Advanced Programming Methods

Iuliana Bocicor
maria.bocicor@ubbcluj.ro

Babes-Bolyai University

2024

Overview

Lifecycle and states of a thread I

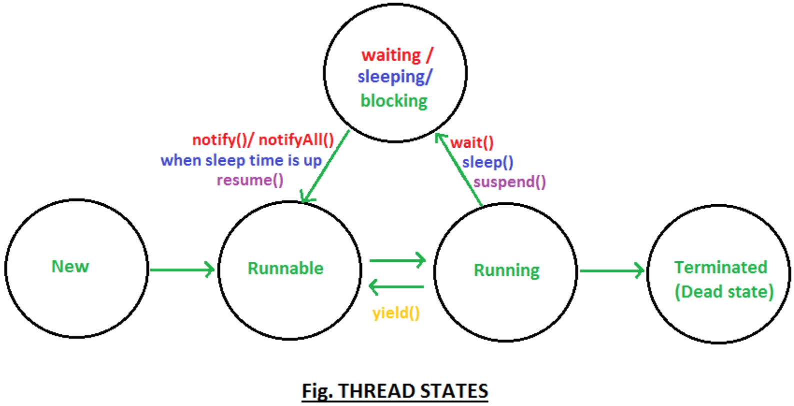

Figure: Figure source: Thread states/Thread lifecycle in Java

Lifecycle and states of a thread II

- **New thread**
  - the Thread object has been created (instantiated).
  - The start() method has not yet been called, thus the thread cannot run.
  - The thread has not yet started.
- **Runnable**
  - The thread enters its runnable state when its start() method is called.
  - This method does the following:
    - allocates necessary system resources.
    - schedules the thread to be executed by the CPU.

Lifecycle and states of a thread III

- A thread can also enter the runnable state after coming back from a *blocked*, *waiting* or a *sleeping* state.
- A thread in the runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as processor.

- **Running**
  - The thread is selected by the thread scheduler and is now actually running.
  - The thread is selected from the pool of runnable threads based on its priority.
  - By default, a thread inherits its priority from the thread that created it, but the priority can be modified (by the setPriority() method.
  - The method run of the object represented by the thread is called and the thread is running.

Lifecycle and states of a thread IV

- While the thread is running, its yield() method can make it pass to *Runnable* state, by "yielding" its running place to another thread.

- **Waiting/blocked/sleeping**
  - If a thread is in one of these states, it is still alive, but it is not running, because of one of the reasons:
    - it is *sleeping* - by calling its method sleep(); the thread will wait for the indicated sleep time to be over to return to its runnable state.
    - it is *waiting* - by calling its wait() method; the thread will wait for other threads to release the object monitor or lock.
    - it is *blocked* - when it is trying to acquire a lock on an object, but there is another thread that is holding the lock.

Lifecycle and states of a thread V

- **Dead (terminated)**
  - A thread is being normally terminated when the run() method completes.
  - A thread can also terminate if some unusual erroneous event happens, like segmentation fault or an unhandled exception.
  - A dead thread no longer consumes any cycles of CPU.

Thread synchronisation I

- All threads have access to the same heap.
- This can cause problems, when a certain object (that is a common resource for all threads) is accessed from two or more different threads.
- A **race condition** happens when when multiple threads try to access (modify) a shared resource at the same time (they race each other). The outcome is unpredictable and this leads to inconsistent results.
- Little or no synchronisation can result in data loss or corruption.
- Too much synchronisation can result in a deadlock (each thread is waiting for the other threads to take action and all are blocked).

Thread synchronisation II

- A **critical section** is a block of code that accesses a shared resource and which can be owned by only one thread at a time.
- Whenever a thread wants to access a critical section it must use some synchronisation mechanism to find out whether there is another thread executing the critical section.
- If there are no other threads, the thread can execute the critical section.
- Otherwise it is suspended by the synchronisation mechanism until the other thread that is executing the critical section finishes the execution.

Thread synchronisation III

- A **monitor** is a synchronisation construct.
- Each object in Java is associated with a monitor, which a thread can lock or unlock.
- Only one thread at a time may hold a lock on a monitor.
- Other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor.
  - The **mutual exclusion (mutex)** mechanism - is supported via object locks and allows threads to work independently on shared data, without interfering with each other.
  - **Cooperation** - communication via conditions; is supported via the wait() and notify() methods and enables threads to work together towards a common goal.

