Parallel and Concurrent Programming with Java

# **Chapter 2: Threads and Processes**

Thread vs Process

Process

* Instance of a program executing
* Includes code, data and state info
* Each process is independent.
* Has own address space in memory
* Computers have many programs, many instances of a program, i.e., many processes, running at any time
* OS’s job is to manage all processes

Thread

* A process is made up of threads
* An independent path of execution through the program
* A sequence of instructions
* Can only exist as part of a process
* OS schedules thread for execution, allocates time to run.
* The most basic units an OS manages

Analogy:

Program to make salad

* 2 cooks = 2 active threads
* 1 thread to retrieve vegetables from pantry
* Another thread to chop vegetables
* If have more tasks to handle, can start new threads. E.g., make salad sauce.
* Once a thread completes executing its instructions, it will exit. Leaves remaining threads running.

Diagram

Description automatically generated

Threads from same process share same memory space

* Access to same resources
* Access to same code and data
* Analogy will be the kitchen space
* Gives shared access to cookbook/recipe (execution codes) and ingredients (data/variables).
* Easy for cooks (threads) to work together, but need to coordinate actions

Sharing resources between processes is harder than sharing resources between threads

* Because every process has separate memory address space
* Example: bake cake vs make salad processes
* Instruction codes (recipes) and data (ingredients) are different
* Need to use system-provided inter-process communication (IPC) mechanisms

Text

Description automatically generated

Which application/program structure is better? Multiple processes or multiple threads?

* Depends on OS and programming languages
* If using multiple computers, multiple processes (instances of same program) is good.
* Multiple threads
  + more light weight, i.e., easier to create and terminate. Use less resources.
  + faster for OS to switch between threads of same process, than to switch between processes

Thread vs Process: Java demo

* Each Java application runs within its own instance of the JVM.
* OS treats each instance of the JVM as a separate, independent process.
* If run multiple Java applications at same time, each will be independent JVM process, with own memory space.
* [Ctrl]+[Shift]+[Esc] opens Task Manager
* Multi-core CPUs have 4/8/12 separate physical processing cores. They support hyper-threading
* Hyper-threading
  + allows each physical core to run 2 independent applications at same time, giving 2X the number in logical processors.
  + Does not 2X performance.
  + optimises use of idle resources. If one thread is waiting or not using certain resource, the other thread may use it.

JDK18 documentation

<https://docs.oracle.com/en/java/javase/18/>

JDK18 Thread Class documentation

<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/lang/Thread.html>

/src/ThreadProcessDemo.java demonstrates the effect of threads running.

While loop for threads to run forever.

Go to /production/ThreadProcessDemo.class folder, type “cmd” in directory bar, type “java ThreadProcessDemo” in cmd window to run. [Ctrl]+C to terminate.

OS sees more threads running for JVM (e.g., 24) than coded for in application/program (e.g., 6). Compare codes against process ID (PID) and Threads count with Task Manager.

* Because of background processes, e.g., garbage collection and runtime compilation.

Concurrent vs Parallel Execution

|  |  |
| --- | --- |
| Concurrent | Parallelism |
| Program broken into parts(threads) that can run independently of each other | Program can be executing same, related, or different unrelated tasks |
| Threads run asynchronously, i.e., partially/completely out of order with one another | Threads run at same time together.  Synchronous execution  Doing things at same time. |
| About program structure | |
| No effect on output/result | |
| For computationally intensive tasks.  Example:  Matrix multiplication | Example:  One thread - Graphical User Interface (GUI)  Another thread – I/O dependent operation.  Concurrency avoids locking up GUI while executing operation. |

Diagram

Description automatically generated

Analogy:

Program – salad recipe

Threads – tasks

Processor – knife

Instructions – chop, mix

Data – lettuce, cucumbers

|  |  |
| --- | --- |
| Single-core | Multi-core |
| If only 1 processor, the 2 threads must take turn. Only 1 thread executing at any instance in time. | Two threads running at same time. |
| concurrent 2 processes/threads  Description automatically generated | Concurrent, 2 threads  Description automatically generated |
| Single core. |  |
|  | Example:  I/O devices – mouse, keyboard, hard disk.  OS concurrently manages their software drivers.  Drivers executed independently as and when needed.  Execution may get split among available processors in multi-core CPU.  But I/O operations infrequent.  All threads can run just as fine on single processor. No benefit from parallel processing. |

# Scheduler

processes, threads

Description automatically generatedScheduler, queue

Description automatically generated

Modern computer has hundreds of processes, thousands of threads.

