Concurrency with Threads

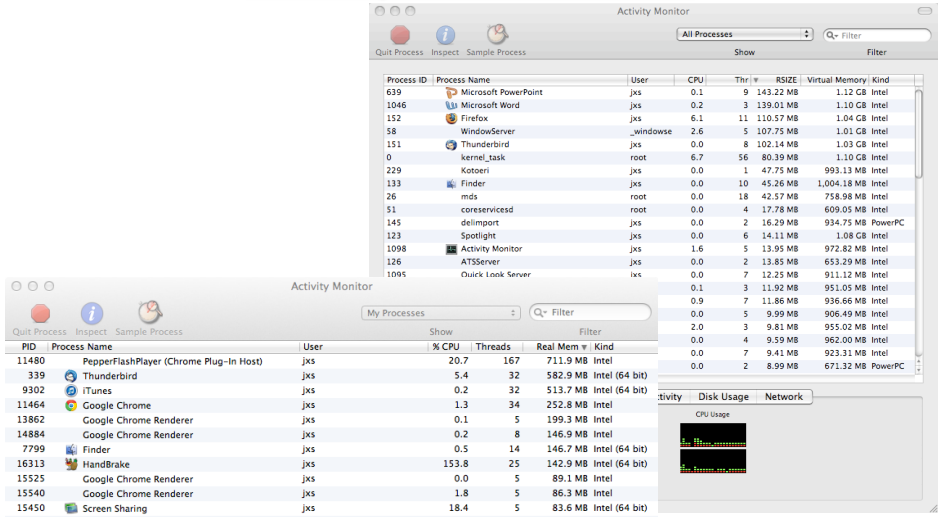
Processes

- One of the most important concepts in all modern operating systems
- A process is a container (or execution environment) for a program (software) in execution.
- Any software is structured as a collection of processes.
  - Java VM (JVM), MS Word, Excel, PPT, Firefox, iTunes, Google Chrome, etc.

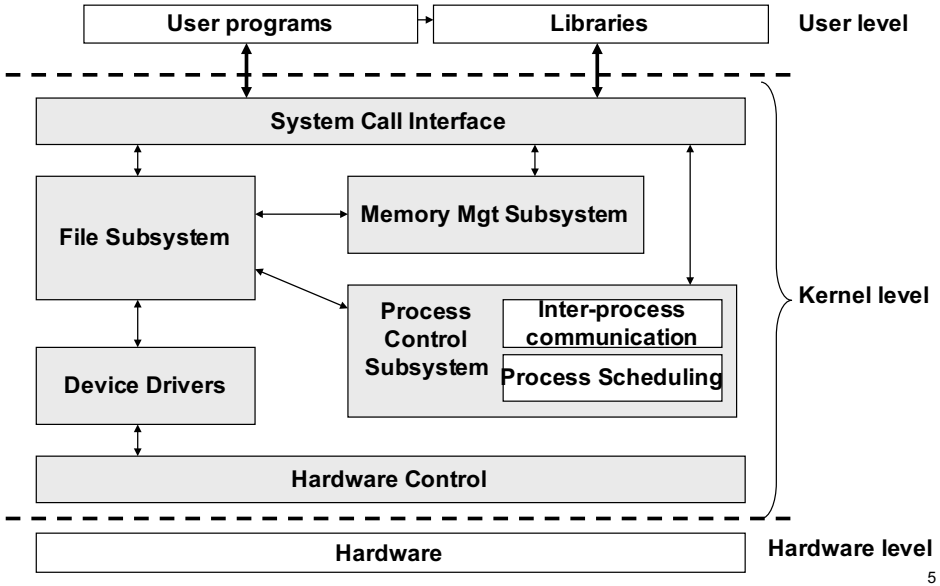
Multi-Tasking (Time Sharing): Concurrency with Processes

- All modern OSes support multi-tasking.
  - e.g., writing a document with MS Word while playing a music with iTunes and downloading files with a Web browser
- At any moment, a single CPU (more precisely, a single CPU core) can execute a single program.
- *Pseudo parallelism*: an OS periodically assigns one program to another to each CPU (core).
  - every several tens of milliseconds to several hundreds of milliseconds
  - A CPU core is multiplexed among processes; processes are not executed in a completely parallel manner.

