Java threading features

Java threading

- Concurrency
- Thread states
- Thread context switch
- Synchronization
- The liveness property

Concurrency

- A software that can perform many operations simultaneously is said to be *concurrent*
  - For example, an application can download files, play music, process some data, and update its display, all at the same time
- The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries

Processes

- In general terms, a *process* is an instance of a program, with a self-contained execution environment
- A process has a complete, private set of basic run-time resources
  - In particular, each process has its own memory space
- Most operating systems provide support for *inter-process communication* (IPC)
  - This enables processes, not necessarily on the same machine, to perform information exchange

Threads

- *Threads* are lightweight processes that share resources, such as memory, and open files
- Each process consists of at least one thread, often called the *main thread*
  - In case of Java, for each application there is an additional, *system* thread that performs memory management, signal processing, etc.
- When compared to processes:
  - Threads require less resources to create and execute
  - Inter-thread communication is much faster, with no additional overhead
  - But, the use of shared memory requires special handling, to avoid execution problems

Thread objects

- In Java, there are two ways of defining a thread:
  - By implementing the *Runnable* interface
  - By inheriting the *Thread* class, which, in turn, provides an empty implementation of the *Runnable* interface
- The second idiom is easier to use in simple applications, but is limited by the fact that the class must be a descendant of *Thread*
- In all cases, the code to be executed by the new thread is contained in the *run()* method: **public void run()**
- Since *Runnable* is a functional interface, lambda expressions can be used to define the *run()* method

Creating and running threads

- To create a thread, it is enough to instantiate an object of the class extending *Thread*
- On the other hand, the *Runnable* object needs to be passed to the *Thread* constructor
- The actual execution of a thread begins only after its *start()* method has been called

Creating and running threads - example

```
class MyThread1 implements Runnable {
  @Override
  public void run() {
    System.out.println("Implementing Runnable");
  }
}

class MyThread2 extends Thread {
  @Override
  public void run() {
    System.out.println("Extending Thread");
  }
}

public class SimpleThreads {
  public static void main(String[] args) {
    new Thread(new MyThread1()).start();
    new MyThread2().start();
  }
}
```

Creating and running threads – rules

- Never invoke the *run()* method directly!
  - This is because the *start()* methods performs additional steps before invoking *run()*

- It is never legal to start a thread more than once
  - In particular, a thread may not be restarted once it has completed execution

Java threading

- Concurrency
- Thread states
- Thread context switch
- Synchronization
- The liveness property

Thread states

- A thread can be in one of the following states:
  - *Ready*: the thread has been created, but it is not executing yet (i.e. its *start()* method has not been called)
  - *Running*: the thread is executing its *run()* method
  - *Suspended*: the thread's execution is paused
  - *Blocked*: the thread is waiting for some conditions to be met
  - *Terminated*: a thread is terminated when its *run()* method terminates

Suspending a thread

- Invoking the static *sleep()* method of the *Thread* class causes the current thread to suspend its execution
  - The method receives one parameter: sleep time
- This approach is often used to make processor time available to other threads of an application or other applications that might be running on a computer system
- That is, whenever a thread needs to execute a loop with a lot of iterations, it is good practice to put a *sleep()* in each iteration
  - Even short sleep times, such as 5 milliseconds, can reduce the overall CPU/core usage of the application from 100% to < 1% !

Suspending a thread - example

```
class ThreadNoSleep extends Thread {
  @Override
  public void run() {
    while (true) ;
  }
}

class ThreadSleep extends Thread {
  @Override
  public void run() {
    while (true)
      try {
        Thread.sleep(1);
      } catch (InterruptedException e)
      { ... }
  }
}
```

- new ThreadNoSleep().start(); // 100% CPU/core usage
- new ThreadSleep().start(); // CPU usage insignificant

Interrupting threads

- An *interrupt* is an indication to a thread that it should stop what it is doing and do something else
  - It is up to the developer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate
- A thread's support for interruption depends on what the thread is doing:
  - If the thread is frequently invoking methods that throw *InterruptedException*, such as the *sleep()* method, it simply catches that exception
  - Otherwise, the thread should periodically invoke the *Thread.interrupted()* method, which returns *true* if an interrupt has been received
- The interrupt mechanism is implemented using an internal flag known as the *interrupt status*
  - When a thread checks for an interrupt by invoking the static method *Thread.interrupted()*, interrupt status is cleared
  - The non-static *isInterrupted()* method does not change the flag