Scheduler, I/O queue

Description automatically generatedOS’s **scheduler** controls which processes and threads get executed on available CPUs.

Scheduler makes it possible to run multiple programs on single-core processor.

When process is created and ready to run -> it is loaded into memory -> placed in Ready Queue. Scheduler assigns ready processes to available processor/s.

1. A process runs until complete. Then scheduler assigns another process to run on available processor.
2. scheduler, substitute

   Description automatically generatedIf process blocked and need to wait for I/O event, will be moved into I/O queue. Another process assigned to the freed processor.
3. Context switch. If process has spent fair share of time running, scheduler swaps it out for another process in Ready Queue.

* OS must store context/state of paused process/thread to resume later.
* Then load context of replacement process/thread that is about to run.
* Context switching consumes time.
* Different OS Scheduler implement different strategies that affect frequency of switch.
* Example of algo: Pause low-priority thread, run high-priority thread in Ready Queue.
* Choice of algo depends on goal of Scheduler. Example: maximise work done or minimise lag.
* Developer has no control over Scheduler.
* Choice of algo and goal may vary from run to run.
* Do not write codes expecting threads/processes to execute in certain order, or for equal amount of time.

scheduling algorithms

Description automatically generatedscheduling goals

Description automatically generated

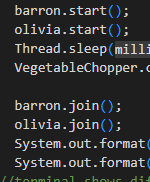
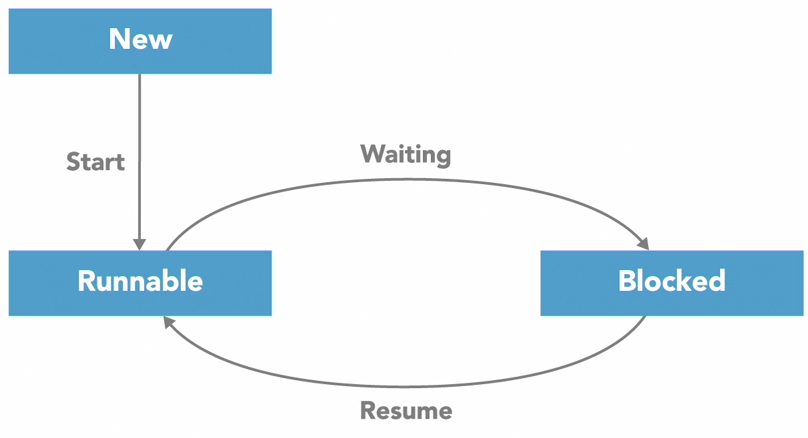
Execution scheduling: Java demo

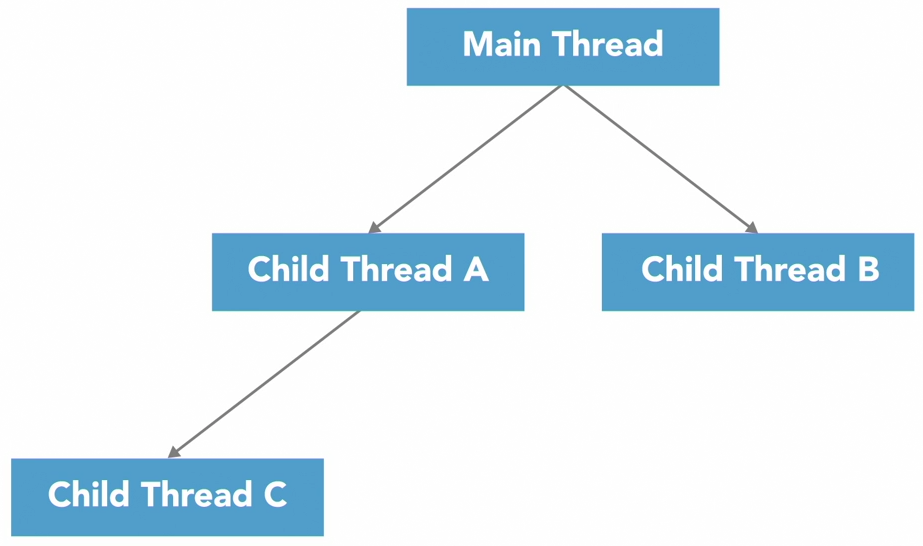
/src/ExecutionSchedulingDemo

Program shows that OS schedules each thread to run for different number of times. 2 instances (Barron and Olivia) of VegetableChopper class are initiated. Each instance runs as an independent thread. Count+ with each run. Shows cannot rely on Scheduler to run threads for equal number of times, or in particular order.