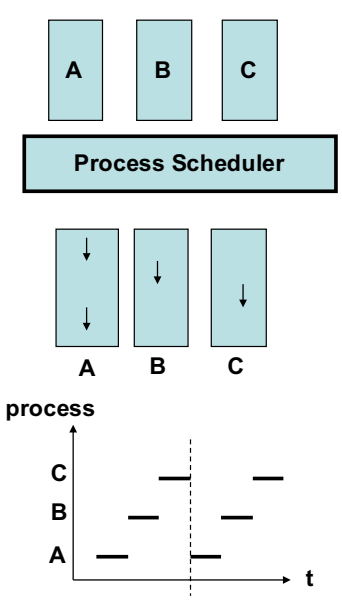
Example Processes

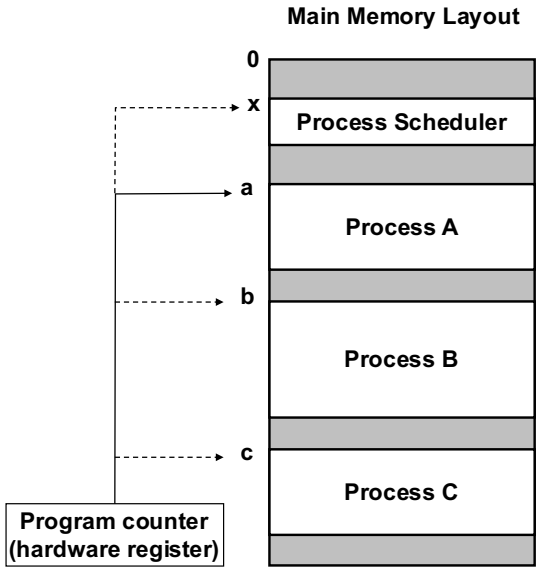


An Architectural View of an OS



Process Scheduling



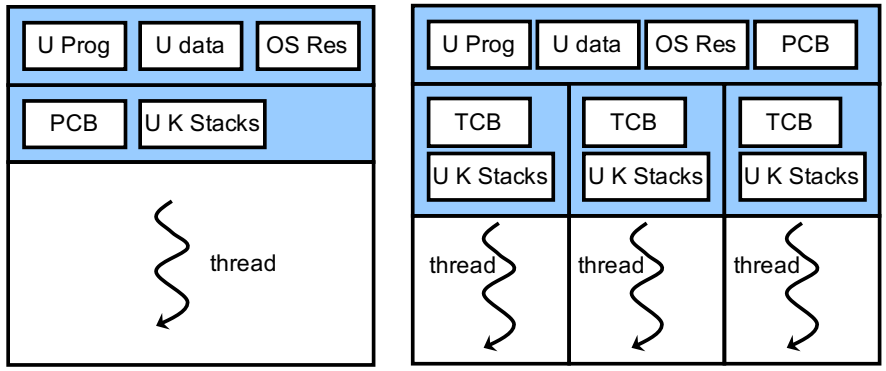


Why Threads?

- Good old days: Coarse-grained concurrency
  - Writing a document with an editor, while sending/downloading emails
- Now: Fine-grained concurrency
  - Each program is expected to do different things concurrently.
    - Word
      - Displaying text and images
      - Responding to keystrokes and mouse inputs from the user
      - Downloading fancy document templates from MS web site
      - Performing spelling and grammar checking in the background
    - Web browser
      - Displaying text and images
      - Responding to keystrokes/mouse inputs
      - Checking and downloading software updates
      - Multiple tabs
    - iTunes
      - Playing music
      - Downloading music and its metadata (album's cover img, song titles, lyrics…)
      - Downloading podcasts

- Fine-grained concurrency (cont'd)
  - A program is expected to do the same or similar things at the same time.
    - Web server
      - Accepts and parses an HTTP request
      - Finds a target file
      - Makes an HTTP message (header, payload, etc.)
      - Returns a target file with the HTTP message
- Assign threads to in-program concurrent tasks

- Process-creation is heavyweight.
  - Time-consuming and resource intensive.
    - Creating a process is 30 times slower than creating a thread.
    - Process switching is 5 times slower than thread switching.



Single-threaded (traditional) process

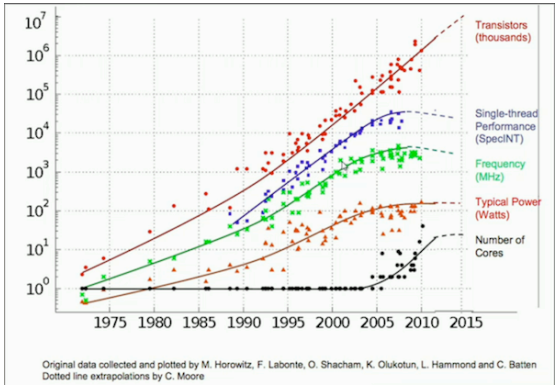
Multi-threaded process

Summary: Why Threads?

- Responsiveness
  - Threads allow a program to continue running even if a part of it is blocked for I/O or is performing a long operation.
- Resource sharing
  - Threads share the memory and other resources of a process that they belong to.
- Efficiency
  - Threads are more lightweight than processes.
    - Process creation is expensive.
    - Switching processes is heavyweight.

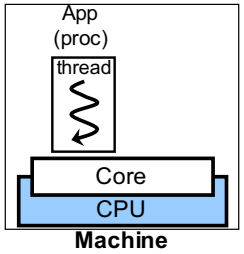
Other Viewpoints to Threads

- CPU speed does not increase any more.
  - Physical material barrier to increase clock speed and # of transistors in a CPU
    - Heat (and cooling) problems

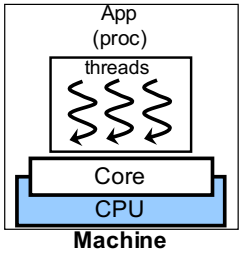


- Each CPU increases its density of cores, not its speed.
  - Intel Core i7: up to 4.0 GHz and up to 8 cores
  - Intel Xeon: up to 4.0 GHz and up to 8 cores
  - Multiple relatively-slower CPU cores for mobile devices to reduce power consumption
    - Amazon Kindle Fire ($50): Quad-core 1.3 GHz
    - Apple iPhone 6: Dual-core 1.38 GHz
- GPUs are designed with a massive # of cores from the beginning.
  - NVIDIA Tesla K80: 4992 cores
- You are expected to increase your app's performance by increasing its concurrency.

