Programming in Java

7. Concurrency in Java

Basic Definitions

- Agent
  - Anything capable of executing a task – person, CPU, host, etc.
- Multi-tasking
  - One agent working on more than one task over a period of time
  - Only one task is being executed at any given moment in time
  - The agent switches between tasks according some strategy
- Preemptive time slicing
  - Most common strategy for multi-tasking in operating systems
  - A time period is divided into equal time slices (e.g. 1 second is divided into 1000 slices)
  - Each task gets exclusive access to system resources for one time slice
  - Then it is “preempted” and forced to give up the resources so another task can run
  - Higher priority tasks get more slices (more turns to use resources) rather than longer slices
  - Necessary to prevent any task from locking up the system

Multitasking versus Parallel Processing

- Working from home with multiple tasks needing to be done
  - Cooking dinner, writing code, doing laundry, grocery shopping, etc
  - You have to switch between tasks – but you are the only agent doing these tasks
  - You will use some sort of switching strategy to move from task to task
- Parallel Processing
  - A set of agents are available
  - Each task can be executed exclusively by one agent
  - Multi-tasking without having to switch tasks
- Working from home with a staff
  - My chef cooks dinner
  - My housekeeper does the laundry
  - My chauffeur does the grocery shopping
  - I just write code

Multiprocessing versus Multi-threading

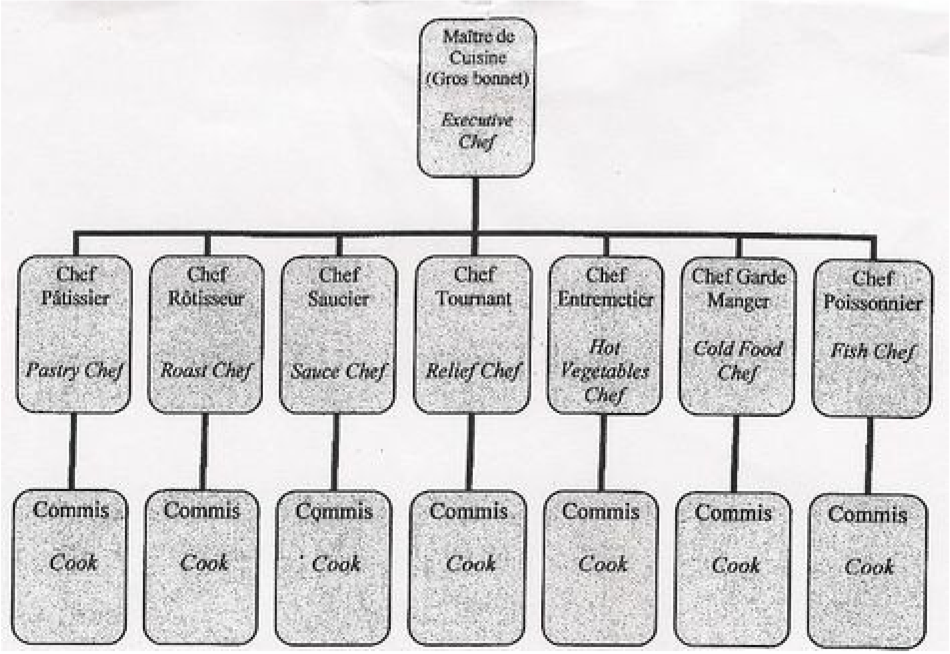
- Multiprocessing
  - Each task is as a self-contained in a totally isolated environment
  - Switching tasks requires saving the executing environment and loading the new task's environment
  - Can be very slow
  - Often called heavyweight multi-tasking
- Recall multi-tasking example
  - To switch from coding to grocery shopping to cooking dinner requires a complete change environment
- Multi-threading
  - Often called lightweight multi-processing
  - A group of related tasks in the same environment is are called threads
  - The agent can switch between tasks without having to change the environment
  - The agent just switches from task to task inside the same environment

Multiprocessing versus Multi-threading

- Cooking dinner involves a number of related tasks
  - Making salad, cooking rice, grilling meat, etc
  - These all take place in the same environment
  - The agent can switch from task to task in the same environment
  - Much more efficient than multi-processing
- Modern operating systems kernel
  - Have multi-threading capabilities to utilize multiple cores
  - Process – defines the general process properties (like open files) and address space
  - Kernel thread – a sequential execution stream within a process
- The JVM is a process
  - JVM threads (called user threads) are defined by the programmer
  - These are mapped to kernel threads available by the JVM