Interrupting threads – example

```java
class InterruptibleThread1 extends Thread {
  @Override
  public void run() {
    while (true) {
      // perform some data processing

      try {
        Thread.sleep(5);
      } catch (InterruptedException e) {
        return; // terminate
      }
    }
  }
}
```

Interrupting threads – example (cont’)

```
class InterruptibleThread2 extends Thread {
  @Override public void run() {
    while (true) {
      // perform some data processing

      if (Thread.interrupted())
        return; // terminate
    }
  }
}

Thread t1 = new InterruptibleThread1();
t1.start();
Thread t2 = new InterruptibleThread2();
t2.start();
// let the threads execute for some time
Thread.sleep(10000);
// terminate threads
t1.interrupt();
t2.interrupt();
```

Joins

- The *join()* method allows one thread to wait for the completion of another; example:

```
class OutputProducer extends Thread {
  private String fileName;
  private boolean ok;

  public OutputProducer(String fileName) { this.fileName = fileName; }

  public String getFileName() { return fileName; }

  public boolean isOk() { return ok; }

  @Override public void run() {
    try {
      FileWriter f = new FileWriter(fileName);
      f.append("Hello");
      f.close();
      ok = true;
    } catch (IOException e) { ... }
  }
}
```

Joins – example (cont')

```
class InputConsumer extends Thread {
  private OutputProducer output;

  public InputConsumer(OutputProducer output) {
    this.output = output;
  }

  @Override public void run() {
    try {
      output.join(); // wait for the producer to finish
      if (output.isOk()) {
        BufferedReader f = new BufferedReader(
          new InputStreamReader(
          new FileInputStream(output.getFileName())));
        System.out.println(f.readLine());
        f.close();
      }
    } catch (Exception e) { ... }
  }
}
```

Java threading

- Concurrency
- Thread states
- Thread context switch
- Synchronization
- The liveness property

Thread context switch

- Parallel execution of multiple threads on single-CPU (single-core) systems is not truly parallel
- Instead, the CPU executes a few instructions of one thread for a brief amount of time, and then switches the execution to another thread
- By default, Java threads will run on multiple cores if they are available, but if the number of threads exceeds the number of cores, switching will have to occur
- The process of saving the execution state of one thread and then restoring the state of another is known as *thread context switch*
- It gives the perception of threads being executed simultaneously, when, in reality, they are not

Thread context switch consequences

- The first important thing to remember is that any Java expression or statement, no matter how simple it looks, may actually be translated into a number of sub-steps. For example, the instructions for **i++** might include:
  - Reading the current value of **i** into memory
  - Increasing the memory value by 1
  - Saving the increased value back to **i**
- The second important thing to remember is that the context switch can occur *at any point* between sub-steps and *no assumptions can be made about when it will occur*
  - So, in the above example, the thread might execute **i++** without interruptions, or a context switch might occur after any (or all) sub-steps

Context switch – example

```
class Counter {
  private int value;

  public void increase() {
    value++;
  }

  public int getValue() {
    return value;
  }
}

class CounterThread extends Thread {
  private Counter counter;

  public CounterThread(Counter counter) {
    this.counter = counter;
  }

  @Override
  public void run() {
    counter.increase();
    System.out.print(getName() + ": " + counter.getValue() + "; ");
  }
}
```

Context switch – example (cont')

```
public class ContextSwitch {
  public static void main(String[] args) {
    Counter c = new Counter();
    Thread t0 = new CounterThread(c);
    Thread t1 = new CounterThread(c);
    t0.start();
    t1.start();
  }
}
```

- Some of the possible outputs are:

```
Thread-0: 1; Thread-1: 1;

Thread-0: 1; Thread-1: 2;

Thread-0: 2; Thread-1: 1;

Thread-0: 2; Thread-1: 2;
```

Context switch – example (cont')

- The output "Thread-0: 1; Thread-1: 1;" appears as the outcome of the following scenario:
  - T0 reads the counter's value into memory and adds 1;
  - Context switch occurs
  - T1 reads the counter's value into memory, adds 1 to it, and saves the result back into the variable
  - Context switch occurs
  - T0 saves its result back into the variable and prints out the result (Thread-0: 1)
  - Context switch occurs
  - T1 prints out the result (Thread-1: 1)
- As noted, this is only one of many possible outcomes; the actual behavior is unpredictable