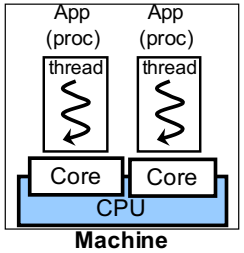
Terminology: Single- or Multi-threaded?



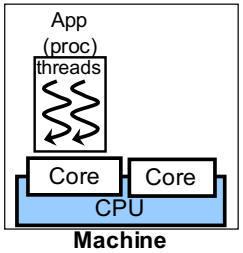
Single-threaded (sequential)



Multi-threaded (concurrent)

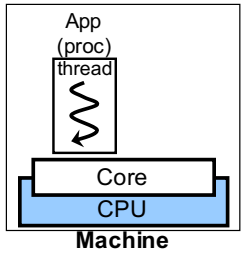


Single-threaded (sequential)



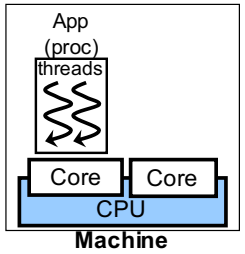
Multi-threaded (concurrent)

Terminology: Concurrent or Parallel?



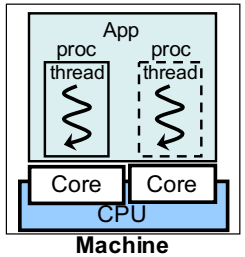
[1] Single-threaded (sequential)

Not parallel



[2] Multi-threaded (concurrent)

Not parallel



[3] Multi-threaded (concurrent)

Parallel

The terms "concurrent" [2] and "parallel" [3] are not always equivalent.

If you think [3] is a special case of [2], you may prefer the term "concurrent."

If you think [2] is a special case of [3], you may prefer the term "parallel."

In Java…

- The terms "concurrent" and "parallel" are somewhat mixed up in Java.
  - Java 5 (2004) introduced `java.util.concurrent`, which was further enhanced by Java 6 (2006).
    - "Concurrent" collections such as `ConcurrentHashMap`
    - The Executor framework
      - An extension/abstraction over low-level threads
  - Java 7 (2011)
    - The Fork/Join framework, an extension to the Executor framework
      - The term "parallel" appeared in its documentation, although it was placed in `java.util.concurrent`.
    - "Parallel" garbage collector
  - Java 8 (2014)
    - "Parallel" collection streams, which allows for "parallel" (threaded) operations on collections with lambda expressions
      - e.g. MapReduce-inspired API extension for collections

- Java 9 (2016?)
  - An extension to collection streams?
  - "Automatic parallelization" with OpenCL?
    - Automatically converting sequential code into multi-threaded code

Java Threads

Java Threads

- Every Java program has at least one *thread of control*.
  - main() runs with a thread on a JVM.
    - The "main" thread is *implicitly* created when a JVM starts.
- If you need extra threads in addition to the main thread, you need to *explicitly* create them.
- 4 things to do:
  - Define a class implementing the **`java.lang.Runnable`** interface
    - **`public abstract void run();`**
  - Write a threaded/concurrent task in **`run()`** in the class
  - Instantiate **`java.lang.Thread`** and associate a Runnable object with the thread
  - Start (call **`start()`** on) the instantiated thread.
    - **`run()`** is automatically called on the thread.

An Example Code: Creating a Thread

- HelloWorldTest.java
- GreetingRunnable.java
- Output:
  - Mon Mar 26 15:14:43 EDT 2007 Hello World
  - Mon Mar 26 15:14:44 EDT 2007 Hello World
  - Mon Mar 26 15:14:45 EDT 2007 Hello World
  - Mon Mar 26 15:14:46 EDT 2007 Hello World
  - Mon Mar 26 15:14:47 EDT 2007 Hello World
  - Mon Mar 26 15:14:48 EDT 2007 Hello World
  - Mon Mar 26 15:14:49 EDT 2007 Hello World
  - Mon Mar 26 15:14:50 EDT 2007 Hello World
  - Mon Mar 26 15:14:51 EDT 2007 Hello World
  - Mon Mar 26 15:14:52 EDT 2007 Hello World

Thread.start()

- Creating a Thread object does not mean creating a new thread.
  - It is start() that actually creates a thread.
- start()
  - Allocates memory and initializes a new thread on a JVM.
  - Calls run() of a specified Runnable object.
    - Do not call run() directly, but let start() call run() on behalf of yourself.