Kitchen Example

- In a Michelin four star restaurant
  - A kitchen is divided into stations that specialize in one aspect of meal preparation
  - Called the kitchen brigade system
  - Every meal is divided tasks which are sent to the stations
    - *All pasty request go to the pastry station*
    - *Salad request go to the salad station*
  - Each station represents an environment
  - Each request to a station is a thread
    - *"I need two Caesar salads, one Waldorff salad and three Cobb salads"*
  - The salad chef can easily multi-task in the salad station environment
- This is an example of real world multi-threading



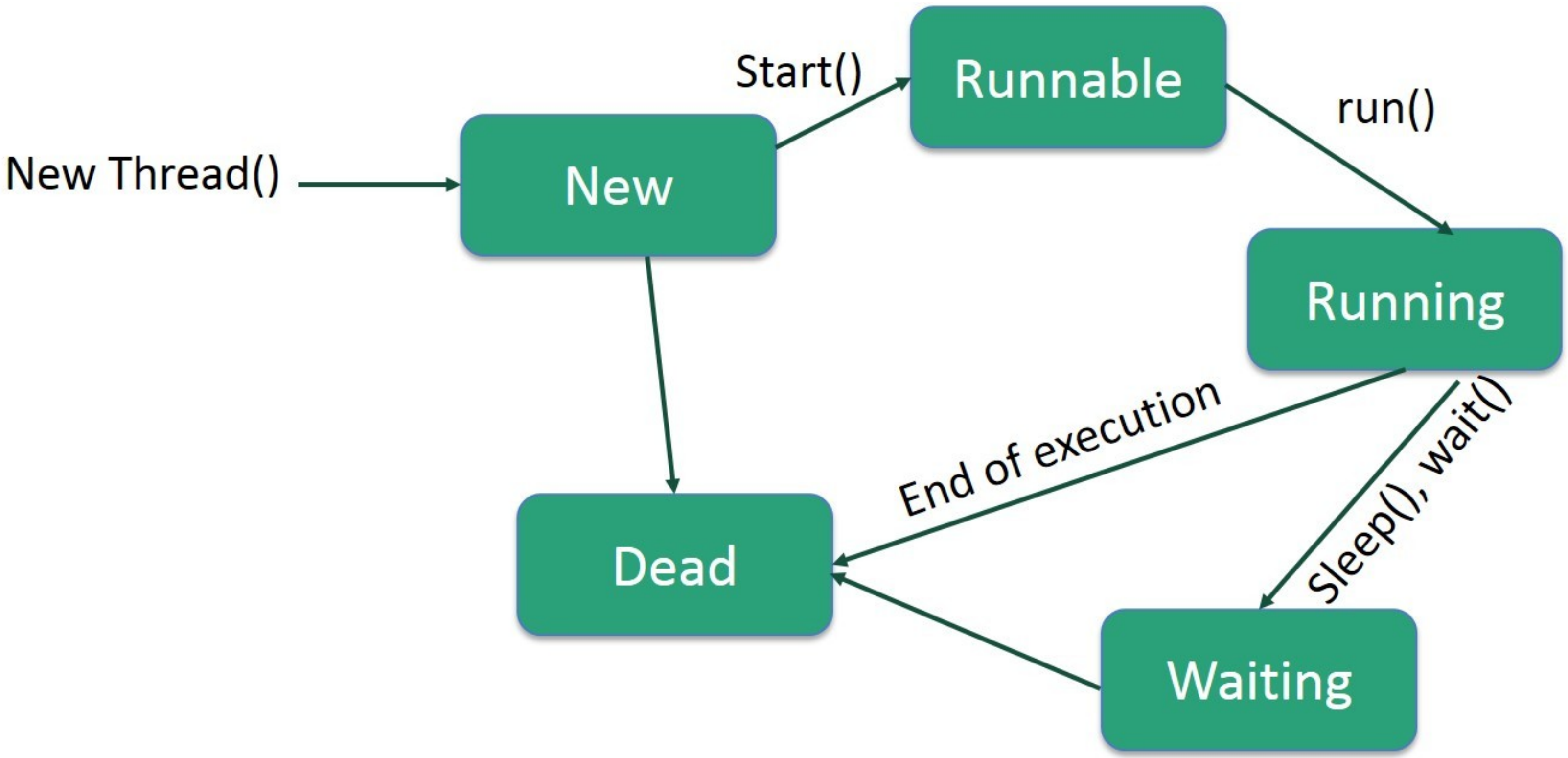
Java Threads

- The Java Thread class is used to create and manage Java threads
- **Java Thread** is a single sequence of execution – it is the smallest unit of processing that can be scheduled for a time slice
- **Java Thread Group** is a group of threads that are managed as a unit
  - We often want to start and stop all the threads in a group together for example
- **User** threads are defined by the user
  - The *main()* method creates a user thread when it starts
  - If no user threads are running, the JVM exits
- **Daemon** threads perform background tasks
  - Memory management, garbage collections, etc
  - A JVM exits if only daemon threads are running

