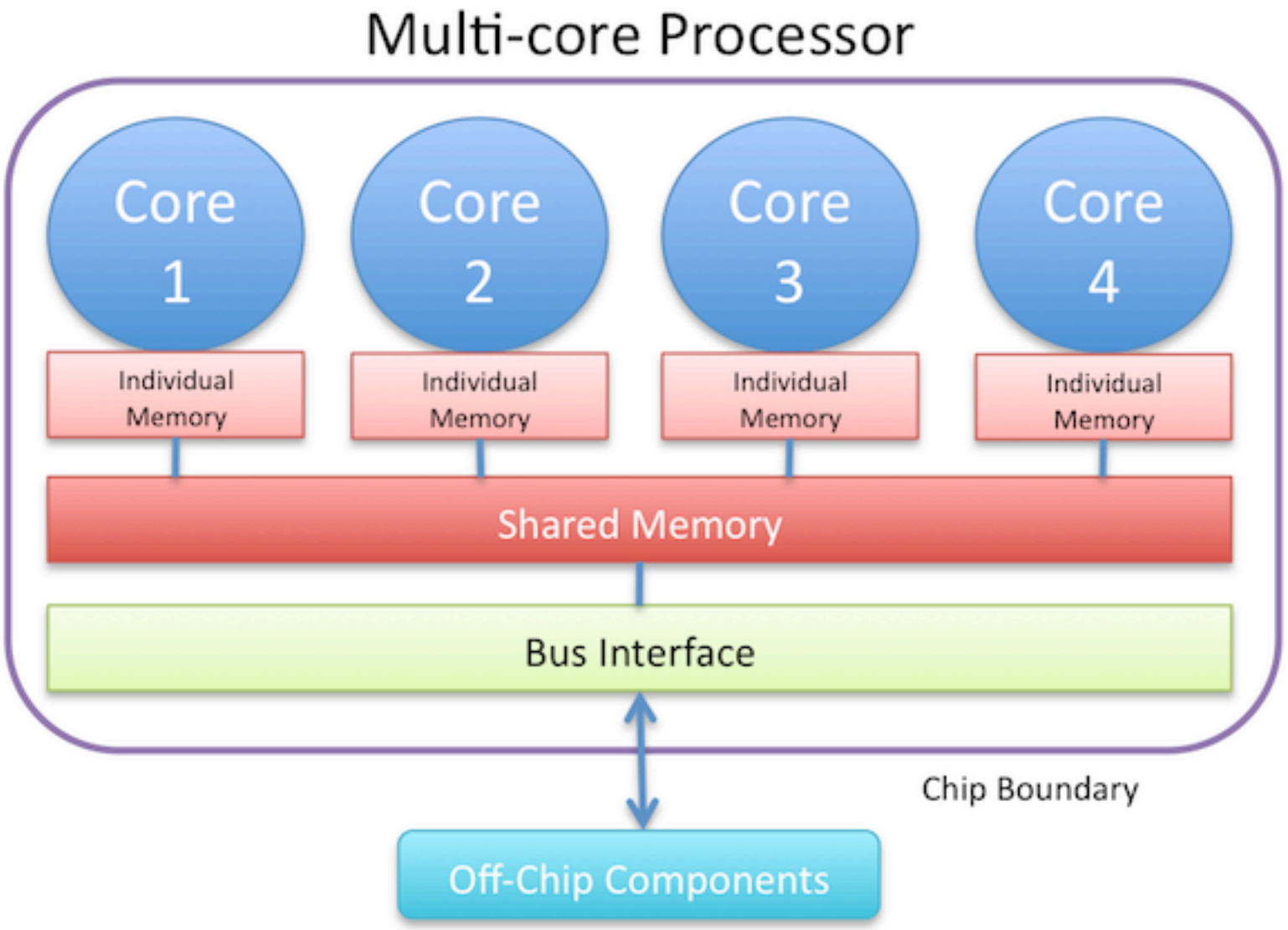
"Maximum expected improvement to an overall system when only part of the system is parallelized."