An Example Code: Creating Threads

- HelloWorldTest2.java and GreetingRunnable.java
- Output:

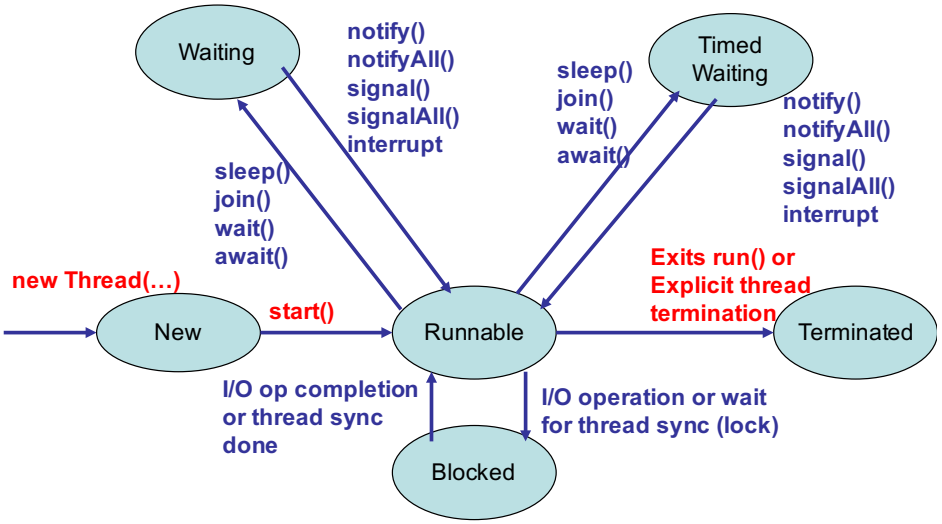
```
Mon Mar 26 15:28:45 EDT 2007 Goodbye World
Mon Mar 26 15:28:45 EDT 2007 Hello World
Mon Mar 26 15:28:46 EDT 2007 Hello World
Mon Mar 26 15:28:46 EDT 2007 Goodbye World
Mon Mar 26 15:28:47 EDT 2007 Hello World
Mon Mar 26 15:28:47 EDT 2007 Goodbye World
Mon Mar 26 15:28:48 EDT 2007 Goodbye World
Mon Mar 26 15:28:48 EDT 2007 Hello World
Mon Mar 26 15:28:49 EDT 2007 Goodbye World
Mon Mar 26 15:28:49 EDT 2007 Hello World
Mon Mar 26 15:28:50 EDT 2007 Goodbye World
Mon Mar 26 15:28:50 EDT 2007 Hello World
Mon Mar 26 15:28:51 EDT 2007 Goodbye World
Mon Mar 26 15:28:51 EDT 2007 Hello World
Mon Mar 26 15:28:52 EDT 2007 Hello World
Mon Mar 26 15:28:52 EDT 2007 Goodbye World
Mon Mar 26 15:28:53 EDT 2007 Hello World
```

- Two message sets (Hello and Goodbye) are not exactly interleaved.

The Order of Thread Execution

- JVM's thread scheduler gives you NO guarantee about the order of thread execution.
- There are always slight variations in the time to execute a threaded task
  - especially when calling OS system calls (typically I/O related system calls)
- Expect that the order of thread execution is somewhat random.

States of a Thread



- New
  - A Thread object is created. start() has not been called on the object yet.
- Runnable
  - Java does not distinguish runnable and running. A running thread is still in the Runnable state.
- Terminated/dead
  - A thread automatically dies when run() returns.

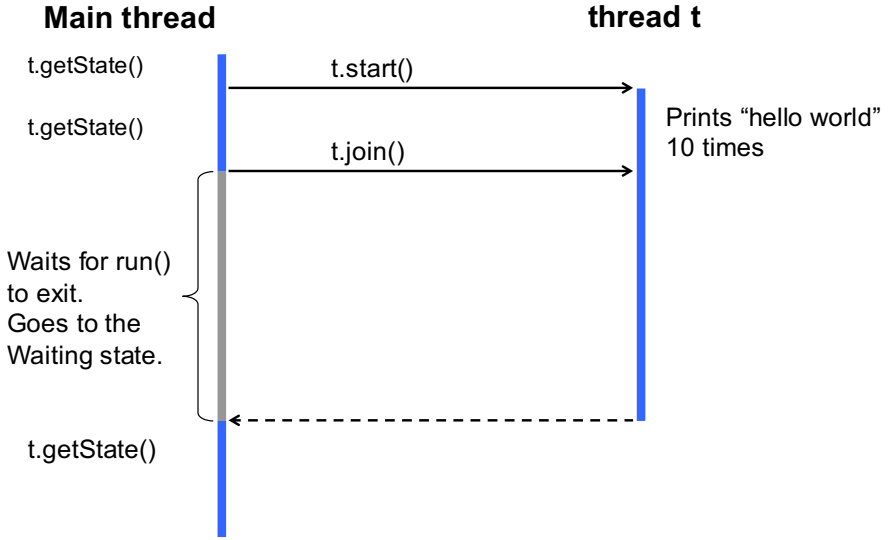
```
public class Thread{
  public enum State{
    NEW, RUNNABLE, BLOCKED, WAITING,
    TIMED_WAITING, TERMINATED }

  public Thread.State getState()

  public boolean isAlive()
```

- Alive
  - in the Runnable, Blocked, Waiting or Timed Waiting state.
  - isAlive() is usually used to poll/check a thread to see if it is terminated.