Multi-thread synchronisation - producer-consumer I

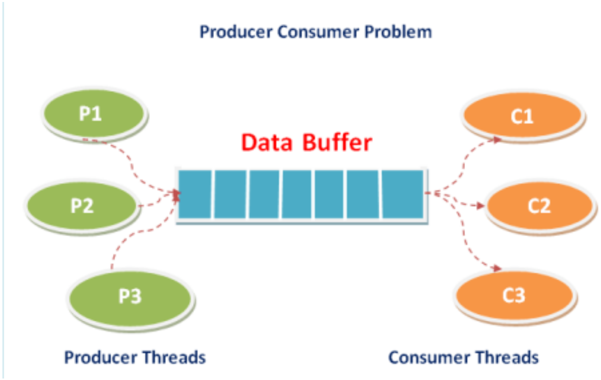

Figure: Figure source: Producer Consumer Problem

Multi-thread synchronisation - producer-consumer II

- Two threads, the producer and the consumer, share the same buffer.
- The producer puts data in the resource, while the consumer is consuming the data.
- The problem is to make sure the producer will not add data to a full buffer and the consumer will not try to extract data from an empty buffer.
- **Example**: depositing and extracting money from a bank account.

Producer - consumer

producer_consumer.Main

Mutual exclusion (mutex) I

- A **mutex** is a simple type of synchroniser that ensures that only one thread can manipulate a certain resource at a time.
- If a thread calls a **synchronized** method on an object, first it is checked if the object is available.
- If yes, the object is marked as locked and the thread begins executing the method. When the method completes, the object is again marked as available.
- If not, it means that there is another thread which executed a syncronized method on the same object. If that is the case, the thread will wait until the object becomes available.

Mutual exclusion (mutex) II

- One way to achieve synchronisation in Java is using the synchronized keyword in the method's prototype.

```
public synchronized void deposit(double amount)
```

- When one thread executes a synchronized method, no other threads are allowed to execute any synchronized methods with the same object as lock.

Communication via conditions I

- Mutual exclusion is necessary for solving a concurrency problem, but it is not sufficient.
- The threads must be able to notify each other when they finished a given task.
- If the consumer thread tries to withdraw money from an empty account (nothing has yet been deposited), the *withdraw* method will do nothing.
- What we really want is the consumer to **wait** until the producer deposits something.
- The producer should **notify** the consumers when something was deposited.

Communication via conditions II

- wait():
  - when a thread calls this method, the thread will be put ”on hold”.
  - this method blocks the thread until the continuation condition is fulfilled.
  - while the thread waits, the lock on the object is released, thus the object is available to other threads.
- notifyAll():
  - all other threads that are blocked in the waiting state (with the wait() method) **of the same object** are un-blocked and can resume.
  - this method notifies all threads that wait for the continuation condition that this has been fulfilled.

Producer - consumer

producer_consumer.CommunicationConditions

Deadlock

- A **deadlock** can occur if when a certain thread executes a synchronized method (thus the object is locked), however the execution cannot be finalised unless a certain condition is fulfilled.
- If such a condition is fulfilled only when another thread calls a synchronized method on the same object, the threads would wait after one another infinitely (the second thread can never call its synchronized method, given that the object was locked by the first thread).

Deadlock

producer_consumer.Deadlock

java.util.concurrent

- Starting with JDK 5.0 the JVM has been significantly improved for concurrent programming (classes can take advantage of hardware concurrency support).
- The java.util.concurrent package offers functions and data structures for building concurrent applications that are powerful, reliable and easily maintainable.

Thread pools I

- A classic mechanism for managing a large group of tasks is combining a **work queue** with a **thread pool**.
- A work queue is a queue of tasks that must be processed.
- A thread pool is a collection of threads that remove tasks from the queue and execute them.
- A thread pool reuses previously created threads to execute current tasks.
- Given that the thread already exists when a request arrives, there is no need to exhaust resources on thread creation, therefore the application will be more responsive.

Thread pools II

- Implementing good synchronisation is difficult when several complex tasks must be run in parallel and the result depends on their finalisation.
- To achieve this Java offers the Executor interface, which allows creating sets of threads, synchronising and executing them.
- By using the executor we only have to implement the Runnable objects and send them to the executor to execute.

```
public interface Executor {
    void execute (Runnable command);
}
```

Thread pools III

- A set of threads can be represented by an instance of the class ExecutorService.
- ExecutorService provides mechanisms for safely starting, closing down, submitting, executing tasks.
- This can have several types:
  - *Fixed thread pool executor*: creates a thread pool that contains a fixed number of threads and reuses them.
  - *Cached thread pool executor*: creates a thread pool that will reuse available threads, but will also create new ones if needed.
  - *Scheduled thread pool executor*: creates a thread pool that allows scheduling tasks to be executed after a given delay, or periodically.

Thread pools IV

- To terminate the threads inside the ExecutorService call the shutdown() method.
- This will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished their current tasks, the ExecutorService shuts down.