# Understanding Java threads, Part 3: Thread scheduling and wait/notify

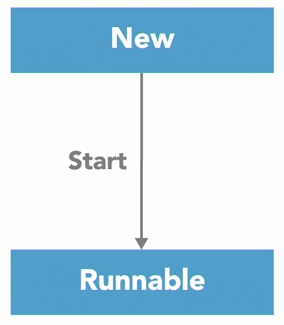
[ahttps://www.infoworld.com/article/2071214/java-101--understanding-java-threads--part-3--thread-scheduling-and-wait-notify.htmlnd wait/notify | InfoWorld](https://www.infoworld.com/article/2071214/java-101--understanding-java-threads--part-3--thread-scheduling-and-wait-notify.html)

Thread lifecycle

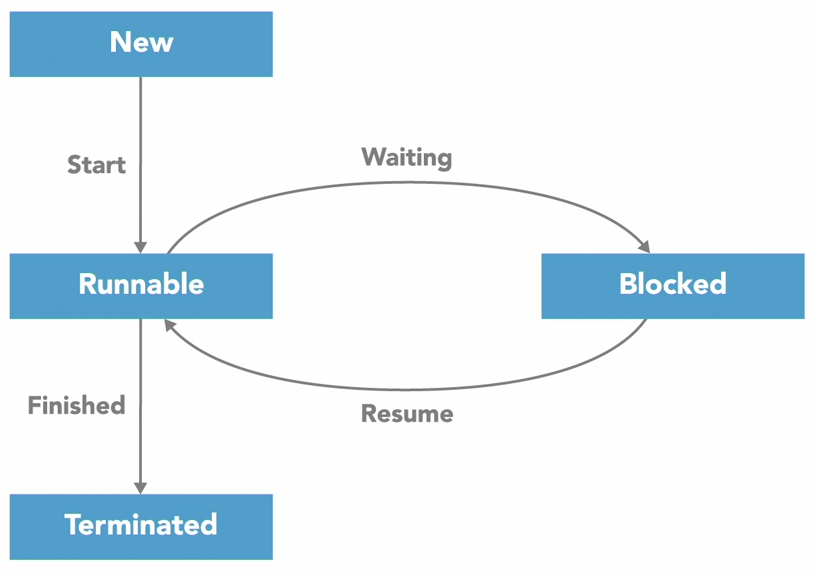
Every time a program starts running, it begins with a single thread – Main thread.

Main thread can start new threads

* When need to execute other tasks
* Same process (program instance)
* Run independent from parent thread
* Called child threads
* When finish executing, child threads will notify parent thread and terminate
* 1st generation parent thread usually last to terminate for program

A thread has 4 main states (6 states for Java)

1. New
   1. All threads begin in new state
   2. Does not take up CPU resources
   3. Assigned new function / code to execute
   4. Some programming language needs explicit instruction for thread to start running
2. Runnable
   1. in JVM
   2. In Ready Queue
   3. Execute when context switch by OS Scheduler to an available processor
3. Blocked
   1. Running thread goes into blocked state when need to wait for I/O input or has a timer
   2. Not using CPU resources in blocked state. Other processes can use CPU resources.
   3. Waiting for monitor lock in java.
   4. OS return thread to runnable state when ready.
   5. If thread A needs to wait for thread B to complete its execution, can use threadB.join() method in code. Waiting thread A will enter blocked state.
4. Terminate
   1. When completed execution, or aborted because of errors