$$\frac{1}{(1-P)+(\frac{P}{N})}$$

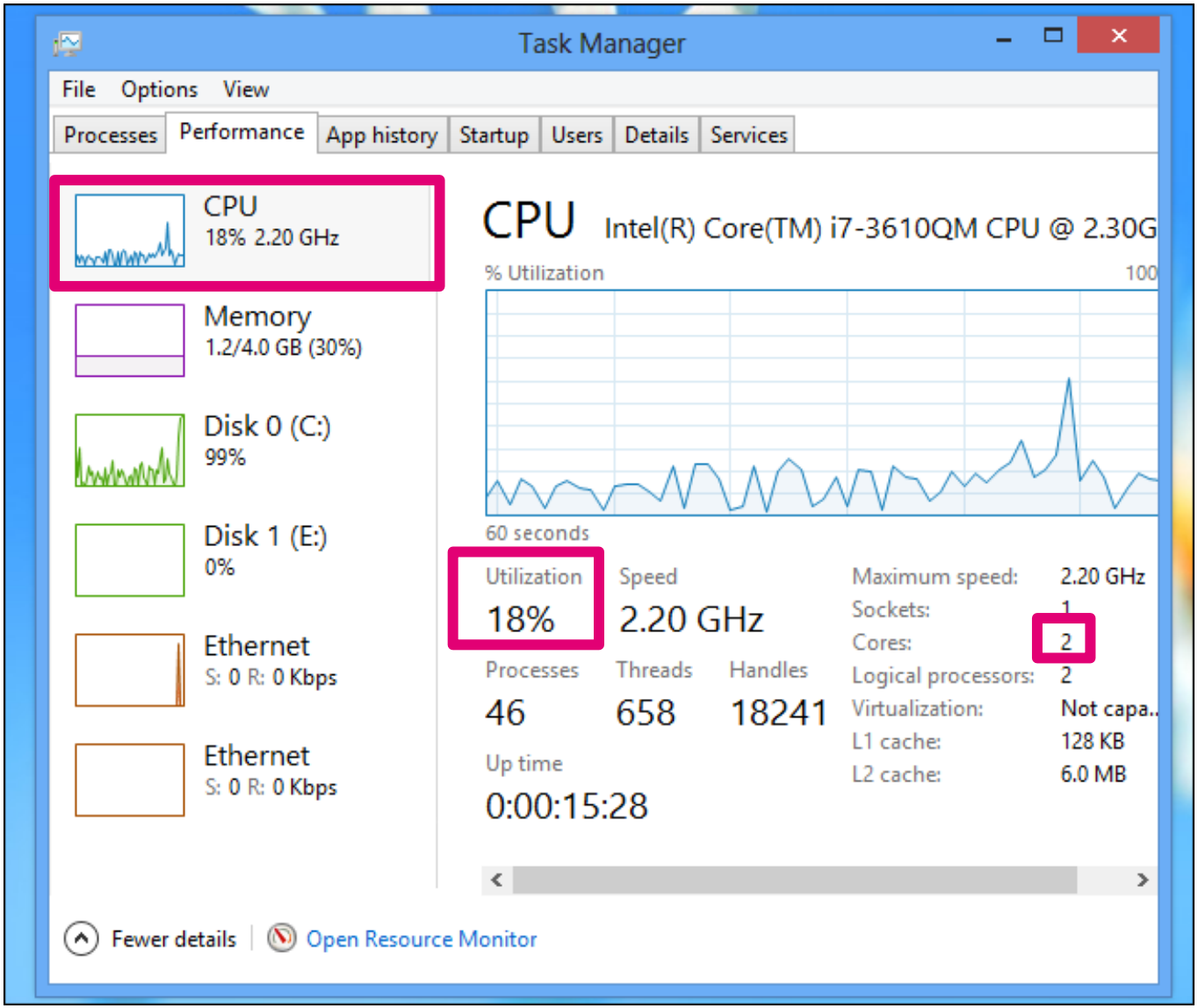


P = % parallelizable
N = # of threads

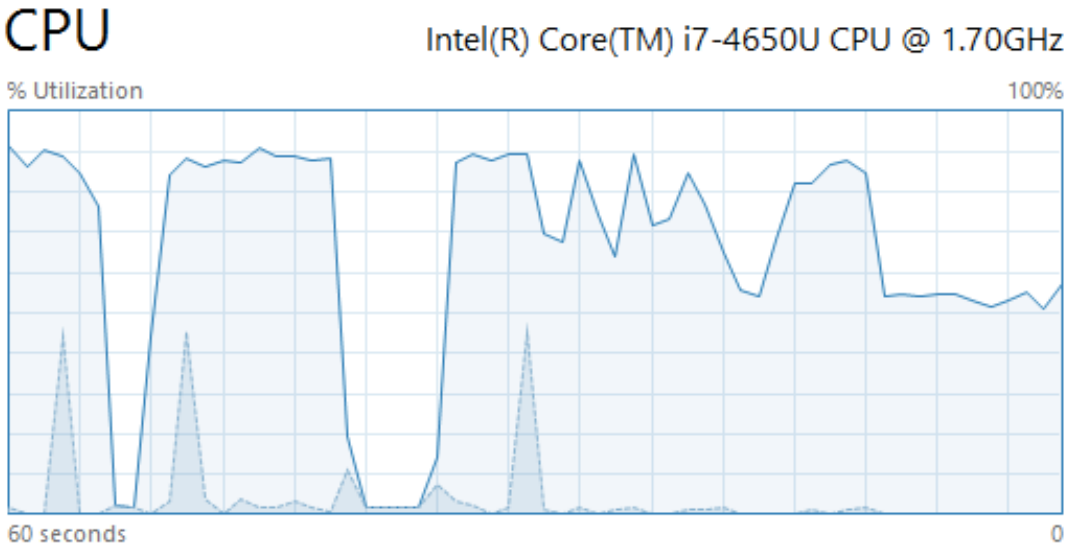
Resource utilization

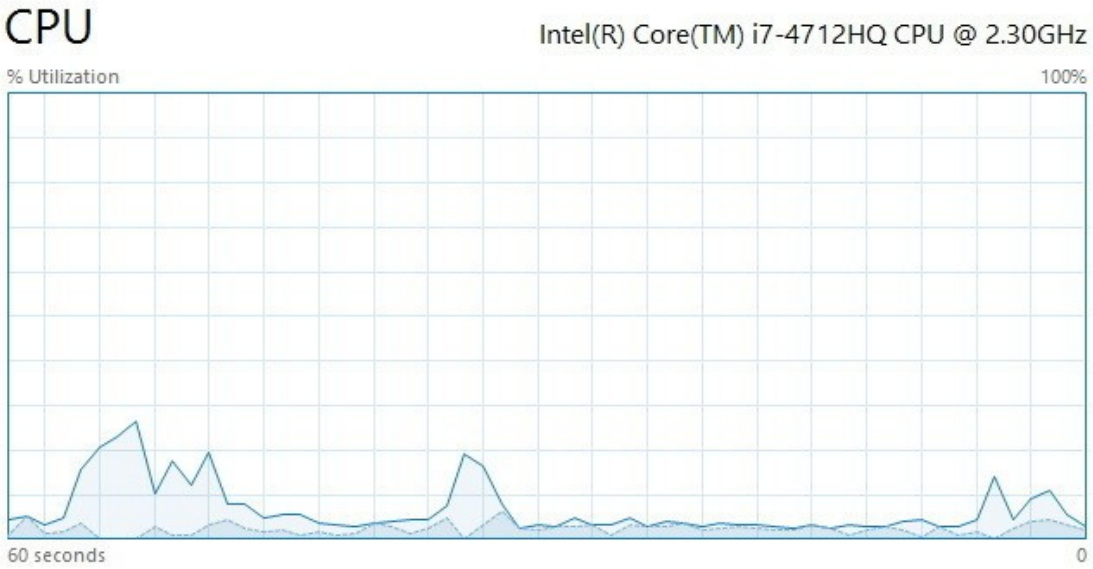


Resource utilization

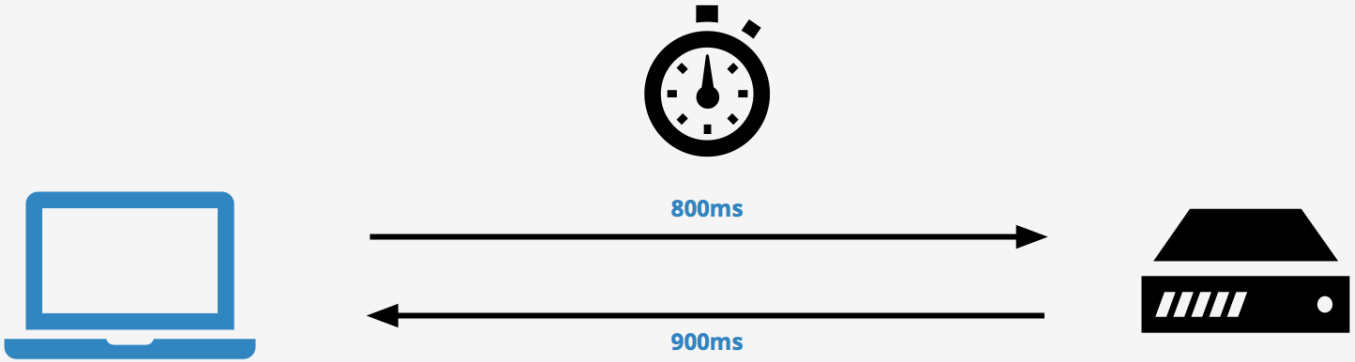


Resource utilization