ExecutorService

executor_service.ExecutorServiceExample

Thread pools V

- Advantages of using thread pools:
  - Response time is reduced - threads are already created.
  - By just replacing the ExecutorService's implementation we can change our application from multiple thread to single thread and viceversa.
  - System stability is increased (the given number of threads are created based on system load and available resource).
  - The programmer is freed from thread management and can better focus on business logic.

Synchronisation classes I

- **Semaphore**
  - Allows a given number of threads to access a certain resource.
  - Uses a counter to control access to the resource.
  - When a thread wants to access a resource, it has to obtain permission from the semaphore.
  - If the semaphore's count is greater than 0, the thread is granted the permission and the count decreases.
  - If the count is zero the thread will not be granted permission.
  - When the thread no longer needs the resource, it siglans this to the semaphore, which increases the counter.
  - A mutex is a semaphore with an access count of 1.

Synchronisation classes II

- **CyclicBarrier**
  - Allows a set of threads to wait for each other in order to reach a common execution point, called a barrier.
  - A thread can signal having reached the barrier point by calling the await() method.
  - A thread that calls this method suspends its execution until a specified number of threads have called the same method on the barrier.

Synchronisation classes III

- **CountDownLatch**
  - Can be used to cause a thread to block until other threads have completed a given task.
  - It has a counter field that can be decremented using the countDown() method.
  - When a CountDownLatch object is created, the number of threads it should wait for is specified.
  - The other threads decrease the counter once they completed their tasks.
  - As soon as the counter reaches zero, the waiting thread can start.

Synchronisation classes IV

- **ReentrantLock**
  - The syncronisation using synchronized is somewhat rigid:
    - a thread ca lock only once;
    - after the resource is released, any thread can grab the lock - there is no queuing possibility;
    - this could mean that some other thread will wait for a very long period of time.
  - Reentrant locks provide greater flexibility.
  - They provide synchronization for shared resources.
  - They allow a thread to enter a lock more than once.
  - With a reentrant lock, a lock can be held even between methods.
  - This class' constructor accepts an optional *fairness* parameter and if this is set to true, locks favour granting access to the longest-waiting thread.

Virtual threads I

- Virtual threads have been introduced as a major new feature in JDK 21.
- These are lightweight threads managed by the JVM, rather than the operating system.
- Unlike traditional threads, which are mapped to OS threads, virtual threads are managed by the Java runtime (they are more efficient with regard to memory and context-switching overhead).
- They are more lightweight - require a small amount of memory and thus thousands or millions of virtual threads can be created.

Virtual threads II

- Virtual threads are particularly useful in cases where there are blocking operations, e.g. network or disk I/O operations.
- When a virtual thread performs a blocking operation (e.g. I/O), while it waits for the operation to complete, it yields control to the JVM, which can schedule another virtual thread to be executed in the meantime (using the same OS thread).

Thread-safe collections I

- Thread-safe code is code that manipulates data such that if several threads are working at the same time on the same data, the data will be safely managed.
- Hashtable, Vector - are thread-safe.
- Other collection types can be made thread-safe using methods in Collections: Collections.synchronizedMap(), Collections.synchronizedList() and Collections.synchronizedSet().
- Some new thread-safe collections offered in java.util.concurrent are ConcurrentHashMap, CopyOnWriteArrayList - improved version of Vector and CopyOnWriteArraySet - improved version of ArrayList.

Thread-safe collections II

- There are two new interfaces for manipulating queues: Queue and BlockingQueue.
- There are two implementations for the Queue interface:
  - ConcurrentLinkedQueue - FIFO access.
  - PriorityQueue.
- BlockingQueues are used for blocking a thread if certain queue operations cannot be executed (e.g. consumers consume data in the queue a lot slower than producers add it; using a blocking queue, producers are blocked until an element in the queue is extracted).
- Implementations for this interface are: LinkedBlockingQueue, PriorityBlockingQueue, ArrayBlockingQueue, SynchronousQueue.

javafx.concurrent

- JavaFX also offers a small concurrent framework API,
- It provides a Worker interface which represents a task that needs to be performed in one or more background threads.
- The Task class implements the above mentioned interface.
- The Task's call() method should be implemented to do whatever needs to be done in the background thread.

Background worker - JavaFX

FibonacciGUI

Summary

- Given that all threads have access the the same heap, some synchronisation mechanism must be put in place.
- Synchronisation mechanisms:
    - Mutual exclusion (mutex).
    - Read/write locks.
    - Communication via conditions.
    - Semaphores.
- The java.util.concurrent package offers functions and data structures for building concurrent applications that are powerful, reliable and easily maintainable.
- *Next week*:
    - Design patterns.