Threads

COMP 401 Fall 2018

Lecture 21

Threads

- As a generic term
  - Abstraction for program execution
    - Current point of execution.
    - Call stack.
    - Contents of memory.
  - The fundamental unit of processing that can be scheduled by an operating system.
- Multithreading
  - A program running two or more threads concurrently.
    - Separate points of execution.
    - Separate call stacks.
    - Shared memory.

Rise of Threads

- In the beginning was the command line...
  - Neal Stephenson
- First computers were batch uniprocessors
  - Ran one program at a time.
  - Non-interactive
  - No need for multithreading.

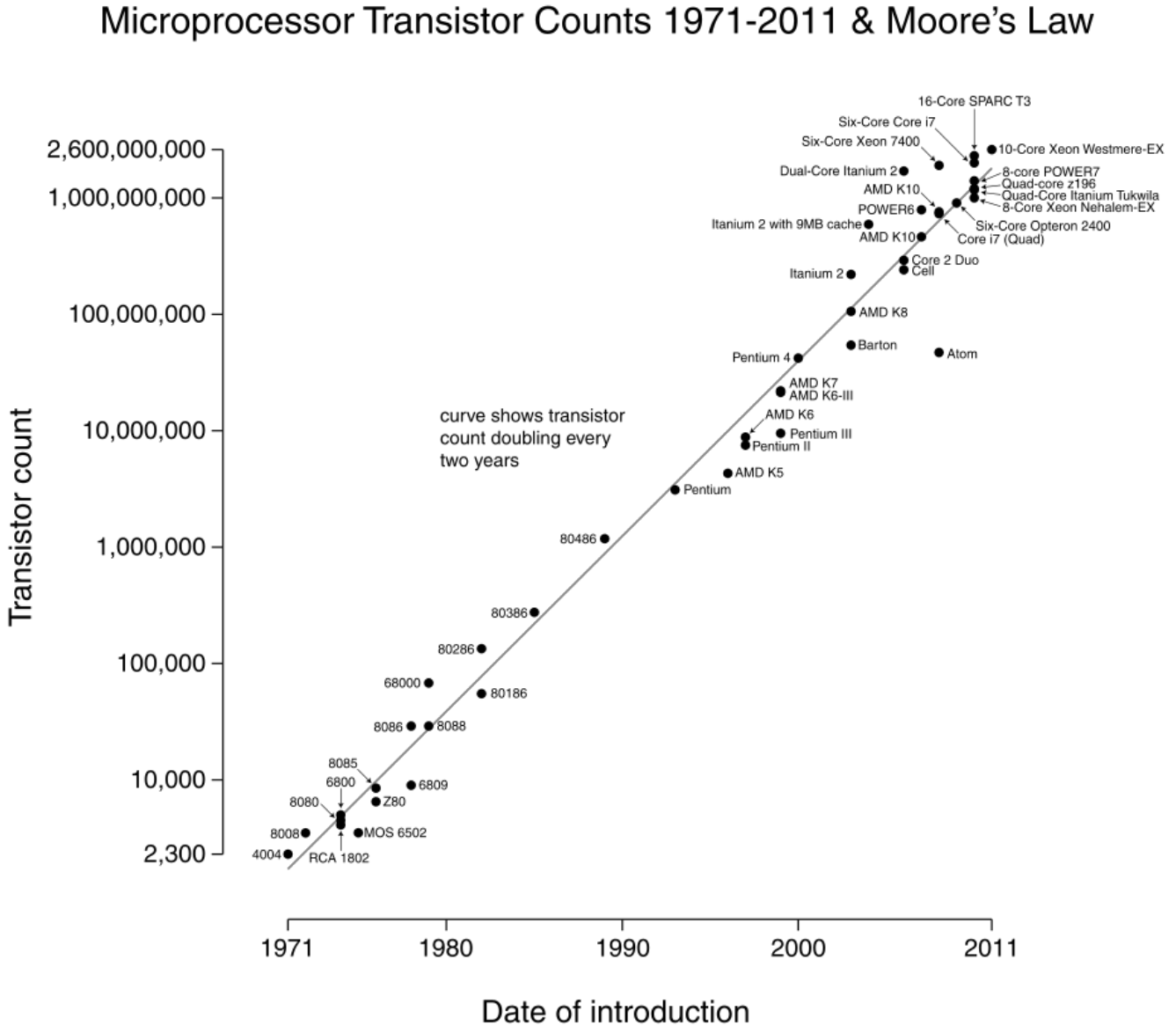
Time Division

- Computers getting faster/cheaper gave rise to interactive computing with GUIs.
  - More than one program at a time.
  - Processor time division
    - Operating system rapidly switches between separate processes.
      - Resource sharing.
      - Not quite multithreading
        - » Separate processes (i.e., programs)
        - » No memory sharing.
      - Illusion of private resource.