Latency

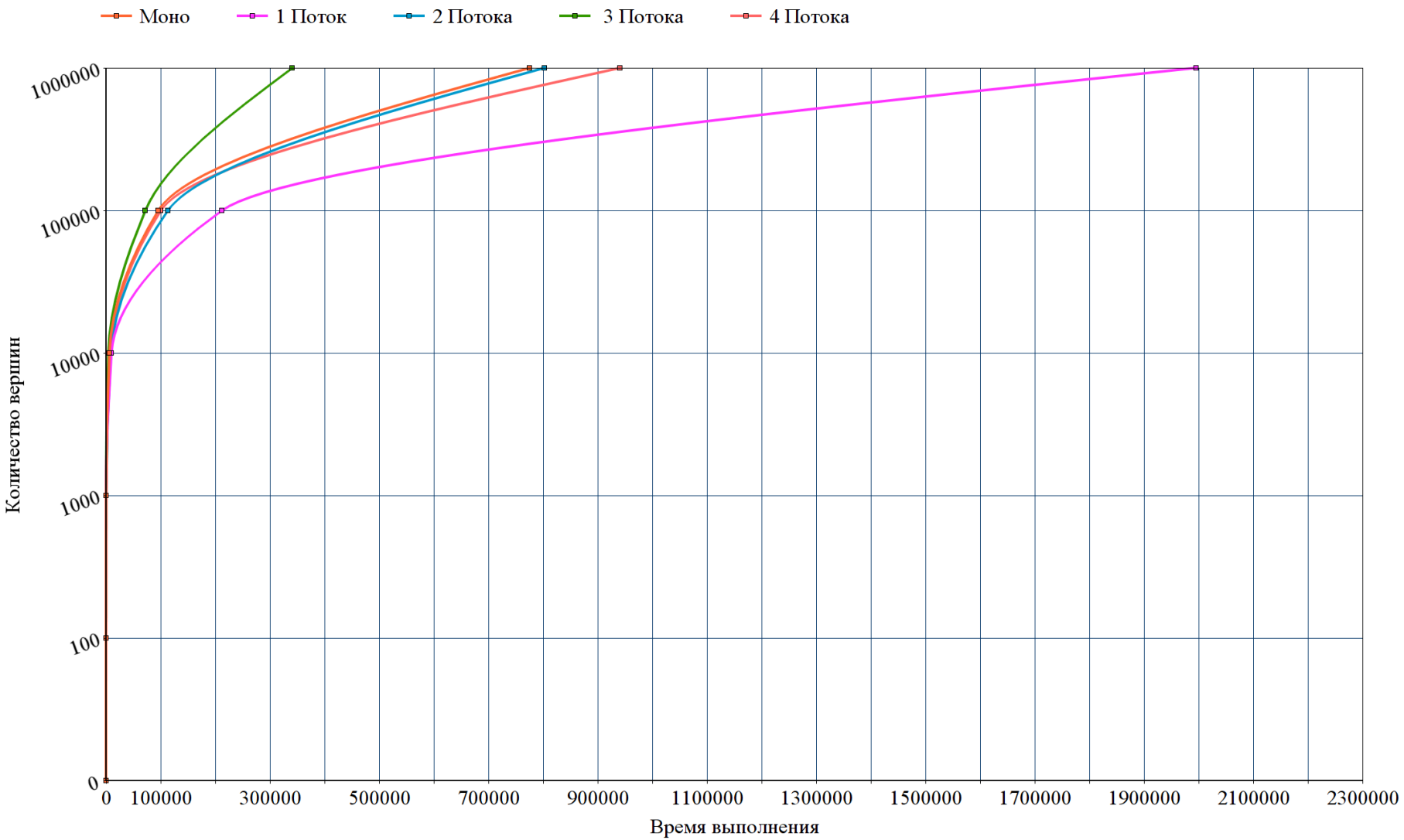


Latency = 800ms + 900ms = 1.7s

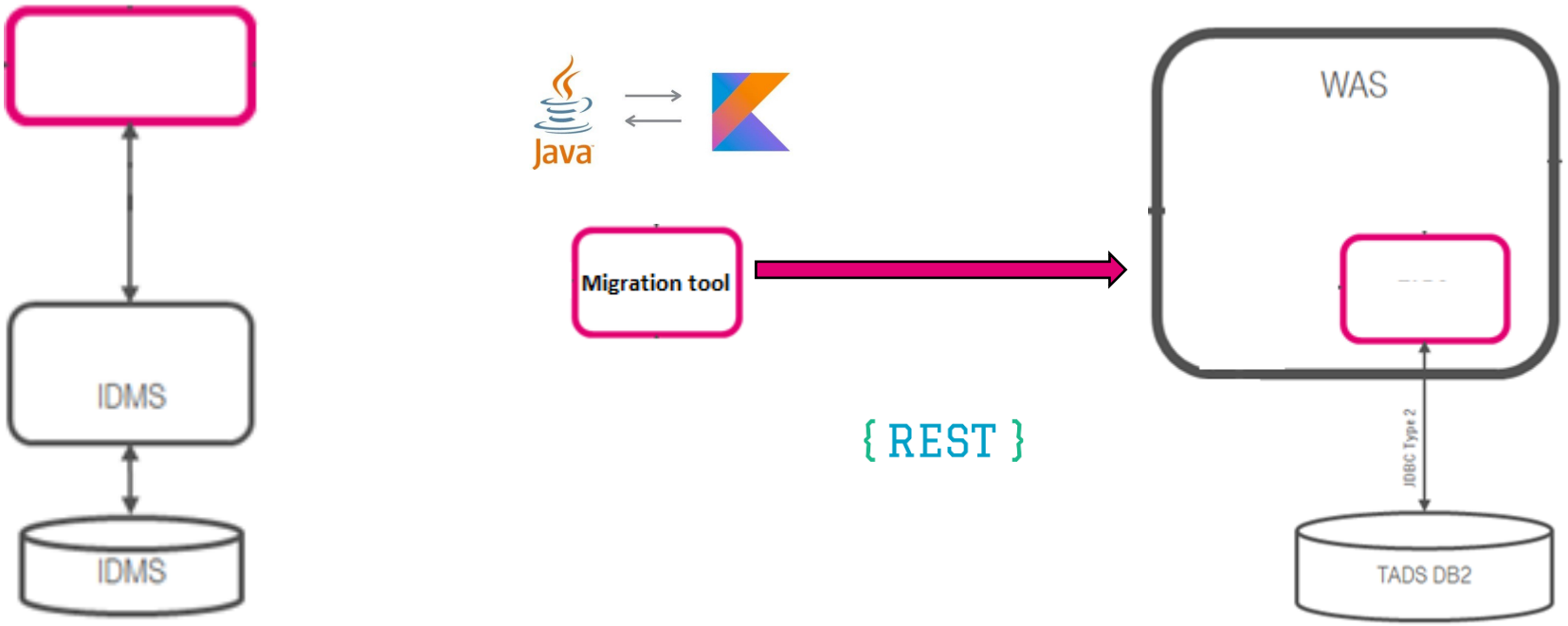
Modern way


BigBasket
Shop By Category
DISCOUNTS
Starter Kit
Network Hospitals
Health & Wellness
My Coverage
My Family
Policy Guide
My Claims
Other Contacts
Claims on the go