The Thread Class

- The Thread class has all of the functionality to create and run a thread
  - We create threads by extending the Thread class or instantiating it directly
- The *start()* method
  - Initializes the thread object so that it can be scheduled to run as a task
  - Once the thread is started, the code in the *run()* method is executed
- The *run()* method
  - Contains the code we want the thread to execute
  - We are limited to a single method that returns void and takes no arguments
- The *x.join()* method
  - Waits until the thread x stops before running
  - Used to synchronize threads that have to work together

Thread States



Thread States

- New
  - Initial state of a thread
  - The thread stays in this state until the JVM schedules the thread
- Runnable
  - After the thread is scheduled, it can be executed and is considered runnable
- Waiting
  - When waiting for another thread to complete a task (the join() operation)
  - Thread switches to runnable when it receives a signal that it can proceed
- Timed Waiting
  - Enters the waiting state for a specified interval of time
  - Transitions back to the runnable state when either the time interval expires or when the event it is waiting for occurs
- Terminated
  - A thread that has been killed or has completed its task

Creating Threads

- The older style of creating threads is to use the Thread class
- Extending
  - We extend the Thread class and override the Thread run() method
- Implementing
  - We create a Thread object passing a runnable object in the constructor
  - A Runnable object is any class that implements the Runnable interface
  - Lambda functions are by definition Runnable so they can be used as well

Lab 7-1

Creating and Running Threads

Thread Attributes

- Threads have a number attributes that we can access
  - Name: we can set a printable name for a thread, or let the JVM generate one (like “Thread-0”)
  - ID: A unique id for the thread object
  - State: The current state of the thread
  - Daemon: Whether the thread is daemon or not
- Each thread has a priority
  - Lowest is MIN_PRIORITY (1)
  - Default is NORM_PRIORITY (5)
  - Highest is MAX_PRIORITY (10)
- Priority is used to schedule threads
  - Higher priority threads get more turns to run
  - However, priorities cannot ensure any specific order of execution
  - The environment the JVM is executing in has an impact on order of execution

Thread Lifetimes

- The JVM will not exit until all the user threads have completed
- Throwing an exception in one thread does not cause other threads to stop
- Threads spawned by the main thread will continue if the main thread exits

Demo

Thread Properties and Exceptions

The Resource Problem

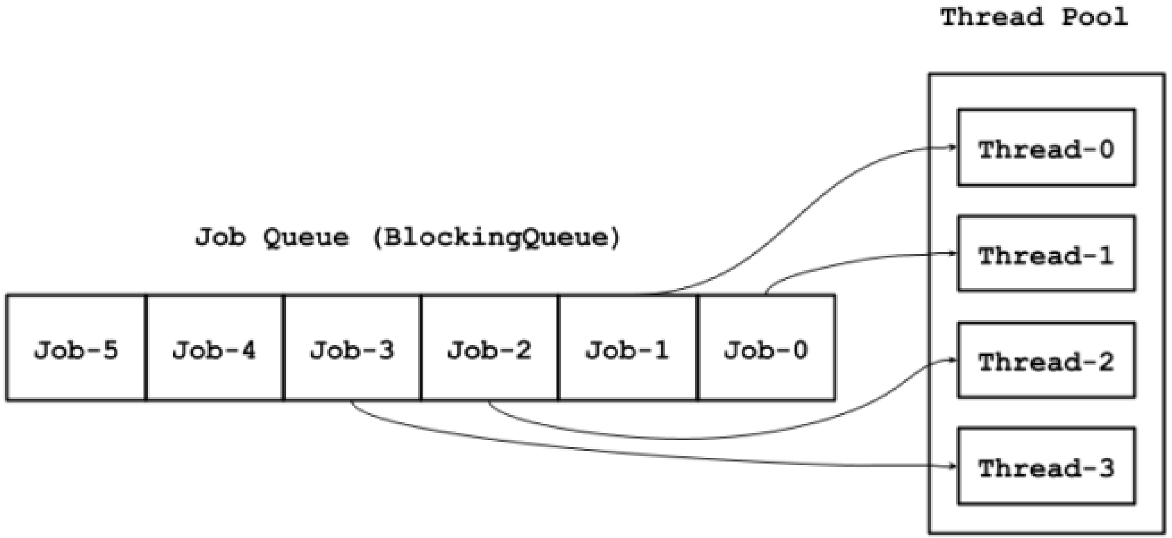
- Threads are often used for task that are:
  - Sort in duration
  - Called very frequently
- The problems with managing any kind of resource with these characteristics are
  - The amount of time spent creating and shutting down threads starts to become significant – the system starts to “thrash” trying to manage the threads
  - Too many threads can cause out of memory issues
- Managing threads at the low level we have been doing:
  - Requires writing a lot of boilerplate code
  - Is time consuming and error prone
- The solution is to delegate the actual creating and running of threads to the JRE