Threads For Time Division

- Threads extend OS mechanisms for process-level time division to within a program.
  - Program needs/wants to make progress on two or more tasks.
    - Similar to the idea of two separate processes sharing the processor to make "simultaneous" progress.
    - But in this case, two threads that are part of *same* program
      - Need to share memory and coordinate actions.
- Example: GUI
  - Want GUI to remain responsive when CPU heavy task is occurs.
  - lec21.v1

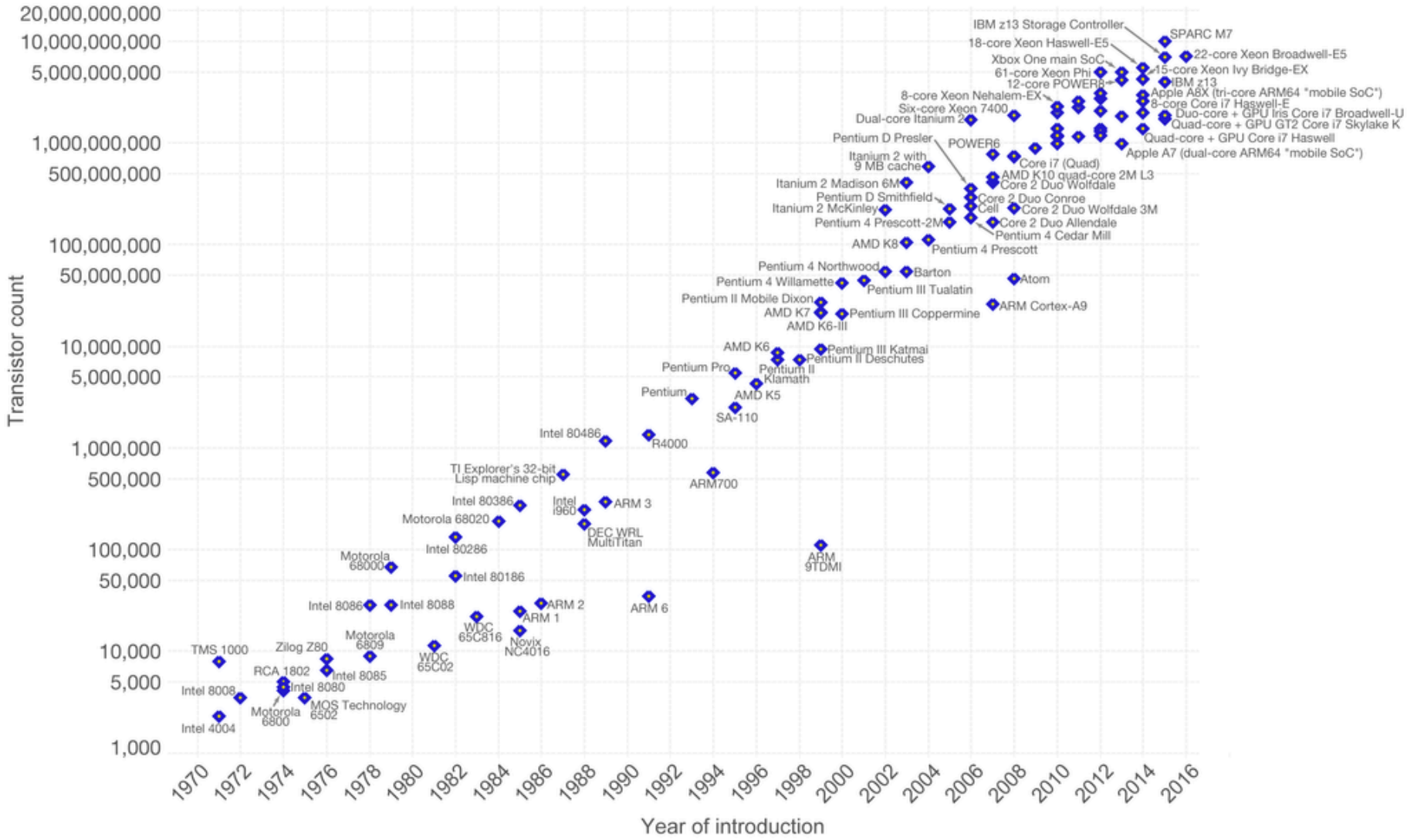
The Multicore Revolution



Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



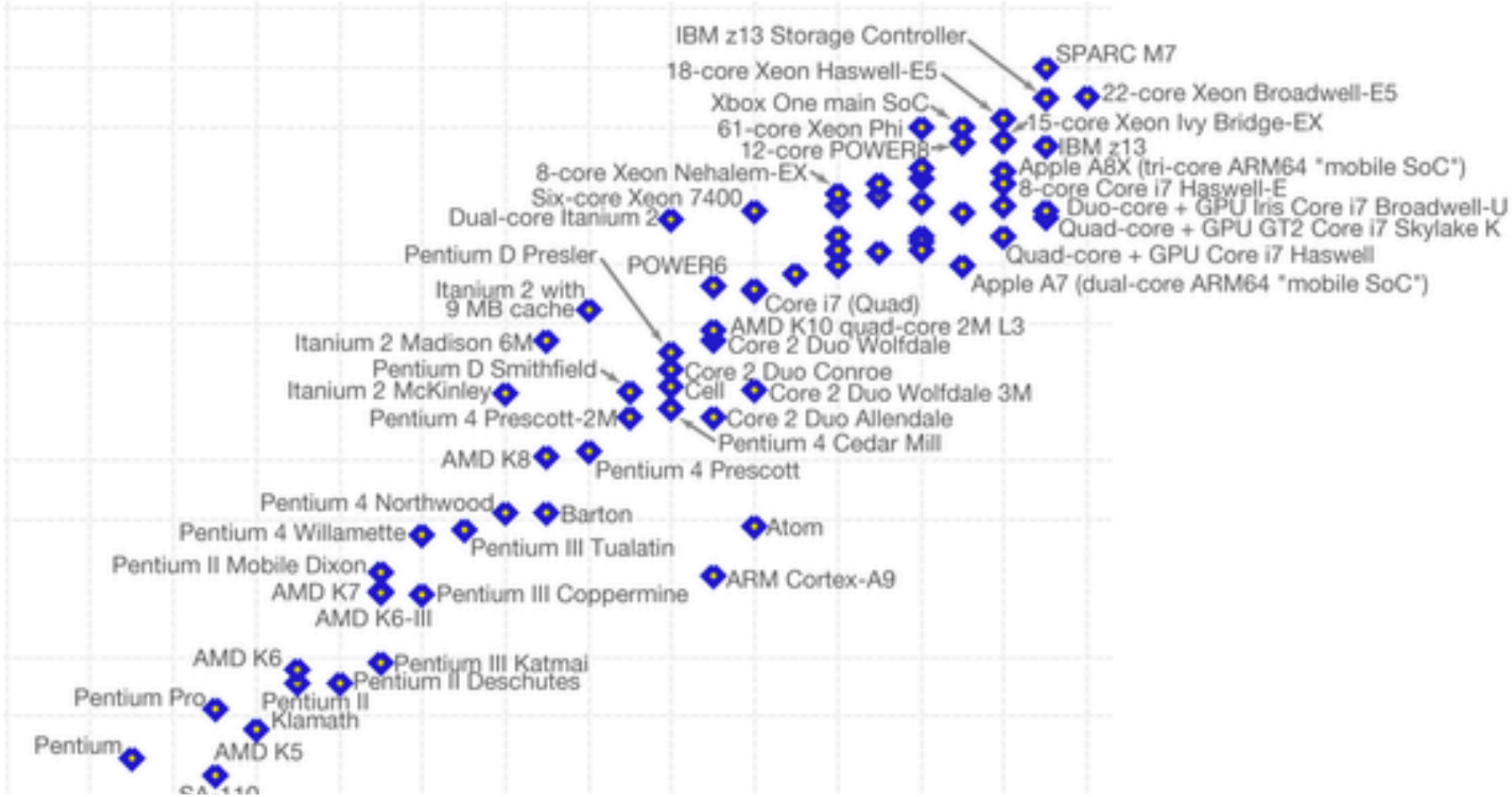
Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