Concurrent Dijkstra algorithm



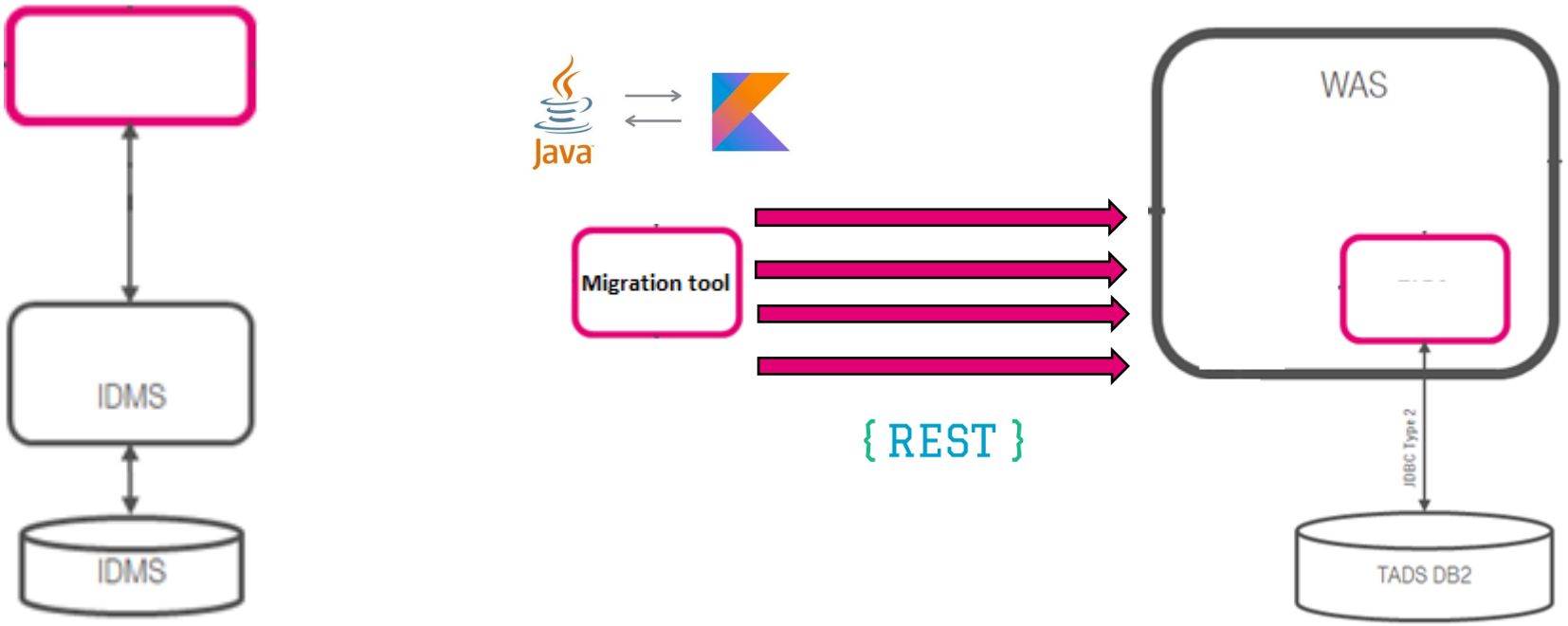
Multi thread migration



Single thread migration

7h21 min 1.7 r/s

Multi thread migration



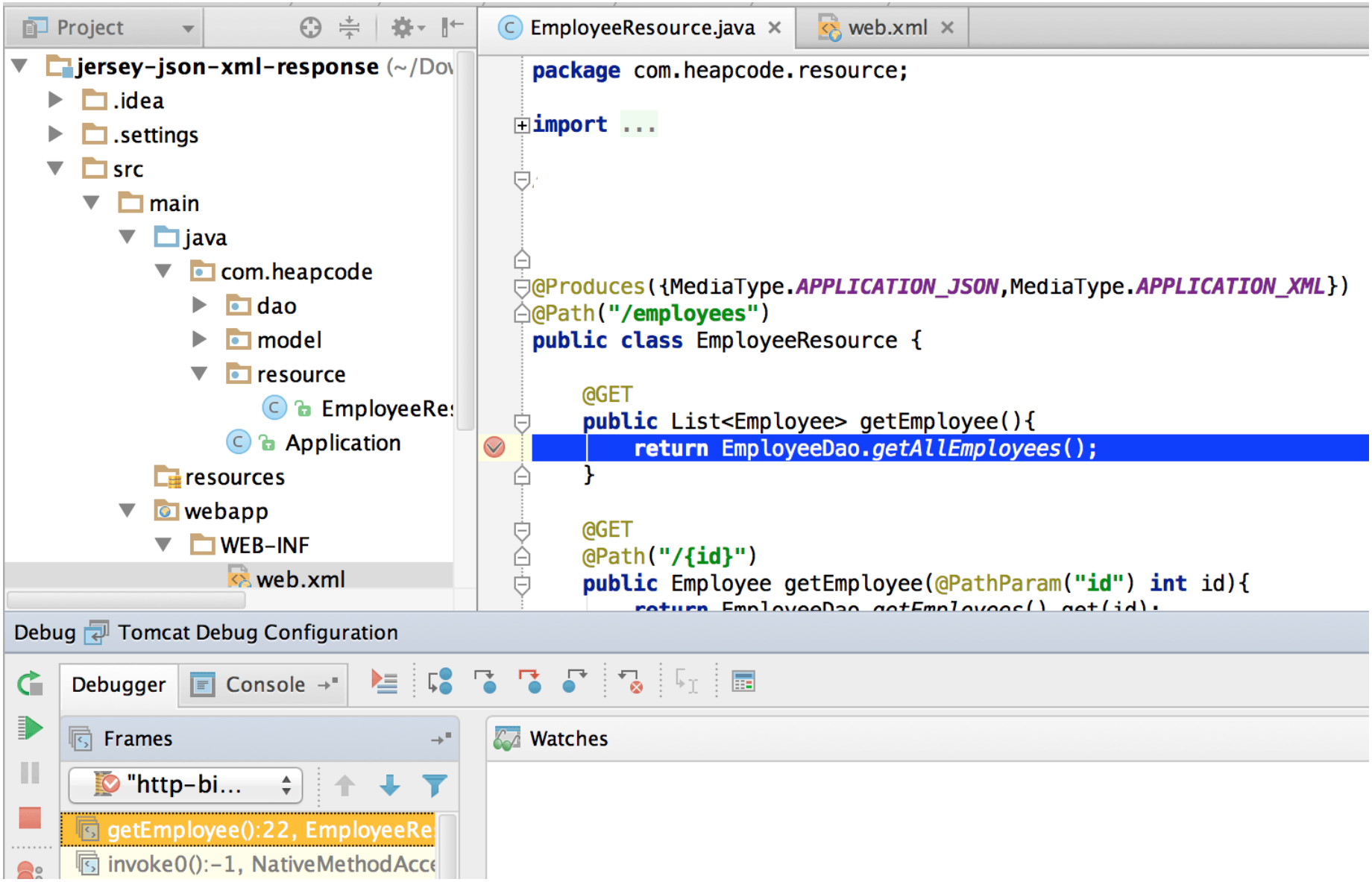
Multi thread migration

7h21 min 1.7 r/s

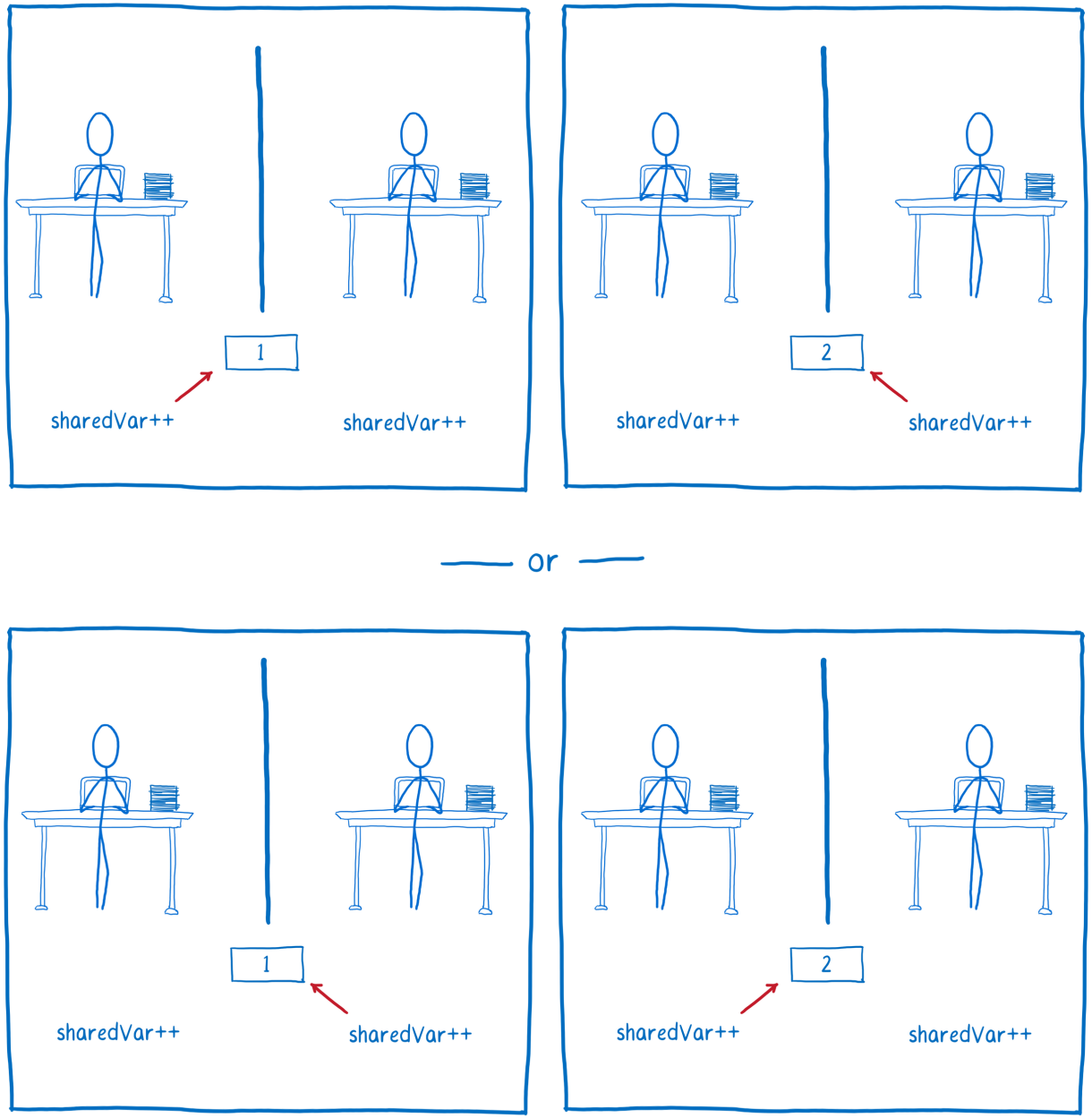
28 min 23 r/s

The show begins

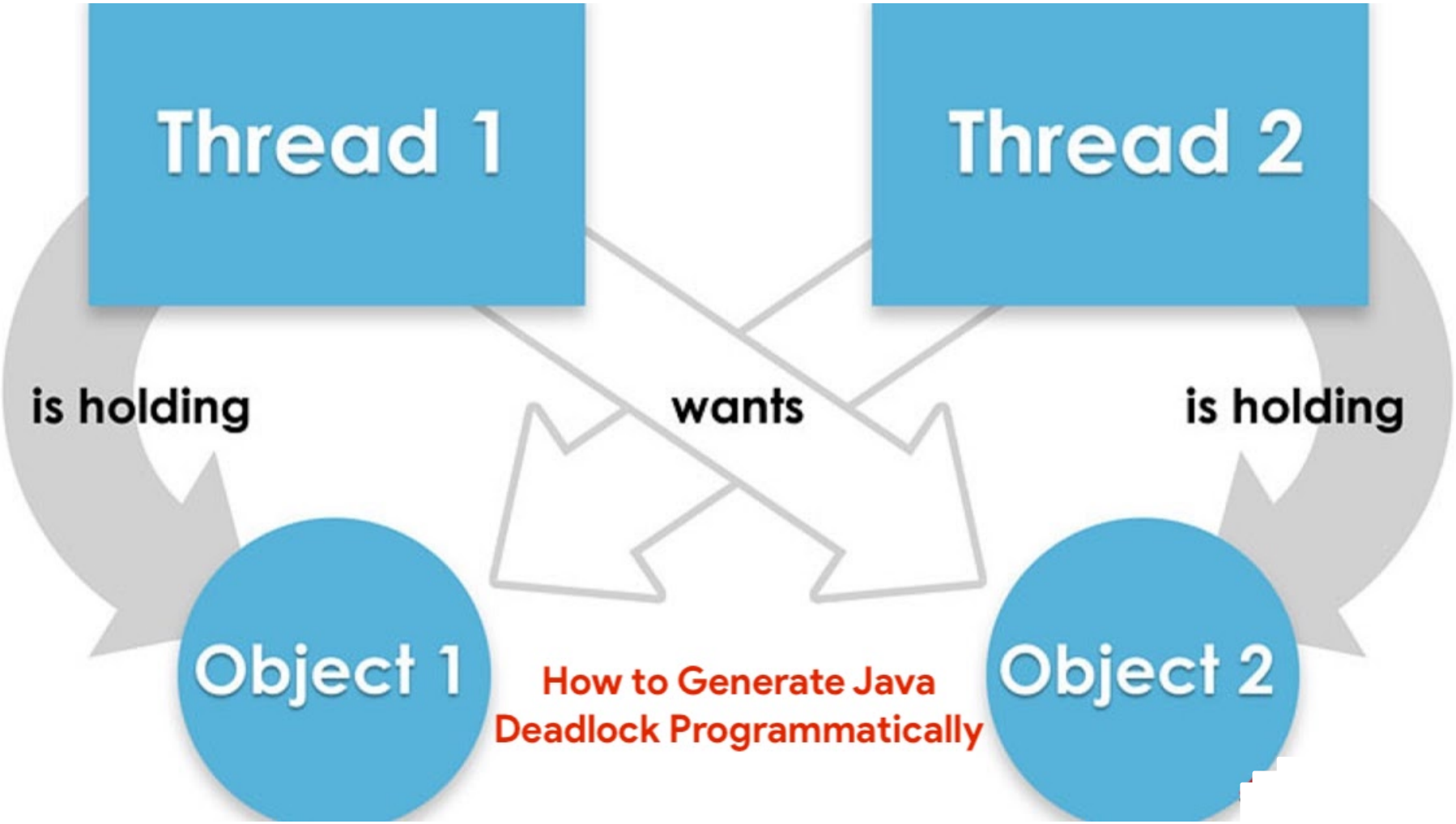
#1 Difficult to debug



#2 Race condition



#3 Dead locks


Thread 1
Thread 2
is holding
wants
is holding
Object 1
Object 2
How to Generate Java
Deadlock Programmatically

Concurrency mad stuff

- **“But I don’t want to go among mad people," Alice remarked.**
  - "Oh, you can’t help that," said the Cat: "we’re all mad here. I’m mad. You’re mad."
  - "How do you know I’m mad?" said Alice.
  - "You must be," said the Cat, "or you wouldn’t have come here.”