Pooling Resources

- Resource pool - collection of pre-created resources that are available on demand
  - This is a standard architectural strategy
  - Flyweight design pattern
- A thread pool contains a number of pre-created threads
  - Delay introduced by thread creating is eliminated, a thread is just selected from the pool
  - The thread is passed a Runnable object and then executes it
  - When the thread finishes executing, it is returned to the pool to be reused later
- Reduces thread life-cycle overhead and thrashing
- Allows resource limits to be set, like the maximum number of threads
- Allows programmers to concentrate on the executable code inside the run() method instead of the overhead of thread management
- This is the preferred way to create and use threads in modern Java programming

Executor Services

- The Executor interface
  - Creates a pool of threads and a queue to hold jobs
  - A runnable object to be executed in a thread is called a “job”
  - When a request comes in via a job queue, it is allocated to a thread which performs the task
  - When the task is finished, the thread is returned to the pool



Executor Services

- Java provides a concurrency library which supports a built-in Java thread pool
- Implemented as three interfaces
- An *Executor* interface that provides a replacement to the standard thread syntax.
  - *(new Thread(runnablecode).start()* can be replaced by *e.execute(runnablecode)* where e is an instance of Executor
- *ExecutorService* interface extends the *Executor* interface to include a submit method for a run() method which returns a value
  - We will not be covering this interface in this class
- *ScheduledExecutorService* which adds methods to allow scheduling of threads

Executor Services

- Start by allocating a Thread pool using one of the constructors (factory method)
  - The following code implements a fixed size Executor service
  - The shutdown() message
    - *Stops the service from accepting new tasks*
    - *Shuts the service down when all the executing threads have exited*

```
    public static void main(String[] args) {

        // Creates a new Thread Pool with 3 executors
        ExecutorService myPool = Executors.newFixedThreadPool(3);

        // Shuts the pool down once all the threads have terminated
        myPool.shutdown();

    }
}
```

Submit a Task

- Once the pool is started
  - Runnable tasks are submitted via the execute() method
  - Execute queues up the task, and when a thread is available, passes the task to the thread then executes the start() method on the thread

```
// Creates a new Thread Pool with 3 executors

ExecutorService myPool = Executors.newFixedThreadPool(3);
for (int i = 1; i < 5; i++) {
    myPool.execute(new MyTask("Task " + i));
}
// Shuts the pool down once all the threads have terminated
myPool.shutdown();
```

Customized Executor

- We can also create our own service with customized parameters
  - Core threads – the number of threads to start with
  - Max threads – the number of threads that can the executor service can scale up to
  - Keep alive – the amount of time to keep an executor running when idle
  - Time units – the time units used to measure the keep alive
  - BlockingQueue – the queue object to be used by the pool

- Executors are designed to be highly configurable

Customized Executor Submission

- Exactly the same as before
- The parameters are tuned for performance based on our requirements
- And based on performance history

```
public static void main(String[] args) {

    int corePoolSize = 3;
    int maxPoolSize = 5;
    long keepAliveTime = 3000;
    BlockingQueue<Runnable> pool = new ArrayBlockingQueue<Runnable>(100);


    ExecutorService myPool = new ThreadPoolExecutor(
            corePoolSize, maxPoolSize, keepAliveTime, TimeUnit.MILLISECONDS,pool);
```

Lab 7-2

Thread Executor Services

Java™