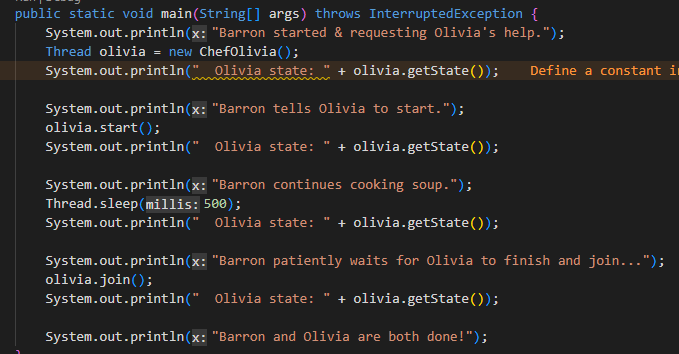
2 extra states for Java

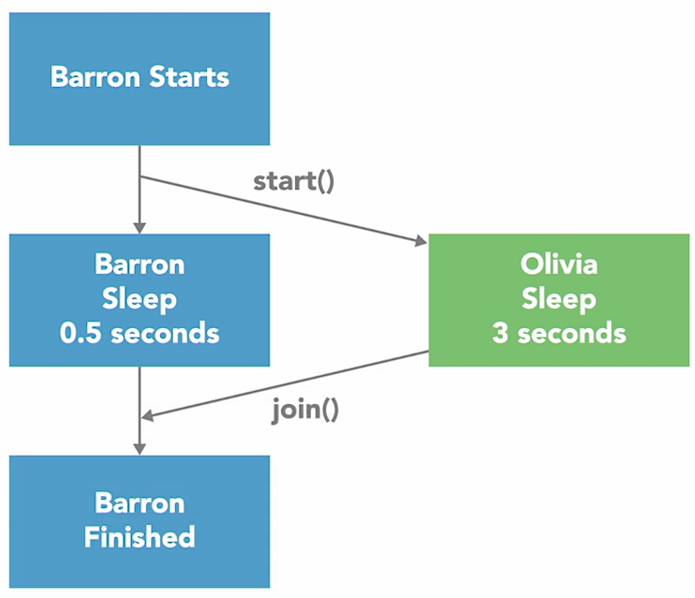
* more specific variation of “blocked” state

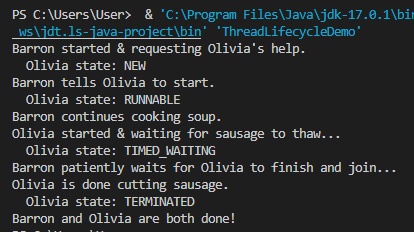
1. Waiting
   * wait indefinitely
2. Timed\_waiting
   * wait for a specific duration

Thread lifecycle: Java demo

/src/ThreadLifecycleDemo.java

- demonstrates the various states of lifecycle





Thread attributes

Properties and methods for thread class

1. thread.getState()

- returns one of the 6 states

2. thread.currentThread()

- returns instance of the Thread that is currently executing

3. thread ID

- a positive long integer

- generated when thread created

- unique, unchanged during lifetime of thread

- caution: when thread terminated, the number can be reused for another thread

- thread.getId()

4. thread Name

- string for identification

- multiple threads can have same name. Thread Id is unique.

- a few ways to pass in thread name when thread created

Method 1

//Helper class

class ThreadNaming extends Thread {

//parameterized constructor

ThreadNaming (String name) {

//call to constructor of Thread class as super keyword refers to parent class

super(name)

}

}

Method 2

ThreadNaming t1 = new ThreadNaming(“tread-1”);

- if thread name not specified during thread creation, java will assign default using *Thread-N* (e.g. Thread-0, Thread-1) format.

- if thread already exists, use setName method: thread.setName(String name)

- get name: thread.getName()

5. thread Priority

- tells JVM’s Scheduler which thread to run first.

- 1 (lowest priority) -> 10 (highest priority)

- to change a thread’s priority: void thread.setPriority(int newPriority)

- must be in the range MIN\_priority to MAX\_priority

- to check a thread’s priority: int thread.getPriority()

Child thread does not have reference to parent thread.

- parent thread affects child thread’s attributes, e.g. priority

- allows parent thread to be garbage collected by JVM to reclaim memory. Cannot be done if child thread points to parent thread.

Implement Runnable interface vs inheriting Thread class: Java demo

Thread class implements Runnable interface.