Alice's Adventures in Wonderland by Lewis Carroll

Dev practices

Everything you're comfortable believing is no longer reliable. It might work and it might not. Most likely it will work under some conditions and not in others, and you'll have to know and understand these situations in order to determine what works.

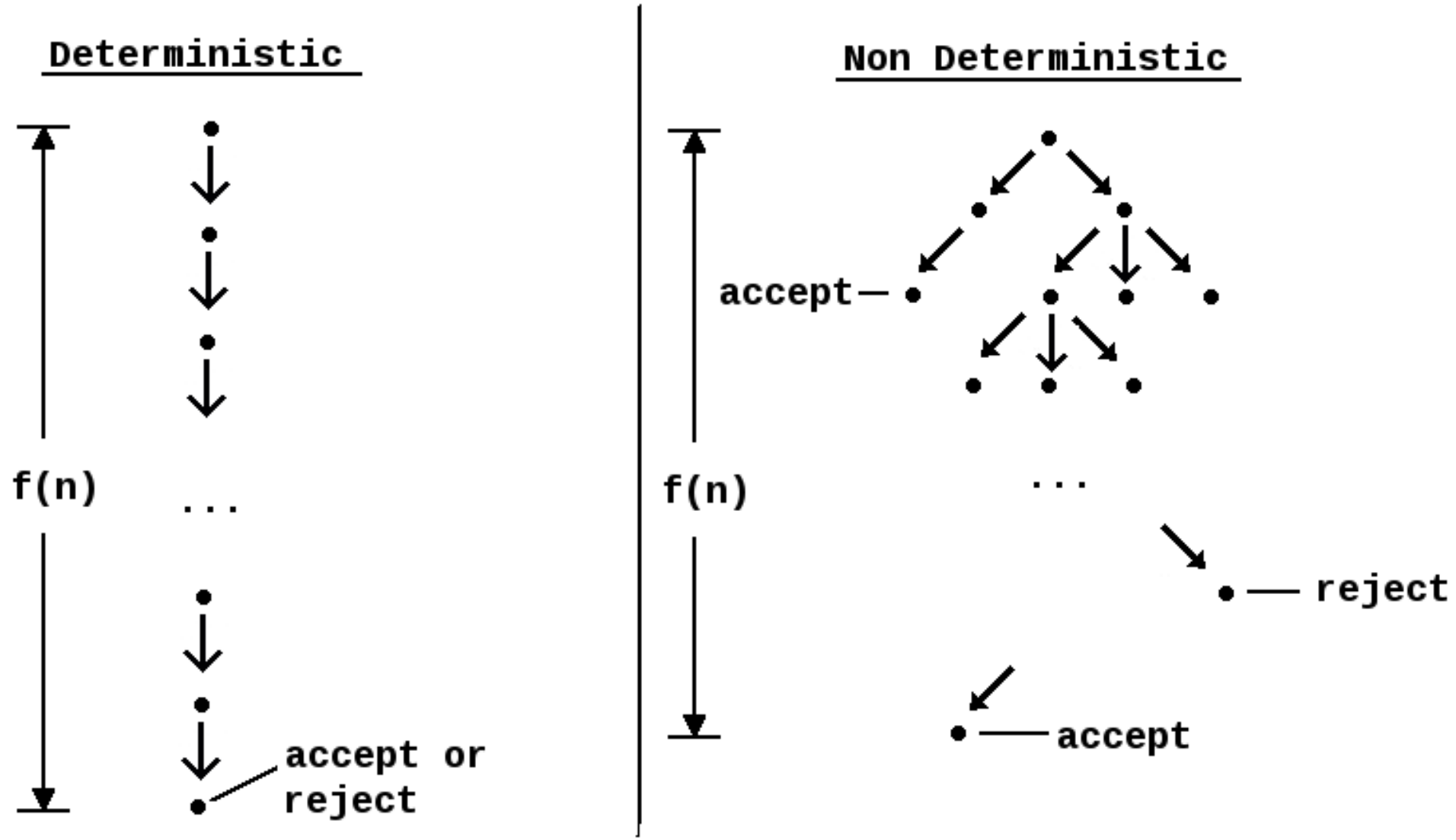
On Java 8 by Bruce Eckel

Dev practices

concurrency is to get your program to run faster. In Java, concurrency is very tricky and difficult, so absolutely do not use it unless you have a significant performance problem—and even then, use the easiest approach that produces the performance you need, because concurrency rapidly becomes unmanageable.

On Java 8 by Bruce Eckel

#4 Non deterministic behavior



#4 Non deterministic behavior

```
public class VanHack {
    public static void main(String[] args) {
        int i = 0;

        i++;

        System.out.println(i); // 0
    }                          // 1
}
```

#4 Non deterministic behavior

```
public class VanHack {
    public static void main(String[] args) {
        int a = 0;
        int b = 0;

        a = a + 1;
        b = b + 2;

        System.out.println(a + " " + b);
        //a=1, b=0
        //a=0, b=2
        //a=1, b=2
        //a=0, b=0
    }
}
```

#4 Non deterministic behavior

```
public class VanHack {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();

        map.put(1, "Artyom");
        map.put(2, "Alicia"); //Can be boom

        System.out.println(map);
    }
}
```