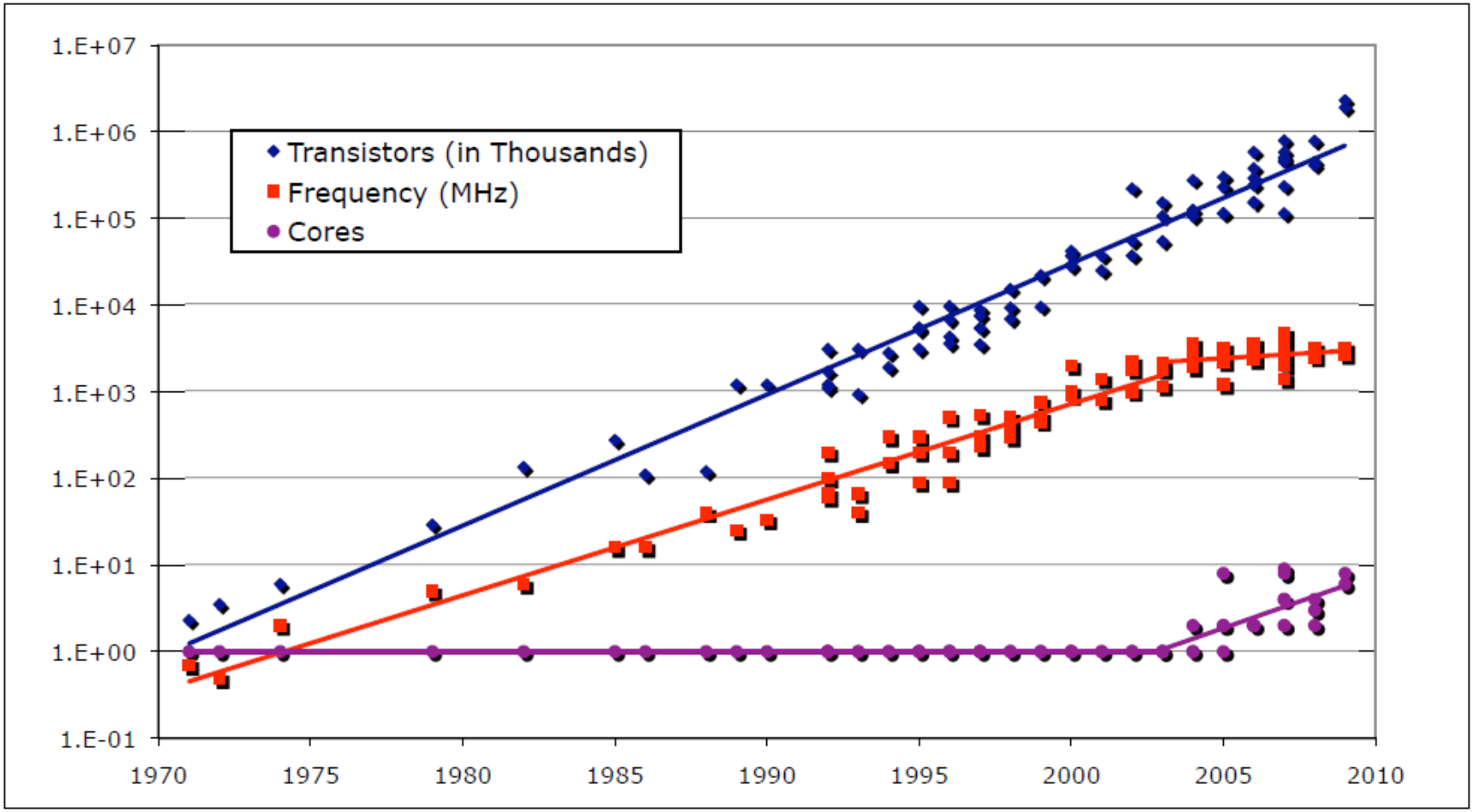
rs on integrated circuit chips (1971-2016)