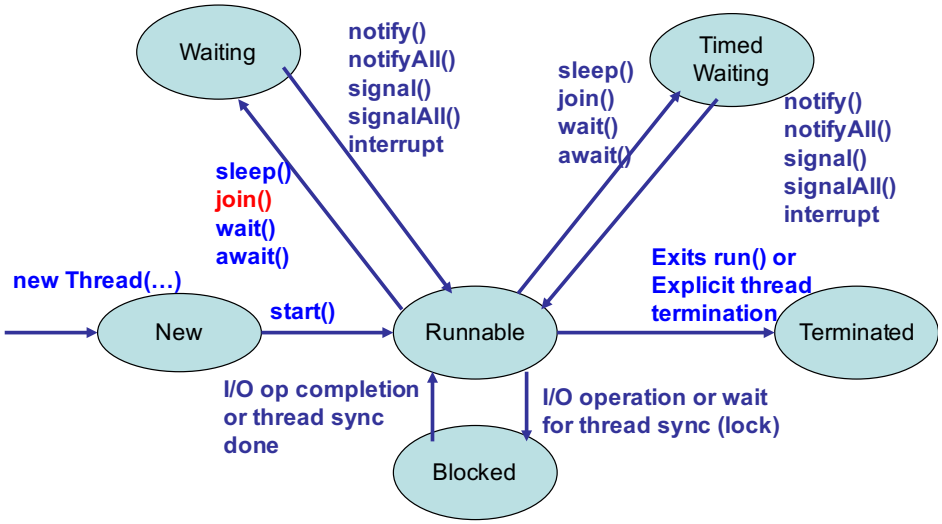
HelloWorldTest3.java

- Output:
  - NEW
  - NEW
  - false
  - false
  - RUNNABLE
  - RUNNABLE
  - true
  - true
  - Wed Mar 28 04:25:01 EDT 2007 Goodbye World
  - Wed Mar 28 04:25:01 EDT 2007 Hello World
  - Wed Mar 28 04:25:02 EDT 2007 Hello World
  - Wed Mar 28 04:25:02 EDT 2007 Goodbye World
  - Wed Mar 28 04:25:03 EDT 2007 Hello World
  - Wed Mar 28 04:25:03 EDT 2007 Goodbye World
  - ……
  - TERMINATED
  - TERMINATED
  - false
  - false

Thread.join()



States of a Thread



Thread Termination

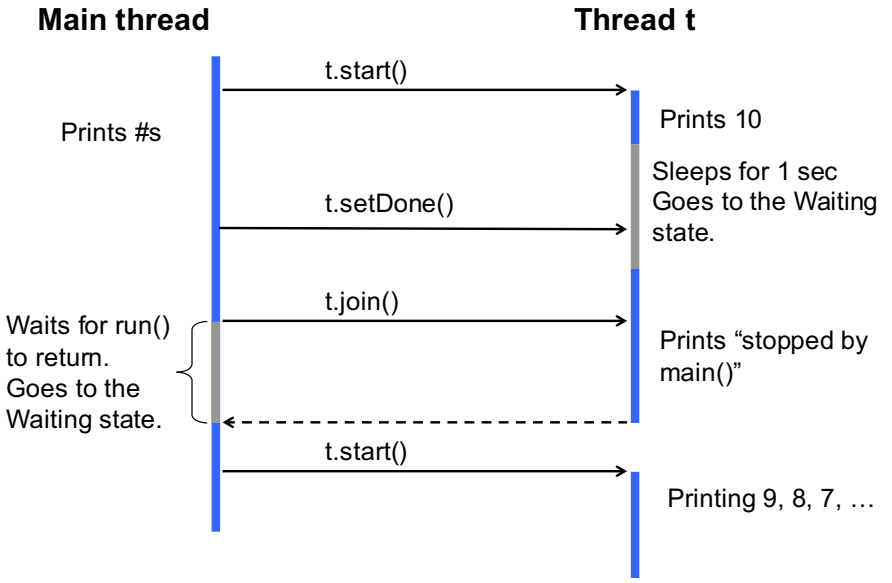
- Implicit termination
  - A thread triggers its own death when run() returns.
    - Once a thread starts executing run(), it continues execution until it returns.
- Explicit termination
  - A thread terminates another thread.
    - when a certain condition is satisfied.
      - e.g., when a tab in a web browser is closed.
  - Two ways
    - Setting a flag
    - Interrupting a thread