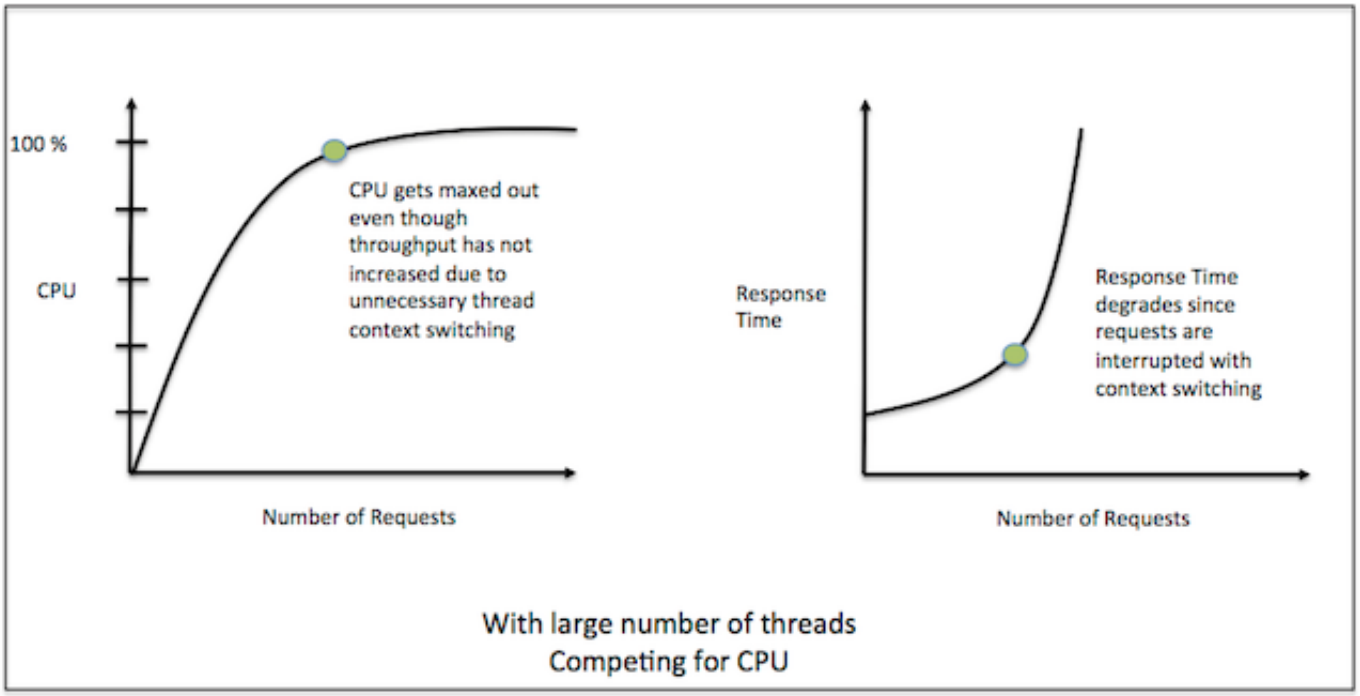
Are you still here?

Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe

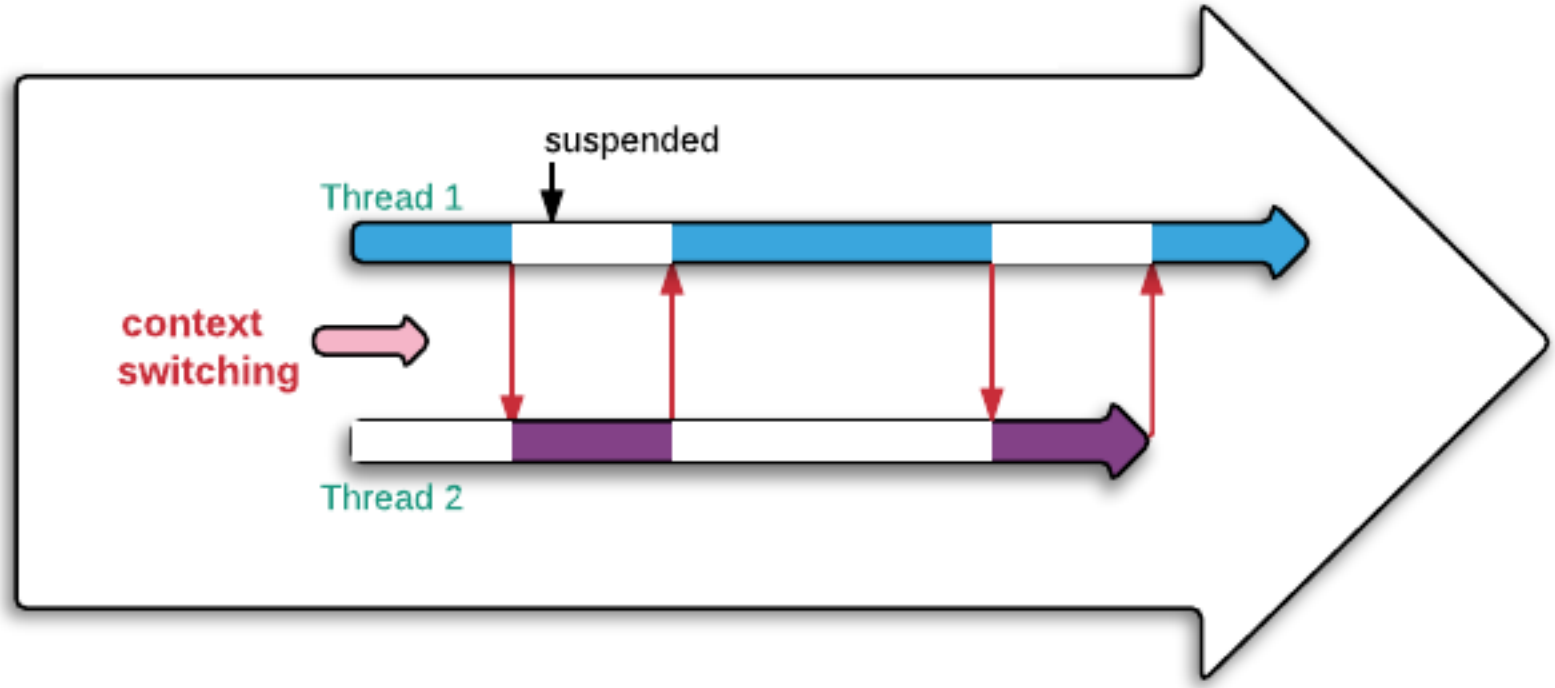
Jabberwocky by Lewis Carroll

Let’s continue?