Thread interference

- If several threads happen to operate on the same data, the problem of *thread interference* might occur
  - As described in the previous scenario, thread T1 interferes with T0’s instructions for incrementing the counter, producing the invalid result
- In order to avoid this behavior, a thread must “protect” its execution while operating on shared data
  - In other words, threads that operate on the same data must *synchronize* their execution to prevent thread interference and memory consistence errors
- There are some situations where thread synchronization is not necessary; for example reading a variable that has been declared as *final* is always safe

Java threading

- Concurrency
- Thread states
- Thread context switch
- Synchronization
- The liveness property

Synchronization

- Basic synchronization techniques in Java include:
  - Synchronized methods
  - Synchronized statements
  - Locks
- Additionally, the Java library provides more advanced synchronization solutions, such as semaphores

Synchronized methods

- Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors
- It is not possible for two invocations of synchronized methods on the same object to interleave:
  - When one thread is executing a synchronized method for an object, all other threads that invoke any (not necessarily the same) synchronized method for the same object stop their execution (become blocked) until the first thread is done with the object
- When a synchronized method exists, it is guaranteed that changes to the state of the object are visible to all threads
- So, if an object is visible to more than one thread, all reads or writes to that object’s variables should be done through synchronized methods

Synchronized methods – example

```
class SynchronizedCounter {
  private int counter;

  public synchronized void increase() {
    counter++;
  }

  public synchronized void decrease() {
    counter--;
  }

  public synchronized int getValue() {
    return counter;
  }
}
```

- Now, multiple threads sharing and accessing the same *SynchronizedCounter* object will always have the correct view of its state

Intrinsic locks

- Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock* (or simply, *monitor*)
- Every object has an intrinsic lock associated with it
  - The lock enforces exclusive access to the object’s state
- When a thread invokes a synchronized method, it automatically *acquires* the intrinsic lock for that method’s object and *releases* it when the method returns
  - The lock release occurs even if the return was caused by an uncaught exception
- When a thread tries to acquire a lock that is already owned by another thread, it will block until the lock is released

More on intrinsic locks

- Although a thread cannot acquire a lock owned by another thread, it *can* acquire a lock that it already owns
- Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*
- Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block (e.g. when writing recursive synchronized methods)

Synchronized statements

- Instead of entire methods, only some statements might be protected with the *synchronized* keyword
- Synchronized statements are used to optimize or fine-tune the synchronization process in order to avoid unnecessary blocking
- Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock

Synchronized statements – example

```
class SyncStatements {
  private int c1;
  private int c2;
  // two intrinsic locks are used: one for c1,
  // and one for c2. this is because there is no
  // need to block the access to c1 when c2 is
  // being modified and vice versa
  private Object lock1 = new Object();
  private Object lock2 = new Object();

  public void inc1() {
    synchronized(lock1) { c1++; }
  }

  public void inc2() {
    synchronized(lock2) { c2++; }
  }
}
```

Lock objects

- Intrinsic locks are easy to use, but they impose one serious limitation: a thread cannot “back out” of acquiring an intrinsic lock
  - I.e. it either acquires the lock or becomes blocked
- Interface *Lock*, and its implementing class, *ReentrantLock*, offer several lock-acquiring methods:
  - **void lock()**: acquires the lock; if the lock is not available, the thread is blocked and the lock is acquired
  - **boolean tryLock()**: acquires the lock if it is available and returns immediately with the value *true*; if the lock is not available then this method will return immediately with the value *false*
  - **void lockInterruptibly()**: acquires the lock if it is available and returns immediately; otherwise, the thread is blocked until either the lock is acquired or until some other thread interrupts the current thread
- Each thread must manually release the acquired lock object, by calling its **unlock()** method
  - To insure that the release is always performed, the call should be put inside a *try … finally* block

Lock objects - example

```
import java.util.concurrent.locks.*;

class LockObjects extends Thread {
  private static Lock lock = new ReentrantLock();

  @Override public void run() {
    while (!Thread.interrupted()) {
      if (lock.tryLock())
        try {
          // access the resource protected by this lock
        } finally {
          // make sure the lock is always released
          lock.unlock();
        }
      else {
        // perform some other stuff
      }
    }
  }
}
```