Explicit Thread Termination with a Flag

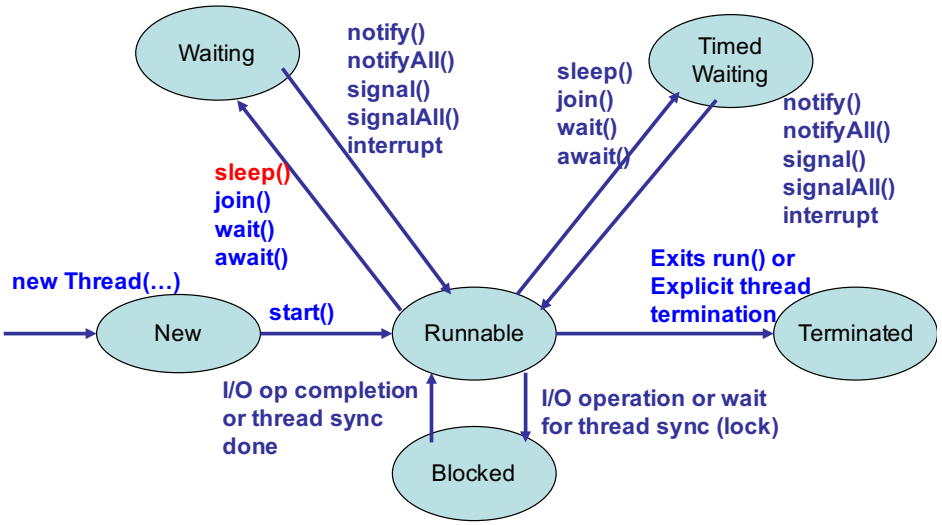
- Stop a thread (i.e., let run() return) by setting a flag to inform that the thread should die.
- The thread periodically checks out the flag to determine if it should stop/die.

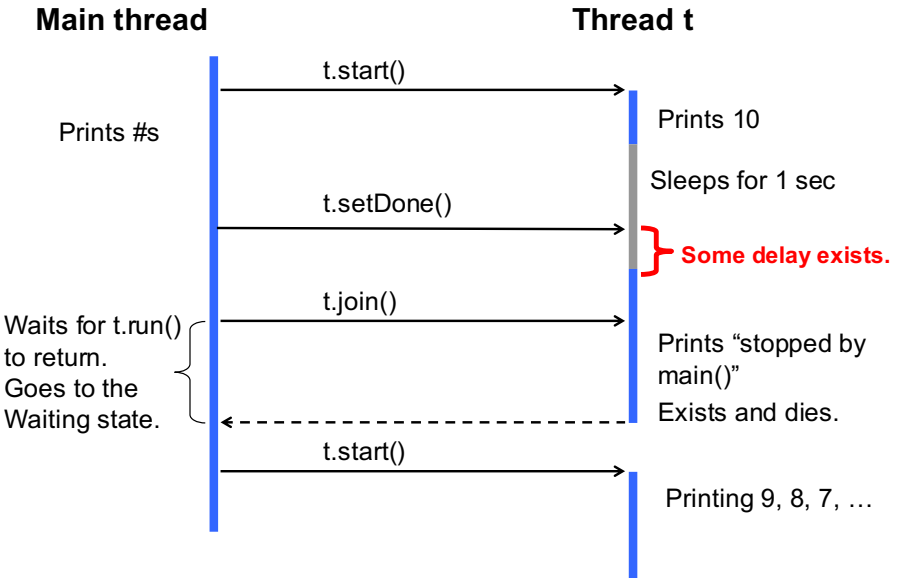
SummationRunnable.java

- Output:
  - #
  - 10
  - #
  - #
  - #
  - #
  - stopped by main()!
  - 9
  - 8
  - 7
  - 6
  - 5
  - 4
  - 3
  - 2
  - 1
  - 0
  - print done



States of a Thread





Thread Interruption

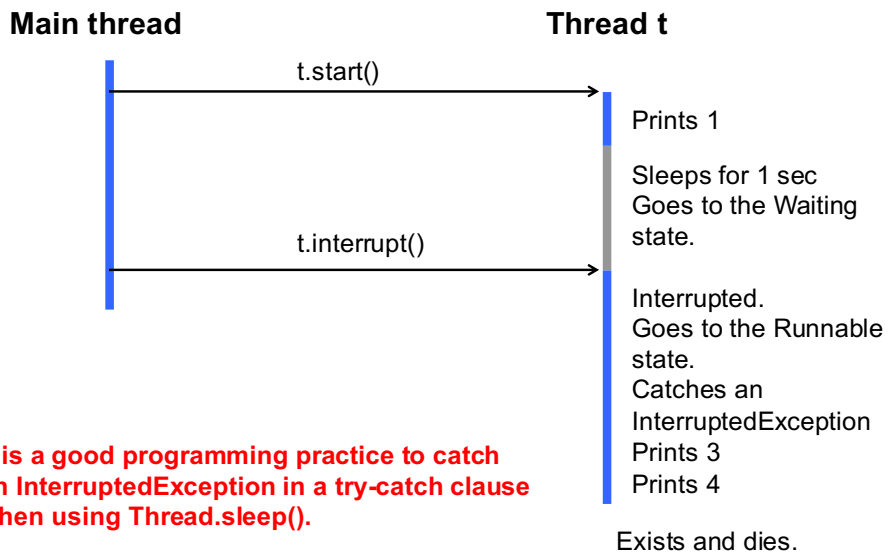
- Interrupt a thread to minimize this delay.
  - the delay between setting a flag and terminating a thread.
  - Signals a thread that the thread should stop/exit immediately.