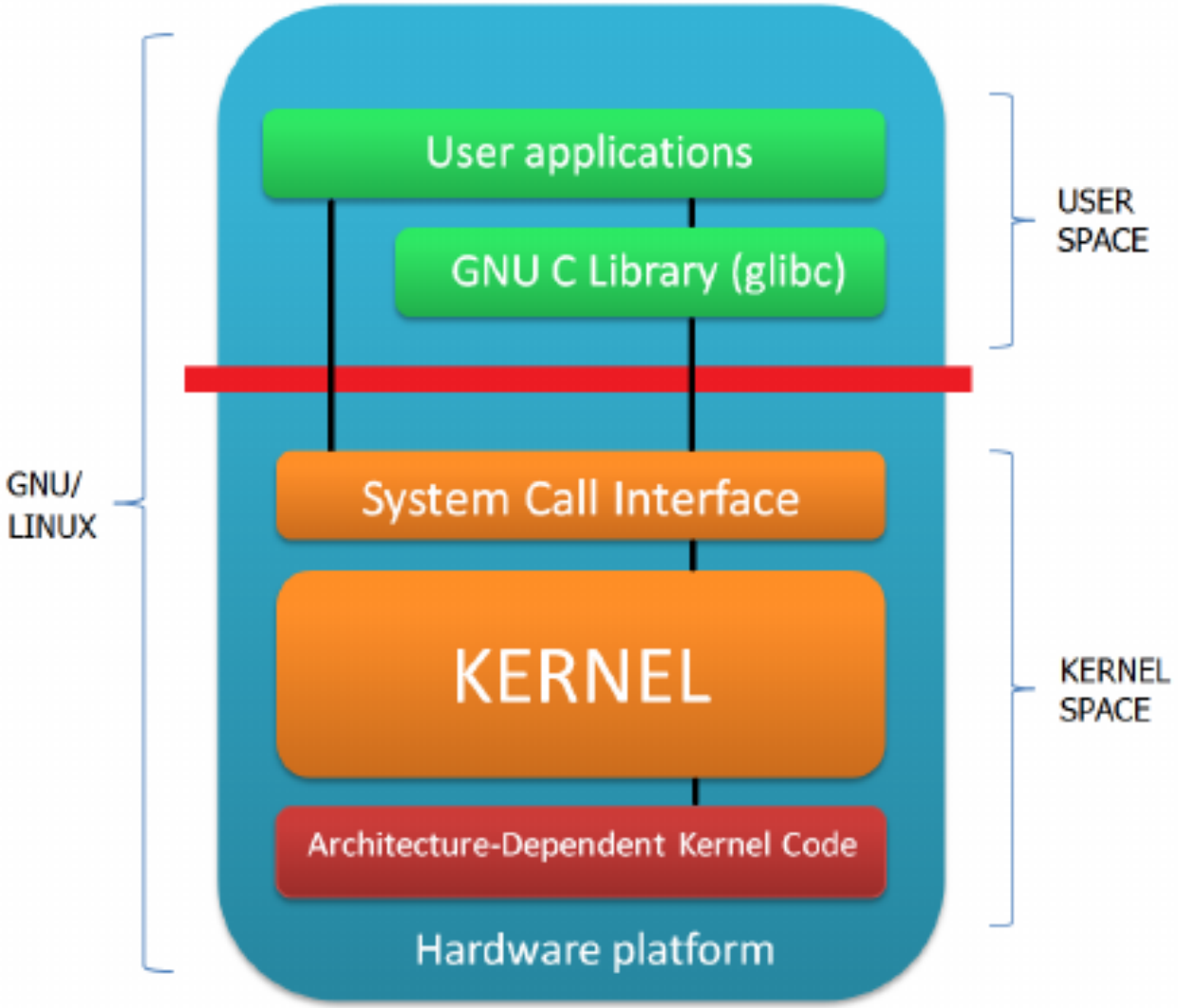
#5 Performance degradation



#5 Performance degradation

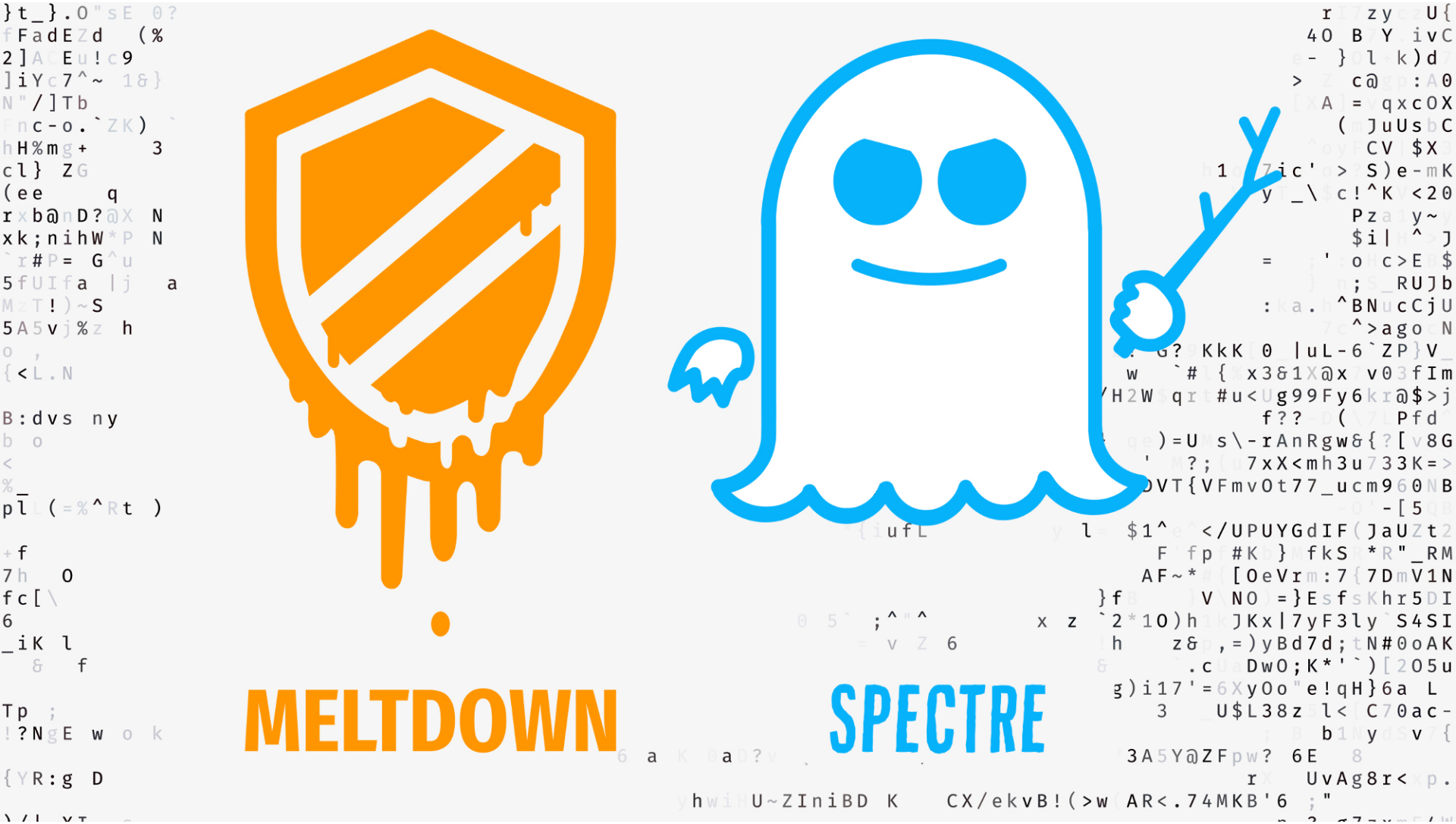


#6 OS limitation



Stop it? Hell no…

#7 Hardware unpredictable optimizations



Concurrency in real world

Java Concurrency

After grappling with Java concurrency over many years, I developed these four maxims:

1. Don’t do it
2. Nothing is true and everything matters
3. Just because it works doesn’t mean it’s not broken
4. You must still understand it

On Java 8 by Bruce Eckel

Conclusion

- Concurrency is powerful practice, you can achieve a lot using it
- Don’t use concurrency if you can avoid it
- Use best practices by experts
- Use frameworks with big professional community
- Minimize synchronization and data sharing
- Perform load and stress testing
- Use as much as possible existed solutions
- Don’t forget about mad world and wonders

Resources

- Java concurrency in practice (Brian Goetz)
- On Java 8 (Bruce Eckel)
- The Art of Multiprocessor Programming (Maurice Herlihy , Nir Shavit)

THANK YOU!

artyom-karnov@yandex.ru, Saint-Petersburg, 2019