2 ways to create Thread:

|  |  |
| --- | --- |
| inheritance | implements interface |
| class Main **extends** Thread | class Main **implements** Runnable |
| Main is a custom class that inherits attributes and methods of Thread class.  Thread class – superclass (parent)  Main class – subclass (child)  Override run() method of Thread class, provide own custom codes.  class Main extends Thread {  @override  public void run() {...}  } | Runnable interface is an abstract “class” that groups abstract methods with “empty” bodies.  To access the interface methods, the interface must be implemented. Method bodies are provided by the “implement” Main class.  Runnable interface  \* cannot be used to create objects  \* cannot have constructor  \* interface methods are abstract and public by default  \* interface attributes are public, static and final by default |
| A class can **only** extend from **one** superclass. | A class can implement multiple interfaces  Example: class Main implements firstInterface, secondInterface {........}  Can still extend 1 other class.  Example: class Main extends superClass implements firstInterface, secondInterface {...} |
| Create an instance of the thread subclass and call start() method.  public class Main extends Thread {  public static void main (String[] args) {  Main thread = new Main();  thread.start();  System.out.println(“main method by main thread”);  }  public void run() {  System.out.println(“run method by child thread”);  }  } | Create an instance of the class. Pass to Thread object’s constructor. Call thread’s start() method.  Class Main extends Runnable {  public static void main (String[] args) {  Main instance = new Main();  Thread thread = new Thread(instance);  //same: Thread thread = new Thread(new Main);  thread.start();  System.out.println(“main method by main thread”)  }  public void run() {  System.out.println(“run method by child thread”);  } |
| If need more threads, create more instances and run them as threads.  Main thread2 = new Main();  thread2.start();  ...  Each instance is a seperate unique object. Own memory space.  Order cannot be predicted, determined by JVM. | Can create multiple threads from same instance.  Thread thread2 = new Thread(instance);  thread2.start();  ...  Benefit: 1 instance for multiple threads reduce memory use.  Any instance variables will be used and affected by all threads. |

Aside (Revision): SOLID Principle

Reduces tight coupling (mutual dependency of classes).

Entity: class, function, module etc

S – Single Responsibility Principle

A software entity should have only one responsibility/purpose. Does one operation only.

If a class handles many operations, any change to code may have unwanted effect on implementation of other operations.

Class with single method can be reused more easily. Flexible - can be combined with other functions for other tasks. Extensible.

Code is clean. No repetition. Only one place to change if need up update/modify.

Each function can be tested seperately.

O - Open/Closed Principle

Closed for modification, Open for extension.

Developer A write class A. Developer B should be able to extend class A for modifications, but not change class A itself.

L - Liskov’s Substitution Principle

Parent class should be able to be substituted by child class.

public class Animal {

public void makeNoise{“making noise”);

}

public class Dog extends Animal {

public void makeNoise(“wow wow”);

}

I - Interface Substitution Principle

Interface should be specific for function. Many specific interfaces better than 1 general interface.

If interface covers multiple methods, developer will be forced to implement unneeded methods.

D – Dependency Inversion Principle

class B should depends on abstraction A, but not on concrete class.

Have instances/objects of interfaces A to communicate with concrete class B. If there are changes to concrete class B, abstract class A does not need to know details.

Daemon Thread

Threads are often created to provide service or perform periodic tasks in support of main thread.

Can be normal child thread or daemon thread.

|  |  |
| --- | --- |
| Normal child thread | Daemon (background) thread |
| Executes codes independent of parent thread | Detached from main program. |
| - Parent main thread that has completed execution cannot exit, if child thread is still running.  - If child thread runs in continuous loop, e.g. garbage collection, parent thread will wait forever. Program will never exit. | - Does not prevent program from terminating.  - main thread is normal thread, so all child threads are by default non-daemon.  - Have to explicitly turn a thread into a daemon thread.  - Use thread.setDaemon(true) method to change status before starting thread.  - new threads from daemon thread will inherit daemon status  - When main thread terminates, and no non-daemon threads are still running, process ends. JVM stops. Daemon threads abandoned, terminated abruptly. No “finally” block. No unwind.  -> ok for garbage collection because all memory used by process will be cleared upon termination.  -> but, if daemon thread used for background I/O operation, .e.g writing to file, premature abrupt termination could corrupt data. |

Garbage collector

- automatic memory mangement

- reclaims memory no longer used by program.

- runs in background

- often part of standard runtime environment

*src*/DaemonThreadDemo

olivia.setDaemon(true);

- Addition of thread.setDaemon(true) before thread.start() change status to daemon thread.