The ReentrantReadWriteLock class

- In order to improve code performance, it is often recommended to distinguish between *read* access and *write* access to a shared resource
  - That is, in general case, while only a single thread at a time can modify the shared resource, any number of threads can concurrently read the resource
- The *ReentrantReadWriteLock* class maintains a pair of associated locks: one for reading, and one for writing operations
  - The read lock may be held simultaneously by multiple reader threads, so long as there are no writers
  - The write lock is exclusive
- The locks are implemented as inner classes *ReadLock* and *WriteLock*, and can be accessed via *readLock()* and *writeLock()* methods of their enclosing class, respectively

Java threading

- Concurrency
- Thread states
- Thread context switch
- Synchronization
- The liveness property

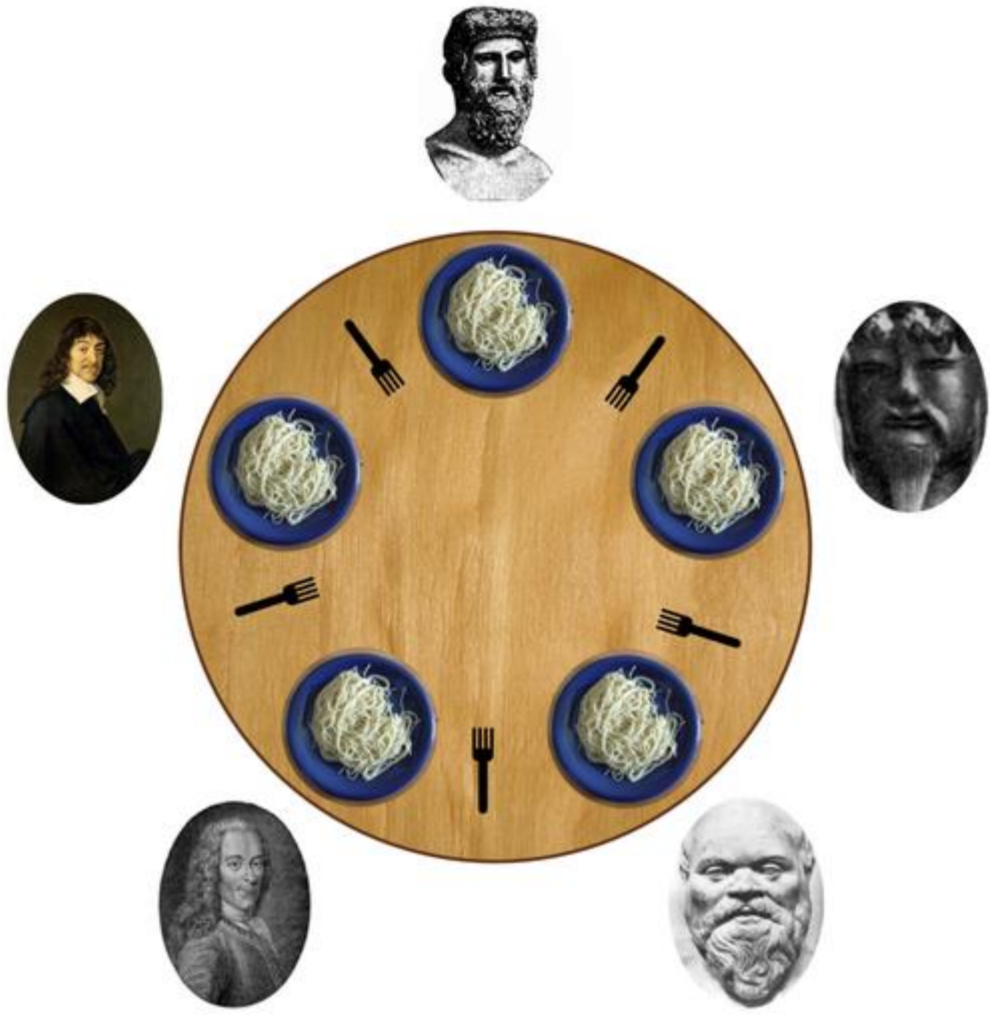
The liveness property

- A concurrent application's ability to execute in a timely manner is known as its *liveness*
- Poor concurrent design can lead to one of the following problems:
  - *Deadlock*: two or more threads are blocked forever (e.g. each waiting for the other to release a lock)
  - *Starvation*: a thread is unable to gain regular access to shared resources and is unable to make progress (thanks to other "greedy" threads)
  - *Livelock*: two or more threads cannot make progress because actions of each thread depend on actions of another thread (similar to pedestrians heading towards one another trying to avoid each other)
- *Assuring concurrent application's liveness is often the most difficult task of thread synchronization*