Our World in Data

ransistors on integrated circuits doubles approximately every two years.
ogress – such as processing speed or the price of electronic products – are



Core Count Driving Moore's Law



https://madusudanan.com/blog/scala-tutorials-part-9-intro-to-functional-programming/

Parallel Programming

- Multicore increases performance only if we can find ways to parallelize our task.
  - Sometimes very easy.
    - No data or logic dependencies.
    - Aggregating data operations.
  - Sometimes very hard.

Java Threads

- Two basic mechanisms for threads:
  - Extend Thread
    - Override run()
    - Call start()
      - Creates a new thread.
      - Executes run() in that thread.
    - lec21.v2
      - Note: only works once.
  - Implement Runnable
    - Provide reference to “runnable” object as parameter of Thread constructor.
    - Call start() on the thread object.
    - lec21.v3

Consistency

- Multi-threaded access to memory can be hazardous.
  - Operation ordering
    - May need all threads to see changes to memory in a consistent way.
      - Actual order may or may not matter as long as all threads see the same effective order.
  - Exclusive access
    - Operation may involve many steps which must be done with exclusive access to memory.
      - Other threads must be prevented from accessing memory while operation in progress.
- lec21.v4

Object Monitor

- Every object in Java associated with a "monitor".
  - Provides ability to "lock" the object.
    - Only one thread can "own" the monitor at a time.
    - Strategy is to lock object before executing any unsafe code.
  - Two ways to acquire the monitor lock:
    - synchronized methods
    - synchronized statements

Synchronized Methods

- "synchronized" keyword
  - public synchronized void a_method() {...
- All synchronized methods obtain monitor lock of object before execution.
  - Automatically released when method returns.
  - This means that for all synchronized methods of an object, only one can be executing at any given time.
- Does not affect unsynchronized methods.
  - This means that all methods that could cause consistency problems must all be synchronized.

Synchronized Statements

- Syntax:

```
synchronized (object) {
   ...
}
```

- Lock obtained on monitor associated with object before statement block is executed.
- Allows synchronization to be wrapped around a small bit of code.
- Allows synchronization using object reference.

Synchronization Equivalence

- These two are equivalent:

```
public synchronized void a_method() {
...
}

public void a_method() {
   synchronized(this) {
      ...
   }
}
```

Consistency Example Revisited

- lec21.v5

Deadlock

- Synchronization can be tricky.
  - Consider:
    - Synchronized method in object A calling a synchronized method in object B.
      - Obtained lock on A
      - Waiting for lock on B
    - At same time in another thread, suppose synch. method of object B is calling synch. method in object A.
      - Obtained lock on B
      - Waiting for lock on A
    - Now neither thread can continue.
    - This is called deadlock.
- lec21.v6

Join

- Sometimes one thread needs to wait for another thread to be done before continuing.
  - A common form of thread coordination.
- join()
  - Method provided by Thread.
  - When called in a different thread, the calling thread is blocked until called thread is done.
    - Done = run() method has finished.
    - Blocked means join() does not return until called thread is done (thus calling thread is waiting).
- lec21.v7

wait() and notify()

- Mechanisms for coordination between running threads.
  - These are methods defined by Object
    - Must own the lock associated with the object in order to call any of these methods.
      - Effectively means that call to wait() / notify() must be within a synchronized method / statement.
  - wait()
    - Calling thread waits until notified.
  - notify()
    - Releases one waiting thread (as soon as lock is available)
  - notifyAll()
    - Releases all waiting threads
      - Each resumes in turn as lock becomes available.
- lec21.v8

sleep() and timing execution

- Thread.sleep(int ms)
  - Sleeps for ms milliseconds
    - Accuracy not guaranteed
- System.nanoTime()
  - Highest resolution timer available in Java
  - Return value is *long*
  - Difference in value between two different calls represents duration in nanoseconds.
    - Actual resolution is courser than a nanosecond.
- lec21.v9

Demonstrating Parallelism

- Generating 100 million random numbers
  - With varying number of threads from 1-16
  - How many cores does my machine have?