- Thread.interrupt()
  - Can interrupt (or wake up) a sleeping thread.

Thread Termination with interrupt()

- InterruptableTask.java

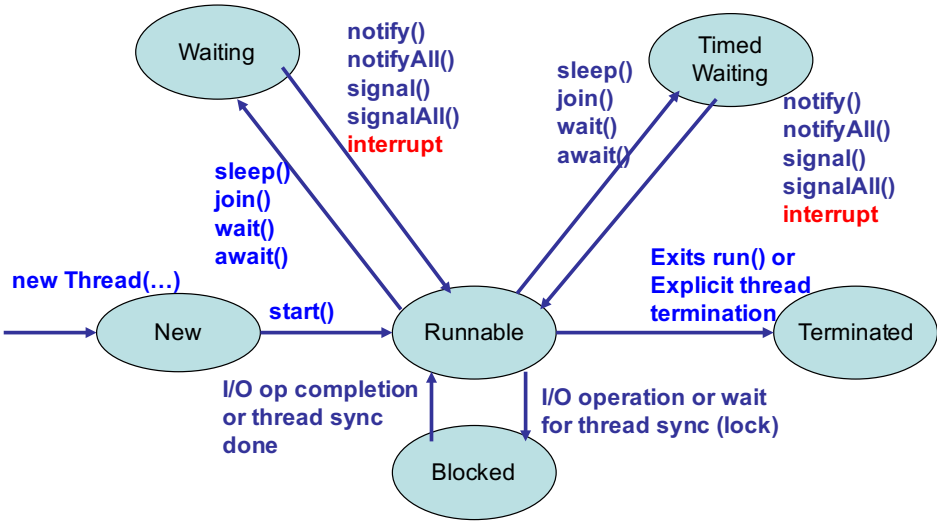
InterruptableTask.java



It is a good programming practice to catch an InterruptedException in a try-catch clause when using Thread.sleep().

- NOTE: interrupt() can be called on a thread in the Waiting or Blocked state.
  - An InterruptedException is raised on an interrupted thread.
  - A corresponding catch clause is executed.

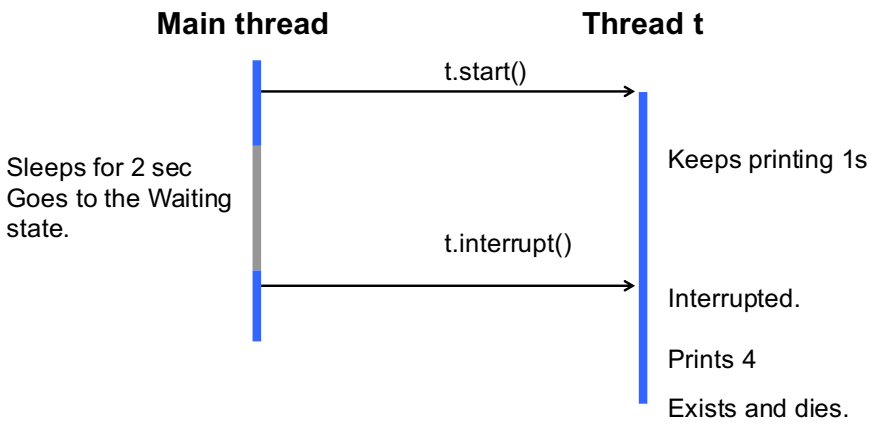
States of a Thread



InterruptableTask2.java

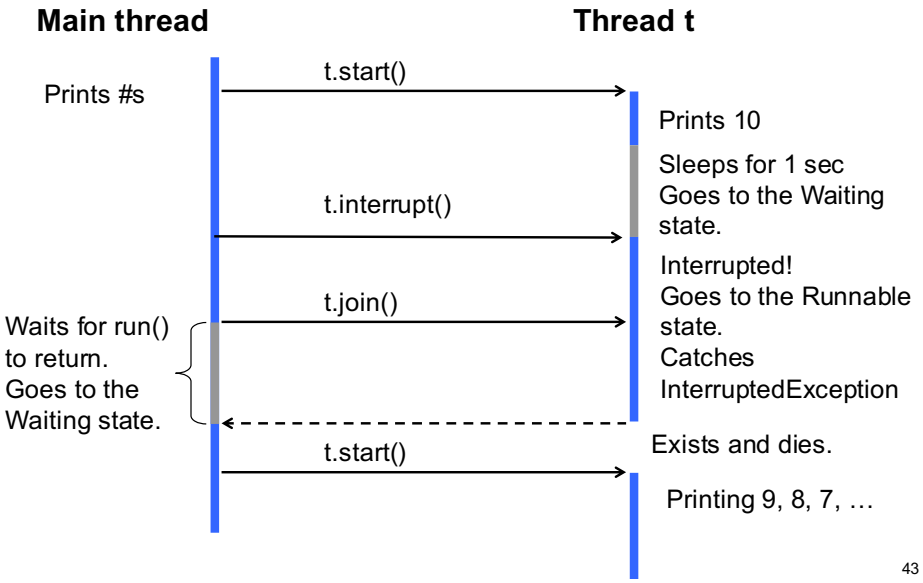
- In fact, interrupt() can be also called on an active thread with the Running state.
- Thread.interrupted() returns true if the currently-executed thread has already been interrupted by another thread.
  - The interrupted state is cleared by this method.