Case study – the *dining philosophers* problem

- The dining philosophers problem (*DPP*) will be used to demonstrate common pitfalls of concurrent application development
- Problem formulation:
  - 5 philosophers are sitting at a round table
  - In front of each philosopher there is a bowl of spaghetti
  - A fork is placed in between each pair of adjacent philosophers, and as such, each philosopher has one fork to his left and one fork to his right
  - A philosopher needs two forks for eating; he can only use the forks on his immediate left and immediate right
  - Each philosopher is either thinking or eating; if he's thinking, he's not eating, and vice versa
  - The philosophers never speak to each other

Case study – the *dining philosophers* problem



DPP solution 1

- The simplest solution would be as follows:

```
while (true)
{
  think();
  takeLeftFork(); // waits until the fork's available
  takeRightFork(); // waits until the fork's available
  eat();
  putLeftFork();
  putRightFork();
}
```

- Although straightforward, the solution can lead to a deadlock – how?

DPP solution 2

- Improvement: if you’ve got the left fork, and the right one is taken, put the left one down:

```
while (true) {
  think();
  while (true) {
    // wait until the fork's available
    takeLeftFork();
    // don't block if the fork's not available
    if (tryTakeRightFork())
      break;
    putLeftFork();
    wait(delta);
  }
  eat();
  putLeftFork();
  putRightFork();
}
```

- This solution can still lead to the starvation of an “unlucky” philosopher – why?

DPP solution 3 – introducing a “waiter”

- “Waiter” is an object that makes sure none of its customers are starving
- It keeps a count of how many times each philosopher wanted to eat, but couldn’t get the forks
- Once its count exceeds a certain threshold, the philosopher is assigned a higher priority:
  - A new rule is introduced that a philosopher cannot pick up forks if his neighbor is starving

DPP solution 3 – the “Philosopher” class

```
public enum State {
  THINKING, HUNGRY, EATING
}


public class Philosopher implements Runnable {
  private State state;
  private int id;
  private Waiter waiter;
  private static final Random rnd = new Random();

  public Philosopher(int id, Waiter waiter) {
    this.id = id;
    this.waiter = waiter;
    state = State.THINKING;
    new Thread(this).start();
  }
```

DPP solution 3 – the “Philosopher” class (cont’)

```
  @Override public void run() {
    try {
      while (true) {
        think();
        while (!waiter.takeForks(id))
          Thread.sleep(randomTime(1000, 2000));
        eat();
        waiter.putForks(id);
      }
    } catch (InterruptedException e) { }
  }

  private void think() throws InterruptedException {
    Thread.sleep(randomTime(1000, 5000));
  }

  private void eat() throws InterruptedException {
    Thread.sleep(randomTime(1000, 5000));
  }

  private int randomTime(int min, int max) {
    return min + rnd.nextInt(max - min);
  }
}
```

DPP solution 3 – the “Waiter” class

```
public class Waiter {
  private static final int NUM_PH = 5;
  private static final int MAX_WAIT = 10;
  private Philosopher[] phil;
  private int[] waitCount;
  private Set<Integer> starving;

  public Waiter() {
    waitCount = new int[NUM_PH];
    starving = new HashSet<Integer>();
    phil = new Philosopher[NUM_PH];
    for (int i = 0; i < NUM_PH; i++)
      phil[i] = new Philosopher(i, this);
  }

  private boolean starving(int id) {
    return starving.contains(id);
  }
```

DPP solution 3 – the “Waiter” class (cont’)

```
  public synchronized boolean takeForks(int id) {
    phil[id].setState(State.HUNGRY);
    int left = (id + NUM_PH - 1) % NUM_PH;
    int right = (id + 1) % NUM_PH;
    // id can eat only if none of its neighbors are eating
    boolean canEat = (phil[left].getState() != State.EATING)
        && (phil[right].getState() != State.EATING);
    // both forks are free. id can eat either if it's starving
    // or none of its neighbors are starving
    if (canEat)
      canEat = starving(id) || (!starving(left) && !starving(right));
    if (canEat) {
      phil[id].setState(State.EATING);
      waitCount[id] = 0;
      starving.remove(id);
      return true;
    }
    // id has to try again later
    if (++waitCount[id] > MAX_WAIT)
      starving.add(id);
    return false;
  }

  public synchronized void putForks(int id) {
    phil[id].setState(State.THINKING);
  }
}
```