InterruptableTask2.java

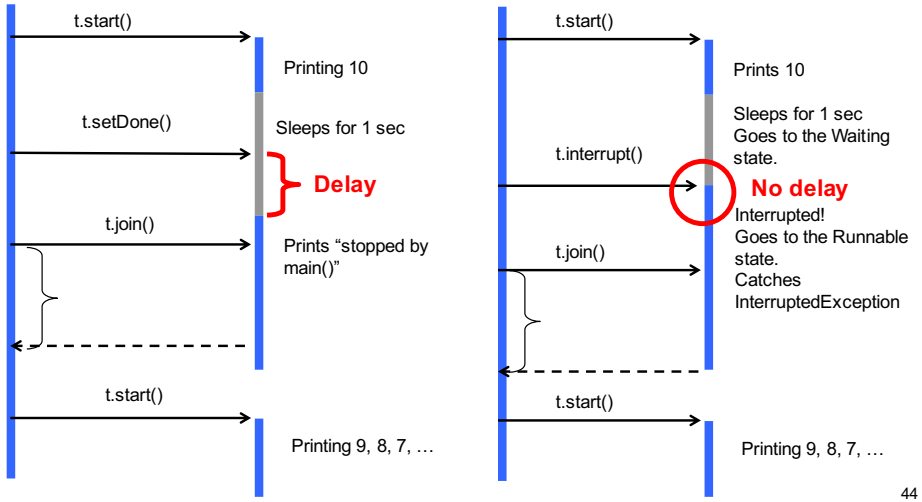


SummationRunnableInterruptable.java

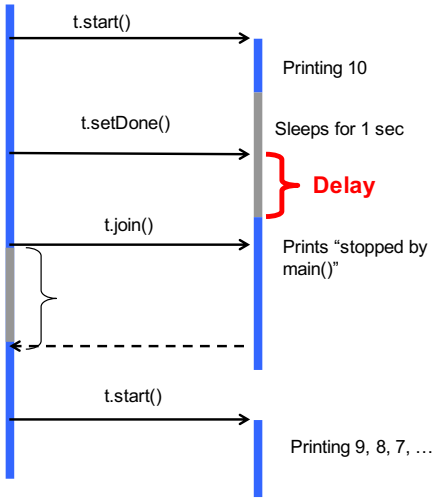
- Output
  - #
  - 10
  - #
  - #
  - #
  - #
  - interrupted by main()!
  - 9
  - 8
  - 7
  - 6
  - 5
  - 4
  - 3
  - 2
  - 1
  - 0
  - print done



SummationRunnable.java and SummationRunnableInterruptable.java



Thread.interrupt() and Blocking Method

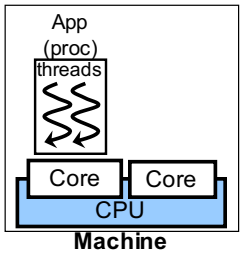


- This kind of delay occurs when a thread calls a *blocking method*.
  - Thread-related methods
    - sleep(), wait(), join()
  - I/O-related methods
    - read()
- If a blocking method never returns on a thread, the thread never stops. (The delay becomes infinite.)
  - e.g. tries to read data with read() from a socket but data never comes in.

If You have a Multicore CPU…

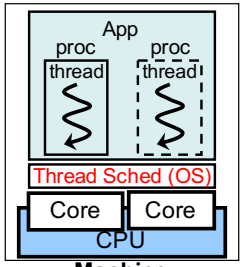
- MCTest.java
  - Mac OS X 1.8.xx on a MacBook Pro
    - Intel Core i7 2.8 GHz (dual core with hyper threading)
    - 8 GB RAM
    - 25 * 25 for 10 billion times
  - With JDK 1.8 (version 1.8.0_72 from Oracle)

# of threads	Time (sec)
1	10.55
2	10.55 (Two threads run in parallel!)
4	17.35 (>10.55, but still << 10.55 * 4)
8	36.29
16	62.05



Multi-threaded (concurrent)

Not parallel



Multi-threaded (concurrent)

Parallel

HW 6

- Modify MCTest.java to use a lambda expression.
  - MCTest.java currently uses an anonymous class to implement run().

```
Thread t = new Thread(
      new Runnable(){
        public void run(){
          int n = 25;
          for (long j = 0; j < nTimes; j++){
            n *= 25;
          }
        }
      });
```

    - Skip implementing a class that implements Runnable.
  - Replace an anonymous class with a lambda expression.
    - Runnable is a functional interface.

Note that…

- It makes less/no sense to use a lambda expression when the code block is long and complex.
- Lambda expressions are useful/powerful when their code blocks are reasonably short.